Understanding the Impact of Refactoring on Smells: A Longitudinal Study of 23 Software Projects

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

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

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

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

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

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

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

ABSTRACT

Code smells in a program represent indications of structural quality problems, which can be addressed by software refactoring. However, refactoring intends to achieve different goals in practice, and its application may not reduce smelly structures. Developers may neglect or end up creating new code smells through refactoring. Unfortunately, little has been reported about the beneficial and harmful effects of refactoring on code smells. 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 16,566 refactorings distributed in 10 different types. Even though 79.4% of the refactorings touched smelly elements, 57% did not reduce their occurrences. Surprisingly, only 9.7% of refactorings removed smells, while 33.3% induced the introduction of new ones. More than 95% of such refactoring-induced smells were not removed in successive commits, which suggest refactorings tend to more frequently introduce long-living smells instead of eliminating existing ones. We also characterized and quantified typical refactoring-smell patterns, and observed that harmful patterns are frequent, including: (i) approximately 30% of the *Move Method* and *Pull Up Method* refactorings induced the emergence of *God Class*, and (ii) the *Extract Superclass* refactoring creates the smell *Speculative Generality* in 68% of the cases.

CCS CONCEPTS

• **Software and its engineering → Software evolution**;

KEYWORDS

Refactoring, Code Smells, Structural Quality

ACM Reference Format:
Diego Cedrim, Alessandro Garcia, Melina Mongiovi, Rohit Gheyi, Leonardo Sousa, Rafael de Mello, Baldoino Fonseca, Márcio Ribeiro, and Alexander Chávez. 2017. Understanding the Impact of Refactoring on Smells: A Longitudinal Study of 23 Software Projects. In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 4–8, 2017 (ESEC/FSE'17),* 11 pages.
https://doi.org/10.1145/3106237.3106259

1 INTRODUCTION

Code smells represent indications of software structural problems in a program [8]. Several types of code smells are recognized as critical by software developers [25, 36]. Examples of such smell types include *God Class*, *Long Method* and *Speculative Generality* [17, 25, 36]. Thus, code smells may suggest structures in the source code that require refactoring [8]. Refactoring is a common practice employed by practitioners along software maintenance [22, 29]. Refactoring is a program transformation used for improving the structure of a program while preserving its observable behavior [8]. Examples of commonly applied types of refactoring involve [22]: (i) restructuring or moving class members, such as *Extract Methods*, *Move Method* and *Pull Up Method*, and (ii) extracting new elements, such as *Extract Superclass* and *Extract Interface*.

The goals of refactoring widely vary in practice [3, 14, 27]. Refactoring goals include combating design degradation, reducing maintenance effort, and facilitating feature additions or bug fixes [3, 14, 27]. In order to achieve these goals, developers apply two tactics during refactoring [22], namely *root-canal refactoring* and *floss refactoring*. Root-canal refactoring is used for strictly improving the source code structure and consists of pure refactoring. Floss refactoring consists of refactoring the code together with non-structural changes as a means to reach other goals, such as adding features or removing bugs.

Independently of refactoring tactic employed, the increase of code smells in a program is harmful. The increased density of code smells often relates to design degradation [17, 19, 24, 34], fault proneness [13, 26], and higher maintenance effort [9, 35]. However, as floss refactoring is often applied [22], developers may often end up introducing, rather than reducing, code smells. Even in situations

---

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. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ESEC/FSE'17, September 4–8, 2017, Paderborn, Germany
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5105-8/17/09…$15.00
https://doi.org/10.1145/3106237.3106259

where root-canal refactorings are performed, developers may either fail to remove a smell or introduce new ones.

Unfortunately, the impact of refactoring on smells is rarely investigated in depth. To the best of our knowledge, there is no study that thoroughly characterizes both positive and negative effects of refactoring on code smells. Only recently, Bavota *et al.* [3] 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 increase, rather than decrease, code smells in a program. Moreover, they did not make a distinction between root-canal and floss refactoring in their analyses. Finally, given the nature and size of their sample, they could not characterize recurring relationships between refactoring and smell types.

Thus, we conduct a longitudinal study that analyzes both the beneficial and negative impact of refactoring changes on the density of smells. We analyze not only if refactoring reduces smells, but also if and to what extent specific types of refactoring are often related to the introduction of new smells. Instead of being limited to the analysis of a few major versions in a few projects, we consider 113,306 versions distributed among 23 open source projects. We classify each refactoring instance according to its interference on the existing and new smells located in the refactored elements. In our study, we classify a given refactoring instance in one of the three cases: (i) *positive* if the absolute number of smells in the elements decreases after the program transformation; (ii) *negative* if it increases; or (iii) *neutral* if it remains the same. This classification is used to analyze whether certain refactoring types tend to improve or decrease the smelly structure of a program. This analysis was also performed in samples of root-canal and floss refactorings.

We identified and analyzed 16,566 refactorings classified in 10 commonly used [22] refactoring types. Thirteen code smell types are used to classify the collected refactorings. These code smell types were selected because they are conceptually associated with the definition of the refactoring types [8], i.e., the definition of each refactoring type is explicitly associated with one or more code smells addressed in our study. Surprisingly, our study revealed that either neutral or negative effects of software refactoring are much more frequent than positive effects. In particular, we derived the following new findings as compared to previous studies.

Refactorings often touch smelly elements, but they are neutral. The aforementioned study [3] found 42% of the refactorings touched smelly elements in their smaller sample. We observed a higher frequency in our sample: approximately 80% of the refactorings touched smelly elements. This frequency was consistent across the majority of the refactoring types. Moreover, 57% of the refactorings are neutral. Surprisingly, even root-canal refactorings often did not reduce the density of code smells in the refactored elements. These findings suggest developers need more guidance to remove a code smell once they start restructuring a smelly element.

Stinky refactorings. We found 33.3% of refactorings are *stinky*, i.e., they were negative refactorings, which were related to the introduction of new smells. Such refactorings occurred three times more frequently than positive ones. Only 9.7% of refactorings removed smells. We also concluded that more than 95% of refactoring-induced smells were not removed afterwards in successive commits. Stinky refactorings were also surprisingly frequent in root-canal refactorings, i.e., when developers perform pure refactoring. These findings shed light on how refactorings, if intended to improve program structure or not, may degrade the smelly structure of a program. In particular, it seems developers should be at least warned of smells being introduced along root-canal refactorings.

Harmful refactoring-smell patterns. We characterized and quantified recurring patterns related to the beneficial, neutral and harmful effects of specific refactoring types on code smells. Harmful patterns were more frequent than beneficial ones. For instance, refactorings intended at moving methods – such as *Move Method* and *Pull Up Method* – tended to induce the emergence of *God Class* in the target class without removing smells in the source class. 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. Several other types of refactoring were surprisingly often related to smells emerging after the transformation. For instance, the *Extract Superclass* refactoring creates the *Speculative Generality* smell in 68% of the cases.

This work is organized as follows: Section 2 provides basic concepts. Section 3 presents the study planning. Sections 4 and 5 present our results. Section 6 describes the threats to validity. Section 7 relates our study with previous work. Section 8 concludes the study.

2 CONCEPTS AND MOTIVATION

A smell is a surface indication that usually corresponds to a deeper structural problem [8, 10, 17, 20, 23]. For instance, a class with several responsibilities is known as *God Class*. This smell makes the class hard to read, modify and evolve. 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 structure 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. However, it might be not always the case that refactorings are successful in removing smells. Even worse, a new code smell can be introduced by refactoring. For instance, suppose that a developer tried to generalize the *Person* class. So, he applied an *Extract Superclass* refactoring in the *Person* class, creating its superclass called *LivingBeing*. However, this generalization was never explored, so the developer created a smell, called *Speculative Generality*, via refactoring. There has been little effort to characterize these occasions, which happen on a non-ignorable frequency, as we can observe in Section 5.2.

2.1 Refactoring Classification

This section presents a scheme [5] 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 a set of software projects. 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 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 $E(v_i)$. For instance, $E(v_2)$ is composed of *Person* and *TelephoneNumber* classes (Section 2), including their 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 of two elements: the refactoring type (rt_i) and the set of refactored elements represented by e_i. So, the function R returns the set of all refactorings detected in a pair of versions. Thus, $R(v_1, v_2) = \{r_1(\text{Extract Class}, e_1)\}$, where $e_1 = \{Person, TelephoneNumber\}$.

Refactored Elements. In this work, we consider as refactored elements all those directly affected by the refactoring. For instance, let us 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, are $\{m, A, B\}$. All callers of m are indirectly affected by this refactoring, but we do not consider them as refactored elements. Similar reasoning applies to the other refactoring types; thus, for each refactoring type, a different set of refactored elements is used. The complete list of what is considered refactored elements for all refactoring types is available at [4].

CS is a code smell detection function where $CS(e_i) = \{cs_1, \cdots\}$ returns a set of code smells present in a set e_i of software elements. We only analyze code smells related to the refactored elements. In this way, we can say that $CS_{b[r]}(e_i)$ is the set of code smells of e_i before the application of the refactoring r. On the other hand, $CS_{a[r]}(e_i)$ is the set of code smells found after the application of r. Considering the aforementioned r_1 refactoring, we have: $CS_{b[r_1]}(\{Person\}) = \{\text{God Class}\}$, $CS_{a[r_1]}(\{Person, TelephoneNumber\}) = \emptyset$.

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_i is a set of software elements, r is a refactoring and $|CS_{b[r]}(e_i)| = x$. After r refactoring, $|CS_{a[r]}(e_i)| = y$. Depending on x and y, it is possible to classify r. If $x > y$, r reduced the number of smells on e_i and, because of that, r is considered a *positive refactoring*. Otherwise, if $x < y$, r increased the number of smells on e_i; thus, r is a *negative refactoring*. When $x = y$, r is a *neutral refactoring*. For instance, $|CS_{b[r_1]}(e_1)| = 1$ and $|CS_{a[r_1]}(e_1)| = 0$. Thus, the r_1 refactoring performed in *PhoneBook* system is a positive one.

2.2 Refactoring-Smell Patterns

We determine the relationship between each refactoring instance on the removal or addition of a code smell. Moreover, the refactoring classification process can also reveal to what extent and which types of refactorings often increase, rather than decrease, the number of code smells in software projects. This classification enables us to characterize recurring relationships between code smells and refactorings. 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 refactoring and smell types as *patterns*.

Let $K = \{r_1, r_2, \cdots, r_n\}$ be the set of all detected refactorings after analyzing the set S. Thus, K_{rt} is the subset of K of refactorings of the type rt. The set $K^{+}_{rt,cs}$ is the K subset composed of refactorings of the type rt that added code smells of type cs in any refactored element, while $K^{-}_{rt,cs}$ is the K subset that removed code smells of type cs. Finally, $K^{*}_{rt,cs}$ is the K subset composed of refactorings of the type rt that satisfies 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.

Applied in Smelly Code. e is a smelly element if and only if $CS(e) \neq \emptyset$. Consider $r = \{rt; e\}$. We say r was applied in program elements hosting at least one code smell if any element belonging to e is a smelly element, i.e., r is *Applied in Smelly Code* if and only if there is a code smell of any type in the *refactored elements*.

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. Thus, creational patterns represent cases of stinky refactorings.

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. We are also interested in studying what types of code smells 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: $|K^{*}_{rt,cs}|/|K_{rt}| \geq \gamma$.

3 STUDY PLANNING

This section presents the study planning.

3.1 Research Questions

Software refactoring might interfere in the presence of code smells. As illustrated in Section 2, the number of code smells located in refactored elements should be ideally reduced. Therefore, our study aims at addressing the following research question:

RQ1. Does refactoring reduce the density of code smells?

We address this question by relying on the classification of each refactoring detected in real projects. This procedure enables us to compute how frequent each refactoring classification occurs across the projects. First, all instances of refactorings and code smells present in a set S of software were detected. Then, all refactoring instances were classified according to Section 2.1. Also, we divided the refactorings into root-canal and floss refactoring (Section 3.2.3). Let p the number of refactorings classified as positive; n the number of negative refactorings; and k representing the number of neutral refactorings. If $n > p$ and $n > k$, we can state that the application of refactorings are likely increasing the number of code smells of projects. Otherwise, if $p > n$ and $p > k$, the answer to our research question is **yes**, refactorings tend to remove code smells. Another possible case is when $k > p$ and $k > n$. In this scenario, refactorings would tend to neither introduce nor remove code smells.

It is also important to understand and distinguish the impact of specific refactoring types on code smells. Some types of refactoring might consistently remove (or fail to do so) or even frequently introduce specific smell types across software projects. Section 2.2 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 by analyzing the impact of refactoring types on smells located in the refactored elements. The knowledge about non-removal and creational patterns make developers informed about the possibilities and risks of missing and introducing certain smells along either root-canal or floss refactorings.

3.2 Study Phases

This section 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 stars; (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 table presents the (i) name, (ii) lines of code and (iii) number of commits for each project.

3.2.2 Phase 2: Smell and Refactoring Detection. This phase is in charge of detecting refactorings in all subsequent pairs of versions v_i and v_{i+1}. It 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 [30, 31] to support the detection of refactoring instances. This tool implements a lightweight version of UMLDiff [33] algorithm for differencing object-oriented models. The precision of 96.4% reported by Tsantalis *et al.* [31] led to a very low rate of false positives, as confirmed in our validation phase (Section 3.2.4). This tool supports the detection of 11 refactoring types, which are amongst the ones reported by Murphy-Hill *et al.* [22] as the most common refactoring types. All refactoring types detected by Refactoring Miner were considered in this study, except the *Rename Method* refactoring. We discarded this refactoring type as it was not directly related to one of the code smells addressed in our study. 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.

Code Smell Detection. 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 [15, 17] to detect code smells. This procedure is the implementation of *CS* function defined in Section 2.1. The specific metrics and thresholds for code smell detection were defined in [17, 18]. These rules were used because: (i) they represent refinements of well-known rules proposed by Lanza *et al.* [15], which are well documented and used in previous studies (e.g., [21, 39]); and (ii) they have, on average, precision of 0.72 and recall of 0.81 [16].

These rules detect five code smells: *God Class*, *Long Method*, *Feature Envy*, *Shotgun Surgery* and *Divergent Change*. Table 2 shows examples of rules used to identify code smells. The rules use the following metrics: (i) Lines of Code — LOC; (ii) Coupling Between Objects — CBO; (iii) Number of Methods — NOM; (iv) Cyclomatic Complexity — CC; (v) Lack of Cohesion of Methods — LCOM; (vi) Fan-out — FO; and (vii) Fan-in — FI. We also considered eight additional smell types: *Complex Class*, *Lazy Class*, *Long Parameter List*, *Message Chain*, *Refused Bequest*, *Spaghetti Code*, *Speculative Generality*, and *Class Data should be Private*.

The selection of these smell types was due these code smells are very common and tend to be related to design degradation symptoms [17]. Another reason to select these smell types was their direct relation with the most frequent refactoring types [8]. Murphy-Hill *et al.* [22] reported the refactorings often performed

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	23,597
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
Total	2,982,190	113,306

by developers and we analyzed which code smell types these refactorings may intend to remove [8]. Refactoring Miner is capable of detecting the refactoring types that remove these smells.

We implemented a tool to detect all the 13 types of code smells [6]. In addition to the rules presented in Table 2, we also implemented all rules proposed in a recent study [3]. These rules detect the eight additional types of code smells. The complete list of the rules and the tool we implemented are available in our study's website [4].

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 $>\varpi$)

Rules for detecting smells play a central role in our study. Thus, we must guarantee that our results are not biased by a single set of detection rules. Different thresholds can lead to different results [7, 11]. 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, known as *tight* set, represents the thresholds previously validated in the study by Macia *et al.* [17]. We named this strategy as *tight* because it relies on the use of high threshold values aiming to detect only critical code smells across the projects. 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) set of thresholds, we also used the detection rules proposed by Bavota *et al.* [3].

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 to 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 [31]. The algorithm always compares subsequent versions of the projects. Let us suppose a project that has only three commits: 1, 2, and 3. In this project, the R function would be computed for the following pairs: R(1, 2) and R(2, 3). The set $V(s)$ of a Git repository s is the list of all non-merge commits in the master branch ordered chronologically.

Root-Canal vs. Floss Refactoring. There are two tactics of refactoring [22]: (i) root-canal refactoring; and (ii) floss refactoring. 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 solely improving code structure and involves a process consisting of exclusive refactoring. In this way, this study comprises a manual inspection of a randomly selected sample of refactorings. In this manual inspection, we evaluate if a refactoring is root-canal or floss. We analyzed manually whether the changes performed during the refactoring do not modify the behavior. We classify a transformation as floss when we identify behavioral changes, such as an addition of methods or changes in a method body not related to refactoring transformations. When no behavioral changes are detected, we classify the refactoring as root-canal. This manual inspection will enable us in revealing the percentage of positive, negative and neutral effects in the context of both root-canal and floss refactorings (Section 4). This inspection was performed by three researchers. Two of them are very experienced refactoring researchers. The most experienced one solved the conflicts. 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%.

3.2.4 *Phase 4: Manual Validation.*

The last phase is responsible for all data validation. As the first three phases rely on 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 our 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 10 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 ensure an acceptable confidence level in 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. We recruited ten undergraduate students from another research group to also analyze the samples. The samples were divided into ten disjointed sets, and each student validated a different one. In general, it was observed a high accuracy for each refactoring type, with a mean of 88.36 %. Details can be found in the study's website [4].

4 REFACTORING AND SMELLS

This section presents and discusses the data used to answer the first research question. The refactoring detection procedure identified 16,566 refactorings. Table 3 presents the refactoring types ordered by the number of their 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 *Extract Method*, similarly to a previous study that analyzed refactoring frequencies in other systems [22]. Table 3 confirms most of the other refactoring types also occur frequently in our sample.

Most refactorings touch smelly elements. Before addressing RQ1, we first analyze the frequency of refactoring types that touch smelly elements. Thus, we also compute how many times each type of refactoring was *applied in smelly code* (Section 2.2). The results are shown in the third column in terms of both absolute number of occurrences and percentages (in brackets). We can observe that developers tend to often apply refactorings in smelly elements of a

Table 3: The impact of common refactorings types

Refactoring Type	Occurences	Applied in Smelly Code	Neutral	Negative	Positive
Extract Method	7,517	6,411 (85.2%)	2,917 (38.8%)	3,914 (52.1%)	686 (9.1%)
Move Field	4,356	3,362 (77.1%)	3,784 (86.9%)	438 (10.1%)	134 (3.1%)
Inline Method	1,528	1,134 (74.2%)	732 (47.9%)	214 (14.0%)	582 (38.1%)
Move Method	1,404	1,049 (74.7%)	1,008 (71.8%)	297 (21.2%)	99 (7.1%)
Pull Up Method	629	511 (81.2%)	430 (68.4%)	155 (24.6%)	44 (7.0%)
Pull Up Field	465	333 (71.6%)	338 (72.7%)	103 (22.2%)	24 (5.2%)
Extract Superclass	342	131 (38.3%)	89 (26%)	246 (71.9%)	7 (2.0%)
Extract Interface	133	65 (48.8%)	11 (8.3%)	122 (91.7%)	0 (0%)
Push Down Method	114	98 (85.9%)	76 (66.7%)	11 (9.6%)	27 (23.7%)
Push Down Field	78	58 (74.3%)	62 (79.5%)	13 (16.7%)	3 (3.8%)
Totals	16,566	13,152 (79.4%)	9,447 (57%)	5,513 (33.3%)	1,606 (9.7%)

program. Seven refactoring types have been applied in smelly code elements in more than 70% of the occurrences (Table 3).

One could wonder if many elements are tagged as smelly in the analyzed programs, thereby increasing the probability of refactorings often touching smelly elements. Then, we computed the probability of randomly choosing a smelly element in our dataset (|smelly elements|/|all elements|), which is 0.3%. This low probability shows that, in our dataset, refactorings did not target smelly elements by coincidence. Refactorings indeed tend to concentrate on smelly elements, which were confined to a vast minority of the program elements. This behavior was consistently observed for both root-canal and floss refactorings.

4.1 Smell-Neutral Refactorings are Common

The three last columns of Table 3 present respectively the incidence rate of neutral, negative and positive refactorings. Surprisingly, the neutral classification was the most frequent one for 7 refactoring types, namely *Move Field*, *Inline Method*, *Move Method*, *Pull Up Method*, *Pull Up Field*, *Push Down Method*, and *Push Down Field*. Even though refactorings are frequently applied in smelly elements, they often do not reduce the smells.

The data presented in Table 3 was produced with smell detection strategies based on a set of *tight* thresholds. To make sure our findings were not biased by this particular set of thresholds, we have also classified the refactorings using *relaxed* thresholds. Finally, we have also used another set of detection strategies, the same used by Bavota *et al.* [2]. Figure 1 shows the general proportion of neutral, positive and negative refactorings using all these three classification methods, labeled as *Tight*, *Relaxed* and *Bavota*.

An analysis of Figure 1 confirms there is indeed a general trend: independently of the smell detection strategy, neutral refactorings are much more frequent than positive and negative refactorings. When we analyze each individual project, the same classification distribution is observed, i.e., neutral refactorings represent the vast majority in all the projects. In our manual validation (Section 3.2.3), we found that 31.5% of the refactorings are root-canal. Even when this tactic is applied, refactorings often do not reduce the density of code smells in the refactored elements. These findings suggest developers need more guidance to remove a code smell once they start restructuring a smelly element, i.e., when they perform root-canal refactorings.

4.2 Stinky Refactorings

Surprisingly, 33.3% of the refactorings were found to be *stinky* (Table 3); they are related to an increase of smells in the refactored elements. Moreover, when we analyzed the commits performed after the negative refactorings, we also concluded that more than 95% of refactoring-induced smells were not removed afterwards. Only 9.7% of refactorings removed smells, according to Table 3. Negative refactorings were more frequent than positive refactorings according to our three classification methods presented in Figure 1. Stinky effects are more frequent than positive ones in the context of both root-canal and floss refactorings as well.

Negative refactorings were more frequent than neutral ones in the context of three refactoring types: *Extract Method*, *Extract Superclass* and *Extract Interface*. Interestingly, *Extract Method* is the most frequent type of refactoring (Table 3). Section 5 discusses different patterns involving this refactoring type. Moreover, the refactorings that involve multiple changes in a class hierarchy, such as *Extract Superclass* and *Extract Interface*, tend to be negative. This fact might indicate developers need more guidance on refactoring class hierarchies even in the context of root-canal refactoring.

These results enable us to answer RQ1: refactorings made by developers in real projects often do not remove code smells. On the contrary, most of the refactorings are neutral or stinky. This observation also prevails if we only consider refactoring types that, according to their description in Fowler's catalog [8], are explicitly associated with specific code smell types addressed in our study. For instance, the mechanics for applying *Move Method*, *Pull Up Method* and *Move Field* refactorings are 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*.

Our data suggest that most refactorings do not remove smells even in the context of root-canal refactorings. There are possible interpretations of this finding. First, critical design problems in a program may not be related to code smells. If so, this fact may explain why developers either neglect or introduce code smells

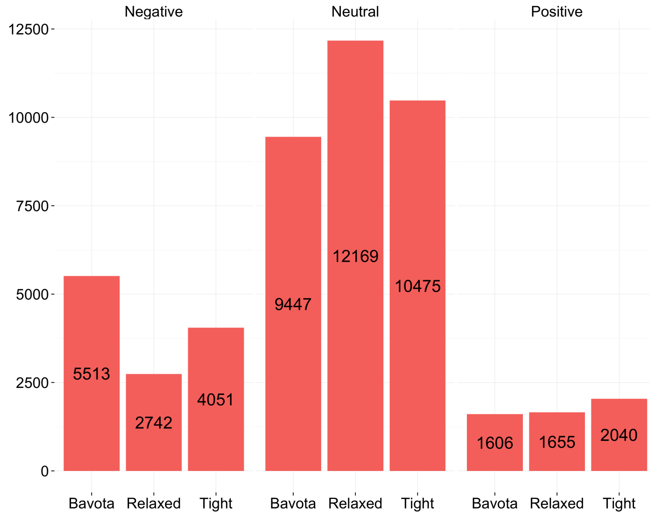

Figure 1: Results of the data collection phase

through refactoring. However, previous studies [17, 23, 24, 38] indicate that design problems are often located in modules containing two or more code smells. Second, similarly to previous studies, we use metrics and thresholds to detect all smells. The proper choice of metrics and thresholds may be sensitive to particular developers [11] and other project-specific factors [7]. As a consequence, our detection of code smells may not reflect what developers truly consider as smells. However, our previous studies involving developers [7, 11] suggest that heuristics used by developers are often not essentially different from smell detection strategies adopted in our study. Our understanding is that refactoring indeed ignores or introduces technical debt in the source code. Finally, it may be the case that our set of studied refactoring types are not among those used by developers to actually remove design problems. Still, it is troublesome that developers introduce smells through refactoring regardless its type and tactic.

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. Section 2.2 defines three categories of such patterns, i.e., removal, non-removal and creational patterns. Section 5.1 focuses on discussing the removal and non-removal patterns, while Section 5.2 discusses the creational (i.e., stinky) patterns.

We will focus on discussing patterns in which more than 15% of the instances of a refactoring type was related 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 lifetime of the prevailing and introduced smells related to the non-removal and creational patterns. We checked if such smells – prevailing or emerging in commits involving one or more refactorings – were either removed or not in subsequent commits. We considered *subsequent commits* all those ones performed until the last commit of each project. Therefore, we were able to identify precisely when a particular code smell was removed. 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 in the next commits.

5.1 Removal vs. Non-Removal Patterns

Table 4 presents the cases of removal and non-removal patterns observed. 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 15% are shown in bold. The last column presents the percentage of root-canal refactoring for each pattern. For instance, the first row informs that 37.5% of the *Extract Interface* refactorings related to *God Class*, either by non-removal or removal pattern, are root-canal refactoring.

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 42.6% (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 Envy* smells as methods to other classes. However, the vast majority of those extracted methods (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 the *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 Class* 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 51.3% and 29.4% 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 Class* 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%).

We can observe a non-ignorable frequency of root-canal refactorings spread across the patterns in Table 4. Even when the root-canal frequency is as high as 50%, the developers are not able to remove the code smell. Since the refactorings belonging to the patterns could be just the first step towards the code smell removal, we computed the code smells' lifetime after the refactoring. In 95% of the cases, the code smells were not removed. This shows that, even when developers refactor purely to improve the code structure (root-canal), they do not succeed on removing the code smells.

Table 4: Removal and non-removal patterns

Refactoring	Code Smell	Non-Removal	Removal	Root-Canal
Extract Interface	God Class	**20.3%**	2%	37.5%
Extract Method	Divergent Change	**34.6%**	7%	25%
Extract Method	Feature Envy	**42.6%**	11%	28.3%
Extract Method	God Class	**48.6%**	0%	26.2%
Move Field	God Class	**29.4%**	**27%**	48%
Move Method	God Class	**51.3%**	**23%**	8.0%
Pull Up Field	God Class	**44.7%**	8%	76.1%
Pull Up Method	God Class	**61.3%**	10%	2.3%
Push Down Field	God Class	**55.7%**	12%	57.1%
Push Down Method	God Class	**54.3%**	15%	22.2%
Push Down Method	Lazy Class	2.6%	**52%**	11.1%
Push Down Method	Refused Bequest	9.2%	**23%**	50%

5.2 Creational Patterns

Interesting data also emerged from creational patterns detected in our dataset. We divided these patterns into three groups [8] 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 5 presents all creational patterns found with the same structure presented in Table 4.

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 Table 5 that *Pull Up Method*, and *Pull Up Field* refactorings are related to the creation of *God Class* smells in 28%, and 61% 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 95% 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 (commit *002901b*). 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.

Table 5: Creational patterns

Refactoring	Code Smell	Creational	Root-Canal
Extract Method	Divergent Change	40.5%	24.7%
Extract Method	Feature Envy	63.8%	32.1%
Extract Superclass	Lazy Class	33.2%	17.8%
Extract Superclass	Refused Bequest	20.3%	0%
Extract Superclass	Spec. Generality	68.3%	0%
Move Method	Complex Class	15%	0%
Move Method	God Class	35%	17.6%
Move Method	Lazy Class	16%	16%
Pull Up Field	God Class	61.2%	2%
Pull Up Field	Spec. Generality	34.1%	90.2%
Pull Up Method	God Class	28.3%	2.9%
Pull Up Method	Spaghetti Code	23.2%	0%

5.2.2 Feature-Moving Patterns. Refactorings aiming at moving features between objects were also part of our catalog of detected creational patterns. *Move Method* refactorings were related to the creation 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 [22]. 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 the *GeneratorDisplay* class to the *MessageNotationUml* class. Before the refactoring, the *GeneratorDisplay* class had three types of code smells, namely *God Class*, *Complex Class* and *Refused Bequest* smells. On the other hand, the *MessageNotationUml* class had only one code smell: *Refused Bequest*. After the refactoring, the *MessageNotationUml* class received the other three types of code smells that were affecting the *GeneratorDisplay* class. However, the *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. Furthermore, it was introduced another code smell in the *FixCRLF* class. After the refactoring, it also had a *Spaghetti Code* smell. In this example, the *Extracted Method* refactoring was responsible for creating an instance of a method extraction pattern. 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.

Table 5 presents the percentage of root-canal refactorings for each creational pattern. Considering all instances involved in those patterns, 26.5% are root-canal refactorings. This is an alarming rate. Developers introduce smells when refactoring even when they are performing solely structural-improvement activities. To make the matter worse, this behavior occurs consistently between specific refactoring-smell pairs. This non-tolerable rate presented indicates developers might need proper support during refactoring to avoid structural degradation even when they, clearly, want to improve the structure via root-canal refactoring.

6 THREATS TO VALIDITY

Internal Validity. The data collection using the Refactoring Miner represents a threat to internal validity because it may find false positives. To minimize this threat and check the tool's precision, we randomly select a sample of 2,584 refactorings and manually validate them. Section 3.2.4 presents the procedure used to estimate the precision of this tool in our dataset. We observed a high precision for each refactoring type, with a median of 88.36%. 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 sample analyzed represent a key factor to provide confidence in the results reported in this work.

We could not reach the developers to ask their intentions (root-canal or floss) in all refactorings detected. Therefore, we performed a manual validation analysis to check whether each refactoring instance was part of a root-canal or floss refactoring. This manual validation also represents a threat to internal validity. To mitigate this threat, part of the sample was validated by two researches. The third researcher solved cases of conflict.

External Validtiy. The selection of subjects in empirical studies is a recurrent external threat to validity. We mitigate this threat by establishing a process to sample a set of valid projects randomly from GitHub. As a result, we yield relevant Java projects with an interesting diversity of structure and size. Moreover, we also included in the sample the three projects used by Bavota *et al.* [3], so that we could contrast our findings with theirs (Section 7).

7 RELATED WORK

Elements touched by refactoring. Bavota *et al.* [3] mined the evolution history of 3 Java open source projects to investigate if refactorings occur on code elements that certain indicators suggest a need for refactoring. Their considered indicators include structural quality metrics and the presence of smells. They also measure the effectiveness of refactorings regarding their ability to remove smells. According to their results, quality metrics do not show a clear relationship with refactoring and 42% of the refactorings are applied on smelly elements, in which only 7% of them remove smells. Different from Bavota *et al.*, our results indicate that most of the refactorings (80%) are performed in elements with code smells, in which 9.7% of them remove smells and 33.3% induce new smells.

The procedure that we followed may explain why our results are different from the one presented by Bavota *et al.*. We collected refactorings between commits while they collected refactorings using only the projects' major versions. Usually, between two major versions, developers perform significant changes in the source code structure. Therefore, they probably missed refactorings when they followed this procedure since refactorings might be hidden or unidentifiable. In our study, we mitigate this threat by collecting the refactorings between consecutive commits. In summary, our study improves several aspects of the study reported by Bavota *et al.*. First, we analyzed 23 projects while they analyzed only 3. Second, we collected refactorings in consecutive commits. Third, we used a refactoring detection tool (Refactoring Miner [31]) with a good precision rather than the well-known Ref-Finder's low precision [28]. In addition, we evaluated if and when refactorings are stinky by introducing new smells in the context of both root-canal and floss refactoring. We have also characterized refactoring-smell patterns.

Motivations to refactor. Our data showed that most of refactorings (80%) touch smelly elements. Even though the specific motivation of the developers is unknown, we actually observed a similar behavior when developers apply both root-canal and floss refactoring. Mainly in the former case, one would expect developers explicitly intend to improve code structure. Regarding developers' motivation, Silva *et al.* [27] investigated the reasons that drive developers to refactor their code. They identified refactorings on 748 Java projects in the GitHub repository. Then, they asked developers why they performed the identified refactorings. Their results indicate that fixing a bug or changing the requirements, such as feature additions, mainly drives refactorings. Their results show that the refactored code may contain code smells, although developers did not mention it explicitly as the intention to refactor. On the other hand, the study of Yamashita & Moonen [37] reports that developers often consider smells as critical.

Benefits to refactor. Kim et al. [14] conducted a three-folded investigation at Microsoft about refactoring through a survey, interviews, and data analysis of the Windows version history. Their results indicate that the refactoring in practice seems to differ from the rigorous definition of behavior-preservation transformation found in the literature [8]. This result provides us an additional support to investigate floss refactorings since developers also perform non-behavior-preservation transformations while refactoring. The survey participants also reported the benefits they have observed from refactoring. The two most cited benefits were improved readability and maintainability. Although the participants did not mention code smells, the presence of code smells affects negatively the two benefits they claim they want to achieve [12, 34, 38].

Introduction of code smells. Tufano *et al.* [32] investigated the circumstances that led to the introduction of smells, not specifically in the context of software refactoring. The authors analyzed issues and tags associated with commits that introduced smells on open source projects. Their results indicate that most of code smells are introduced during enhancement activities (between 60% and 66%). They also found that between 4% and 11% of the smell-introducing commits were tagged as refactoring. However, our study goes beyond: our findings indicate that stinky refactorings are frequent: refactorings are related to the introduction of code smells in 33.3% of the cases. We also characterized and quantified typical refactoring-smell patterns, and observed that certain stinky patterns are very frequent.

Negative refactorings. We conducted a preliminary study [5] that analyzes how often the commonly-used refactoring types affect the density of 5 types of smells along the version histories of 25 projects. Our findings are based on the analysis of 2,635 refactorings distributed in 11 different types. Surprisingly, 95.1% of refactorings are neutrals. Only 2.24% of refactoring were positive, and 2.66% negatives. Our new results show that negative refactorings are much more frequent. This discrepancy between the studies can be explained by the nature of the projects. In the preliminary study, we considered only small projects with fewer developers.

8 CONCLUSIONS

We conducted a study aiming to understand the relationship between refactorings and code smells in 23 projects. First, we have observed that although refactorings touch smelly elements, they are often smell-neutral. Second, stinky refactorings occur more often than positive refactorings. Stinky refactorings were also surprisingly frequent in root-canal refactorings, i.e., when developers are solely focused on improving the program structure. These findings suggest developers need more guidance to fully remove a code smell once they restructure a smelly element.

In order to better guide developers, we have investigated which recurring refactoring-smell pairs tend to produce stinky, neutral or positive effects. We achieved this goal by revealing and characterizing removal, non-removal and creational patterns. We found a wide range of creational and non-removal patterns, which were much more frequent than positive patterns. Extract Method is a refactoring type frequently involved in both stinky and non-removal patterns. Moreover, we decomposed creational patterns in three groups. The first group included refactorings dealing with generalization: they were often related to the creation of *God Class* and *Speculative Generality* smells. The second group represents feature-moving refactorings, which induced the creation of *God Class*, *Complex Class*, and *Lazy Class* smells. Finally, the last group comprises the *Extract Method* refactorings, which were related to the creation of *Divergent Change* and *Feature Envy* smells.

The aforementioned findings can help practitioners and tool engineers. Practitioners using refactoring are now better informed of when they may typically introduce neutral or stinky refactorings in their programs. Moreover, a refactoring assistance tool can be built in order to: (i) detect when developers performed refactoring in a commit, and (ii) depending on the refactoring characteristics (e.g. occurrence of a root-canal refactoring), immediately produce warnings or recommendations for the developer. For example, a refactoring assistant could warn developers of an emerging God Class (related to a stinky refactoring) and suggest her to move the class member to a more appropriate class.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous referees for their valuable comments and helpful suggestions. This work is funded by CAPES/Procad (175956, 117875), CNPq (309884/2012-8, 483425/2013-3, 477943/2013-6, 465614/2014-0, 308380/2016-9), and FAPERJ (22520 7/2016, 102166/2013).

REFERENCES

[1] R. Arcoverde, I. Macia, A. Garcia, and A. von Staa. 2012. Automatically detecting architecturally-relevant code anomalies. In *Proceedings of the 3rd International Workshop on Recommendation Systems for Software Engineering (RSSE)*. IEEE Computer Society, Washington, DC, USA, 90–91. https://doi.org/10.1109/RSSE.2012.6233419

[2] Gabriele Bavota, Bernardino De Carluccio, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Orazio Strollo. 2012. When Does a Refactoring Induce Bugs? An Empirical Study. In *Proceedings of the 12th International Working Conference on Source Code Analysis and Manipulation (SCAM '12)*. IEEE Computer Society, Washington, DC, USA, 10. https://doi.org/10.1109/SCAM.2012.20

[3] Gabriele Bavota, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Fabio Palomba. 2015. An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software* 107 (2015), 1 – 14. https://doi.org/10.1016/j.jss.2015.05.024

[4] Diego Cedrim. 2017. Experiment data of the research. http://diegocedrim.github.io/fse-2017-data/. (2017). [Online; accessed 6-july-2017].

[5] Diego Cedrim, Leonardo da Silva Sousa, Alessandro F. Garcia, and Rohit Gheyi. 2016. Does refactoring improve software structural quality? A longitudinal study of 25 projects. In *Proceedings of the 30th Brazilian Symposium on Software Engineering*. ACM, New York, NY, USA, 73–82. https://doi.org/10.1145/2973839.2973848

[6] Diego Cedrim and Leonardo Sousa. 2017. Organic. https://github.com/diegocedrim/organic. (2017). [Online; accessed 6-july-2017].

[7] Manuele Ferreira, Eiji Barbosa, Isela Macia, Roberta Arcoverde, and Alessandro Garcia. 2014. Detecting Architecturally-relevant Code Anomalies: A Case Study of Effectiveness and Effort. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing (SAC '14)*. ACM, New York, NY, USA, 1158–1163. https://doi.org/10.1145/2554850.2555036

[8] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. 1999. *Refactoring: Improving the Design of Existing Code* (1 ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. 464 pages.

[9] Kenji Fujiwara, Kyohei Fushida, Norihiro Yoshida, and Hajimu Iida. 2013. Assessing Refactoring Instances and the Maintainability Benefits of Them from Version Archives. In *Product-Focused Software Process Improvement*. Vol. 7983. Springer Publishing Company, Incorporated, Berlin, Germany, 313–323. https://doi.org/10.1007/978-3-642-39259-7_25

[10] Alessandro Gurgel, Isela Macia, Alessandro Garcia, Arndt von Staa, Mira Mezini, Michael Eichberg, and Ralf Mitschke. 2014. Blending and Reusing Rules for Architectural Degradation Prevention. In *Proceedings of the 13th International Conference on Modularity (MODULARITY '14)*. ACM, New York, NY, USA, 61–72. https://doi.org/10.1145/2577080.2577087

[11] Mario Hozano, Alessandro Garcia, Nuno Antunes, Baldoino Fonseca, and Evandro Costa. 2017. Smells Are Sensitive to Developers!: On the Efficiency of (Un)Guided Customized Detection. In *Proceedings of the 25th International Conference on Program Comprehension (ICPC '17)*. IEEE Press, Piscataway, NJ, USA, 110–120. https://doi.org/10.1109/ICPC.2017.32

[12] Cory J. Kapser and Michael W. Godfrey. 2008. "Cloning Considered Harmful" Considered Harmful: Patterns of Cloning in Software. *Empirical Software Engineering* 13, 6 (Dec. 2008), 645–692. https://doi.org/10.1007/s10664-008-9076-6

[13] Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. 2012. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Software Engineering* 17, 3 (2012), 243–275. https://doi.org/10.1007/s10664-011-9171-y

[14] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2012. A Field Study of Refactoring Challenges and Benefits. In *Proceedings of the 20th International Symposium on the Foundations of Software Engineering (FSE '12)*. ACM, New York, NY, USA, Article 50, 11 pages. https://doi.org/10.1145/2393596.2393655

[15] Michele Lanza and Radu Marinescu. 2010. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems* (1st ed.). Springer Publishing Company, Incorporated, Berlin, Germany.

[16] Isela Macia. 2013. *On the detection of architecturally relevant code anomalies in software systems*. Ph.D. Dissertation. Pontifical Catholic University of Rio de Janeiro.

[17] Isela Macia, Roberta Arcoverde, Alessandro Garcia, Christina Chavez, and Arndt von Staa. 2012. On the Relevance of Code Anomalies for Identifying Architecture Degradation Symptoms. In *Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR '12)*. IEEE Computer Society, Washington, DC, USA, 277–286. https://doi.org/10.1109/CSMR.2012.35

[18] Isela Macia, Alessandro Garcia, Christina Chavez, and Arndt von Staa. 2013. Enhancing the Detection of Code Anomalies with Architecture-Sensitive Strategies. In *Proceedings of the 17th European Conference on Software Maintenance and Reengineering*. IEEE Computer Society, Washington, DC, USA, 177–186. https://doi.org/10.1109/CSMR.2013.27

[19] Isela Macia, Joshua Garcia, Daniel Popescu, Alessandro Garcia, Nenad Medvidovic, and Arndt von Staa. 2012. Are Automatically-detected Code Anomalies Relevant to Architectural Modularity?: An Exploratory Analysis of Evolving Systems. In *Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development (AOSD '12)*. ACM, New York, NY, USA, 167–178. https://doi.org/10.1145/2162049.2162069

[20] Isela Macia Bertran, Alessandro Garcia, and Arndt von Staa. 2011. An Exploratory Study of Code Smells in Evolving Aspect-oriented Systems. In *Proceedings of the 10th International Conference on Aspect-oriented Software Development (AOSD '11)*. ACM, New York, NY, USA, 203–214. https://doi.org/10.1145/1960275.1960300

[21] Leandra Mara, Gustavo Honorato, Francisco Dantas Medeiros, Alessandro Garcia, and Carlos Lucena. 2011. Hist-Inspect: A Tool for History-sensitive Detection of Code Smells. In *Proceedings of the 10th International Conference on Aspect-oriented Software Development Companion (AOSD '11)*. ACM, New York, NY, USA, 65–66. https://doi.org/10.1145/1960314.1960335

[22] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. 2009. How We Refactor, and How We Know It. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, Washington, DC, USA, 287–297. https://doi.org/10.1109/ICSE.2009.5070529

[23] Willian Oizumi, Alessandro Garcia, Thelma Colanzi, Manuele Ferreira, and Arndt von Staa. 2015. On the relationship of code-anomaly agglomerations and architectural problems. *Journal of Software Engineering Research and Development* 3 (2015), 11.

[24] Willian Oizumi, Alessandro Garcia, Leonardo da Silva Sousa, Bruno Cafeo, and Yixue Zhao. 2016. Code Anomalies Flock Together: Exploring Code Anomaly Agglomerations for Locating Design Problems. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 440–451. https://doi.org/10.1145/2884781.2884868

[25] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrea De Lucia. 2014. Do They Really Smell Bad? A Study on Developers' Perception of Bad Code Smells. In *Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution*. IEEE Computer Society, Washington, DC, USA, 101–110. https://doi.org/10.1109/ICSME.2014.32

[26] Jacek Ratzinger, Thomas Sigmund, and Harald C. Gall. 2008. On the Relation of Refactorings and Software Defect Prediction. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories (MSR '08)*. ACM, New York, NY, USA, 35–38. https://doi.org/10.1145/1370750.1370759

[27] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. 2016. Why We Refactor? Confessions of GitHub Contributors. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 858–870. https://doi.org/10.1145/2950290.2950305

[28] Gustavo Soares, Rohit Gheyi, and Tiago Massoni. 2013. Automated Behavioral Testing of Refactoring Engines. *IEEE Transactions on Software Engineering* 39, 2 (Feb. 2013), 147–162. https://doi.org/10.1109/TSE.2012.19

[29] Konstantinos Stroggylos and Diomidis Spinellis. 2007. Refactoring–Does It Improve Software Quality?. In *Proceedings of the 5th International Workshop on Software Quality (WoSQ '07)*. IEEE Computer Society, Washington, DC, USA, 10–. https://doi.org/10.1109/WOSQ.2007.11

[30] Nikolaos Tsantalis. 2017. RefactoringMiner github page. https://github.com/tsantalis/RefactoringMiner. (2017). [Online; accessed 6-july-2017; version 0.2.0].

[31] Nikolaos Tsantalis, Victor Guana, Eleni Stroulia, and Abram Hindle. 2013. A Multidimensional Empirical Study on Refactoring Activity. In *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research (CASCON '13)*. IBM Corp., Riverton, NJ, USA, 132–146. http://dl.acm.org/citation.cfm?id=2555523.2555539

[32] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. 2015. When and Why Your Code Starts to Smell Bad. In *Proceedings of the 37th International Conference on Software Engineering (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 403–414. http://dl.acm.org/citation.cfm?id=2818754.2818805

[33] Zhenchang Xing and Eleni Stroulia. 2005. UMLDiff: An Algorithm for Object-oriented Design Differencing. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE '05)*. ACM, New York, NY, USA, 54–65. https://doi.org/10.1145/1101908.1101919

[34] Aiko Yamashita. 2014. Assessing the Capability of Code Smells to Explain Maintenance Problems: An Empirical Study Combining Quantitative and Qualitative Data. *Empirical Software Engineering* 19, 4 (Aug. 2014), 1111–1143. https://doi.org/10.1007/s10664-013-9250-3

[35] Aiko Yamashita and Steve Counsell. 2013. Code smells as system-level indicators of maintainability: An empirical study. *Journal of Systems and Software* 86, 10 (2013), 2639 – 2653. https://doi.org/10.1016/j.jss.2013.05.007

[36] Aiko Yamashita and Leon Moonen. 2013. Do developers care about code smells? An exploratory survey.. In *Proceedings of the 20th Working Conference on Reverse Engineering*, Ralf Lämmel, Rocco Oliveto, and Romain Robbes (Eds.). IEEE Computer Society, Washington, DC, USA, 242–251.

[37] Aiko Yamashita and Leon Moonen. 2013. Do developers care about code smells? An exploratory survey. In *Proceedings of the 20th Working Conference on Reverse Engineering*. IEEE Computer Society, Washington, DC, USA, 242–251. https://doi.org/10.1109/WCRE.2013.6671299

[38] Aiko Yamashita and Leon Moonen. 2013. Exploring the Impact of Inter-smell Relations on Software Maintainability: An Empirical Study. In *Proceedings of the 35th International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 682–691. http://dl.acm.org/citation.cfm?id=2486788.2486878

[39] Aiko Yamashita and Leon Moonen. 2013. To What Extent Can Maintenance Problems Be Predicted by Code Smell Detection? - An Empirical Study. *Information and Software Technology* 55, 12 (Dec. 2013), 2223–2242.