The "Dark Side" of Code Refactoring: Revealing Harmful Effects on Structural Quality

Diego Cedrim
PUC-Rio, Brazil
dcgrego@inf.puc-rio.br

Alessandro Garcia
PUC-Rio, Brazil
afgarcia@inf.puc-rio.br

Melina Mongiovi
UFCG, Brazil
melina@copin.ufcg.edu.br

Rafael de Mello
PUC-Rio, Brazil
rmaiani@inf.puc-rio.br

Alexander Chávez
PUC-Rio, Brazil
alopez@inf.puc-rio.br

Rohit Gheyi
UFCG, Brazil
rohit@dsc.ufcg.edu.br

Baldoino Fonseca
UFAL, Brazil
baldoino@ic.ufal.br

Márcio Ribeiro
UFAL, Brazil
marcio@ic.ufal.br

Leonardo Sousa
PUC-Rio, Brazil
lsousa@inf.puc-rio.br

ABSTRACT

Code smells in a program represent indications of structural quality problems, which can be addressed by software refactoring. There is an explicit assumption that software refactoring improves the structural quality of a program by reducing its density of code smells. However, little has been reported about whether and to what extent developers neglect or end up creating code smells through refactoring. This paper reports a longitudinal study intended to address this gap. We analyze how often commonly-used refactoring types affect the density of 13 types of code smells along the version histories of 23 projects. Our findings are based on the analysis of 29,318 refactorings distributed in 10 different types. Even though more than 80% of the refactorings touched smelly elements, 63.2% did not reduce their occurrences. Surprisingly, only 12.3% of refactorings removed smells, while 24.5% induced the introduction of new ones. More than 95% of such refactoring-induced smells were not removed in successive commits. All these findings suggest that refactorings tend to more frequently introduce long-living smells instead of fully eliminating existing ones. Also, we analyzed which types of refactoring affects specific types of code smells more often. Through this analysis, we were able to identify refactoring-smell pairs which occurred more frequently.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Misc; D.2.8 [**Software Engineering**]: Metrics

Keywords

Refactoring, Code Smells, Structural Quality

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICSE 2017 Buenos Aires, Argentina
Copyright 2017 ACM X-XXXXX-XX-X/XX/XX ...$15.00.

1. INTRODUCTION

Code smells in a program represent indications of structural quality problems [5]. Examples of code smells include *God Class*, *Feature Envy* and *Long Method*. Refactoring is a common practice employed by practitioners along software maintenance in order to improve the structural quality [14, 19]. Refactoring is defined as a program transformation intended at preserving the observable behavior and improving its internal structure by reducing the density of code smells [5]. This definition entails a positive expectation about the effects of each refactoring. In other words, it is often assumed there is a strong relationship between refactoring and improving internal software quality, i.e., each refactoring is likely to reduce the density of code smells in the program elements affected by the program transformation.

In practice, programmers apply two tactics when refactoring their code structure [14]. First, the so-called *root-canal refactoring* is used for reversing the quality deterioration of smelly source code and involves a protracted process consisting of exclusive (i.e., pure) refactoring. Second, the programmer employs *floss refactoring* for improving its structural quality as a means to reach a specific end, such as facilitating the addition of a feature or enhancing program testability. Thus, independently of its tactic of use, there is a common expectation that refactorings often improve aspects of the program structure, ideally reducing the density of code smells. However, mainly in the latter case, programmers might be inattentive or misinformed and end up increasing rather than reducing code smells. Thus, even though refactoring is intended to improve code structure, this expectation might not be met in real settings as either root-canal or floss refactorings are often performed manually [7, 14].

Little has been reported about whether and to what extent developers successfully improve structural quality through code refactoring. To the best of our knowledge, there is no study that thoroughly investigates and characterizes both positive and harmful structural effects of software refactoring. Existing studies tend to focus only on analyzing the relationship of code smells and quality attributes [6, 15, 22]. Only recently, Bavota *et al.* [2] performed a study aiming to investigate if refactoring tends to remove code smells in the context of some major versions of only three software projects. However, they could not reveal to what extent and

which types of refactorings often decrease, rather than increase, the structural quality in software projects. Moreover, given the nature and size of their sample, they could not characterize recurring desirable or undesirable relationships between code smells and specific refactoring types.

Different from existing research [14, 2], we conduct a longitudinal study that analyzes not only the beneficial but also the negative impact of refactoring changes on the structural quality of 23 open source projects. Instead of being limited to the analysis of a few major versions in a few projects, we consider 113,306 versions in our analysis. We analyze not only if refactoring reduces code smells, but also if and to what extent specific types of refactoring are often related to the introduction of new code smells. We classify each refactoring instance according to its interference on the existing and new smells located in the refactored elements. In our study, the classification of a given refactoring instance lies in one of the three cases: (i) *positive* if the absolute number of code smells decreases after the program transformation; (ii) *negative* if it increases; or (iii) *neutral* if it remains the same. This classification is used to understand whether certain refactoring types tend to improve or decrease the internal structure of a program.

We identified and analyzed 29,318 refactorings classified in 10 well-known refactoring types. According to Murphy-Hill *et al.* [14], these refactoring types are amongst the commonly used ones. Thirteen code smell types are used to classify the collected refactorings. We revealed recurring beneficial or harmful effects of software refactoring, such as (but not limited to):

- As expected, most refactoring instances touched smelly elements. However, 10,475 refactorings (63.2%) are neutral ones. Either root-canal or floss refactorings did not reduce the density of code smells in the refactored elements. Moreover, we found negative refactorings occur more frequently than positive ones: 24.5% against 12.3%. Similarly, we also concluded that most refactorings do not affect any structural quality measure. Even worse, more than 95% of refactoring-induced smells were not removed afterwards in successive commits. These findings shed light on how refactorings may (in)directly or indirectly degrade the structural quality. They also suggest developers need more guidance to fully remove a code smell once they start restructuring a smelly element.

- Thus, we characterized and quantified recurring patterns governing beneficial and harmful effects of specific refactoring types. Again, harmful patterns were more frequent than beneficial ones. For instance, refactorings intended at moving methods – such as *Move Method* and *Pull Up Method* – to other classes tended to induce occurrences of *God Class* in the target classes in addition to the prevalence of smells in the source classes. The *Move Method* refactoring induced the emergence of *God Classes* in 35% of the cases, while the *Pull Up Method* tended to be related to this smell type in 28% of the cases. Moreover, 95% of such smell instances, induced by such refactorings, remained as *God Classes* in successive commits.

- Several other types of code refactoring were surprisingly often related to smells emerging after the program transformation. For instance, the *Extract Superclass* refactoring creates the code smell *Speculative Generality* in 68% of the cases.

This work is organized as follows: Section 2 provides basic concepts. Section 3 presents the study planning. Section 4 provides answer to our first research question, whereas Section 5 presents the answer to the second one. Section 6 describes our effort on mitigating threats to validity. Section 7 relates our study with previous work. Section 8 concludes the study.

2. CONCEPTS AND MOTIVATION

This section introduces concepts and motivating examples.

2.1 Motivating Examples

A smell is a surface indication that usually corresponds to a deeper structural problem in the system [5]. For instance, a class with several responsibilities is known as *God Class*. This smell makes the class hard to read, modify and evolve [5]. The structural quality of a program can be quantified by the occurrences of code smells found on it. For instance, let us suppose a *Person* class that has, amongst many other members, at least three attributes representing two loosely-coupled concepts: person and telephone number. This version of *Person* can be considered a *God Class*. In order to remove this smell, the developer can extract part of the class structure into another class: *TelephoneNumber*. After this transformation, called *Extract Class* refactoring, the program no longer has the *God Class* and still realizes the same functionality. After this refactoring, the number of code smells would be reduced and the software structural quality would be improved. However, it might be not always the case that refactorings are successful in removing smells and improving structural quality. Even worse, a new code smell can be introduced by refactoring edits.

For instance, let us suppose that the same developer who applied the *Extract Class* refactoring tried to generalize the *Person* class. To accomplish this task, he then applied an *Extract Superclass* refactoring in the *Person* class, creating its superclass called *LivingBeing*. However, this generalization was never explored in the system, so the developer created a smell called *Speculative Generality*. Thus, the developer created a new code smell via refactoring. [redacted]

2.2 Refactoring Classification

Refactorings may interfere positively or negatively in the existence of smells. This section presents a scheme to classify each refactoring instance by computing the number of smells introduced or removed along the refactoring changes. Let $S = \{s_1, \cdots, s_n\}$ be the set of software projects to be analyzed in our study. Each software s has a set of versions $V(s) = \{v_1, \cdots, v_m\}$. Each version v_i has a set of elements $E(v_i) = \{e_1, \cdots\}$ representing all methods, classes and fields belonging to it. In the previous section, the set $S = \{PhoneBook\}$ represents a software system, called $PhoneBook$. This software S has two versions $V(PhoneBook) = \{v_1, v_2\}$, where v_1 is the version before the refactoring and v_2, after. Finally, each version v_i has a set of elements $E(v_i)$ composed by *Person* and *TelephoneNumber* classes, methods and fields.

In order to be able to detect refactorings, we must analyze transformations between each subsequent pair of versions. In this way, we assume R is a refactoring detection function where $R(v_i, v_{i+1}) = \{r_1(rt_1; e_1), \cdots, r_k(rt_k; e_k)\}$ gives us a set of tuples composed by two elements: the refactoring type (rt) and the set of refactored elements represented by e. So, the function R returns the set of all refactoring activities detected in a pair of software versions. If we apply the R function in the *PhoneBook* software, the result would be: $R(v_1, v_2) = \{r_1(\text{Extract Class}, e')\}$ where $e' = \{Person, TelephoneNumber\}$.

Refactored Elements. In this work, it is considered as refactored elements all those directly affected by the refactoring. If a refactoring is applied only in a method body, only this method is considered as refactored element. For instance, lets consider the *Move Method* refactoring. In this refactoring type, a method m is moved from class A to B. Hence, the considered refactored elements in this case would be $\{m, A, B\}$. All m method callers are affected by this refactoring, but we do not consider them as refactored elements. As another example, let us consider the *Rename Method* refactoring. In this scenario, a new name is given to the method m and the refactored element set would be just $\{m\}$. For each refactoring type a different refactored element set is used. The complete list of what is considered refactored elements for all refactoring types is available at [3].

Let CS be a code smell detection function where $CS(e) = \{cs_1, \cdots\}$ returns a set of code smells present in a software element set e. As the objective is to classify refactorings, the classifying procedure only analyzes code smells related to elements affected by a refactoring. In this way, we can say that $CS_{b[r]}(e)$ is the set of code smells of e before the application of the refactoring r. On the other hand, $CS_{a[r]}(e)$ is the set of code smells found after the application of r. Considering r_1 refactoring applied on classes of *PhoneBook* system, CS function would present the following results: $CS_{b[r_1]}(\{Person\}) = \{\text{God Class}\}$, $CS_{a[r_1]}(\{Person, TelephoneNumber\}) = \emptyset$.

Positive, Neutral and Negative Refactorings. Using data collected by the functions defined before, it is possible to classify a refactoring by looking how it interferes in existing code smells. Suppose e is a software element set, r is a refactoring edit and $|CS_{b[r]}(e)| = x$. After r refactoring, $|CS_{a[r]}(e)| = y$. Depending on x and y, it is possible to classify r. If $x > y$, r reduced the number of code smells on e and, because of that, r is considered a *positive refactoring*. Otherwise, if $x < y$, r increased the number of code smells on e and, because of that, r is a *negative refactoring*. When $x = y$, r is a *neutral refactoring*. For instance, $|CS_{b[r_1]}(e)| = 1$ and $|CS_{a[r1]}(e)| = 0$. Thus, the r_1 refactoring performed in the *PhoneBook* system is a positive one.

Independent of its type, a code smell should be addressed by refactorings along software maintenance. In other words, any instance of code smell demands at least a single refactoring to be removed. The classification is based on this principle and, because of that, the number of code smells is used to classify a refactoring.

2.3 Refactoring-Smell Patterns

In this paper, structural quality is defined as the density of code smells. Thus, the aforementioned classification helps us to understand how developers' refactorings affect structural quality. This process can reveal to what extent and which types of refactorings often decrease, rather than increase, the structural quality in software projects. Also, we determine the relationship between each refactoring instance on the removal or addition of a code smell. By considering the refactoring types involved with code smells creation, we can advance our research towards the characterization of recurring desirable or undesirable relationships between code smells and specific refactoring changes. For instance, if the *Move Method* refactoring introduces *God Class* frequently, is possible to infer that there is a pattern governing these two types. We use a threshold-based rule to state a relation between types of refactoring and code smell as *patterns*. It is necessary to define some notations before introducing the rule.

Let $K = \{r_1, r_2, \cdots, r_n\}$ be the set of all detected refactorings after analyzing a software set S. Thus, K_{rt} is the K subset of refactorings of the type rt. The set $K^+_{rt,cs}$ is the K subset composed by refactorings of the type rt that added code smells of type cs in any element involved in the refactoring, while $K^-_{rt,cs}$ is the K subset that removed code smells of type cs. Finally, $K^*_{rt,cs}$ is the K subset composed by refactorings of the type rt that satisfy the following conditions: (i) the refactoring was applied in classes or methods containing at least one code smell instance of the type cs; and (ii) the refactoring did not remove the instance of the code smell of cs type.

Creational Patterns. The definition of creational pattern between types of code smell and refactoring can be established using the above notation. A creational pattern occurs when a specific refactoring type involves code transformations that often introduces a specific code smell. We define this concept as a threshold-based rule. If $|K^+_{rt,cs}|/|K_{rt}| \geq \gamma$, is possible to affirm that there is a creational pattern between rt and cs. This kind of pattern captures scenarios where developers apply a refactoring and, somehow, end up creating at least one new code smell.

Removal and Non-Removal Patterns. The definition of *removal pattern* also lies in a threshold-based rule. If $|K^-_{rt,cs}|/|K_{rt}| \geq \gamma$, we can affirm that there is a removal pattern between rt and cs. It means that developers consistently removes instances of cs when performing rt refactorings.

Creational patterns help us to understand cases of reckless refactorings but do not capture the underlying characteristics of the refactored classes or methods. In this way, we are also interested in studying what code smells instances are commonly present in classes and methods and, somehow, end up remaining in the source code after refactoring. This third type of pattern is called *non-removal pattern* and it is defined by another threshold-based rule: $|K^*_{rt,cs}|/|K_{rt}| \geq \gamma$. The non-removal patterns help us to understand which smells are touched by a refactoring type but ended up not being removed.

3. STUDY PLANNING

This section presents the study planning.

3.1 Research Questions

Software refactoring is likely to interfere in the presence of code smells (Section 2.1). As code smells provide the motivation to perform refactoring changes [5], someone may expect that the latter contribute to the reduction of the former. As illustrated in Section 2.1, the number of code smells located in refactored elements is expected to be reduced, thereby

improving software structural quality. Therefore, our study aims at addressing the following research question:

> **RQ1.** Does refactoring improve the software structural quality?

We address this question by relying on the classification of each refactoring instance (Section 2.2). Our assumption is that a positive refactoring contributes to improving the structural quality. On the other hand, a negative refactoring may indicate a decrease on the structural quality of a program. Similarly, a neutral refactoring does not affect the structural quality and, therefore, the refactoring change is not achieving the purpose of improving structural quality.

Our study answers the aforementioned question by analyzing refactorings performed in real projects. This procedure enables us to compute how frequent each refactoring classification occurs across the analyzed projects. Let S to be a set of software projects. First, all instances of refactorings and code smells present in this set were detected and analyzed in our study. Then, all refactoring instances were classified according to Section 2.2. Let p the number of refactorings classified as positive; n the number of negative refactorings; and k representing the number of neutral refactorings. If n is greater than p and k, we can state that the application of refactorings are likely downgrading the structural quality of projects. Otherwise, if p is greater than n and k, the answer to our research question is **yes**, refactorings tend to improve the structural quality of programs. Another possible case is when k is greater than p and n. In this scenario, refactorings would tend to not improving the structural quality. However, developers can improve a relevant aspect of the program structure without fully removing code smells. Thus, we also investigate the likelihood of refactorings improving or degrading a single structural quality measure.

Some types of refactoring might consistently remove (or fail to do so) or even frequently introduce specific smell types across software projects. Section 2.3 defined three categories of possible patterns between types of refactoring and smells. Discovering these patterns is the focus of our second research question.

> **RQ2:** What are the patterns governing types of refactoring and code smells?

By answering RQ2, we are able to reveal harmful actions made by developers on refactored elements. We detect removal, non-removal and creational patterns (Section 2.3) by analyzing the impact of refactoring types on code smells located in the refatored elements. The knowledge about non-removal and creational patterns make developers more informed about the possibilities and risks of missing and introducing certain smells along either root-canal or floss refactorings. Tool developers can also understand how to design better recommender systems to guide programmers on fully removing complex smells, such as *God Classes* and *Feature Envies*.

The second research question aims to understand how each refactoring, in fact, introduces or removes types of code smell. We also consider refactorings that do not change the existing code smells, i.e., non-removal patterns. Our study tries to reveal non-removal patterns by analyzing the neutral refactorings collected. For each one, we observe the code smell instances existing in the refactored software elements. In this way, we can observe how often each refactoring targets elements affected by each type of code smell. The non-removal patterns might shed light on the objective of each studied refactoring. We can observe how consistently developers apply certain kinds of refactorings in each type of code smells and, yet, did not remove them.

3.2 Study Phases

This subsection presents all phases of the study design.

3.2.1 Phase 1: Selection of Software Projects

The first step of this study is to choose a set S of software projects to compose the study sample. First, we established GitHub, the world's largest open source community, as the source of software projects. We focused our analysis on open source projects so that our study could be easily replicated and extended. This study uses 23 GitHub projects that met the following quality criteria: (i) high popularity, i.e., among the projects with most starts; (ii) active issue tracking system, i.e., users actively use the GitHub issue management system for bug reporting and improvement suggestions; (iii) has at least 90% of the code repository effectively written in Java. The full list of selected projects is in Table 1. This list is resulting from the execution of the presented criteria in GitHub in May 2016. This table presents the following data about the projects: (i) name; (ii) lines of code and (iii) number of commits. More details about the selected projects are available in our online supplementary material at [3].

Table 1: Projects used

Name	LOC	Commits
alibaba/dubbo	104,267	1,836
AndroidBootstrap/android-bootstrap	4,180	230
apache/ant	137,314	13,331
argouml	177,467	17,654
elastic/elasticsearch	578,561	23597
facebook/facebook-android-sdk	42,801	601
facebook/fresco	50,779	744
google/iosched	40,015	129
google/j2objc	385,012	2,823
junit-team/junit4	26,898	2,113
Netflix/Hystrix	42,399	1,847
Netflix/SimianArmy	16,577	710
orhanobut/logger	887	68
PhilJay/MPAndroidChart	23,060	1,737
prestodb/presto	350,976	8,056
realm/realm-java	50,521	5,916
spring-projects/spring-boot	178,752	8,529
spring-projects/spring-framework	555,727	12,974
square/dagger	8,889	696
square/leakcanary	3,738	265
square/okhttp	49,739	2,645
square/retrofit	12,723	1,349
xerces	140,908	5,456

3.2.2 Phase 2: Smell and Refactoring Detection

The second phase is in charge of detecting refactorings in all subsequent pairs of versions v_i and v_{i+1}. This phase also encompasses the detection of all smells in each version $v_i \in V(s)$. These activities are described in the following.

Refactoring Detection. We choose Refactoring Miner

[20] (version 0.2.0) [1] to support the detection of refactoring instances. This tool implements a lightweight version of UMLDiff [21] algorithm for differencing object-oriented models. The precision of 96.4% reported by the authors [20] led to a very low rate of false positives, as confirmed in our validation phase. The only drawback of this tool is the number of refactoring types detected (11 types). Fortunately, these 11 types were amongst the ones reported by Murphy-Hill as the most common refactoring types [14]. Refactoring Miner gives us as output a list of refactorings $R(v_i, v_{i+1}) = \{r_1, \cdots, r_k\}$ as defined before, where k is the total number of refactorings identified. As mentioned in Section 2.2, each r_i is a tuple containing the refactoring type rt_i and the element refactored e_i. The refactoring detection primarily relied on the use of o tools to analyze pairs of software versions v_i and v_{i+1}, generating all refactorings $R(v_i, v_{i+1})$ as output.

Code Smell Detection. In order to complete the data collection procedure, we need to identify smells present in each e_i returned in R tuples. Code smells are often detected with rule-based strategies [1]. Each strategy is defined based on a set of metrics and thresholds. Therefore, the application of rule-based strategies requires the collection of metrics for all source files in a project. After the collection of metrics, we apply a set of previously defined rules [9, 11] to detect code smell occurrences. This procedure is the implementation of CS function defined in Section 2.2. The specific metrics and thresholds for code smell detection were defined in [11, 12]. These rules were used due to two main reasons: (i) they represent refinements of well-known rules proposed by Lanza *et al.* [9], which are well documented and used in previous studies (e.g., [13, 23]); and (ii) they have, on average, precision of 0.72 and recall of 0.81 [10].

These rules detect five code smells: *God Class*, *Long Method*, *Feature Envy*, *Shotgun Surgery* and *Divergent Change*. The selection of these smell types was due to two main reasons: (i) these code smells are very common and tend to be related to design degradation symptoms [11]; and (ii) they subsume the occurrence of other code smells. Another reason to select these smell types was their direct relation with the most frequent refactoring types. Murphy-Hill *et al.* [14] reported the refactorings often performed by developers and we analyzed which code smell types these refactorings are intended to remove [5]. Refactoring Miner is capable of detecting the refactoring types that remove these smells.

Table 2 shows examples of rules used to identify code smells; the full set of rules is presented in our online suplementary material. These rules use the following metrics: (i) Lines of Code — LOC; (ii) Coupling Between Objects — CBO; (iii) Number of Methods — NOM; (iv) Ciclomatic Complexity — CC; (v) Lack of Cohesion of Methods — LCOM; (vi) Fan-out — FO; and (vii) Fan-in — FI. All metrics values were collected using the Understand software [16] with non-commercial license. [redacted] We focused on detecting 8 additional types of smells (in addition to the five ones in Table 2): *Complex class*, *Lazy class*, *Long parameter list*, *Message chain*, *Refused bequest*, *Spaghetti code*, *Speculative generality*, and *Class data should be private*.

Rules for detecting code smells play a central role in our study. In this way, we must guarantee that our results are not biased by a single set of detection rules. Different thresholds can lead to different results. Therefore, choices of thresholds can pose a threat to this study. Thus, two sets of thresholds were used to mitigate this menace. The first set, as known as *tight* set, represents the thresholds previously validated in the study by Macia *et al.* [11]. [redacted] The second strategy, named as *relaxed*, uses relaxed thresholds designed to detect as many smells as possible. In addition to the two previously mentioned (tight and relaxed) thresholds set, we also used the detection rules proposed by Bavota *et al.* [2]. The Bavota's rules are presented in [2] and [3]. These three smells detection strategies allowed us to derive refactoring classifications based on three different sets of code smells, mitigating the threat of the thresholds choice.

Table 2: Rules for Code Smell Detection

Code Smell	Detection Rule
God Class	[(LOC $>\alpha$) and (CBO $>\beta$)] or [(NOM $>\delta$) and (CBO $>\beta$)]
Long Method	(LOC $>\epsilon$) and (CC $>\eta$)
Shotgun Surgery	(CC $>\theta$) and (FO $>\omega$)
Divergent Change	(FI $>\iota$) and (LCOM $<\upsilon$) and (CC $>\varsigma$)
Feature Envy	(FO $>\zeta$) and (LCOM $<\varrho$) and (CC $>\varpi$)

3.2.3 Phase 3: Refactoring Classification

The objective of the third phase is to classify all refactorings detected in the prior phase. We classified each detected refactoring by observing its interference in the number of code smells. After this classification, it is possible to quantify how frequent refactorings are labeled according each possible category in our software set S. As mentioned in Section 3.2.1, all projects are Git repositories stored on GitHub servers. The data collection process starts by cloning a Git repository. This study considers as a version every commit in the repository. We skipped merge commits during the analysis since this kind of commit could lead us to compute twice the same refactoring [20]. The algorithm always compares subsequent versions of the projects. [redacted] The set $V(s)$ of a Git repository s is the list of all non-merge commits in the master branch ordered chronologically.

Quality Metrics Analysis. As discussed in Section 3.1, in addition to the impact of refactorings on code smells, we are also interested in measuring their impact on quality metrics. We collected and analyzed a set of 16 quality metrics for each version of all projects. We compute each metric value before and after every single refactoring detected. The value of each metric can increase, decrease or remain the same. This variation study is used to classify each metric in three cases: (i) improved; (ii) worsened; or (iii) no effect. The complete table of all metrics and criteria is available at [3].

Root Canal vs. Floss Refactoring. According to Murphy-Hill *et al.* [14], there are two ways of performing refactoring: (i) root canal refactoring; and (ii) floss refac-

[1] Available at https://github.com/tsantalis/RefactoringMiner

toring. During floss refactoring, the programmer uses refactoring as a means to reach a specific end, such as adding a feature or fixing a bug. Root-canal refactoring, in contrast, is used for correcting deteriorated code and involves a process consisting of exclusive refactoring. We are also interested in knowing the percentage of root canal and floss refactoring in our dataset. In this way, this study comprises a manual inspection of a sample of refactorings. In this manual inspection, we will evaluate if a refactoring is root canal or floss refactoring. This will help us to identify when a refactoring introduced, in fact, a smell or if the smell is a product of another code changes. Our online supplementary material provides additional details on how we computed root canal and floss refactorings.

3.2.4 *Phase 4: Manual Validation*

The last phase is responsible for all data validation. As the first three phases use tools to detect refactorings and code smells, there is a threat to validity related to false positives and negatives yielded by these tools. To mitigate this threat, the fourth phase is required. In this phase, a manual procedure was executed in a smaller dataset. A manual validation of each output was made in this phase to ensure the reliability of or data. In this vein, we conduct different data validation activities.

We randomly sampled refactorings from each type to support the analysis. We decided to sample by the 11 refactoring types since the precision of the Refactoring Miner could vary due to the rules implemented in the tool to detect each refactoring type. To deliver an acceptable confidence level to the results, we calculated the sample size of each refactoring type based on a confidence level of 95% and a confidence interval of 5 points. We used such confidence to all sampling activities performed in this study. Details can be found in [3]. We recruited ten undergraduate students from another research group to analyze the samples.

4. REFACTORING AND QUALITY

This section presents and discusses the data used to answer RQ1.

4.1 Classification Results

The set of refactorings and code smells in our dataset covered a wide range of refactoring types and smell types. All refactoring types detected by Refactoring Miner were considered in this study. The refactoring detection procedure identified 29,318 refactorings distributed in 11 different types. Table 3 presents the refactoring types ordered by the number of occurrences across the projects analyzed. The first column shows each refactoring type followed by the corresponding number of its occurrences (second column) in all projects analyzed. The most common refactoring type is *Rename Method*, similarly to a previous study that analyzed refactoring frequencies in other systems [14]. The second most common refactoring type is *Extract Method*, and we observe several occurrences frequency of *Move Field*, *Move Method* and *Pull up field* refactorings.

We also computed how many times each type of refactoring was applied in program elements hosting at least one code smell. In other words, we count how many refactorings are applied in smelly code elements (see column *Applied in Smelly Code*). The remaining columns present the occurrences of positive, negative and neutral refactorings for each type of

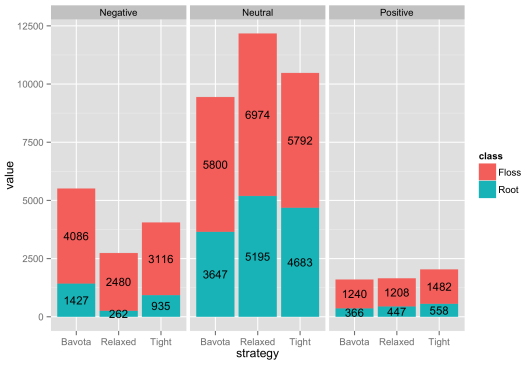

Figure 1: Results of the Data Collection Phase

refactoring. Since the definition of *Rename Method* is not directly related to one of the code smells addressed in our study, we did not use its instances during the refactoring classification phase.

If we compare the data of the second and third columns, we can observe that developers tend to often apply refactorings in smelly elements of a program. Notice that 8 refactoring types have been applied in smelly code elements in more than 70% of the occurrences. The exceptions are *Extract superclass* (38.30%) and *Extract interface* (48.87%). These frequencies were very similar in the data gathered with *relaxed* thresholds [3]. Surprisingly, the neutral classification was by far the most frequent one for 7 refactoring types. The negative classification was more frequent for the 3 remaining ones: *Extract method*, *Extract superclass* and *Extract interface*. In other words, even though refactorings often target smelly elements, they often do not reduce the smells present in those elements after refactoring edits.

We focus now on the data concerning code smells, refactorings and three different code smells detection strategies (*Bavota*, *Relaxed* and *Tight*), as shown in Figure 1. All refactorings were labeled according to the classification presented in Section 2. We observe that even with different strategies for detecting code smells, the rates of neutral refactorings are the most common classification detected in our data set.

4.2 Most of Refactorings are Neutral

This section explicitly focuses on addressing the RQ1: Does refactoring improve software structural quality? In order to answer this question, all detected refactorings were classified thrice as presented Section 4.1. The main classification concerns the code smells detected using the *tight* threshold set, while the secondary classification relies on the *relaxed* one. Bavota rules composes the third code smells set. Figure 1 shows the general proportion of neutral, positive and negative refactorings using the three classification procedures. When we analyze each individual project, the same classification distribution is observed, i.e., neutral refactorings represent the vast majority in all the projects.

This recurring refactoring behavior across the projects, using either *tight*, *relaxed* or *Bavota* rules, supports our answer to RQ1: refactoring edits made by developers in real projects often do not improve program structural quality. Even if we consider each refactoring type, neutral refactorings tended to be the most common ones (Table 3). Surprisingly, these results also prevail if we only consider refactoring types

Table 3: Most common refactorings types collected

Refactoring Type	Occurences	Applied in Smelly Code	Neutral	Negative	Positive
Rename method	12,752	6,031 (47.29%)	-	-	-
Extract method	7,517	6,411 (85.28%)	2,917	3,914	686
Move field	4,356	3,362 (77.18%)	3,784	438	134
Inline method	1,528	1,134 (74.21%)	732	214	582
Move method	1,404	1,049 (74.71%)	1,008	297	99
Pull up method	629	511 (81.24%)	430	155	44
Pull up field	465	333 (71.61%)	338	103	24
Extract superclass	342	131 (38.30%)	89	246	7
Extract interface	133	65 (48.87%)	11	122	0
Push down method	114	98 (85.96%)	76	11	27
Push down field	78	58 (74.35%)	62	13	3

that, according to their description in Fowler's catalog [5], are explicitly associated with specific code smell types addressed in our study. For instance, the motivation for applying *Move Method*, *Pull up Method* and *Move Field* refactorings is associated with smells that represented methods or fields that are misplaced. The misplacement of these members are captured by occurrences of either *Feature Envy*, *Divergent Change*, *Shotgun Surgery* or *God Class*. Someone could also hypothesize that refactorings performed in real project settings are not primarily aimed at removing code smells. However, Table 3 shows that refactoring edits are often targeted at smelly elements of a program. The problem is that those edits often did not suffice to remove code smells.

Root Canal vs. Floss Refactoring. This study also aims to observe the impact of the two ways that developers can refactor. Developers can perform root canal or floss refactoring [14]. As presented in Section 2.2, we analyzed manually the frequency of each way of refactoring in a randomly selected sample. We found that developers apply root canal refactoring in 31.5% of the cases. The confidence level for this number is 95% with a confidence interval of 5%. The root canal percentage found is a significant number. Even when developers are trying explicitly to perform only a refactoring (root canal), they are not able to remove code smells in a consistent way.

Refactoring Impact on Individual Measures. We also analyzed the impact of each refactoring on individual quality metrics values (Section 3.2.3). This analysis followed similar trends: in almost every case, most of the quality metric values do not change after refactoring. It means that even at the metric level, the refactorings are neutral in the most of the cases. Similarly, when the metrics values are disturbed by refactorings, the values tend to get worse rather than getting improved. Therefore, we observed similar behaviors between the analysis in the code smell level and metrics level. The complete analysis of the metrics impact study is available at [3].

Refactoring Miner Estimated Precision. The Refactoring Miner precision can pose a threat to our results. Since we rely on the refactorings collected by this tool, the results depend on its precision. Section 3.2.4 presents the procedure executed to estimate the precision of Refactoring Miner in our dataset. In general, it was observed a high precision to each refactoring type, with a median of 88.36% (excluding rename method). The precision found in all refactoring types are close to the standard deviation (7.73). By applying the Grubb outlier test (alpha=0.05) we could not find any outlier, indicating that no refactoring type is strongly influencing the median precision found. Thus, the results found in the representative sample analyzed represent a key factor to provide reliability to the other results reported in this work. The complete results of this sample study is available at [3].

5. REFACTORING-SMELL PATTERNS

In order to address our second research question, we analyzed what are the patterns emerging from the relationship between refactorings and smells touched by them. Section 2.3 defined three categories of such patterns, i.e., removal, non-removal and creational patterns. This section presents and discusses the patterns observed in our dataset. Section 5.1 focuses on discussing the removal and non-removal patterns, while Section 5.2 discusses the creational patterns.

We tended to focus on discussing patterns in which more than 20% of the instances of a refactoring type was related (removed, touched but not removed, introduced) to instances of a specific smell type. For these patterns, we inspected all the pattern instances in order to understand what happened in each case. In particular, we also confirmed whether the refactoring was directly related to the removal or introduction of the smell. This was an important step as we had pattern instances occurring in the context of either root canal refactoring or floss refactoring.

For the non-removal and creational patterns, we also analyzed the life time of the prevailing and introduced smells related to the non-removal and creational patterns. We checked if such refactoring-related smells – prevailing or emerging in commits involving one or more refactorings – were either removed or not in subsequent commits. We considered *subsequent commits* those ones performed within one-month period after the occurrence of a pattern instance. Our goal was to understand whether the refactoring-related smell was (or not) temporarily prevailing in the code because the developer was planning to remove the smell shortly, i.e. in the next commits.

5.1 Removal vs. Non-Removal Patterns

Table 4 presents the cases of removal and non-removal patterns observed in our dataset. They are alphabetically ordered by the refactoring type. Each row represents a removal and/or non-removal pattern involving a pair of refactoring type and smell type. The first column shows the refactoring type, followed by the smell type in the second column.

The next two columns present for each refactoring type the percentage of its instances related to the removal (fourth column) or prevalence (third column) of the corresponding code smell. Patterns with an incidence strength higher than 20% are shown in bold. The third column also shows the absolute number of non-removal pattern instances in brackets. Additional data is presented in our online material [3].

At a first glance, it is already possible to observe there was a much higher incidence of non-removal patterns than removal ones. The percentages of non-removal patterns (third column) are often higher than their removal counterparts. Thus, we can also conclude there are specific types of refactorings tending to consistently affect a particular type of smell. However, those refactorings more frequently are unsuccessful (non-removal) rather than successful (removal) with respect to that particular smell type. For instance, *Extract Method* refactoring was often targeted at methods hosting a *Feature Envy* smell. However, as expected, most of those *Extract Method* refactorings could not remove this smell. Table 4 shows 42.61% (against 11%) of such refactorings touched this smell, but they were not able to eliminate it.

After analyzing these pattern instances, we confirmed that proper action of developers should also include moving (not only extracting) those *Feature Envie* smells as methods to other classes. However, the vast majority of those extracted method (higher than 95% of its instances) were neither moved to neighbor classes in subsequent commits. In fact, those (11%) of successful *Extract Method* refactorings were performed in conjunction with other method-moving refactorings in the same commit, such as *Move Method*, *Pull up Method* or *Push down Method* refactorings.

Another interesting observation is that *God Class* smell was the most frequent target of removal or non-removal refactorings. In fact, this smell dominates the rows of Table 4. Several refactoring types were often related to changes moving out members from *God Classe* smells. Two of the refactoring types – namely, *Move Method* (23%) and *Move Field* (27%) refactorings – were significantly successful in contributing to the removal of a *God Class* smell within a commit. However, even for these refactoring types, there was higher incidence of non-removal patterns. Table 4 shows 51.37% and 29.44% of *Move Method* and *Move Field* refactorings touched *God Class* smell but were not sufficient to eliminate it, independently if they were part of root-canal or floss refactorings. Those refactorings were often performed in conjunction with other member-moving refactorings in the same commit, but were not sufficient to remove *God Classes*. In 99% of the cases, the prevailing *God Classes* smell were not removed in the successive commits either. There were only two refactoring-smell patterns that more predominantly removed (rather than not) the code smell. They were patterns involving the *Push down Method* refactoring and *Lazy Class* smell (52%) and *Refused Bequest* smell (23%).

5.2 Creational Patterns

Interesting data also emerged from creational detected in our dataset. We divided these patterns into three groups [5] considering the purpose of the refactoring type: (i) refactorings targeted at improving generalization; (ii) refactorings responsible for moving features between objects; and (iii) refactorings targeted at restructuring members of a class. The following subsections respectively present and discuss creational patterns involving refactorings in these groups.

Table 4: Removal and Non-Removal Patterns

Refactoring	Code Smell	Non-Removal	Removal
Extract Interface	God Class	**20.33%** (25)	2.00%
Extract Method	Divergent Change	**34.68%** (1,115)	7.00%
Extract Method	Feature Envy	**42.61%** (1,419)	11.00%
Extract Method	God Class	**48.65%** (1,620)	0.00%
Extract Method	Long Method	11.92% (397)	8.00%
Move Field	God Class	**29.44%** (1,139)	**27.00%**
Move Method	God Class	**51.37%** (544)	**23.00%**
Pull up Field	God Class	**44.79%** (142)	8.00%
Pull up Method	God Class	**61.32%** (317)	10.00%
Push down Field	God Class	**55.71%** (39)	12.00%
Push down Method	God Class	**54.35%** (50)	15.00%
Push down Method	Lazy Class	2.63% (2)	**52.00%**
Push down Method	Refused Bequest	9.21% (7)	**23.00%**

Figure 2 presents all creational patterns found based on a heat map. Each cell represents the percentage of the times that the refactoring in the Y-axis is related to the introduction of the smell (i.e., in the same commit) in the X-axis.

5.2.1 Generalization Patterns

Refactorings dealing with generalization were often related to the creation of *God Class* and *Speculative Generality* smells. We can observe in Figure 2 that *Pull Up Method*, *Pull Up Field* and *Extract Superclass* refactorings are related to the creation of *God Classe* smells in 28%, 61% and 14% of the cases, respectively. *Extract Superclass* refactoring creates the *Speculative Generality* smell in 68% of the cases, while 34% of the *Pull Up Field* refactoring instances introduce this same smell in the target superclass. What is more troublesome was the fact that more than 98% of such introduced smells were not removed in successive refactorings.

A typical example of generalization-related creational pattern can be illustrated by the case involving the *DefaultProjectListener* class from the Xerces project. The *DefaultProjectListener* class is the default implementation of a listener that emulates the old ant listener notifications. *Extract Superclass* refactoring was applied on *DefaultProjectListener* class, thereby creating the *AbstractProjectListener* class from it. However, the new abstract class did not seem to justify the refactoring. There was only one class that extended *AbstractProjectListener* class, i.e. the *DefaultProjectListener* class itself. Thus, the refactoring created the *AbstractProjectListener* class with a *Speculative Generality* smell. Moreover, this refactoring had another negative consequence on the affected classes as it introduced another code smell. The *DefaultProjectListener* class overrides all the methods defined on the *AbstractProjectListener* class. Consequently, the *DefaultProjectListener* class became affected by a *Refused Bequest* smell. One of the reasons for this problem is that all the bodies of the methods defined on the *AbstractProjectListener* class are empty; they do not have any implementation. Ideally, the *AbstractProjectListener* abstract class should have been instead defined as an interface. Moreover, all these smells were not removed in successive commits, thereby affecting other listener subclasses created latter. Therefore, in this example, the *Extract Superclass* refactoring is responsible for creating an instance of a generalization-related creational pattern and propagating a smelly structure to other classes.

5.2.2 Feature-Moving Patterns

Refactorings responsible for moving features between objects were also part of our catalog of detected creational patterns. *Move Method* refactorings were related to the cre-

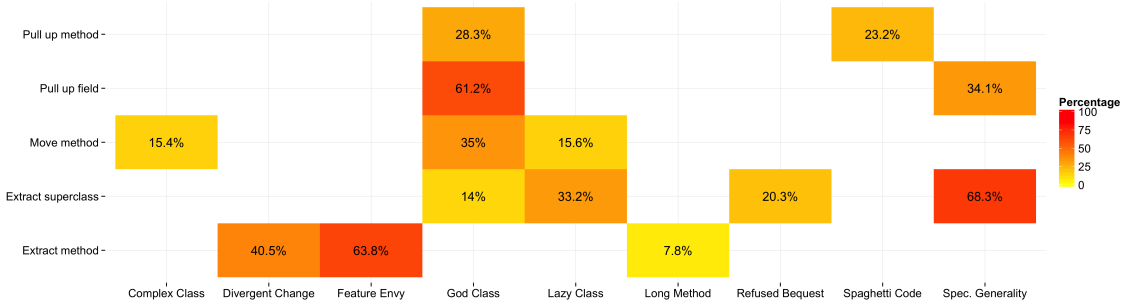

Figure 2: Creational patterns

ation of three types of smells. This refactoring created *God Class*, *Complex Class*, and *Lazy Class* smells in 35%, 15%, and 16% of the cases, respectively. Interestingly, this type of refactoring was amongst the most common ones in a previous study [14]. When analyzing all these pattern instances, we confirmed that developers were consistently creating smells through *Move Method* refactorings in the target classes (i.e. those receiving the moved methods) without removing those smells in the source classes. Again, the vast majority of these introduced smells (more than 98%) prevailed in the successive commits. This observation shows that tooling support should warn developers about the risks related to such recurring creational patterns.

A typical case of creational feature-moving pattern can be illustrated by refactoring changes affecting two classes from the ArgoUML project. The *generateMessageNumber* method was moved from *GeneratorDisplay* class to the *MessageNotationUml* class. Before the refactoring, *GeneratorDisplay* class had three types of code smells, namely *God Class*, *Complex Class* and *Refused Bequest* smells. On the other hand, *MessageNotationUml* class had only one code smell: *Refused Bequest*. After the refactoring, *MessageNotationUml* class received the other three types of code smells that were affecting *GeneratorDisplay* class. However, *GeneratorDisplay* class continued having the three types of code smells. That is, in addition to introducing code smells in the target class, *Move Method* refactoring did not remove the code smells from the source class. To make matters worse, *Move Method* refactoring also introduced a fifth type of code smell that was not affecting any one of the both classes before. It introduced a *Spaghetti Code* smell since the moved method interacts, through a method call, with an existing method of the *MessageNotationUml* class that was long (in terms of LOC). This *Move Method* refactoring instance is a critical one since it is responsible for creating two out of three relevant code smells (*God Class* and *Complex Class* smells), and it is also responsible to introduce the *Spaghetti Code* smell.

5.2.3 Method Extraction Patterns

In the last category, we only found creational patterns involving the *Extract Method* refactoring. This refactoring type was often related to the creation of two types of smells: *Divergent Change* smell in 41% of the cases, and *Feature Envy* smell in 64% of the cases. However, when we analyzed these pattern instances, we observed that most of them occurred in the context of: (i) floss refactorings, or (ii) composite root-canal refactorings. Therefore, *Extract Method* refactoring was often not the only factor potentially contributing to the emergence of those code smells. Still, the high incidence of such creational patterns may warn developers that *Extract Method* refactorings should be often followed by *Move Method* refactorings in order to eliminate possibly prevailing *Feature Envy* or *Divergent Change* smells.

The *FixCRLF* class from the Apache Ant project had the *Complex Class* smell. Also, the *execute* method from this class had two code smells, namely *Feature Envy* and *Long Method* smells. This method had two functionalities: executing a scanning task on a source code folder and processing files found in the folder. Through the *Extracted Method* refactoring, the *execute* method was split into a second method called *processFile*. After the refactoring, the *execute* method had only the *Feature Envy* smell while the *processFile* method kept both smells: *Feature Envy* and *Long Method* smells. However, it was also introduced a *Divergent Change* smell in the *processFile* method. [redacted] Also, this refactoring contributed to introduce a code smell at class level, *Spaghetti Code*, which did not exist in the class before the refactoring, and together with the *Complex Class* smell, they can decrease the reusability of the system.

6. THREATS TO VALIDITY

The data collection using the Refactoring Miner tool represents a threat to internal validity because it may find some false-positives and false-negatives. To minimize this threat and check the precision of the tool, we randomly select 2,584 samples by refactoring type and manually validate them. In this sense, we improve confidence regarding the Refactoring Miner precision. Notice that distinct researchers from the ones that applied the tool perform the manual validation. We could not reach the developers to ask their intentions (root canal or floss) in all refactorings detected. Therefore, we include the validation whether the refactoring is root or floss in our manual task, which is a threat. Notice that such analysis is limited to two versions of the source code directed related to the refactoring, not considering all versions in the repository. Moreover, the manual analysis only considers behavior preservation in the classes of the refactoring scope.

The answers for both research questions (Section 3.1) rely on the code smells. Thus, different thresholds can lead to results completely distinct. Therefore, choices of thresholds can pose a threat to this study. Thus, two sets of thresholds

are used to mitigate this menace: *tight* [11] and *relaxed*. We also use the code smells detection rules proposed by Bavota *et al.* [2] to detect three code smells of our study. These three smells detection strategies allow us to derive refactoring classifications based on three different sets of code smells, mitigating the threat of the thresholds choice. Following such approach, we could not find any project from the analyzed sample influencing the results in a significant way. The set of code smells types can be also considered a threat to validity. However, we consider very common smell types, not to mention we also used Bavota *et al.* [2] technique, allowing us to improve the coverage of different code smell types.

[redacted]

7. RELATED WORK

Bavota *et al.* [2] mine the evolution history of three Java open source projects to investigate whether refactorings edits reduce the occurrence of code smells in a program or not. They use Ref-Finder [8] to identify the refactorings, and analyze the projects' major versions. [redacted] In summary, our study improves some aspects of the study reported by Bavota *et al.*: (i) we analyze 23 projects while they analyze only 3; (ii) we collect refactorings in consecutive commits while they consider only major versions; (iii) different from them we evaluate whether a refactoring introduces new smells; (iv) different from them we analyze the code smells lifespan. Next, we discuss some differences and similarities between the studies.

Bavota *et al.* [2] identify 15,008 refactorings edits using Ref-Finder. They manually validate the detected refactorings due to the well-known Ref-Finder's low precision [17]. [redacted] Although Ref-Finder had a good precision in their study, we evaluate Ref-Finder in refactorings collected between major versions of 14 projects analyzed by them and obtained 15% of precision. Moreover, Soares *et al.* [18] perform a manual validation of Ref-Finder analysis and also obtain a low precision (25%). [redacted] We recruited ten students for performing a manual validation to improve confidence in our results and Refactoring Miner has a precision of 88% in our data set.

Bavota *et al.* [2] collect the refactoring edits using only the projects' major versions. Usually, between two major versions, developers perform significant changes in the source code structure. These changes might impact directly on the refactoring collection process, i.e., refactoring edits might be hidden or unidentifiable. A commit may override a refactoring applied in the previous commit. Therefore, the tool cannot detect the refactoring. In our study, we minimize this threat by collecting the refactorings between consecutive commits. Even analyzing pairs of commits, we found some of them changing more than 20 files.

Fujiwara *et al.* [6] and Ratzinger *et al.* [15] investigate the influence of software changes, such as refactoring, on bug fixes required in later versions. They studied whether refactoring reduces the probability of software defects and whether refactoring is more important than bug fixing for software quality. The authors find that an increase in refactoring has a positive interference on software quality. Results show that number of defects in the target period decreases if more refactorings are applied. They also observed a defect decrease if these refactorings increase compared to bug fixes. While the authors correlate refactorings with bug fixes and the interference of such changes on software defects, we are correlating smells with refactorings. We focus on analyzing the interference of refactoring on structural quality.

8. CONCLUSIONS

We conducted a study aiming to understand the relationship between refactorings and code smells. We analyzed 23 projects in order to detect the occurrence of both phenomena. Every single refactoring was classified according to its effect on code smells as positive, negative or neutral. The first finding of this research is that a significant part of detected refactorings tends to be neutral. In other words, they seem not to affect the density of code smells. Second, negative refactorings occur more often than positive refactorings.

Another finding emerged from the removal, non-removal and creational patterns. We could conclude that there are specific types of refactorings that tend to consistently affect a particular type of smell. However, those refactorings more frequently are unsuccessful to remove a particular smell type. That happened, for example, when *Extract Method* refactoring targeted methods with *Feature Envy* smell. Interesting data also emerged from creational patterns, at which three groups emerged from them. The first one included refactorings dealing with generalization and were often related to the creation of *God Class* and *Speculative Generality* smells. The second group represents feature-moving refactorings related to the creation of *God Class*, *Complex Class*, and *Lazy Class* smells. Finally, the last group comprises the *Extract Method* refactorings related to the creation of *Divergent Change* and *Feature Envy* smells.

Future work should comprise further analyses of these data with different goals. A refactoring assistance tool can be built as result of this analysis in order to support developers during refactoring. For instance, this tool could warn developers about the risks of performing a refactoring incorrectly. This tool could also be able of detecting additional code smells during refactorings and recommend alternative or additional refactorings to address the issues.

9. REFERENCES

[1] R. Arcoverde, I. Macia, A. Garcia, and A. von Staa. Automatically detecting architecturally-relevant code anomalies. *Proc. of RSSE '12*, pages 90–91, 2012.

[2] G. Bavota, A. D. Lucia, M. D. Penta, R. Oliveto, and F. Palomba. An experimental investigation on the innate relationship between quality and refactoring. *JSS*, 107:1 – 14, 2015.

[3] D. Cedrim. Experiment data of the research. `http://diegocedrim.github.io/icse-2017-data/`, 2017. [Online; accessed 26-august-2016].

[4] R. de Mello, K. Stolee, and G. Travassos. Investigating samples representativeness for an online experiment in java code search. In *Proc. of ESEM '15*, pages 1–10, 2015.

[5] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1 edition, 1999.

[6] K. Fujiwara, K. Fushida, N. Yoshida, and H. Iida. Assessing refactoring instances and the maintainability benefits of them from version archives. In *Product-Focused Software Process Improvement*, volume 7983, pages 313–323. 2013.

[7] X. Ge, Q. L. DuBose, and E. Murphy-Hill. Reconciling manual and automatic refactoring. In *Proc. of ICSE '12*, pages 211–221, 2012.

[8] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit. Ref-finder: a refactoring reconstruction tool based on logic query templates. In *Proc. of FSE '10*, pages 371–372, 2010.

[9] M. Lanza, R. Marinescu, and S. Ducasse. *Object-Oriented Metrics in Practice*. Springer, 2005.

[10] I. Macia. *On the detection of architecturally relevant code anomalies in software systems*. PhD thesis, Pontifical Catholic University of Rio de Janeiro, 2013.

[11] I. Macia, R. Arcoverde, A. Garcia, C. Chavez, and A. von Staa. On the Relevance of Code Anomalies for Identifying Architecture Degradation Symptoms. *Proc. of CSMR '12*, pages 277–286, 2012.

[12] I. Macia, A. Garcia, C. Chavez, and A. von Staa. Enhancing the detection of code anomalies with architecture-sensitive strategies. In *Proc. of CSMR '13*, pages 177–186, 2013.

[13] L. Mara, G. Honorato, F. D. Medeiros, A. Garcia, and C. Lucena. Hist-inspect: A tool for history-sensitive detection of code smells. In *Proc. of AOSD '11*, pages 65–66, 2011.

[14] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. *Proc. of ICSE '09*, pages 287–297, 2009.

[15] J. Ratzinger, T. Sigmund, and H. C. Gall. On the relation of refactorings and software defect prediction. In *Proc. of MSR '08*, pages 35–38, 2008.

[16] Scitools. Understand Software. `https://scitools.com`, 2016. [Online; accessed 18-april-2015].

[17] G. Soares, R. Gheyi, and T. Massoni. Automated behavioral testing of refactoring engines. *IEEE Transactions on Software Engineering*, 39(2):147–162, 2013.

[18] G. Soares, R. Gheyi, E. Murphy-Hill, and B. Johnson. Comparing approaches to analyze refactoring activity on software repositories. *JSS*, 86(4):1006 – 1022, 2013.

[19] K. Stroggylos and D. Spinellis. Refactoring–does it improve software quality? In *Proc. of WoSQ '07*, 2007.

[20] N. Tsantalis, V. Guana, E. Stroulia, and A. Hindle. A multidimensional empirical study on refactoring activity. In *Proc. of CASCON '13*, pages 132–146, 2013.

[21] Z. Xing and E. Stroulia. Umldiff: An algorithm for object-oriented design differencing. In *Proc. of ASE '05*, pages 54–65, 2005.

[22] A. Yamashita. Assessing the Capability of Code Smells to Explain Maintenance Problems: an Empirical Study Combining Quantitative and Qualitative Data. *Empirical Software Engineering*, 19(4):1111–1143, 2013.

[23] A. Yamashita and L. Moonen. To what extent can maintenance problems be predicted by code smell detection? - an empirical study. *Information and Software Technology*, 55(12):2223–2242, 2013.