Does refactoring improve software structural quality? A longitudinal study of 25 projects

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

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

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

ABSTRACT

Code smells in a program represent indications of structural quality problems, which can be addressed by software refactoring. Refactoring is widely practiced by developers, and considerable development effort has been invested in refactoring tooling support. 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 successfully remove code smells through refactoring. This paper reports an empirical study intended to address this gap. We analyze how often refactorings affect the density of code 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, 2,506 refactorings (95.1%) did not reduce code smells. Moreover, refactorings tended to induce new smells as often as removing existing smells. These findings suggest that practitioners should be careful when performing refactoring edits. In fact, several smells induced by refactoring edits tended to live long, and were only eventually removed when smelly elements became more complex and costly to get rid of.

Categories and Subject Descriptors

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

Keywords

Refactoring, Code Smells, Structural Quality

1. INTRODUCTION

Refactoring is a common practice employed by practitioners along software evolution [19, 26]. In order to satisfy evolving requirements, developers often need to improve the software structure. Refactoring is a popular procedure for improving the internal structure of software systems [3, 5]. Indeed, case studies suggest that refactoring is often used to improve the structural software quality [3, 19]. However, even though refactoring aims at improving code structure, this expectation might not be met in real settings as refactoring changes are often performed manually [7, 19].

A code smell is a structural problem in the program [5], and examples include *God Class*, *Feature Envy* and *Long Method*. It is expected that refactorings reduce the occurrence of code smells in a program [5]. However, previous work has not challenged this common assumption. Yamashita and colleagues [31] focused only on analyzing whether code smells reflect actual maintenance problems and which code smells should be removed first by refactoring. In a similar vein, Liu *et al.* [14] developed a study to recommend code smells that should be prioritized through refactoring edits. However, there is little understanding on how refactoring interferes on the prevalence of code smells, i.e, whether refactorings remove (or not) smelly elements of a program. As a consequence, practitioners remain unaware about the effectiveness of their refactoring edits.

In order to address this gap, we conducted a study that addresses the following research question: Does refactoring improve the software structural quality? We answered this research question by analyzing the relationship of refactorings and code smells through the version histories of 25 open source projects. We studied this relationship by classifying each refactoring instance according to its interference on the code smells located in the refactored elements. Let n_{bf} be the number of existing code smells before a refactoring and n_{af} the number of instances after. In our study, a given refactoring is classified in one of three cases: (i) positive if $n_{af} < n_{bf}$; (ii) negative if $n_{af} > n_{bf}$; or (iii) neutral if $n_{af} = n_{bf}$. This classification is used to understand whether refactorings tend to improve (or not) the internal structure of a program.

Our findings are based on the analysis of 2,635 refactorings distributed in 11 refactoring types. Surprisingly, 2,506 refactorings (95.1%) are neutral ones, i.e., they did not reduce the density of code smells. Moreover, refactorings tended to induce new smells as often as removing existing ones. We found that negative refactorings occur as frequently as positive ones: 2.66% and 2.24%, respectively. These findings shed light on how refactorings are applied in real settings and suggest that practitioners should be more careful when performing refactoring edits. In fact, several smells induced by refactoring edits tended to live long, and were only eventually removed when smelly elements became more complex and costly to get rid of. Developers might consider detecting reminiscent smells earlier as several critical smells tended to

ICSE 2016 Austin, Texas USA

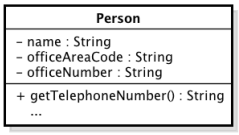

Figure 1: Person class with God Class code smell

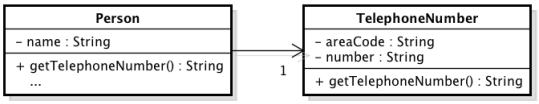

Figure 2: Person class after Extract Class refactoring without God Class

last for several months in their projects.

This work is organized as follows: Section 2 provides basic concepts and highlights the motivation for our study. Section 3 presents the study planning. Section 4 provides the collected data, whereas Section 5 presents the discussion of findings and some recommendations to practitioners. Section 6 describes our effort on mitigating threats to validity. Section 7 relates our study with previous work. Section 8 concludes the study and outlines future directions.

2. CONCEPTS AND MOTIVATION

This section introduces some basic concepts and motivating examples (Section 2.1). Section 2.2 describes a method to classify each refactoring instance in terms of its interference in code smells.

2.1 Motivating Examples

A code smell is a surface indication that usually corresponds to a deeper structural problem in the system [5]. There are different types of code smells documented in the literature [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, Figures 1 and 2 show the *Person* class in two different forms. In Figure 1, the *Person* class 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* (Figure 2). After this transformation, the program no longer has this *God Class* and still realize the same functionality. The version of the program with a code smell in Figure 1 has inferior structural quality as compared to the smell-free version in Figure 2.

Projects with inferior structural quality tend to be hard to read and maintain. Therefore, whenever a code smell is identified in a program, software refactoring should be applied to remove it [5]. A key claim associated with refactoring benefits is its ability to remove smells and, therefore, improve software structural quality. In fact, case studies suggest that refactoring is a common practice adopted by practitioners [19, 30] and often targeted at improving software structure [3, 12, 18, 34].

Let's consider the Figures 1 and 2 again. As mentioned before, the module *Person* in Figure 1 has a *God Class* code smell. This single class is accumulating responsibilities that should be fulfilled, instead, by two classes at least. In order to address this smell, the *Extract Class* refactoring should be performed, i.e., a programmer should create a new class and move the relevant fields and methods from the old class into the new one. The refactoring, in this case, would remove the *God Class* smell. 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.

2.2 Refactoring Classification

Refactorings may interfere positively or negatively in the existence of code smells. This section presents a scheme to classify each refactoring instance by computing the number of code smells introduced or removed along the refactoring changes. Let $S = \{s_1, s_2, \cdots, s_n\}$ be the set of software projects to be analyzed in our study. Each software s has a set of versions $V(s) = \{v_1, v_2, \cdots, v_m\}$. Each version v_i has a set of elements $E(v_i) = \{e_1, e_2, \cdots\}$ representing all methods, classes and fields belonging to it. In Figures 1 and 2, the set $S = \{PhoneBook\}$ is composed just by one software. This software *PhoneBook* has two versions $V(PhoneBook) = \{v_a, v_b\}$ represented by Figures 1 and 2, respectively. 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 give to us all refactoring activities detected in a pair of software versions. If we apply the R function in the *PhoneBook* software, the result would be:

$$R(v_a, v_b) = \{r_1(\text{Extract Class}, e')\}$$

$$\text{where } e' = \{Person, TelephoneNumber\}$$

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 a *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, lets 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.

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 will only analyze 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:

$$CS_{b[r1]}(\{Person\}) = \{\text{God Class}\}$$

$$CS_{a[r1]}(\{Person, TelephoneNumber\}) = \emptyset$$

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 a positive refactoring. In other way, if $x < y$, r increased the number of code smells on e and, because of that, r is a positive refactoring. When $x = y$, r is a neutral refactoring.

Independent of its type, a code smell should be addressed by refactorings during software evolution. In other words, any instance of code smell demands refactorings to be removed. The classification is based on this principle and, because of that, only the number of code smells is used to classify a refactoring. For instance, lets imagine a class A containing a *Long Method* code smell. In order to address this smell, a developer used a *Extract Method* and removed the *Long Method*. On the other hand, both new methods caused a new instance of code smell: *Divergent Change*. According to the proposed refactoring classification, this is a neutral refactoring. This classification does not make distinction among code smell types since all types eventually should be addressed by a refactoring within the same change or along further changes. Despite of solving one problem, the developer created another one and the development team will work in the same piece of code again in the future.

Revisiting the aforementioned system, we know the number of code smells was reduced, i.e., $|CS_{b[r1]}(e)|= 1$ and $|CS_{a[r1]}(e)|= 0$. The r_1 refactoring reduced the number of code smells present in {*Person, TelephoneNumber*} element set. Therefore, r_1 is classified as a positive refactoring. By observing the refactoring classification in a software set S, it is possible to quantify how frequent each type of refactoring is positive, negative or neutral in these software versions.

3. STUDY PLANNING

This section presents the study planning. Section 3.1 presents our research question. In order to address this question, the study is structured into four phases: (i) identification of software projects to be analyzed; (ii) detection of refactoring and code smell instances; (iii) refactoring classification, and (iv) validation of study findings. Sections 3.2.1-3.2.4 describe each of these phases.

3.1 Research Question

Software refactoring is likely to interfere in the presence of code smells (Section 2.1). As code smells provide the motivation to perform refactoring activities [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 code 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 this research question by analyzing refactorings performed in real projects. This procedure enables 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 changing the structural quality.

3.2 Study Phases

This section describes the four phases of our study.

3.2.1 Phase 1: Selection of Software Projects

The first step of this study is to choose a software set S to be analyzed. We focused our analysis on open source projects so that our study could be easily replicated and extended. As GitHub is the world's largest open source community, it was used as the source of software projects selected for the study. We chose GitHub projects that match the following criteria: (i) projects with reasonable number of contributions (at least 100 pushes); (ii) projects written in Java; and (iii) projects of different domains. The first criterion ensures that projects analyzed has long-enough version histories. The Java language was chosen because previous studies (e.g. [15, 16]) have already validated rules for detecting code smells.

The entire GitHub archive is available as a public dataset on Google BigQuery [8]. This Google project provides a way to write SQL like queries to find projects satisfying the aforementioned criteria. The following data about each project were collected from BigQuery: name, number of pushes, description and GitHub URL. The target software set S chosen is composed of 25 projects belonging to this query result. The full list of selected projects and their descriptions are available at the study website [4].

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}. The second phase also encompass the detection of all code smells in each version $v_i \in V(s)$. These activities are described in the following.

Refactoring Detection. The refactoring detection primarily relied on the use of two tools to analyze pairs of software versions v_i and v_{i+1} and generate all refactorings $R(v_i, v_{i+1})$ as output. In other words, these tools imple-

ment the R function described in Section 2.2. The first tool selected was Ref-Finder [11]. This tool is an Eclipse plug-in that identifies refactorings using a template-based reconstruction technique. It expresses each refactoring type in terms of template logic queries and uses a logic programming engine to infer concrete refactoring instances [11]. The selection of Ref-Finder is justified by the fact it currently supports 63 refactoring types described in Fowler's catalog [11]. Ref-Finder seemed to be, in principle, a good candidate to support refactoring detection in our study. However, we struggled with some practical problems using this tool: (i) the precision of Ref-Finder was as low as 15% for most of the refactoring types (as observed in our validation phase), causing a considerable number of false positives; and (ii) the detection algorithm was inefficient when executed in a large dataset.

Because of these problems, Ref-Finder was discarded and another tool was selected in this phase. Ref-Detector [1] [27] was the tool chosen to support the detection of refactoring instances. This tool implements a lightweight version of UMLDiff [29] algorithm for differencing object-oriented models. This tool was way more efficient than Ref-Finder since it was designed to be executed in a large dataset. In addition, its precision is 96.4% (as observed in our validation phase), leading to a very low rate of false positives. The only drawback of this tool is the number of refactoring types detected. While Ref-Finder detects 63 types of refactoring, Ref-Detector detects 11 types. Fortunately, these 11 types were amongst the ones reported by Murphy-Hill as the most common refactoring types [19]. Ref-Detector 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.

Code Smell Detection. In order to complete the data collection procedure, we need to identify code smells present in each e_i returned in R tuples. Code smells are often detected with rule-based strategies [2, 21]. Each rule-based 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 [13, 15] 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 [15, 16]. These rules were used due to two main reasons: (i) they represent refinements of well-known rules proposed by Lanza et al. [13], which are well documented and used in previous studies [9] [17] [32]; and (ii) and they have, on average, precision of 0.72 and recall of 0.81.

These rules detect five types of code smells: *God Class*, *Long Method*, *Feature Envy*, *Shotgun Surgery* and *Divergent Change*. The selection of these code smell types was due to two main reasons: (i) these code smells are very common and tend to related to design degradation symptoms [15]; and (ii) they subsume the occurrence of other code smells. For instance, a class labeled as *God Class* tends to be more problematic than classes with other simpler smells [15]. A *God Class* is often a source of smells, such as *Duplicated Code*. It is also an indicator of major design problems, such as *Component Concern Overload* and *Overused Interface* [16].

Another reason to select these smell types was their direct relation with the most frequent refactoring types. Murphy-Hill *et al.* [19] reported the refactorings often performed by developers and we analyzed which code smell types these refactorings are intended to remove according to Fowler's catalog [5]. *Move Method* can contribute to the removal of *God Class* and *Feature Envy* smells. *Shotgun Surgery* smells can be removed by *Move Method* and *Move Field*. *Extract Method* is primarily targeted at removing *Long Method* smells. Although *Extract Class/Superclass* are not too common, they can be also used to remove *Divergent Change*.

Table 1 shows the rules used to identify code smells. 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 [23] with non-commercial license. This software was chosen because it collects all metrics values needed by the rules applied.

Table 1: 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 $>\varpi$)

Different thresholds can lead to results completely distinct. Therefore, choices of thresholds can pose a threat to this study. Thus, two sets of thresholds were used in order to mitigate this threat. The first set, as known as *pessimistic*, represent the thresholds previously validated in the study by *Macia et al.* [15]. We quality this strategy as *pessimistic* because it relies on the use of high thresholds values aiming to detect only critical code smells across the projects. The second strategy, named as *optimistic*, uses relaxed thresholds aiming to detect as many code smells as possible. These optimistic thresholds represent lower bound thresholds, which were tuned through our validation phase. These two strategies allowed to derive refactoring classifications based on two different sets of code smells.

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. The three initial phases of our study are expressed as a pseudocode in Algorithm 1. The first line of this algorithm represents the first phase: software set definition. The lines 4 and 6 comprise the detection of refactoring and code smells, i.e. the second phase. The lines 7-18 are responsible for the third phase: refactoring classification.

[1] available at https://github.com/tsantalis/RefDetector

Algorithm 1 First three phases of the research

Require: A set of projects S
Ensure: A set of classified refactorings
1: $S \leftarrow \{s_1, s_2, \cdots, s_n\}$ {Phase 1}
2: **for all** $s \in S$ **do**
3: **for all** $v_i \in V(s)$ **do**
4: Collect $CS(e)$ for all $e \in E(v_i)$ {Phase 2}
5: **if** $v_{i+1} \in V(s)$ **then**
6: Collect all refactorings $R(v_i, v_{i+1})$ {Phase 2}
7: **for all** $r \in R(v_i, v_{i+1})$ **do**
8: $e \leftarrow r[1]$ //r[1] is the refactored element e
9: $Before \leftarrow |CS_{b[r]}(e)|$
10: $After \leftarrow |CS_{a[r]}(e)|$
11: Classify r as neutral {Phase 3}
12: **if** $After < Before$ **then**
13: Classify r as positive
14: **end if**
15: **if** $After > Before$ **then**
16: Classify r as negative
17: **end if**
18: **end for**
19: **end if**
20: **end for**
21: **end for**

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 [27]. The algorithm always compares subsequent versions of the projects. Lets suppose a project that has only 3 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.

During the first three phases, all 25 software projects were analyzed, totaling 8,274 versions. Considering all Java files belonging to all versions, we analyzed 2,320,953 classes during refactorings and code smells detection. The number of detected and classified refactorings was 2,635, whereas 736,194 code smells instances were obtained.

3.2.4 Phase 4: 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 similar to Algorithm 1 was executed in a smaller dataset. A manual validation of each output was made in this phase to ensure the reliability of the output.

Four systems were used during validation phase: OODT, Health Watcher, Mobile Media and OpenConext-cruncher (OC). Health Watcher [25] is a web system for public to register several kinds of complaints, improving the quality of the services provided by health care institutions. Mobile Media is a software product line created to manage photos, music, and videos on mobile devices [33]. The third system is Apache OODT (Object Oriented Data Technology), which is a software framework prompted by NASA for distributing and managing scientific data. The fourth project (OC) was one of the systems used during data collection phase.

The same data collection process presented in Algorithm 1 was used in this reduced software set. It is possible to run a manual analysis on the obtained results since only four projects were used in this phase. Each detected refactoring was evaluated to check whether the result is a false positive or not. Each detected refactoring was compared to Fowler's catalog in order to ensure if it is, in fact, a refactoring and not a misclassified code transformation. This manual analysis increases the level of confidence about the refactoring data collected in this phase.

The verification performed in this phase is useful to validate the results generated during the third phase. Results of both phases were compared in order to detect interference of false positives in data collection phase. If the results of both phases are too different, it may indicate that the automatic extraction strategy for the data collection phase comprises a significant number of false positives, which can invalidate our results.

4. DATA PRESENTATION

This section presents the study results. We present the results in terms of: (i) the frequency of occurrences of each refactoring type, (ii) the frequency of occurrences of each code smell type, and (iii) the overall classification (Section 3.2) of refactoring occurrences found in the projects. The data presented in this section is used to both elucidate the answer of the research question as well as deriving lessons learned and recommendations to practitioners (Section 5). Except when explicitly mentioned otherwise, all data concerning detected code smells are related to the use of *pessimistic* thresholds (Section 3.2.2).

4.1 Smell-Affecting Refactorings

The set of instances of refactorings and code smells in our data set covered a wide range of refactoring types and smell types. All refactoring types detected by Ref-Detector were considered in this study, i.e., 11 types of Fowler's catalog. The refactoring detection procedure identified 2,635 refactorings distributed in 7 different types. Table 2 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 [19]. In addition, we also observe several occurrences frequency of *Move Field*, *Move Method* and *Pull up field* refactorings.

Although the definition of *Rename Method* is not directly related to one of the code smells addressed in our study (Table 3), we still used its instances during the refactoring classification phase. The reason is manifold. First, according to Antoniol *et al.* [1], program elements with unstable structural quality tend to also suffer changes in their identifiers. Second, structural change tends to occur in conjunction with *Rename* refactorings [1]. Changes of method and class identifiers seem to be often motivated by structural problems in their implementation [1]. For instance, a method with a *Feature Envy* is likely to have, soon or later, its name changed after refactoring since the method no longer fulfills some of its original responsibilities (before refactoring).

Table 2: Most common refactorings types collected

Refactoring Type	Occurences	Related to Smelly Elements	Neutral	Negative	Positive
Rename method	1,491	22 (1%)	1,486	4	1
Move field	879	726 (82%)	817	32	30
Move method	209	168 (80%)	151	33	25
Pull up field	29	21 (72%)	28	0	1
Pull up method	21	14 (66%)	20	1	0
Push down method	5	2 (40%)	3	0	2
Push down field	1	1 (100%)	1	0	0
Extract method	0	0	0	0	0
Inline method	0	0	0	0	0
Extract superclass	0	0	0	0	0
Extract interface	0	0	0	0	0

In any case, we compute the occurrences of positive, negative and neutral refactorings for each type of refactoring (Table 2). The goal is to understand the variation of the frequency of positive, negative and neutral refactorings across refactoring types. We computed how frequently each type of refactoring was applied in program elements hosting at least one code smell, i.e., refactorings applied to elements that are not free of code smells either before or after refactoring. This factor is captured in the third column (*Related to Smelly Elements*), which shows how many refactorings are related to smelly elements. The following columns present the refactoring classification according to the criteria defined in Section 2.2.

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. Except for the *Rename Method*, from 40% to 82% of the refactorings were applied to smelly elements of a program. These frequencies were very similar in the data gathered with *optimistic* thresholds [4]. Surprisingly, the neutral classification was by far the most frequent one for all refactoring types. In other words, even though refactorings often target smelly elements, they often do not reduce the smells present in those elements after refactoring edits. Moreover, negative refactorings occurred as often as positive refatorings (Section 4.2).

Table 3: Code smells in refactored elements

Code Smell Type	Occurences
God Class	3,279 (86%)
Shotgun Surgery	351 (9%)
Long Method	110 (3%)
Feature Envy	50 (1%)
Divergent Change	40 (1%)

In our study, the relevant code smells are those affected by refactorings, i.e., those code smells located in program elements touched by refactorings. In this context, a total of 3,830 relevant instances of code smells were collected and analyzed in pursuance of answer to our research question. In this study, the most common type of code smell was *God Class* followed by *Shotgun Surgery* and *Long Method*. Table 3 presents all code smells data collected in this study. As we can see, *God Class* represents 86% of all code smells detected. These results represent an interesting finding as previous studies have been found that *God Classes* and *Long Methods* are often related to: (i) severe symptoms of design degradation [15], and (ii) major maintenance effort involving the affected code elements in later versions of the program [24].

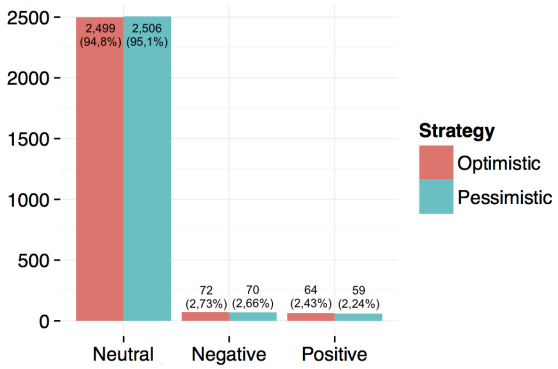

Figure 3: Results of the Data Collection Phase

4.2 Classification Results

We focus now on the data concerning the correlation between code smells and refactorings. All refactorings were labeled according to the classification presented in Section 2. As mentioned in the previous section, the most common classification detected in our data set was the neutral one. The vast majority of refactorings did not change the number of code smells (95.1% of the times). Negative refactorings represent 2.66% of the cases. Finally, positive refactorings represent 2.24% of the cases. Therefore, observing the dataset, refactorings tend to maintain unaltered the density of code smells. On the other hand, when refactorings affect smelly elements of a program, they slightly introduce more smells rather than reduce them. As mentioned in Section 3.2.2, we performed the same classification procedure using the *optimistic* threshold set. As presented in Figure 3, both *pessimistic* and *optimistic* classification procedures presented very similar results. Thus, it is possible to claim that even with different strategies for detecting code smells, the rates of positive, negative and neutral refactorings tend to be similar.

Moreover, in order to further improve confidence on our results and make sure they were not influenced by other particularities of the employed smell detection strategies, we also analyzed the refactoring classification results in our

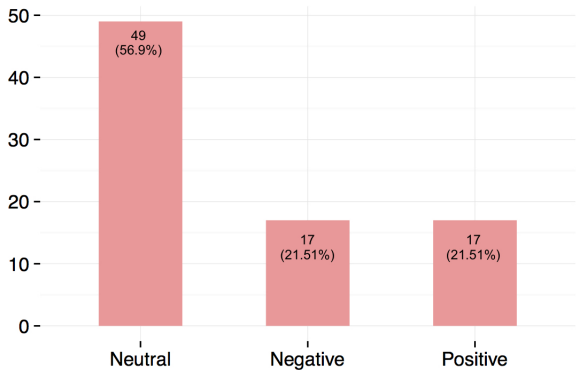

Figure 4: Results of the Validation Phase

validated data set. As explained in Section 3.2.4, all results of the validation phase were analyzed individually in order to remove any spurious data. The results presented in Figure 4 were obtained in the validation phase. Overall, the results in both Figures 3 and 4 follow similar trends, with the following noticeable changes: (i) reduction in the percentage of neutral refactorings, and (ii) increases in the percentages of negative and positive refactorings. In spite of these differences, the most common classification in the validation phase still was the neutral one with 56.9% of all occurrences. Negative and positive refactorings represent each 21.51% of the cases.

5. FINDINGS AND LESSONS LEARNED

This section presents our main findings and lessons learned. We also discuss how such findings and lessons learned can guide developers to improve their software refactoring practices.

5.1 Refactoring and Structural Quality

This section explicitly focuses on addressing our research question: does refactoring improve software structural quality? In order to answer this question, all detected refactorings were classified twice as presented in the previous section. The main classification concerns the code smells detected using the *pessimistic* threshold set, while the secondary classification relies on the *optimistic* one. Figure 3 shows the general proportion of neutral, positive and negative refactorings using both 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 *pessimistic* or *optimistic* thresholds, support 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 2). Surprisingly, these results also prevail if we only consider refactoring types 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 2 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. Moreover, this behavior was also frequently observed in the dataset obtained in the validation phase (Section 4.2).

5.2 Developers Should Be Careful

Our results indicate developers often do not improve the software structural quality during refactoring. Even worse, negative refactorings seem to be as frequent as the positive ones (Section 4.2). These results imply that there is a risk of degrading the software structural quality while developers perform their refactoring edits. Whenever refactoring edits induce new code smells, it means that developers may be require to invest additional maintenance effort in the future in order to remove such smells. Developers should avoid the introduction of new code smells while refactoring.

We analyzed the subset of refactorings directly associated with code smells, i.e., refactorings that started in smelly methods or classes. Our results contain 954 refactorings satisfying this criterion. Interestingly, we observed 853 neutral refactorings, 42 negative ones and 59 positive ones. These 42 negative refactorings represent situations where developers potentially tried to refactor a smelly element, but, after the refactoring edits, the code became even smellier. There were 3 projects – namely *Confetti*, *Ghprb-pluggin* and *Jara* – where the rate of negative refactorings was higher than 10%. Moreover, we observed that the rate of negative refactorings observed in the validation phase was even higher: more than from 20% of the refactorings were negative ones. This high rate was also consistent in the 4 projects analyzed in the validation phase.

While refactoring, developers are supposed to improve the software structural quality. However, the aforementioned findings suggest developers should be more careful while refactoring. Unfortunately, for different reasons, they have not been able to reap certain intended benefits of software refactoring. Therefore, developers should be equipped with means to improve their awareness about the interference of refactoring edits on the structural quality of their programs. For instance, developers should leverage the benefits of continuous integration in order to promptly observing possible quality side effects of their refactoring edits.

5.3 Code Smells Tend to Live Long

Section 5.1 focused on analyzing if code smells are reduced immediately after refactoring edits are applied. In other words, only what happened between two successive commits is considered during our refactoring classification procedure. However, it might be that developers fully remove a code smell in the long run, i.e. after several commits are applied. For instance, lets consider a refactoring r classified as negative. Suppose this refactoring occurred between the versions v_1 and v_2. It means that at least one code smell instance cs was introduced between these two versions. Lets also suppose that cs was removed in version v_3. A valid assumption in this scenario would be: the developer was removing this code smell between versions v_1 and v_3 and, therefore, our original classification does not capture this case.

Therefore, we executed another procedure in order to ob-

serve how long the code smells lived in the analyzed projects. For each code smell instance collected, we computed its seminal and final commit, i.e., the commits in which the code smell was introduced and removed, respectively. Therefore, we were able to compute the average life of each code smell in number of commits. The findings are presented in Table 4 in terms of each smell type. Overall, the average life of all code smells collected is 48.51 commits.

Table 4: Average life (in commits) for each code smell type

Code Smell	Average Life
Long Method	51.93
Shotgun Surgery	51.90
God Class	45.73
Divergent Change	43.14
Feature Envy	37.11

We also analyzed the life of each code smell introduced along with a refactoring editing. Through this analysis we were able to compute how frequently the smells induced by negative refactorings were removed afterwards. The results show that 76 code smells were introduced along with refactoring edits. Only 4 cases were removed afterwards, i.e., code smells introduced by refactoring live much longer than 1 commit. More precisely, they live, on average, 44 commits. Considering that in our dataset, on average, one commit is performed every 3.26 days, the average life of code smells introduced by refactoring is 146 days. These numbers reinforce the lesson that developers should be careful when refactoring.

5.4 Refactor Early

The longer a class remains in a program, the greater tends be its number of dependencies to other classes. A class with many associations is harder to refactor than a class with just few ones. In this way, the longer a smelly class remains in the code, it is likely to become harder will the removal of its smell through refactoring. As presented before, the anomalies introduced during the development have an average life of 48.51 commits (approximately 158 days). If a refactoring is performed right after the code smell introduction, the maintenance effort is reduced.

Lets look to the particular case of the class *AbstractInitiator* contained in *Ikasan* project. We analyzed 1,739 versions of this project and this class was classified as *God Class* between versions 361 and 1015. According to Table 1, a given class can be classified as *God Class* if the values of *LOC* and *CBO* metrics are higher than a specific threshold. Figure 5 presents the values of *LOC* metric for *AbstractInitiator* class between all aforementioned versions. This graphic also presents the trend line of the data (blue dashed line). In a similar vein, Figure 6 presents the values of *CBO* metric. The values of both metrics tend to increase over the time, indicating this smell was becoming increasingly harder to fix as the project progressed. This *God Class* lived in this project between August 21, 2009 and March 14, 2012. In the following version (1016), the *AbstractInitiator* was fully removed from the project. However, the cost of removing it in 2012 was significantly higher than if it was removed early in 2009.

We observed that the degrading behavior observed in *AbstractInitiator* case repeated across multiple projects, i.e., several code smells tend to be worse over time. This observation reinforces the idea of better equipping developers with early refactoring support. The longer the problem lives, the worse it may become. However, several developers may simply be unaware of such side effects. Simple code smells can escalate to bigger problems in the future. According to Macia *et al.* [15], *God Class* and *Long Method* smells are more likely to be related to architectural problems. It means that detecting and removing these code smells early is critical to avoid high level design problems in the future.

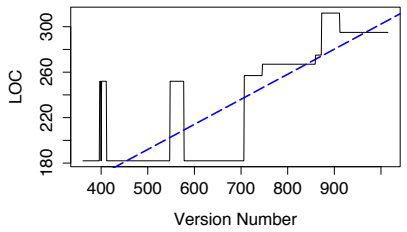

Figure 5: LOC values of the AbstractInitiator class

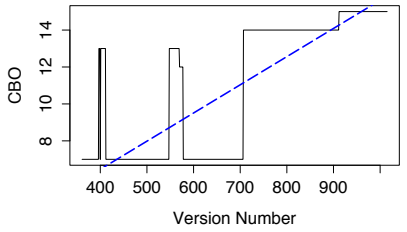

Figure 6: CBO values of the AbstractInitiator class

6. THREATS TO VALIDITY

As several empirical studies involving software, the lack of representative results is also a possible threat to the external validity. For example, the number and domains of programs used in the study may not be enough to generalize the results. In order to further reducing the impact of this threat, we considered a large dataset with programs from a wide range of domains. Nevertheless, all the programs used in the study constitute both programs widely used in other empirical studies and popular Java projects on GitHub.

Threats to internal validity of this work concern the data collection procedure. Macia *et al.* technique uses a group of predefined detection strategies (detection rules) [15], which was applied to identify code smells in our dataset. The same group of rules was used in all five programs of their study. However, in their work, Macia *et al.* have tuned the thresholds of each smell detection rule to each project. Since we used a larger dataset, it is not possible to consult developers and to adjust all thresholds to each one of the projects. In order to mitigate the possible impact of this threat, we have tuned the thresholds for the projects in the systems analyzed in the validation phase. Then, as far as code smells are concerned, we only reported findings that we consistently observed in both the larger and the validated dataset.

The number of code smell types is another threat. As mentioned before, the 5 code smell types are very common

[15] and subsume the occurrence of other code smells. Even so, if we had considered other smell types, the results could have been different. In this way, the selection of considered code smells may have been biased the results. Therefore, to mitigate this threat more code smells types should be used in future studies.

We do not consider the developers' intention during refactoring. We consider that every transformation detected as refactoring was, in fact, an intention of refactoring. This can have biased the results in the sense that not always the developers intended to refactor. If we had known the intention of the developers during the commits, we could have discarded some refactorings of our dataset. These discarded refactorings could have changed the results of this research.

7. RELATED WORK

Fujiwara *et al.* [6] and Ratzinger *et al.* [22] investigated 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 found that an increase in refactoring has a significant positive interference on software quality. Results showed that number of software 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 code smells with refactorings. Differently from them, we focused on analyzing the interference of refactoring on structural quality of programs.

Khomh *et al.* [10] presented an exploratory study to analyze if classes with code smells are more change-prone than classes without smells. The authors first investigated whether classes with code smells have an increased likelihood of changing than other classes. After that, they studied whether classes with more smells than others are more change-prone. The authors affirm that classes with more smells exhibit higher change-proneness. Besides, they observe that code smells have a negative interference on classes and certain kinds of smells are more harmful than others. While they investigated the relations between code smells and changes, we are investigating the interference of a refactoring (a change) over code smells.

Olbrich *et al.* [20] investigated whether classes that are involved in certain code smells are liable to be changed more frequently and have more defects than other classes in the code. They focus on the influence of God Classes and Brain Classes on the frequency of defects detected. The results lead them to conclude that as long as the size of the God and Brain Classes is not "extreme", and as long as there are not many of them, the presence of God and Brain Classes may be beneficial to a software system. We cite as major differences to this work the target taken into account. The authors are only considering the class size in order to analyze only two code smells. We are analyzing the interference of refactoring on code smells.

Tufano *et al.* [28] executed a large empirical study using change history of 200 open source projects from different software ecosystems (Eclipse, Apache and Android). This study has two objectives: (i) to investigate when developers introduce code smells; (ii) and what are the circumstances and reasons behind their introduction. The authors developed a strategy in order to identify commits, which introduced code smells. Using this strategy, they mined over 0.5M commits and did manual analysis of 9,164 commits identified as smell introducing. They found smells are generally introduced during improvement of existing features or during the implementation of new ones. Our research found refactorings degrading structural quality by introducing smells in 2.66% of the observed cases. Our result relates to Tufano *et al.* results because refactorings usually are applied in order to improve the existing design. Since Tufano *et al.* stated smells are introduced during enhancement tasks, developers should be careful to not introduce smells when refactoring.

8. CONCLUSIONS

This research was conducted aiming to understand the relationship between refactorings and code smells. We analyzed 25 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 affect code smells. Second, negative refactorings at least occur as often as positive refactorings. This finding implies that whenever refactoring changes the existing smells in the target code elements, the probability of introducing a code smell is at least as high as the probability of removing a code smell.

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] G. Antoniol, Y.-G. Gueheneuc, E. Merlo, and P. Tonella. Mining the Lexicon Used by Programmers during Sofware Evolution. *Proc. of ICSM '07*, pages 14–23, 2007.

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

[3] F. Bourquin and R. Keller. High-impact refactoring based on architecture violations. In *Proc. of CSMR '07*, pages 149–158, March 2007.

[4] D. Cedrim. Experiment data of the research. `http://diegocedrim.github.io/seip-2016-data/`, 2015. [Online; accessed 23-october-2015].

[5] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1 edition 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 J. Heidrich, M. Oivo, A. Jedlitschka, and M. Baldassarre, editors, *Product-Focused Software Process Improvement*,

volume 7983 of *Lecture Notes in Computer Science*, pages 313–323. Springer Berlin Heidelberg, 2013.

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

[8] Google. Google BigQuery Cloud Platform. `https://cloud.google.com/bigquery/`, 2015. [Online; accessed 18-january-2015].

[9] Y. Guo, C. Seaman, N. Zazworka, and F. Shull. Domain-specific tailoring of code smells: An empirical study. In *Proc. of ICSE '10*, pages 167–170, New York, NY, USA, 2010. ACM.

[10] F. Khomh, M. Di Penta, and Y.-G. Gueheneuc. An Exploratory Study of the Impact of Code Smells on Software Change-proneness. *Proc. of WCRE '09*, pages 75–84, 2009.

[11] 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*, page 371, New York, New York, USA, 2010. ACM Press.

[12] R. Kolb, D. Muthig, T. Patzke, and K. Yamauchi. A case study in refactoring a legacy component for reuse in a product line. In *Proc. of ICSM'05*, pages 369–378. IEEE, 2005.

[13] M. Lanza, R. Marinescu, and S. Ducasse. *Object-Oriented Metrics in Practice*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[14] H. Liu, L. Yang, Z. Niu, Z. Ma, and W. Shao. Facilitating software refactoring with appropriate resolution order of bad smells. *Proc. of FSE 2009*, page 265, 2009.

[15] 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, Mar. 2012.

[16] 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, March 2013.

[17] 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, New York, NY, USA, 2011. ACM.

[18] P. Meananeatra. Identifying refactoring sequences for improving software maintainability. In *Proc. of ASE '12*, page 406, New York, New York, USA, 2012. ACM Press.

[19] 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.

[20] S. M. Olbrich, D. S. Cruzes, and D. I. Sjoberg. Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems. In *Proc. of ICSM '10*, pages 1–10. IEEE, Sept. 2010.

[21] I. Polasek, S. Snopko, and I. Kapustik. Automatic identification of the anti-patterns using the rule-based approach. In *Proc. of SISY '12*, pages 283–286. IEEE, Sept. 2012.

[22] J. Ratzinger, T. Sigmund, and H. C. Gall. On the relation of refactorings and software defect prediction. In *Proc. of MSR '08*, page 35, New York, New York, USA, 2008. ACM Press.

[23] Scitools. Understand Software. `https://scitools.com`, 2015. [Online; accessed 15-january-2015].

[24] D. I. K. Sjoberg, A. Yamashita, B. C. D. Anda, A. Mockus, and T. Dyba. Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering*, 39(8):1144–1156, 2013.

[25] S. Soares, E. Laureano, and P. Borba. Implementing distribution and persistence aspects with aspectj. In *Proc. of the 17th ACM SIGPLAN*, OOPSLA '02, pages 174–190, New York, NY, USA, 2002. ACM.

[26] K. Stroggylos and D. Spinellis. Refactoring–does it improve software quality? In *Proc. of WoSQ '07*, Washington, DC, USA, 2007. IEEE Computer Society.

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

[28] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. D. Penta, A. D. Lucia, and D. Poshyvanyk. When and why your code starts to smell bad. In *Proc. of ICSE '15*. IEEE, 2015.

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

[30] Z. Xing and E. Stroulia. Refactoring Practice: How it is and How it Should be Supported - An Eclipse Case Study. In *Proc. of ICSM '06*, pages 458–468. IEEE, Sept. 2006.

[31] 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, Mar. 2013.

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

[33] T. J. Young. Using aspectj to build a software product line for mobile devices. msc dissertation. In *University of British Columbia, Department of Computer Science*, pages 1–6, 2005.

[34] M. Zhang, T. Hall, and N. Baddoo. Code Bad Smells: a review of current knowledge. *Journal of Software Maintenance and Evolution: Research and Practice*, 23(3):179–202, Apr. 2011.