Recommending Composite Refactorings for Smell Removal: Heuristics and Evaluation

Willian Oizumi
woizumi@inf.puc-rio.br
PUC-Rio, Brazil

Diego Cedrim
dccedrim@amazon.com
Amazon, Brazil

Leonardo Sousa
leo.sousa@sv.cmu.edu
Carnegie Mellon University, USA

Ana C. Bibiano
abibiano@inf.puc-rio.br
PUC-Rio, Brazil

Anderson Oliveira
aoliveira@inf.puc-rio.br
PUC-Rio, Brazil

Alessandro Garcia, Daniel Oliveira
{afgarcia,doliveira}@inf.puc-rio.br
PUC-Rio, Brazil

ABSTRACT

Structural degradation is the process in which quality attributes of a system are negatively impacted. When due attention is not paid to structural degradation, the source code may also become difficult to change. Code smells are recurring structures in the source code that may represent structural degradation. Hence, there are many catalogs and techniques for supporting the removal of code smells through refactoring recommendations, which usually consist of single refactorings such as a Move Method or an Extract Method. However, single refactorings are often not enough for completely removing certain smell occurrences. Moreover, recent studies show that developers most often apply composite refactorings – *i.e.*, sequences of two or more refactorings – for removing code smells. Despite showing the importance of performing composite refactorings, most studies do not provide information on which composite refactoring patterns are recurrent in practice. In this context, a previous study identified 35 smell removal patterns that are frequent across multiple open source systems. However, such study has not explored how the removal patterns could help developers to apply effective composite refactorings. Thus, in this work, we propose a suite of new recommendation heuristics to help developers in applying effective composite refactorings. These heuristics are intended to remove three code smell types, namely Complex Class, Feature Envy, and God Class. After designing the heuristics, we evaluated their effectiveness through a quasi-experiment. This evaluation was conducted with 12 software developers and 9 smelly Java classes. Results indicate that developers considered our heuristics effective or partially effective in more than 93% of the cases. In addition, the evaluation helped us to identify multiple factors that contribute to the acceptance or rejection of the refactoring recommendations. Based on these factors, we defined new guidelines for the effective recommendation of smell-removal composite refactorings.

SBES '20, October 21–23, 2020, Natal, Brazil

ACM ISBN 978-1-4503-8753-8/20/09…$15.00
https://doi.org/10.1145/3422392.3422423

CCS CONCEPTS

• **Software and its engineering → Software design engineering**.

KEYWORDS

refactoring, composite refactoring, refactoring heuristics, composite refactoring patterns, code smells, refactoring recommendations

ACM Reference Format:
Willian Oizumi, Diego Cedrim, Leonardo Sousa, Ana C. Bibiano, Anderson Oliveira, and Alessandro Garcia, Daniel Oliveira. 2020. Recommending Composite Refactorings for Smell Removal: Heuristics and Evaluation. In *34th Brazilian Symposium on Software Engineering (SBES '20), October 21–23, 2020, Natal, Brazil.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3422392.3422423

1 INTRODUCTION

The structural quality of a system often degrades due to the code changes that are carried out throughout its evolution [17]. As a result, maintaining the degraded structure can become increasingly difficult and error-prone. To avoid this and other consequences, developers often need to take actions for identifying and removing structural degradation.

To identify structural degradation, developers can rely on automatically detected symptoms such as code smells [11, 15]. An example of code smell is the God Class, which indicates a class that is large and has too many responsibilities [15]. Refactoring is a popular technique that is often applied for removing code smells [11]. This technique consists of applying micro changes in the source code structure without impacting the system's behavior [11]. If refactorings are carried out properly, the structural quality of the system can be improved, or at least maintained. However, refactorings performed in practice are not always effective in removing smells [7, 30].

Existing catalogs and recommendation techniques (*e.g.*, [11, 12, 31]) usually recommend only single refactorings for removing smells. However, besides not being enough for smells removal, single refactorings are often linked to the introduction of new smells [9]. Moreover, recent studies show that developers often apply more than one refactoring, even if they aim for small structural improvements [6, 8, 19, 25, 33]. In this paper, we call such refactoring sequences as *composite refactorings*. Existing techniques for the recommendation of composite refactorings usually provide recommendations that involve re-designing the whole system [2].

However, refactoring a large number of classes is often not feasible in a real project. Therefore, the literature still lacks techniques that assist developers in the application of composite refactorings for preventing major structural degradation. Despite making the importance of composite refactorings evident, existing studies provide little to no information on which composite refactoring patterns are applied in practice. Moreover, the literature still lacks guidelines for the effective recommendation of composite refactorings.

Given the aforementioned limitations, in this work we propose and evaluate three heuristics for composite refactoring recommendations. We are focused in recommendations for three code smell types, namely Complex Class, Feature Envy, and God Class. Those smell types represent common forms of structural degradation that developers often consider harmful [24], and that are early introduced with the creation of new code elements [33].

To propose the heuristics, we relied on composite refactoring patterns that are often associated with the removal of code smells. In our previous work [28], we mined refactorings and code smells from 48 Java open source projects to identify such patterns. After analyzing 104,505 refactorings, we found the occurrence of 2,946 smell-removal composite refactorings. Such investigation resulted in the identification of 35 patterns, being 9 for Feature Envy, 11 for God Class, and 15 for Complex Class. In this work, for each smell type, we selected and implemented one pattern in the form of a composite refactoring recommendation heuristic.

The evaluation of our heuristics consisted of a quasi-experiment involving 12 software developers. We applied our heuristics to 9 Java smelly classes and asked participants to evaluate the resulting recommendations. The participants had to evaluate the recommendations according to their effectiveness in improving the structural quality of the code. In addition, for each evaluated recommendation, the participants provided a detailed feedback on the impact they perceived. For the analysis of the provided feedback, we applied a qualitative data analysis method. This analysis allowed us to assess the proposed heuristics and to identify factors that contribute to the acceptance (or rejection) of refactoring recommendations. Such factors helped us to propose new guidelines for the effective recommendation of composite refactorings.

Our contributions can be summarized as follows: (1) we propose and evaluate three heuristics that can be used to improve state-of-the-art refactoring recommendation tools; (2) participants of our study considered that the impact of the evaluated recommendations was positive or partially positive in more than 93% of the cases; (3) finally, we proposed guidelines that may improve refactoring recommendation techniques.

2 BACKGROUND AND RELATED WORK

2.1 Code Smells

Code smell is a surface indicator of deeper structural problems in the software system [11, 24, 29]. In fact, the presence of code smells can even indicate the need for a refactoring [11]. An example of code smell is the *God Class*, which indicates a class that is large and implements too many responsibilities. A God Class usually impacts quality attributes such as modifiability and extensibility.

To illustrate how a code smell manifests in the source code, let us consider the `LibraryMainControl` class that contains multiple methods and attributes, as shown in Listing 1.

```
1  public class LibraryMainControl {
2    ...
3    private Float fineAmount;
4    private Person user;
5    private Catalog catalog;
6    private Item currentItem;
7
8    public void doInventory() {...}
9
10   public void checkOutItem(Item item) {...}
11
12   public void checkInItem(Item item) {...}
13
14   public void addItem(Item item) {...}
15
16   public void deleteItem(Item item) {...}
17
18   public void printCatalog(Catalog catalog) {...}
19
20   public void sortCatalog(Catalog catalog) {...}
21
22   public void searchCatalog(String term) {...}
23   ...
24 }
```

Listing 1: Partial view of the *LibraryMainControl* class

The `LibraryMainControl` class was flagged as *God Class* because it implements multiple responsibilities, which should be modularized in other classes. For example, the methods `printCatalog`, `sortCatalog`, and `searchCatalog` should be moved to the `Catalog` class as they are concerned with catalog-related functionalities.

In this study, we focus on three types of code smells, namely *Complex Class* [24], *Feature Envy* [15], and *God Class* [15]. *God Class* and *Complex Class* are frequently considered relevant by developers [27] and there is evidence that *Feature Envy* and *God Class* reflect important maintainability aspects [34]. These three smells are also of major severity if compared with many others, and their resolution would eliminate/reduce other inner smelly structures (e.g., *Long Method*) [1, 9]. Finally, their removal usually requires at least two refactorings [7, 28].

2.2 Composite Refactoring

Refactoring consists in transforming the code structure without changing the functional behaviour of the system [11]. Thus, we consider that refactoring is any structural change that is aimed at improving quality attributes of the system's design. There are multiple refactoring types cataloged in the literature (e.g., [11] and [32]). Each refactoring type is applied to perform a specific structural transformation. For instance, Extract Class aims at creating a new class with methods and attributes of an existing class.

In Listing 1, existing catalogs and techniques [11, 12, 31] would probably recommend the application of Extract Class to solve the God Class smell. Although it is a correct strategy for many cases, it might not be the most appropriate for the *LibraryMainControl* class. The reason is that the additional responsibilities of that class would be better modularized into the *Catalog* and *Item* classes.

A **composite refactoring** (or batch refactoring) happens when two or more related refactorings are applied to one or more code

elements [6, 8, 25, 28, 33]. The composites can be divided into two broader categories, namely temporally-related composite (*i.e.*, refactorings applied in the same commit) and spatial composite (*i.e.*, refactorings applied to structurally related code elements, on the same commit or not) [28]. Even though some studies indicate that the composite refactorings are applied by a single developer [6, 18], there are cases where the developers can work in groups to perform a composite [14] (*i.e.* when they are applying large structural changes in the software system). For the *LibraryMainControl* class, a suitable composite refactoring would be to perform multiple Move Method refactorings, transferring methods from *LibraryMainControl* to the *Catalog* and *Item* classes.

2.3 Related Work

Composite recommnedation. The proposal and evaluation of refactoring tools have been the focus of several studies (*e.g.*, [5, 31]). Despite making important contributions, such studies have not considered the application of composite refactorings for removing code smells. Other studies focused on proposing recommendations for composite refactorings [6, 9]. Cedrim *et* al. [9], for example, investigated the effect of single refactorings on code smells. They found that single refactorings often introduce or do not fully remove code smells. Then they suggest some composites that could have removed code smells based on Fowler's catalog [11]. However, these suggestions were based only on anecdotal evidence. Bibiano *et* al. [6] investigated composite refactorings on 57 software projects. They found some composites that removed code smells, generating recommendations from these results. However, they detected only composites in the scope of individual elements. They did not analyze composites that involve and affect multiple elements.

Search-based techniques. Other studies have used Search-Based Software Engineering (SBSE) techniques to create recommendations of composites (e.g., [2, 22]). Ouni *et* al. [22], for example, introduced an automated approach for refactoring recommendation, called MORE, using a genetic algorithm that is focused on three objectives: (1) improve design quality, (2) fix code smells, and (3) introduce design patterns. In general, such techniques have disadvantages that prevent their adoption and use in practice. The first disadvantage is that they usually provide recommendations that involve re-designing the whole system [2]. However, refactoring a large number of classes is often not feasible in a real project. In addition, such techniques do not allow the application of floss refactorings [18]. Finally, the use of SBSE techniques in large projects may be impracticable due to the computational cost.

Guidelines for refactoring recommendation. Bavota *et* al. [4] presented guidelines to build and evaluate refactoring recommendation tools. However, unlike us, their guidelines are not based on qualitative empirical data. They generated several recommendations based on state-of-the-art papers and based on the authors' prior knowledge on the subject. In addition, according to the authors themselves, there are still limitations and challenges regarding the recommendation of composite refactorings, which are not covered by their guidelines. Tsantalis *et* al. [31] presented lessons learned from 10 years of research with the JDeodorant tool. Their lessons are mostly focused on the adoption of recommendation tools in practice. We, on the other hand, focus on guidelines for the effective recommendation of composite refactorings. As far as we know, we are the first to identify such guidelines through a systematic qualitative study.

Table 1: Refactoring patterns that often remove smells

Smell Type	Pattern
Feature Envy	**Extract Method, Move Method**
	Inline Method{n}, Extract Method{n}
	Extract Method{n}, Move Attribute{n}
God Class	**Move Method{n}**
	Pull Up Method{n}, Move Method, Pull Up Method
	Pull Up Method{n}
Complex Class	**Extract Method{n}**
	Pull Up Method{n}, Move Method, Pull Up Method{n}
	Move Attribute{n}, Extract Method{n}

3 SMELL REMOVAL PATTERNS

In our previous work [28], we analyzed composite refactorings from a dataset of 48 open source projects. Our results revealed composite refactoring patterns that are associated with the introduction or removal of code smells. Such patterns served as basis for creating composite refactoring recommendation heuristics. Next, we summarize the research procedures and the resulting patterns. More details are available in our replication package [23].

Procedures for Finding Patterns. Firstly, to identify code smell removal patterns, we detected the code smells and the refactorings. We implemented rule-based detection strategies for code smells [15] and used the the Refactoring Miner tool [32] for detecting refactorings. After that, we applied a synthesis strategy that is able to identify both temporally-related composites and spatial composites (see Section 2.2). After finding composite refactorings, we identified recurrent composite refactoring patterns related to the removal of Feature Envy, God Class, and Complex Class. This identification was based on the frequency with which each pattern resulted in the removal of each smell type.

Refactoring Patterns. Following the aforementioned method, we found 2,946 smell-removal composite refactorings. Then, we identified 35 smell removal patterns, being 9 for Feature Envy, 11 for God Class, and 15 for Complex Class. Table 1 shows a sub-set of the most frequent patterns for each smell type. In the second column of Table 1 is the list of refactoring types that compose each pattern. The *{n}* symbol after some refactoring types means that the refactoring type was performed an *n* number of times in the analyzed cases. Such number was varied in each occurrence.

Selection of patterns for heuristics implementation. In Table 1, we highlighted in bold the smell removal patterns that were selected for the implementation of our heuristics. We applied the following criteria for selecting them. First, we opted for patterns that would be viable to implement using state-of-the-art algorithms for automated refactoring (e.g., [10, 31]). Second, we chose the patterns involving the least number of different refactoring types. Thus, we believe that the resulting recommendations would be easier to analyze and understand. Finally, we selected patterns that would make sense for most possible scenarios. This last criterion is important because there are patterns that can only be applied for classes with inheritance (e.g., *Pull Up Method{n}*). Given such

restrictions, we selected *Extract Method, Move Method* for Feature Envy, *Move Method{n}* for God Class, and *Extract Method{n}* for Complex Class. In the next section, we present the heuristics that were implemented based on such patterns.

4 SMELL REMOVAL HEURISTICS

In this section we present three heuristics for removing code smells. Each one is focused in removing a particular type of code smell. They were derived from the refactoring patterns presented in Section 3. As presented in that section, there are many patterns that may be applied for removing code smells. For instance, it is possible to remove a *Complex Class* by applying several *Push Down Methods* or by applying a sequence of *Extract Methods*. Since our objective in this study is to check the viability of deriving useful heuristics, we implemented and evaluated only one pattern for each code smell type. Next, we present details about each heuristic.

4.1 Feature Envy Removal

One of the most common patterns found for removing *Feature Envy* is composed of *Extract Method* and *Move Method*. Although this sequence may have been applied independently – i.e., the *Move Method* may have been applied to a different method than the extracted one – many times when developers were successful in removing *Feature Envies*, they first extracted the foreign part of the method into a new one, and then moved the newly created method to a different class. Thus, we define a *Feature Envy* removal heuristic that always applies the *Move Method* to the extracted method. Formally, the *Feature Envy* removal heuristic is composed of four parts: (i) identification of method lines that are more interested in different class; (ii) extraction of these lines into a new method; (iii) identification of the class that suits the newly-created method; and (iv) application of a *Move Method* refactoring to move the new method to the identified class.

Each step of this heuristic poses a different challenge. The first one is that we need to identify lines of the *Feature Envy* method that are more interested in a different class other than the one they are contained in. It is not trivial to recommend lines of code to be extracted from a method because the section of code to be removed must execute a particular functionality in a way that the removal of the whole section makes sense. Several studies propose different techniques to accomplish the objective of discovering extract method opportunities [10, 31]. The technique proposed by Charalampidou *et al.* [10] is based on the functional relevance of the combined lines. They introduce an approach that aims at identifying source code chunks that collaborate to provide a specific functionality and propose their extraction as separate methods. Since their approach fits well with our first part for recommending *Feature Envy* removal, we adopt it.

Therefore, to accomplish the first part of this heuristic, we implemented the approach proposed by Charalampidou *et al.* [10]. As described in their paper, they propose an approach called SRP-based Extract Method Identification (SEMI). In particular, their approach recognizes fragments of code that collaborate for providing functionality by calculating the cohesion between pairs of statements. The extraction of such code fragments can reduce the size of the initial method, and subsequently increase the cohesion of the resulting methods. In our scope, we implemented their technique and treated this component as a black box, where the input is a method and the output is a set of line intervals that can be extracted.

By having these possible intervals, we have several possibilities to recommend extraction. However, only having these intervals is not enough to remove the *Feature Envy*, since we do not want to recommend an extraction that would still maintain the smell that we already had. Therefore, we run a verification step for each interval. We simulate the removal of such lines by disregarding their influence on the *Feature Envy* detection. Hence, we test, for each interval, if its removal would lead to a *Feature Envy*. If the removal of a single interval is not enough to remove the *Feature Envy*, then we look for a combination of two intervals. We keep increasing the number of intervals until we have a *Feature Envy* removal possibility. After completing this step, we can move on for the second part of the heuristic: *Extract Method* refactoring.

After the aforementioned step, we can test if there is still a *Feature Envy*. Unfortunately, the newly-created method could still have the *Feature Envy*. However, we have to leverage in the fact that we know its lines are cohesive and can be moved together to a different class. In this way, we check which class this new method relates to the most, either by method calls or attribute use. After discovering this class, we can recommend a *Move Method* refactoring of the newly-created method to this discovered class.

Therefore, our first heuristic is completed by executing the four described steps. We first recommend an *Extract Method* by combining our code smell detection strategy with the SEMI approach. After extracting the method, we can recommend a *Move Method* by examining the method calls and attributes use of the newly-created method. In this way, we reproduce programmatically the composite-smell pattern *Extract Method, Move Method* presented in Section 3. Notice that we do not only reproduce the removal pattern; we instead execute a verification to make sure that the composite would be able of removing the smells. The combination of the verification with the knowledge about the removal patterns is what increases the chance of our heuristic to get rid of a *Feature Envy*.

4.2 God Class Removal

As presented in Section 3, *Move Methods* play a central role on *God Class* removals. *God Class* is a class that assumes several responsibilities in a system. If we distribute these responsibilities (*i.e.*, methods) over several classes in the system, the developer can remove the smell (Section 3). Indeed, to perform this distribution, we found that developers can apply different types of refactorings; for instance, he can apply a composite composed of *Move Method{n}*. Hence, the heuristic we implemented to remove God Class is based on method-moving refactorings.

According to the rules of *God Class* detection [3], a class has this smell if its cohesion is lower than the average of the system, and it contains more than 500 lines of code. This threshold can be tailored to particular projects or modules by using machine learning techniques [13]. Thus, for each method in the class, we identify a suitable class to which we move it. We used the same strategy presented in Section 4.1 to identify the destination class

of the method. We keep recommending *Move Methods* until the *God Class* is removed. However, such operations can create new *God Classes* in the system [28]. Therefore, before recommending a Move *Method*, we check if the destination class would become a *God Class*. If so, we change the recommendation to the second most suitable class.

It is worth mentioning that we find the suitable class by counting the number of method calls and accesses to attributes. For instance, let us assume that a particular method *m* calls 3 methods and accesses 2 attributes from class *A*. In this case, the "bonding factor" of *m* to *A* is 5. Let us assume the same method *m* calls 4 methods from the class *B*, leading to a bonding factor of 4. Now, assume our heuristic recommended to move *m* to *A*, but *A* would be transformed into a *God Class* if this occurs. In this case, the heuristic would recommend the method-moving change to class B, since in this case, *B* would still be smell-free.

In summary, the second heuristic is a sequence of *Move Methods*. However, it also uses the smell-detection strategy to understand when the target class would not be a *God Class* anymore. Additionally, the smell detection strategy is used to prevent the creation of new code smells after the recommended refactoring.

4.3 Complex Class Removal

In our studies, we consider a class as *Complex Class* if it has at least one method having a high cyclomatic complexity (CC) [15]. So, the strategy to remove such smell is related to the reduction of the complexity of methods with high CC. As presented in Section 3, developers often apply *Extract Methods* to remove such complex structures. Hence, the heuristic to remove *Complex Class* is composed of four parts: identify all methods with high CC, identify *Extract Method* opportunities to reduce the complexity, evaluate the identified opportunities, and recommend *Extract Methods*.

The first part is composed of our code smell detection strategy, which finds all *Complex Methods* in a particular *Complex Class*. After finding them, we use the SEMI approach presented in Section 4.1 to generate possible line intervals to be extracted. After identifying such intervals, we need to evaluate the identified opportunities. For each interval found, we simulate its removal and compute what would be the new complexity of the method. When we find a interval (or a set of intervals) that reduces the complexity, we start recommending the *Extract Methods*.

Therefore, after running the steps of this heuristic, our tool can identify pieces of code that can be extracted to reduce the complexity of the methods found. After the recommendation of a composite of *Extract Methods*, we can distribute the complexity of the class into several smaller methods, getting rid of the original *Complex Class*. It is worth mentioning the recommended extractions can pose a risk and create new code smells, such as a new *Feature Envy*. If this occurs, we can trigger the *Feature Envy* removal heuristic to improve the composite by removing the introduced smell.

5 EMPIRICAL EVALUATION: STUDY DESIGN

This section presents the design of a quasi-experiment [27]. This experiment was designed to evaluate the three proposed heuristics.

5.1 Goal and Research Question

We hypothesize that our heuristics (Section 4) can be effective in real scenarios. Thus, our goal is to *evaluate the smell-removing heuristics regarding their effectiveness in improving the source code structural quality*. To achieve this objective, we observed how the heuristics perform in real scenarios to evaluate their effectiveness. Thus, our quasi-experiment involved software developers and industry systems. To avoid bias, we applied the heuristics to projects that were not used to find the composite refactoring patterns (Section 3).

Our first research question – *RQ1: Are the smell-removing heuristics effective in improving the code structural quality*? – is intended to assess the effectiveness of our heuristics in helping developers to combat structural code degradation. To address RQ1, we first applied the heuristics steps on different smelly code elements. Each heuristic comprises some steps (Section 4), and the application of each step in a code element delivers a new code state. In this way, we documented each code state obtained as a result of the application of each heuristic step. After this, we compiled all the results and asked for the evaluation of software developers. They had to evaluate the code states and inform us about their opinion concerning the impact of the code changes on the structural quality of the code. After this, we conducted quantitative and qualitative analysis.

The second research question – *RQ2: Why do developers accept or reject a composite refactoring recommendation?* – aims to characterize the factors that lead to the acceptance or rejection of refactoring recommendations. These factors are relevant to smell removal recommendations in general (not only ours), but they have never been investigated in previous studies. To answer this question, we relied on the qualitative analysis of responses provided by developers in the quasi-experiment. We applied systematic procedures to find and categorize the reasons that led participants to accept or reject the recommendations. Such procedures led us to build an initial set of guidelines related to the acceptance or rejection of refactoring recommendations. More details about the qualitative data analysis procedures are provided in Section 5.3.

5.2 Experimental Tasks

To evaluate the heuristics, our quasi-experiment was composed by three main activities, described as follows.

Activity 1: Sample Selection and Heuristics Execution. Since our objective is to evaluate the heuristics, we had to execute them in different classes affected by the studied code smells, then we selected 3 different classes for each smell. Then, we executed each of the proposed heuristics in the contexts of three classes containing the corresponding smells. We selected three classes for each smell, so that the experiment's participants could have the proper time to inspect each one of the 9 recommended composites. Also, we selected, for each smell, classes from three distinct projects. Besides that, we chose classes implemented with different purposes, from log-in services to classes that manage students data from an educational institution. Since our goal here was not to compare different heuristics, we decided to select classes with varying complexity and degradation characteristics. This helped us to evaluate the heuristics in heterogeneous scenarios. We then executed the heuristics

for each sample. As presented in Section 4, each heuristic produces as output a list of recommended refactorings.

Activity 2: Recruitment and Characterization of Subjects. We invited 20 software developers to participate in this study, among which 12 met the minimum requirements and agreed to participate. We asked the developers to fill out a questionnaire to gather their information, including educational level, professional experience with software development in terms of years, experience with Java programming (in years), and whether they were familiar with code smells and refactoring or not. The data collected during this activity was used to understand if the participants met the minimum requirements needed to participate in the experiment. Since all code examples are in Java, the participants have to be able to read and understand the code. They also need to know how to refactor a piece of code. Otherwise, it would be very hard for them to understand the heuristics steps, invalidating their answers. Screen shots of the questionnaire are available in our replication package [23]. Table 2 presents the data regarding the participants' years of experience with software development, years of experience with Java, and number of Java developed projects. Most participants have industry experience with software development, and with the Java language. Only one participant has no experience in industry. Nevertheless, such a participant does not pose a threat to the study due to his/her experience with code smells and refactoring research.

Table 2: Participants' characterization data

Answer	Median	Average	Std. Dev.	Max	Min
Years of programming experience	4	5.5	4.6	15	0
Years of experience with Java	1	2.8	3.9	14	0
Number of Java developed projects	1	5.4	13.4	50	0

Activity 3: Experiment Execution. As mentioned before, we executed the heuristics' steps on 9 different smelly classes. Each execution led us to a sequence of refactorings, implicating in several code changes. Each participant had to evaluate each sequence of refactorings generated for each one of the 9 classes. Hence, each participant had to visualize and evaluate 9 composite refactorings. After visualizing each composite, the participants had to answer the following question:

> What is your opinion about the impact of the sequence of refactorings on the code structural quality?

As we can see, this question does not involve the term *code smell*. Although the heuristics had been derived from relationships between composites and code smells, we are ultimately interested in improving the code structural quality. If developers feel that the code had its structural quality improved by our heuristics, this is one more evidence that code smells are, in fact, good estimators to measure the code structural quality. In other words, we can keep developing heuristics focused in code smells because, in the end, the code structural quality can be improved by their removal.

The participants had to choose between Positive, Intermediate, and Negative as an answer to the provided question. In each case, they had to provide a detailed feedback to justify the answer. Table 3 presents the three possible answers that each participant had to give for each one of the nine presented composites. In any case, they had always to provide a justification for the answer in an open text field, so we can use it to better understand their answers. We developed a web application in order to present composites and to collect the developers' answers.

Table 3: Possible answers during the quasi-experiment

Answer	Description
Positive	The code structural quality has improved
Intermediate	There are benefits, but I think there is room for improvement
Negative	The code structural quality has decreased

5.3 Qualitative Data Analysis

One of our goals in this study was to discover and understand the reasons that lead a refactoring recommendation to be (partially) accepted or rejected by developers. Therefore, we asked the participants to explain the reasons for classifying a recommendation as positive, negative, or intermediate. Thus, the experiment provided us with a robust material for building an initial set of guidelines that would be helpful for providing effective recommendations. To identify the guidelines, we have adapted procedures that are commonly used in qualitative data analysis methods such as the Grounded Theory [16]. To avoid bias, three researchers participated in this analysis, conducting discussions whenever necessary. Below we present details about our procedures.

Coding the data. Open coding is a commonly applied procedure to extract relevant codes from textual content. Thus, we started our analysis with open coding, following the recommendations of Lazar *et al.* [16]. Before identifying and extracting the codes, we defined three types of textual sentences that we were interested in. The first type comprises sentences indicating positive motivations that contribute to the acceptance of a recommendation. The second type includes sentences that indicate negative motivations regarding a recommendation; thus, contributing to its rejection. Finally, for the last type, we included sentences related to suggestions of improvement for the recommendations. We are interested in this last type of sentence because we observed that participants provided several relevant suggestions during the experiment. This occurred even when they considered recommendations to be positive or intermediate. Based on the extracted and classified sentences, we identified codes that briefly described each relevant sentence.

Creation of categories and relationships. After identifying the codes, we searched for relevant categories of codes. The categories were created through the analysis and exhaustive comparison of the different codes. This allowed us to identify similar codes that were repeated in the responses of different participants and, therefore, resulted in the creation of categories. Besides that, we also identified the relationships that exist among the different categories, which is an important step in a qualitative data analysis.

Synthesis and description of the guidelines. In this last step, we analyzed the categories and relationships created to synthesize and describe our guidelines in the form of factors that are often related to the acceptance or rejection of refactoring recommendations. Given the number of participants, it was not possible to reach theoretical saturation. However, the data provide a significant contribution for refactoring recommendation techniques and tools.

6 EVALUATION RESULTS

6.1 Effectiveness of Recommendations

Each participant evaluated 9 refactoring recommendations produced by our smell removal heuristics (Section 4). In each evaluation, they had to inform their opinion about the impact of the composite on the code structural quality. Table 4 summarizes the data regarding the answers given by the participants. The first column shows the smell type that was targeted by each recommendation, while the second column shows the name of the smelly class. The number of classifications (positive, intermediate or negative) provided by the participants in each case are summarized in the 3rd, 4th and 5th columns of Table 4.

Most of the recommendations were considered positive or intermediate. The majority of participants considered most recommendations as being positive (74%) or intermediate (20%) (Table 4). Only few recommendations were considered negative by some developers (6%). For 4 out of 12 recommendations, no negative classification was provided. This acceptance rate is slightly higher than that of other similar techniques. Bavota *et* al. [5], for example, reported a study in which more than 70% of their recommendations were considered meaningful.

Suggestions for improving the recommendations. As described in Section 5.3, after executing the quasi-experiment, we also conducted a qualitative data analysis. For all recommendations, at least one participant suggested some improvement. Despite having received only one negative classification, the heuristic for Feature Envy received the highest number of suggestions for improvement (12), followed by Complex Class (9) and God Class (5).

One of the most recurring suggestions was related to the provision of additional information regarding the proposed refactorings and regarding the structural quality of the code. Some of the participants who had little experience with refactoring had difficulty in understanding the structural changes caused by the refactorings. In addition, the participants raised the need for some support to visualize the impact of refactorings on structural quality. We further discuss this suggestion in Section 6.3.

Positive and negative sentences in the detailed feedback. To avoid bias, we analyzed the detailed feedback from the participants for finding sentences that indicate positive and negative aspects of the recommendations (Section 5.3). We identified, for each recommendation, the number of participants that: wrote only positive sentences, only negative sentences, both positive and negative sentences, and did not write any relevant sentences for this analysis. We also identified the total number of positive sentences and the number of negative sentences found in each detailed feedback. We included such results in our replication package [23].

In this analysis, we observed that the percentage of classifications with only positive sentences (59%) is significantly smaller than the percentage of positive classifications (74%). This happened because, even classifying the recommendations as positive, some participants wrote feedback indicating that there were negative aspects that could be improved. Similarly, certain recommendations classified as intermediate contained only negative sentences. Thus, looking from this alternative perspective, the percentage of intermediate and negative feedback is higher, i.e., 14% and 12% respectively. Still, if we look at the total number of sentences, we can see that the number of positive sentences (128) is significantly higher than the number of negative sentences (50). In addition, this result does not invalidate the classifications provided by the participants because, even when writing about aspects that could be improved, the participants considered the recommendations to be positive or intermediate. Next, we present details about the recommendations for each type of smell.

Table 4: Summarized results of the quasi-experiment

Code Smell	Class	Pos.	Int.	Neg.
Complex Class	Clause	6	4	2
	GenericTranspalBean	11	0	1
	DiarioClasseService	10	2	0
Feature Envy	IngressoUniversidadeService	7	5	0
	Media	10	2	0
	UserFactory	10	1	1
God Class	MatriculaAcademicaService	7	4	1
	LibraryMainControl	10	2	0
	EmployeeUtils	9	1	2
All		80 (74%)	21 (20%)	7 (6%)

6.2 Impact of Recommendation Heuristics

We did not compare our heuristics with existing ones, since previous studies detect different types of code smell and use different detection strategies (Section 5). However, in this section, we present the results for each recommendation heuristic and discuss how they can be used for improving existing tools. For this purpose, we used the JDeodorant tool [31] as an example. JDeodorant is a well known tool that is able to detect and recommend refactorings for six code smell types. Like us, JDeodorant is able to recommend refactorings for Feature Envy and God Class but not for Complex Class.

Support for Removing Complex Class. JDeodorant does not have any heuristic for detecting and removing Complex Classes. Thus, our heuristics could be incorporated by JDeodorant to make it possible to recommend composite refactorings that remove Complex Class. As presented in Table 4, our heuristic for Complex Class presented satisfactory results for most classes. Nevertheless, for the *Clause* class, our recommendation was considered positive only by 6 out of 12 participants. When looking at the detailed feedback, 3 participants wrote only positive sentences, 2 wrote only negative sentences, and 6 wrote both positive and negative sentences.

The *Clause* class was flagged as *Complex Class* because of the method *toCriterion*, which has a high cyclomatic complexity. This method is composed of a chain of ifs, leading to low code readability. As presented in Section 4.3, the heuristic recommends a list of *Extract Methods* to distribute the complexity. In this particular case, the heuristic recommended two *Extract Methods*. The *Complex Class* was removed because the class no longer has methods with high cyclomatic complexity.

However, the legibility of the created methods is still not good, as considered by some participants. They suggested a complete rewrite of this method by using different data structures and even a different object model design to implement the same functionality. Unfortunately, the heuristics are not able to recommend that, since the suggested changes are not part of our composite refactoring patterns. Thus, a total reshape is probably the best solution for

the case. Nevertheless, in the perspective of 10 participants, the recommendation achieved some improvements, while still having room for making the class structurally better. Below we quote the feedback of one participant regarding this difficult case.

> "...I do agree the method is less complex, but the complexity was merely spread into the other two methods..." – *Feedback about the Clause class refactoring.*

If we look at the recommendations provided by the JDeodorant tool, a different type of recommendation could be provided for the *Clause* class. JDeodorant supports the recommendation of the *Replace Type code with State/Strategy* refactoring when there is a State Checking smell. In fact, one of the participants recommended this refactoring for the *Clause* class. Therefore, the solution proposed by the JDeodorant tool would be the most appropriate in this case. It is important to note that, as mentioned earlier, we would not be able to recommend such refactoring because we have not identified refactoring patterns for State Checking smells.

Despite this unfavorable case, our Complex Class heuristic is relevant for other cases. Most participants mentioned that it was able to significantly reduce the complexity of the *GenericTranspalBean* and *DiarioClasseService* classes. In these cases, complexity was caused by the presence of conditionals and nested loops. Thus, the recommendation of *Extract Methods* was adequate to reduce complexity and improve readability of the code.

Feature Envy Affecting the Implementation of Multiple Methods. The heuristic of JDeodorant for removing Feature Envies consists of recommending Move Methods for each of the affected classes. As observed in our study, Move Method is indeed applied by developers during the removal of a Feature Envy. However, in many cases, the sole execution of this refactoring is not enough. The reason is that the envious code may be within a method mixed with non envious code. Therefore, we conjecture that JDeodorant recommendations could be improved by our heuristic. Besides recommending the *Move Method* for fully envious methods, JDeodorant would also be able to address more complex scenarios based on the recommendation of composite refactorings. In such cases, our heuristic for *Feature Envy*, would help developers to remove *Feature Envies* that impact the implementation of multiple methods.

In fact, the heuristic for *Feature Envy* removal was the most successful one in our quasi-experiment. Only one participant gave a *negative* answer, while 8 answers were *intermediate*, and 27 were *positive*. These results gave us confidence about the removal patterns we found. We were able to derive a heuristic that was able to remove the smell very often, according to the participants. The developer we quote below is one of the *intermediate* answers.

> "I believe the refactorings improved the code, but its readability is still not perfect. I believe the constructor is still large." – *Participant with 4 years of experience*

In this case, the participant agrees the heuristic was positive, but there is a suggestion to keep improving the source code. Most of the *intermediate* answers mention the need for more refactorings in order to reach an ideal state. However, most of those improvements were not closely related to the purpose of the heuristic being evaluated. The most experienced participant said:

> "I agree with the proposed refactorings. They were enough to remove the code smell." – *Participant with 14 years of experience.*

Removing God Class without a New Class. For removing a God Class, JDeodorant recommends the creation of a new class through the Extract Class refactoring. This heuristic is effective when a new class is necessary. Nevertheless, there are scenarios in which the God Class contains methods and attributes that could be placed in an existing correlated class. Thus, our heuristic for God Class could present effective recommendations for developers in such scenarios. As described in Section 4, in such cases, our heuristic is able to remove God Classes without introducing new ones and without the creation of new classes.

In the three cases that we presented to the participants, we recommended several *Move Methods* to remove the *God Class*. All three classes are very large and contains thousands of lines and dozens of methods. Even in these highly complicated scenarios, the heuristics achieved 7 intermediate, 26 positive, and only 3 negative answers. Interestingly, not even a single one developer criticized a proposed *Move Method*. There were no complaints regarding the suggested refactorings. All complaints were related to the continuity of the improvements, i.e., the developers were expecting more *Move Methods* to solve the problem, as the one we quote below.

> "This class still needs more refactorings, because I think it still contains several responsibilities." – *Participant with 1 year of experience.*

These results are exciting because it shows a difference between what we consider *God Class*, and what some developers think. According to our *God Class* detection rules, the target classes were not affected by the smell anymore after the refactorings, while the developers still think it is. In this case, we could change the rule to be more severe on the *God Class* detection. For instance, we can reduce the threshold of the number of lines a class must have to be classified as *God Class*. If we change the threshold, the heuristic would continue to suggest *Move Methods*, and the unsatisfied developers might be satisfied if we use the new rule. Most developers agree that the outcome is positive, as the one we quote below:

> "I believe the class was very confusing before the refactorings. Now, after the refactorings, the class is way easier to understand and maintain." – *Participant with 2 years of experience.*

Upon data analysis, we can say that the evaluated heuristics are indeed effective in removing code smells. Even with some preliminary heuristics, we were able to achieve a high acceptance from the developers. In this way, it seems worthwhile to follow the path of improving and creating more heuristics. Nevertheless, we do not believe that existing heuristics should be replaced by our heuristics. We also do not expect our heuristics to show satisfactory results in any case. The reason is that, despite finding several composite refactoring patterns, we implemented heuristics corresponding only to one pattern per type of smell. Thus, as discussed above, our results show that state-of-the-art recommendation tools can, in fact, benefit from our results.

6.3 Guidelines: Improving Recommendations

Based on our qualitative analysis, we were able to identify several factors that contributed to the developers classifying our recommendations as positive, intermediate, or negative. These factors compose the answer for our RQ2. In fact, we believe these factors can be more useful than just explaining why the participants accepted or rejected a composite refactoring recommendation. In this sense, we relied on these factors to propose a set of guidelines that can help other researchers to propose recommendation heuristics. These guidelines may increase the chance of developers perceive techniques for composite refactoring recommendations as useful.

From the participants' feedback, we extract 128 positive sentences, 50 negative sentences, and 26 improvement suggestions (see our replication package [23]). Following systematic procedures (Section 5.3), we analyzed the sentences and identified multiple recurring codes that resulted in the creation of 16 factors that contribute to the acceptance and 11 factors that cause the rejection of refactoring recommendations. Next, we present guidelines related to the most relevant factors.

Refactoring need and impact must be clear. A prominent factor noted in this study was related to convincing developers that they should conduct the proposed refactorings. To achieve such objective, we noted that: (1) the need for refactoring must be clear, (2) the effect of the refactorings in the structural quality must be evident, and (3) the developer must agree that the resulting structure is significantly better than before the refactorings.

Regarding the need for composite refactoring, just informing the existence of a smell is usually not enough, except in cases where the degradation is severe. For the common cases, we noted that, the developer must be also informed both about the metrics that indicate the presence of the smell and about the quality attributes that are being impacted. In the experiment, we did not provide such information as we would like the participants to judge the recommendations without any influence on our part. However, our results show that providing such information is essential for composite recommendations to be accepted.

Understanding the impact of composite refactorings on the code's structure also proved to be a fundamental factor. Participants made constant mentions of design principles and quality attributes that were improved. The quality attributes that were frequently mentioned are the following: Legibility, Maintainability, and Reusability. To our surprise, legibility was the most cited quality attribute, even for when removing smells like God Class and Feature Envy, which usually involves significant changes to the code structure. Besides that, there were mentions to design principles such as cohesion, coupling and separation of responsibilities.

Proposed composites should be intuitive. We also observed that the more intuitive the refactorings are, the more likely the developer is to accept them. This is not always possible to achieve because in some cases the removal of a smell may require composite refactorings that are not always easy to understand, especially for less experienced developers. Therefore, as discussed above, providing information about the code structure and the impact of refactorings is essential. For less experienced developers, it may also be necessary to explain what type of structural change each recommended refactoring will perform in the code.

Resulting structure should not be worse than before. Even after the composite refactoring, the developer may consider that the code structure has not improved, or became worse than before. This may happen when the composite removes a smell but introduces others. Such scenario may occur with God Class removal recommendations, for example. While the God Class refactorings have the potential to improve the separation of concerns, in some cases they may also result in the introduction of Lazy Class smells. Finally, the recommendation technique must also be able to remove all relevant smell occurrences from the refactored class. Developers are often not satisfied when a particular smell is removed, but others remain in the class. Thus, it is fundamental to assess the impact of a composite before its recommendation. In our case, we had heuristics for only three smell types, so we were unable to identify and remove other types of smells that already exist or that have been introduced during refactorings.

Developer-driven customization is fundamental. Developers can often disagree with the smells detected by the recommendation technique. They can also believe that the smell has not been completely removed after the recommended composite. This occurs because code smell detection is highly sensitive to the developer [13, 26] and to other contextual factors [20, 21]. Thus, we believe that customized detection strategies should be used for each project or for each development team. There are, indeed, techniques based on Machine Learning that try to provide customized smell detection (*e.g.*, [13]). However, their effectiveness is still far from ideal.

In conclusion, the factors discussed above answer our research question RQ2. They are related to technical and human issues. Such factors can be useful to improve existing techniques and to propose new techniques, since they show what developers believe is important in a composite refactoring recommendation technique.

7 THREATS TO VALIDITY

This section presents some threats that could limit the validity of our findings. For each threat, we present the actions taken to mitigate their impact on the research results. The first threat to validity is related to the number of participants in the study. We have selected a sample of 12 participants, which may not be enough to achieve conclusive results. Nevertheless, our data analysis was mainly based on a qualitative method that does not require a large number of participants to achieve scientifically relevant results [35].

The second threat is related to possible misunderstandings during the study. Participants may have conducted the study differently from what we asked. To mitigate this threat, we wrote thorough instructions in a web page and encouraged them to reach us in case of any doubt. We highlighted that our help would be limited to only clarifying the study in order to avoid bias in the results. In addition, the open answers may have been biased by the objective question, since it allowed only three response options. Nevertheless, our qualitative analysis involving the content of the open answers provided evidence that they were not influenced by the objective question.

In this experiment, we opted for not comparing our recommendations with a baseline. The reason is that existing refactoring recommendation techniques do not detect the same types of code

smells that we are using in this study. Moreover, the detection strategies for the smell types in common are different. As a result, the baseline technique could detect different types of code smells in the classes used in our experiment. To mitigate this threat, we based our study on an in-depth qualitative data analysis.

Finally, there is a threat concerning the selected classes and composites. The complexity of the code and the refactorings may have caused the participants to not perform the experiment properly. To mitigate such threat, we described each class thoroughly, and we were very careful to explain what was going on on each step of the generated composites. In addition, we only selected participants with a minimum knowledge about Java, code smell and refactoring.

8 CONCLUSION

In this paper, we proposed and evaluated three refactoring heuristics for removing code smells. Towards their evaluation, we executed and reported a quasi-experiment. Results show that the heuristics are promising, leading to interesting and well-evaluated recommendations. Such results encourage the use of our heuristics for the creation or improvement of recommendation systems.

Although we got a high number *positive* answers, we still got a reasonable number of *intermediate* ones (20% of all the answers). Thus, it is worthwhile to work towards the reduction of this number since we could increase the developers' satisfaction. To do this, we can explore more of the removal patterns (Section 3) and the guidelines (Section 6.3). As a future work, we intend to build and evaluate a composite refactoring recommendation tool based on findings from this study and also based on the existing literature.

ACKNOWLEDGMENTS

This work was partially funded by CNPq (grants 434969/2018-4, 312149/2016-6, 140919/2017-1), CAPES/Procad (grant 175956), CAPES/Proex, and FAPERJ (200773/2019, 010002285/2019).

REFERENCES

[1] M Abbes, F Khomh, Y Gueheneuc, and G Antoniol. 2011. An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, on Program Comprehension. In *Proceedings of the 15th ESEC; Oldenburg, Germany*. 181–190.

[2] Vahid Alizadeh and Marouane Kessentini. 2018. Reducing Interactive Refactoring Effort via Clustering-based Multi-objective Search. In *Proceedings of the 33rd ASE*. ACM, New York, NY, USA, 464–474.

[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).

[4] Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto. 2014. *Recommending Refactoring Operations in Large Software Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 387–419.

[5] Gabriele Bavota, Malcom Gethers, Rocco Oliveto, Denys Poshyvanyk, and Andrea de Lucia. 2014. Improving Software Modularization via Automated Analysis of Latent Topics and Dependencies. *ACM Trans. Softw. Eng. Methodol.* (2014).

[6] Ana Carla Bibiano, Eduardo Fernandes, Daniel Oliveira, Alessandro Garcia, Marcos Kalinowski, Baldoino Fonseca, Roberto Oliveira, Anderson Oliveira, and Diego Cedrim. 2019. A Quantitative Study on Characteristics and Effect of Batch Refactoring on Code Smells. In *13th ESEM*. 1–11.

[7] Ana Carla Bibiano, Vinicius Soares, Daniel Coutinho, Eduardo Fernandes, João Correia, Kleber Santos, Anderson Oliveira, Alessandro Garcia, Rohit Gheyi, Baldoino Fonseca, Márcio Ribeiro, Caio Barbosa, and Daniel Oliveira. 2020. How Does Incomplete Composite Refactoring Affect Internal Quality Attributes?. In *28th IEEE/ACM International Conference on Program Comprehension (ICPC)*.

[8] Aline Brito, Andre Hora, and Marco Tulio Valente. 2020. Refactoring Graphs: Assessing Refactoring over Time. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE.

[9] 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 the 11th ESEC/FSE*. ACM, 465–475.

[10] S. Charalampidou, A. Ampatzoglou, A. Chatzigeorgiou, A. Gkortzis, and P. Avgeriou. 2017. Identifying Extract Method Refactoring Opportunities Based on Functional Relevance. *IEEE Transactions on Software Engineering* (2017).

[11] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. 1999. *Refactoring: Improving The Design Of Existing Code*.

[12] 2020 Refactoring Guru. 2020. https://refactoring.guru/.

[13] 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 ICPC*. Piscataway, NJ, USA.

[14] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2014. An Empirical Study of Refactoring Challenges and Benefits at Microsoft. *IEEE Transactions on Software Engineering* 40, 7 (2014), 633–649.

[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.

[16] Jonathan Lazar, Jinjuan Heidi Feng, and Harry Hochheiser. 2017. *Research methods in human-computer interaction*. Morgan Kaufmann.

[17] A MacCormack, J Rusnak, and C Baldwin. 2006. Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code. *Manage. Sci.* 52, 7 (2006), 1015–1030.

[18] E. Murphy-Hill, C. Parnin, and A. P. Black. 2012. How We Refactor, and How We Know It. *IEEE Transactions on Software Engineering* 38, 1 (2012), 5–18.

[19] Willian Oizumi, Leonardo Sousa, Anderson Oliveira, Luiz Carvalho, Alessandro Garcia, Thelma Colanzi, and Roberto Oliveira. 2019. On the density and diversity of degradation symptoms in refactored classes: A multi-case study. In *30th International Symposium on Software Reliability Engineering (ISSRE)*.

[20] Roberto Felicio Oliveira, Leonardo da Silva Sousa, Rafael Maiani de Mello, Natasha M. Costa Valentim, Adriana Lopes, Tayana Conte, Alessandro F. Garcia, Edson Cesar Cunha de Oliveira, and Carlos José Pereira de Lucena. 2017. Collaborative Identification of Code Smells: A Multi-Case Study. In *39th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice Track*.

[21] Roberto Felicio Oliveira, Rafael Maiani de Mello, Eduardo Fernandes, Alessandro Garcia, and Carlos Lucena. 2020. Collaborative or individual identification of code smells? On the effectiveness of novice and professional developers. *Inf. Softw. Technol.* 120 (2020).

[22] Ali Ouni, Marouane Kessentini, Mel Ó Cinnéide, Houari Sahraoui, Kalyanmoy Deb, and Katsuro Inoue. 2017. MORE: A multi-objective refactoring recommendation approach to introducing design patterns and fixing code smells. *Journal of Software: Evolution and Process* 29, 5 (2017), e1843.

[23] Replication Package. 2020. https://refactoringheuristics.github.io/.

[24] 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 ICSME*. 101–110.

[25] Fabio Palomba, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. 2017. An exploratory study on the relationship between changes and refactoring. In *2017 IEEE/ACM 25th ICPC*. IEEE, 176–185.

[26] Fabiano Pecorelli, Fabio Palomba, Foutse Khomh, and Andrea De Lucia. 2020. Developer-Driven Code Smell Prioritization. In *Proceedings of the 2020 MSR*.

[27] William R Shadish, Thomas D Cook, Donald Thomas Campbell, et al. 2002. *Experimental and quasi-experimental designs for generalized causal inference/William R. Shedish, Thomas D. Cook, Donald T. Campbell*. Boston: Houghton Mifflin,.

[28] Leonardo Sousa, Diego Cedrim, Alessandro Garcia, Willian Oizumi, Ana Carla Bibiano, Daniel Tenorio, Miryung Kim, and Anderson Oliveira. 2020. Characterizing and Identifying Composite Refactorings: Concepts, Heuristics and Patterns. In *17th International Conference on Mining Software Repositories (MSR)*.

[29] Leonardo Sousa, Willian Oizumi, Alessandro Garcia, Anderson Oliveira, Diego Cedrim, and Carlos Lucena. 2020. When Are Smells Indicators of Architectural Refactoring Opportunities? A Study of 50 Software Projects. In *28th International Conference on Program Comprehension (ICPC)*.

[30] Uchôa *et al.* 2020. How Does Modern Code Review Impact Software Design Degradation? An In-depth Empirical Study. In *36th ICSME*. 1 – 12.

[31] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. 2018. Ten years of JDeodorant: Lessons learned from the hunt for smells. In *2018 IEEE 25th SANER*. IEEE, 4–14.

[32] Nikolaos Tsantalis, Matin Mansouri, Laleh M. Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and Efficient Refactoring Detection in Commit History. In *Proceedings of the 40th ICSE*. ACM, New York, NY, USA, 483–494.

[33] 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 ICSE (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 403–414.

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

[35] Robert K Yin. 2015. *Qualitative research from start to finish*. Guilford publications.