A Quantitative Study on Characteristics and Effect of Batch Refactoring on Code Smells

Ana Carla Bibiano
Informatics Department
PUC-Rio
Rio de Janeiro, Brazil
abibiano@inf.puc-rio.br

Eduardo Fernandes
Informatics Department
PUC-Rio
Rio de Janeiro, Brazil
emfernandes@inf.puc-rio.br

Daniel Oliveira
Informatics Department
PUC-Rio
Rio de Janeiro, Brazil
doliveira@inf.puc-rio.br

Alessandro Garcia
Informatics Department
PUC-Rio
Rio de Janeiro, Brazil
afgarcia@inf.puc-rio.br

Marcos Kalinowski
Informatics Department
PUC-Rio
Rio de Janeiro, Brazil
kalinowski@inf.puc-rio.br

Baldoino Fonseca
Computing Institute
UFAL
Maceió, Brazil
baldoino@ic.ufal.br

Roberto Oliveira
Computing Institute
UEG Posse-GO
Posse, Brazil
roberto.oliveira@ueg.br

Anderson Oliveira
Informatics Department
PUC-Rio
Rio de Janeiro, Brazil
aoliveira@inf.puc-rio.br

Diego Cedrim
Amazon
São Paulo, Brazil
dccedrim@amazon.com

***Abstract*—Background: Code refactoring aims to improve code structures via code transformations. A single transformation rarely suffices to fully remove code smells that reveal poor code structures. Most transformations are applied in batches, i.e. sets of interrelated transformations, rather than in isolation. Nevertheless, empirical knowledge on batch application, or batch refactoring, is scarce. Such scarceness helps little to improve current refactoring practices. Aims: We analyzed 57 open and closed software projects. We aimed to understand batch application from two perspectives: characteristics that typically constitute a batch (e.g., the variety of transformation types employed), and the batch effect on smells. Method: We analyzed 19 smell types and 13 transformation types. We identified 4,607 batches, each applied by the same developer on the same code element (method or class); we expected to have batches whose transformations are closely interrelated. We computed (1) the frequency in which five batch characteristic manifest, (2) the probability of each batch characteristics to remove smells, and (3) the frequency in which batches introduce and remove smells. Results: Most batches are quite simple: although most batches are applied on more than one method (90%), they are usually composed of the same transformation type (72%) and only two transformations (57%). Batches applied on a single method are 2.6 times more prone to fully remove smells than batches affecting more than one method. Surprisingly, batches mostly ended up introducing (51%) or not fully removing (38%) smells. Conclusions: The batch simplicity suggests that developers have sub-explored the combinations of transformations within a batch. We summarized some batches that may fully remove smells, so that developers can incorporate them into current refactoring practices.**

***Index Terms*—code refactoring, code smell, quantitative study**

I. Introduction

Code refactoring consists of applying transformations on code structures aiming to improve them, thereby enhancing the software maintainability [1]. Major companies adopt refactoring in practice [2] [3]. Along with refactoring, developers apply one or more code transformations [4] [5]. An example of a frequent transformation type is Extract Method [5] [6]. Extract Method consists of extracting specific code statements from an existing method in order to create a new method [7]. Previous studies have largely investigated the refactoring effect on software maintainability [3] [4] [5] [8]. These studies often relied on characteristics that constitute each single code transformation, such as the transformation type.

Code transformations are eventually applied for removing poor code structures [3] [6] [9], which potentially harm the software maintainability [10] [11] [12]. These structures may be revealed by code smells [1] [13] [14]. A common smell type is Feature Envy, i.e., a method that partially "envies" the software features realized by another class rather than the method's host class [1] [15]. Hence, Feature Envy instances reflect a poor feature encapsulation across classes and indicate threats to maintainability [14]. It is advertised that refactoring can help remove code smells. However, a single transformation rarely suffices to fully remove a code smell [4] [16]. Each isolated transformation tends to either introduce (33%) or do not fully remove (57%) code smells [4]. Thus, improving the current refactoring practices towards a more effective removal of code smells requires understanding a wider and more complex phenomenon called *batch refactoring* [17].

Batch refactoring means applying two or more interrelated transformations in conjunction rather than a single transformation [18] [19] [20]. A set of interrelated code transformations is called a *batch* [5]. Five batch characteristics and their typical manifestations are discussed by past research: *cardinality* [18], *commit* [21], *scope* [3], *smelliness* [17], and *variety* [5]. Each characteristic can manifest in different ways in practice. Let us take *cardinality*, i.e., the number of transformations within a batch, as an example. It has two manifestations: 2 that represents the shortest set of interrelated transformations possible,

Funded by CNPq (grants 434969/2018-4, 427787/2018-1, 409536/2017-2, and 312149/2016-6), CAPES (175956), and FAPERJ (22520-7/2016).

and ≥ 3 that encompasses a plenty of different numbers of transformations. Batch characteristics and manifestations can help reasoning about the batch application and drawing strategies for successfully improving code structures. Indeed, fully removing code smells typically requires combining various transformations with different types [1] [4] [17].

Up to 60% of code transformations are applied in batches rather than in isolation [5]. Nevertheless, current empirical knowledge about batch refactoring is scarce. Previous studies (e.g., [3] [5] [19]) limit their discussions about batch characteristics based on the authors' assumptions without any empirical validation. Thus, questions such as *Which are the most frequent manifestations of batch characteristics in practice?* were not yet addressed. Even worse, there is scarce understanding of the batch effect on code smells in real software projects. Thus, it remains hard to reason about batches applied by developers in their daily work, but also enhance current refactoring practices (e.g., [19] [22]). A recent work [17] explored the overall effect of batches on code smells. However, the relationship of such effect and the batch characteristics remains unexplored.

In this paper, we present a large-scale quantitative study with 57 open and closed software projects. Our major goal was understanding the usual batch application from two perspectives: characteristics that typically constitute a batch, and the batch effect on smells. We analyzed 19 smell types and 13 transformation types. We relied on a heuristic proposed by a previous work [17] to identify 4,607 batches, each applied by the same developer on the same code element (method or class). The heuristic was carefully selected in order to provide us with batches whose transformations are closely interrelated. We computed (1) the frequency in which five batch characteristic manifest, (2) the probability of each batch characteristics to remove smells, and (3) the frequency in which batches introduce and remove smells. Hereafter we summarize our study findings:

- Most batches are quite simple: although most batches are applied on more than one method (90%), they are usually composed of the same transformation type (72%) and only two transformations (57%). The batch simplicity suggests that developers have sub-explored the combinations of transformations within a batch.
- Batches applied on *one method* are 2.6 times more prone to fully remove smells than batches affecting more than one method. It suggests that major modifications of code structures are more likely to harm the code maintainability than minor modifications.
- Surprisingly, batches mostly ended up introducing (51%) or not fully removing (38%) smells. Nevertheless, we have characterized some particular cases in which batches were able to at least partially improve code structures; some of these batches were not documented by the existing batch catalogs [1] [17].

II. Batch Refactoring at a Glance

Section II-A overviews code refactoring. Section II-B introduces batch refactoring. Section II-C discusses the possible relationship between batch refactoring and code smells.

A. Code Refactoring and Transformation Types

Code refactoring means applying transformations on code structures for enhancing software maintainability [3] [5]. A plenty of transformation types can be employed by developers along with refactoring [1]. Each transformation type defines how developers should modify certain code elements, such as methods and classes. Extract Method is an example of a transformation type popularly adopted by developers [6] [7]. This transformation type consists of extracting particular code statements of a method to create a new method. Extract Method can be used to separate the software features across methods of a software project [23]. Another example of a transformation type is Move Method [24], which consists of moving one method across classes. Move Method can be used to better encapsulate software features across classes [1].

Table I lists the 13 transformation types assessed by our work. The types include Extract Method, Move Method, and Inline Method, which are frequent in practice [5] [6]. The identification of code transformations in the 57 open and closed software projects was supported by the Refactoring Miner tool [6] [25]. The identified transformation types are varied by level of code element affected: three types affect locally an *attribute* (Lines 2 to 4); six types affect *methods* (Lines 5 to 10); and four types have a broader effect and modify *a class or interface* (Lines 11 to 14). In theory, these transformation may help to remove code smells. For instance, Extract Method may reduce too long methods [7], while Move Method may remove Feature Envy [1].

TABLE I
Transformation Types Investigated by This Work

Transformation Type	Definition
Move Attribute	Move an attribute across classes
Pull Up Attribute	Move attribute from child to parent class
Push Down Attribute	Move attribute from parent to child class
Extract Method	Extract a new method from an existing one
Inline Method	Move a method body to an existing method
Move Method	Move a method across classes
Pull Up Method	Move method from child to parent class
Push Down Method	Move method from parent to child class
Rename Method	Update name of an existing method
Extract Interface	Extract an interface from an existing class
Extract Superclass	Extract a superclass from an existing class
Move Class	Move a class across packages
Rename Class	Update name of an existing class

B. A Real Example of Batch Refactoring

Batch refactoring means applying two or more interrelated code transformations [18] [19] [20]. Each set of interrelated transformations composes a *batch*. We exemplify the batch application as follows. Let us consider the `SEach` class of the Elasticsearch open source software project. This class has poor code structures that are hard to read and modify.

Along the commit history of `SEach`, the class became particularly complex for developers to comprehend because of two problems. The major problem is that methods of this class implement software features that are more concerned about features provided by other classes, especially `Variables` and `MethodWriter`, than `SEach` itself. This problem can be revealed by the Feature Envy smell type [1]. In addition, certain methods are too long and complex by themselves, which can be revealed by the Long Method smell type [1].

Figure 1 illustrates a batch performed on the `SEach` class along three consecutive commits i, $i+1$, and $i+2$ (hashes 6dace47, 8db9a971, and b3804c4). In Commit i, three smell instances affected the `SEach` methods: while a Long Method instance affects the `analyze()` method, Feature Envy instances affect both the `analyze()` and `write()` methods. In Commit $i+1$, because `analyze()` is long and implements too many features, the developer applied two Extract Method transformations (Extract Method$_1$ and Extract Method$_2$), thereby creating the methods `analyzeArray()` and `analyzeIterable()`.

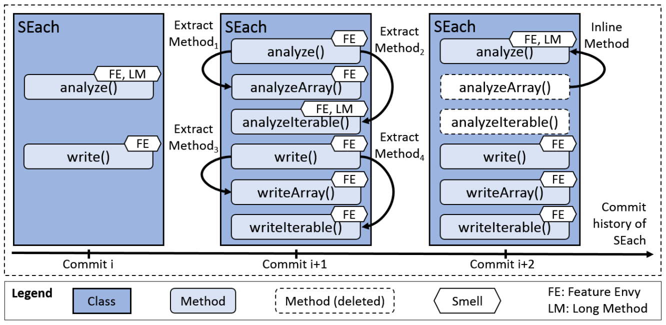

Fig. 1. Example of code transformations applied in a batch

Similarly, the developer applied two Extract Method transformations (Extract Method$_3$ and Extract Method$_4$) on `write()`, thereby creating the `writeArray()` and `writeIterable()` methods. Consequently, in Commit $i+1$, `SEach` obtained five new smells on extracting the methods: four new Feature Envy instances have emerged, thereby propagating the smelliness observed in Commit i, and one Long Method instance was introduced in `analyzeIterable()`. The Long Method instance in `analyze()` was removed.

In Commit $i+2$, the developer applied an Inline Method, thereby rejoining the features implemented by `analyze()` and `analyzeIterable()` in a single method. Differently from Extract Method, Inline Method removed a Feature Envy instance while reintroduced Long Method. Still regarding Commit $i+2$, the developer removed the `analyzeIterable()` method, thereby removing one Feature Envy instance and one Long Method instance. In summary, in order to improve code structures, the developer has applied the batch $b = \{$Extract Method$_1$, Extract Method$_2$, Extract Method$_3$, Extract Method$_4$, Inline Method$\}$. Indeed, all transformations applied on Commits $i+1$ and $i+2$ seem to interrelate towards supporting a common developer motivation: to properly distribute features across methods.

C. The Relationship of Code Smells and Batches

Code smells are typical symptoms of poor code structures affecting a software project [1] [10]. Each single instance of code smell may reveal, at least partially, potential harms to the software maintainability [11] [12] [26]. Code smells vary by the type of poor code structure revealed. An example of common smell type in real projects is Long Method [1] [15]. This smell type is realized by too long and complex methods that are typically hard to read and modify [13] [27]. Table II summarizes the 19 smell types analyzed in this work, which affect a variety of code elements: nine types affect code structures at the *method* level (Lines 2 to 10); and ten types affect *classes* (Lines 11 to 20).

TABLE II
SMELL TYPES ANALYZED THROUGHOUT THIS WORK

Smell Type	**Definition**
Brain Method	Method overloaded with software features
Dispersed Coupling	Method that calls too many methods
Divergent Change	Method that changes often when others change
Feature Envy	Method "envying" other classes' features
Intensive Coupling	Method that depends too much from a few others
Long Method	Too long and complex method
Long Parameter List	Too many parameters in a method
Message Chain	Too long chain of method calls
Shotgun Surgery	Method whose changes affect many methods
Brain Class	Class overloaded with software features
Class Data should be Private	Class that overexposes its attributes
Complex Class	Too complex software features into a class
Data Class	Only data management features into a class
God Class	Too many software features into a class
Large Class	Too large class
Lazy Class	Too short and simple class
Refused Bequest	Child class rarely uses parent class features
Spaghetti Code	Too much code deviation and nesting
Speculative Generality	Useless abstract class

The refactoring scenario of Figure 1 illustrates some insights about how batches affect code smells. The code transformations applied by the developer on both `analyze()` and `write()` methods suggest that the developer was clearly concerned about improving the code structure. In fact, the developer has tried to improve the structure of `SEach` in Commits i+1 and i+2. In order to achieve a proper separation of software features implemented by multiple methods of the same class, the developer has applied five code transformations in conjunction: four Extract Method instances and one Inline Method, plus a method deletion. Unfortunately, in Commit $i+2$, `SEach` contains one Long Method and four Feature Envy smell instances, which is two smell units higher than the total before the batch application (Commit i).

In this work, we hypothesize that some batches are able to partially improve a code structure. In the case of `SEach`, the introduction of Feature Envy instances in Commit $i+1$ was essential for better separating the software feature across methods. However, fully removing Feature Envy would require completing the batch with Move Method transformations applied on the "envious" methods [1] [17]. Additionally, we highlight that Long Method was reintroduced in `analyze()` after successive transformations. This scenario reveals an opportunity for guiding developers in applying batches.

III. Empirical Methodology

All study artifacts are available on our website: https://anacarlagb.github.io/esem2019-batch-refactoring.

A. Goal and Research Questions

According to the Goal Question Metric (GQM) template [28], our study goal is: *analyze* batch refactorings performed by developers on their software projects; *for the purpose of* understanding how batches have been applied and how they affect code smells; *with respect to* batch characteristics and the effect of batches on smell introduction and removal; *from the point of view of* software engineering researchers; *in the context of* 4,607 batch instances computed from 57 open or closed software projects implemented in the Java programming language. We designed two research questions.

RQ$_1$: *Which are the most frequent manifestations of batch characteristics in real software projects?* – The literature (e.g., [3] [5] [21]) is quite fragmented with respect to the definition of characteristics that probably constitute a batch. Previous studies often mention five basic characteristics of batches: *cardinality* [18], *commit* [21], *scope* [3], *smelliness* [17], and *variety* [5]. Table III summarizes these characteristics with their respective manifestations, as reported by previous studies. For instance, *variety* has two typical manifestations: *one type* of code transformation for the entire batch, and *many types* that vary by code transformation within a batch.

TABLE III
Typical Manifestations by Batch Characteristic

Charact.	Manifestation	Description
Cardinality	2	Batch has two transformations
	≥ 3	Batch has three or more transformations
Commit	*One commit*	Batch completed in only one commit
	Many commits	Batch completed in two or more commits
Scope	*Method*	Batch transformed only one method
	Class	Batch transformed multiple methods or class
Smelliness	*Smelly*	Batch affected smelly code element
	Not smelly	Batch did not affect smell code element
Variety	*One type*	Batch has only one transformation type
	Many types	Batch has two or more transformation types

There is still no consensus about which characteristics are relevant to understand the batch effect on code smells. This is because most of the previous studies did not empirically investigate the batch characteristics and their manifestations in real projects. Conversely, these studies mostly provide hints of batch characteristics based on the authors' assumptions and for convenience, e.g., to support a certain data analysis – as in the case of [3] and [5]. *How many code transformations usually constitute batches?* and *How many commits are often required to complete the batch application?* are examples of questions that were poorly addressed by previous studies. Thus, it remains hard to reason about applied batches.

A recent study [17] presented the first study aimed to investigate how batch characteristics manifest in real projects. The authors have investigated four out of the five characteristics selected for analysis in our study (Table III). The characteristics are *cardinality*, *commit*, *smelliness*, and *variety*. We have found the following opportunities to improve the previous investigation. Regarding *smelliness*, the past study assessed only if neutral batches affect the code smell types affecting the modified code structures. Differently, RQ$_1$ assesses how many batches occur in *smelly* code elements – i.e., code elements affected by smells either before or after the batch application. Additionally, RQ$_1$ assesses one additional characteristic (*scope*). Finally, we analyzed the frequency of manifestations by batch characteristics in two ways: each manifestation in isolation (similarly to the previous work [17]) and combined (details in Section V).

RQ$_2$: *How do batches affect code smells in real software projects?* – Code refactoring is usually associated with enhancements for software maintenance [4] [8] [29], and so is batch refactoring [17] [19] [20] [22]. Fowler's refactoring book [1] recommends some batches that may succeed in fully removing code smells. For instance, it is discussed that combining Extract Method with Move Method transformations can fully remove Feature Envy instances [1] [4]. Nevertheless, little empirical knowledge was acquired so far about the effectiveness of batches in fully removing code smells. As a consequence, it remains hard for researchers to recommend batches that can actually remove code smells without major side-effects on the code structure of a software project (e.g., the introduction of other code smell instances).

Previously [17], the batch effect was computed by the number of code smell instances the code elements had before and after the batch application. For instances, if a batch applied to a class *C* has decreased the number of code smells affecting *C*, the batch effect was positive; if the batch increased the number of code smells, the effect was negative; if the number of code smells did not change, the effect was neutral. However, there was a major limitation that should be addressed: the batch effect on code smells was not assessed in terms of the batch characteristics. Thus, some key questions remain without being addressed, such as *Which batch characteristics are more likely to be associated with the code smell removal?*

Through RQ$_1$, we explore a key batch characteristic called *variety*, which regards the variety of transformation types that constitute a batch. Analyzing the most frequent manifestations of *variety* in practice is a good starting point to understand the batch application. However, such analysis helps little to understand the actual contribution of the various existing transformation types (e.g., Extract Method and Move Method) on the code smell introduction and removal. Aiming at performing a more detailed analysis, we have designed RQ$_2$ to explore the batch effect on code smells based on the *nature* of code transformations that constitute a batch. Table IV lists the nature of all 13 transformation types of Table I. The first column defines each of the six natures; the second column lists the transformation types that correspond to each nature.

B. Study Steps

Step 1: Study Preparation. This step consisted of selecting software projects for analysis. We searched for open source software projects available online at GitHub, preferably developed in Java due to the wide popularity of this program-

TABLE IV
NATURE OF TRANSFORMATION TYPES

Nature	Definition	Transformation Types
Extraction	Extracting (parts of) an existing code element aimed to create a new element	Extract Interface Extract Method Extract Superclass
Inline	Copying a code element's body for pasting into an existing element	Inline Method
Motion	Moving a code element within the software project	Move Attribute Move Class Move Method
Pull Up	Moving a code element from subclass to superclass	Pull Up Attribute Pull Up Method
Push Down	Moving a code element from superclass to subclass	Push Down Attribute Push Down Method
Rename	Renaming a code element	Rename Class Rename Method

ming language (https://www.tiobe.com/tiobe-index/) and ease of study replication. We sorted all software projects by stars count, which has been useful for filtering active and popular projects [30]; we selected the top-100 projects. Thereafter, we filtered the software projects in order to keep those with at least 90% of source code written in Java. The resulting set of 54 open source projects sums up 13,400,686 Lines of Code (LOC) and 151,391 commits. Code refactoring is often studied in these projects [4] [8] [13] [29], e.g., Ant, ArgoUML, Elasticsearch, JUnit, and Spring Framework. Three anonymous companies provided us with one closed project each for analysis. These companies expected to contribute and improve their practices based on our findings.

Step 2: Transformation Detection. This step consisted of detecting all code transformation instances applied by developers in each of the 57 analyzed projects. Aimed to support the transformation detection, we have employed a tool called Refactoring Miner [6] [25]. This tool was designed to detect 13 transformation types (which are listed in Table I). The tool's accuracy is satisfactory: 93% of recall and 98% of precision [6] [25]. We collected a total of 24,893 transformations whose distribution by type is: mean = 1,560; minimum = 13; median = 838; and maximum = 6,992.

Step 3: Batch Computation. A recent study [17] introduced batch computation heuristics. Among the proposed heuristics, one of them stood out for its ability to capture fine-grained batches whose code transformations are closely interrelated: the *element-based heuristic*. For each code element, this heuristic analyzes the whole commit history of a software project. Based on the commit history of the code element, the heuristic involves computing all sets of two or more code transformations that satisfy two mandatory conditions: (1) all code transformations within the set were applied on exactly *one code element* – the element can be either a method or a class; and (2) all code transformations within the set were applied by exactly *one software developer*. From the 57 projects analysis, we collected 4,607 batches in total.

The chosen heuristic is quite restrictive with respect to the developer responsible for the batch application and the modified code element. One could argue that, in certain circumstances, the computed batches may be unrealistic once they are constituted of too many code transformations. Having this in mind, we also computed the distribution of number of transformations that constitute the batches and our results were encouraging: most batches (96%) are constituted by ten transformations at most; thus, only a few batches (4%) are likely to have unrealistic numbers of transformations. Additionally, the chosen heuristic does not capture batches that modify more than one class or a whole package. In this case, it is worth mentioning that we are concerned about transformation and smell types that affect either an attribute, a method, or a class (cf. Tables I and II).

Step 4: Smell Detection. This step consisted of computing code smell instances by software projects. We relied on the 19 smell types listed in Table II. For this purpose, we first computed static code metrics, such as Lines of Code (LOC) and Cyclomatic Complexity (CC) [31], supported by the Understand tool (https://scitools.com/features/). After that, we combined metrics in detection strategies [32] by smell type aimed to detect all code smells affecting each software project. The employed detection strategies of code smells were selected from previous studies [10] [15] [29]. These strategies were validated by previous work [29] with a resulting precision and recall [33] of 72% and 81%, respectively. We collected 41,398 code smell instances whose average number by smell type equals 2,435.

Step 5: Batch Effect Computation. We computed three types of batch effects on code smells. The *positive* effect means that the total number of code smell instances in the code elements affected by the batch has reduced after the batch application. Conversely, the *negative* effect the total number of code smell instances in the code elements affected by the batch has increased after the batch application. In the borderline, the *neutral* effect means that even if the code smell types affecting the refactored code has changed, the total number of code smell instances remained unaffected. We opted for an analysis of the code smell introduction and removal because, by definition, code transformations are designed for (at least partially) improving the code structure, thereby potential contributing to remove code smells. However, as typically happens with single transformations applied in isolation, batches could also tend to not remove code smell instances.

Step 6: Data Analysis. This step consisted of analyzing the collected data in order to address our $\mathbf{RQ}_s$. Aimed to answer $\mathbf{RQ}_1$, we have computed the frequency in which the five batch characteristics summarized in Table III manifest in practice. In order to answer $\mathbf{RQ}_2$, we performed two analyses: we computed the probability of each batch characteristic to remove smells; we also computed the the frequency in which batches introduce and remove code smells.

We computed the data distribution for the 57 projects analyzed with respect to four metrics. About the number of code transformations: one quarter of the projects have at least 376 transformations detected, which suggests a considerable refactoring activity across projects. About the number of batches: only a few batches were found in half of the projects (median = 15), but 25% of projects had at least 71 batches;

we found this result reasonable given the restrictiveness of our heuristic (see Step 3). About the ratio of batches by code transformations for each project: there seems to be a balance between the number of projects with high ratio and low ratio (minimum = 4, median = 24, and maximum = 33). About the number of commits performed during the life cycle of each project: although values tend to be low (median = 144), 25% of the projects have at least 790 commits.

IV. Manifestation of Batch Characteristics (RQ$_1$)

Table V summarizes the frequency of each manifestation by batch characteristic of Table III.

TABLE V
FREQUENCY OF MANIFESTATIONS BY BATCH CHARACTERISTIC

Characteristic	Manifestation	# Batches	Frequency
Cardinality	2	2,605	57%
	≥ 3	2,002	43%
Commit	*One commit*	4,265	93%
	Many commits	342	7%
Scope	*Method*	428	9%
	Class	4,179	91%
Smelliness	*Smelly*	2,911	63%
	Not smelly	1,696	37%
Variety	*One type*	3,330	72%
	Many types	1,277	28%

Characteristic 1: *Cardinality*. More than half of batches (57%) are constituted of exactly two code transformations. This result shows that most of the batches are not complex. While such simpler batches may eventually suffice to remove simpler smelly structures, they may not suffice to remove most of the smell types considered in our study. For instance, in order to fully remove a Large Class instance, developers may have to apply various Extract Class, Move Method, and Extract Interface transformations in conjunction [1]. With such simpler batches, developers may be sub-using the possible combinations of code transformations to remove certain smells.

However, we also found that a considerable amount of batches (43%) are constituted of three or more code transformations. We further analyzed these batches by computing the numerical distribution of their *cardinality*. The goal was to find frequent ranges of batches' *cardinality*. We have found that: 768 batches (17%) have *cardinality* = 3; 411 batches (9%) have *cardinality* = 4; 600 batches (13%) have cardinality from 5 to 10. In total, 1,779 out of the 4,607 batches (39%) range from three to ten code transformations. The remaining 4% represents batches with 10 or more transformations. On one hand, batches with more than two transformations are more likely to remove at a least a code smell. On the other hand, they may be affecting very complex smelly structures and, therefore, may not succeed in fully removing the smell.

Characteristic 2: *Commit*. Most batches (93%) required only *one commit* to be completed by the developer. Let us remind that our batch computation heuristic considers all transformations applied on a single code element as part of a batch. By combining this information with the high rates of batches completed in *one commit*, one could assume that commits often mark a change in the developers' motivation for code refactoring. In other words, it is very likely the transformations of each detected batch were cohesively related. It is also interesting to observe that, even in a single commit, there was a considerable proportion of batches with three or more transformations, as discussed above. As far as the time spent to complete batches is concerned: 32% of batches took just one day to be completed, which matches the fact that most batches are applied in one commit; 46% took more than one day and less than 30 days (one month); and only 23% took more than one month to be completed. Fortunately, only 5% of batches took more than 365 days (one year), which is low enough and suggest a certain soundness of our heuristic.

It may be the case that multiple developers work together to form a batch across commits or within the same commit – as in the case of batches composed along with code reviews [20]. This particular scenario is not captured by our heuristic, once all transformations have to be applied by the same developer. Anyway, the role of *commit* in marking a change in the developers' refactoring motivations should be further validated by interviewing actual developers in future studies.

Characteristics 3 and 4: *Scope* and *Variety*. Our study results suggest that most batches (91%) affect *multiple methods* into a class and/or the class itself. Only 9% of batches affect a *single method*. This result is somehow expected because many code smell types affect *multiple methods* of a class together. Thus, it is reasonable that batches are mostly constituted of transformations affecting various methods (or the entire class) together. As for the variety of transformation types, we observed that most batches (72%) consist of transformations with *one type*; still, 28% are batches consisting of *many types*. This result was unexpected because many batch recommendations [1] [4] [17] [22] depend on the combination of different transformation types for fully removing poor code structures, especially code smell instances.

Characteristic 5: *Smelliness*. Table V shows that the majority of the batches (63%) indeed affect smelly elements. In the next section, we will discuss whether these batches tend to remove or not those smells. Our results also show that 37% of batches occur in code elements not affected by code smells neither before nor after the batch application. This result can be explained by the fact that refactoring can be applied for other reasons such as enabling the addition of software features [34] and supporting bug fixes [16]. However, additional studies are necessary in order to confirm this observation, similarly to what has been done in the context of each single code transformation (e.g., [6] and [25]).

Finding 1: Developers are possibly sub-exploring batches to fully remove smells: 72% have one transformation type and 57% have only two transformations, while 43% have three or more transformations. Section V assesses whether such simple or complex batches remove smells.

TABLE VI
BATCH EFFECT ON CODE SMELLS (PLUS FISHER'S TEST) BY BATCH CHARACTERISTIC

Characteristic	Manifestation	Positive Absolute	Positive %	Negative Absolute	Negative %	Neutral Absolute	Neutral %	Odds Ratio (OR)	p-value
Cardinality	2	189	6%	847	29%	570	20%	0.90	> 0.05
	≥ 3	160	5%	647	22%	498	18%		
Commit	*One commit*	328	11%	1,414	49%	994	34%	0.92	> 0.05
	Many commits	21	1%	80	3%	74	4%		
Scope	*Method*	30	1%	52	2%	17	0%	2.60	< 0.05
	Class	319	11%	1,442	50%	1051	38%		
Variety	*One type*	240	8%	1,051	36%	749	26%	0.92	> 0.05
	Many types	109	4%	443	15%	319	12%		

V. BATCH EFFECT ON CODE SMELLS (RQ$_2$)

***Scope* is strongly related with code smell introduction.** We computed the relationship of batch characteristics and batch effect on code smells via Fisher's exact test [35] with confidence interval equals 95% (p-value < 0.05). Fisher's test computes the probability of a property (e.g., the manifestations of a batch characteristic) to co-occur with another property (e.g., the code smell introduction or removal). Table VI presents the test results by characteristic except *smelliness*. All table data regard the 2,911 batches categorized as *smelly* (Table V); thus, it does not make sense to assess the effect of *not smelly* batches (Table III). For each characteristic, we created a contingency table in which: the lines correspond to the possible manifestations (e.g., Lines 2 and 3); and the columns have the absolute numbers of positive and negative effect of batches on code smells (e.g., Columns 3 and 5). We then computed Fisher's test with each table as an input; two outputs are given: Odds Ratio (OR) [36] and p-value. Odds Ratio is the probability of a manifestation (e.g., *one commit* in Line 2) to remove code smells (Positive Absolute in Column 3) when compared to the same probability for the opposite manifestation (*many commits* in Line 3). Achieving a statistically significant OR requires a p-value < 0.05.

For the sake of illustration, let us consider the characteristic of *commit*. Batches applied in *one commit* are 92% more likely to have a positive effect on code smells than batches applied in *many commits*. In this case, the table indicates that statistical significance was not achieved by the *commit* characteristic. In fact, as the only batch characteristics whose OR is statistically significant is *scope*. In this particular case, we have found that batches applied on a *single method* are 260% more likely to have a positive effect on code smells when compared to batches applied at the *class* scope.

Finding 2: Batches at the *method* scope are 2.6 times more prone to fully remove smells than others. One could say that a greater number of code elements modified by a batch implies more complex smells to remove; thus, batches at *class* will probably introduce or not remove smells. However, many batches recommended to remove smells [1] are at *class* scope; it suggests that developers are either sub-using or misusing these recommendations.

Batches usually introduce rather than remove code smells. A previous work [17] has found that most code transformations, when analyzed in isolation, tend to either introduce (33%) or not fully remove (57%) code smells from software projects. However, analyzing the effect of each single transformation on code smells may not suffice for understanding the whole effect of code refactoring on the program comprehension. In the particular case of batch refactoring, we have observed a similar scenario. Our results indicate code smells were introduced by 1,494 (51%) out of the 2,911 *smelly* batches. Additionally, we have found that 1,068 (38%) of the *smelly* batches were not able to fully remove code smells.

We complemented our previous analysis by combining different manifestations to further investigate the batch effect on code smells. We categorized according to all possible combinations of manifestations for the four batch characteristics analyzed in Table VI. A combination follows this order of batch characteristics: *cardinality*, *commit*, *scope*, and *variety*. We aimed to identify combinations that are more prone to introduce or remove code smells. Table VII lists the top-five most frequent combinations based on the 2,911 batches categorized as *smelly* (Table V). We focused on the top-five combinations because, surprisingly, they sum up 2,701 out of the 2,911 batches (93%) categorized as *smelly*. The first column lists the combinations. The following columns present the absolute (Abs.) and relative (%) frequency of batches whose effect is negative, neutral, and positive.

TABLE VII
FIVE MOST FREQUENT COMBINATIONS OF MANIFESTATIONS

Combination	Negative Abs.	Negative %	Neutral Abs.	Neutral %	Positive Abs.	Positive %
2, *One commit*, *Class*, *One type*	**611**	**21**	418	14	123	4
$\geq$3, *One commit*, *Class*, *One type*	**366**	**13**	280	10	84	3
$\geq$3, *One commit*, *Class*, *Many types*	**230**	**8**	176	6	59	2
2, *One commit*, *Class*, *Many types*	**156**	**5**	110	4	35	1
2, *One commit*, *Method*, *One type*	**28**	**1**	4	0	21	1
Total	**1,391**	**48**	**988**	**34**	**322**	**11**

We have found that 2,347 out of 2,911 batches (81%) have either a negative or neutral effect on code smells. Curiously, 58% of these batches are composed by exactly *one transformation type*, while only 23% contain *many transformation types*. Additionally, batches that took only *one commit* to be completed, which are usually applied at the *class scope*, tend to introduce or not fully remove code smells. Even nontrivial batches, i.e., composed by more than one transformation type are more likely to introduce rather than fully remove code smells. This finding confirms our previous observation (Section IV) that developers may have poorly explored the combination of transformation types to remove code smells. Nevertheless, further investigation is required to understand which types of transformations have been usually employed in these ineffective batches. We address this gap in the following by investigating the nature of code transformations that constitute these batches. Table IV in Section III-A described the meaning of each nature considered in our study.

Finding 3: Batches tend to either introduce (51%) or not fully remove (38%) code smells. A previous study has shown that single transformations tend to introduce smells in 33% of the cases [4]. This result suggests that batch application often does not accomplish code structure improvements. This result also reinforces the need for guiding developers along with the batch application.

On the batch effect by nature of code transformations. Finally, we analyzed the batch effect based on the nature of code transformations that constitute a batch. We followed a three-step analysis procedure. **Step 1**: we grouped all 4,607 batches by nature of code transformations (Table IV). It is worth mentioning that a batch may be composed of one or more natures; cases of multiple natures occurring together in a single batch were grouped in isolation from the "pure" groups (with only one nature). **Step 2:** we computed how many code smells were either introduced or removed by group of batches. **Step 3:** we computed the rate of code smells introduced by group according to the formula $I(g) = \frac{i(g)}{i(g)+r(g)}$, where: $i(g)$ is the number of smell instances introduced by a group g, and $r(g)$ is the number of smell instances removed by g.

Table VIII summarizes the batch effect on code smells according to the nature of transformations. The table data consider all 4,607 batches regardless the manifestations of batch characteristics. In the second and third columns, no batch was counted for more than one category. The remaining columns present the $I(g)$ values by code smell type analyzed in this work. We marked with "*" all $I(g)$ values for which $i(g) + r(g) < 5$; we aimed to warn about $I(g)$ values computed on only a few smell instances (which may not be as relevant as values computed on many instances). Red-colored cells indicate $I(g) > 50$, i.e., groups of batches that tend to introduce rather than remove code smells. Green-colored cells indicate $I(g) < 50$, i.e., groups that tend to remove rather than introduce smells. White-colored cells indicate either $I(g) = 50\%$, i.e., groups of batches that introduce and remove smells in an equivalent rate, or values marked with "*" and, therefore, considered of little practical relevance. Due to space constraints in the paper, we display the results for ten out of the 19 code smell types under analysis (Table II). This decision was taken because the occurrence of many smell types is very rare, which made it unfeasible to compute $I(g)$ values. Batches were more likely to remove code smells for two smell types only: Feature Envy and Message Chain.

Regarding **Feature Envy**, batches of three natures (*Pull Up*, *Inline*, and *Inline/Motion*) removed rather than introduced smells. *Pull Up* batches (all with at least one Pull Up Method) removed 12 Feature Envy instances; the "envious" methods were possibly moved to the superclass so that the smell affecting the subclass vanished. *Inline* batches with only Inline Method transformations removed 39 Feature Envy instances. Our results are quite surprising because previous catalogs [1] [17] recommend *Extraction/Motion* batches to fully remove Feature Envy. An example is combining Extract Method and Move Method [1], which introduced 26 Feature Envy instances (contrary to expectations). Curiously, *Extraction/Motion* batches had $I(g) = 98.1$; thus, they rarely remove Feature Envy, at least as they have been applied so far. Two scenarios may justify this observation: (1) developers correctly applied Extract Method on the "envious" code, but they did not apply Move Method precisely on the extracted (and "envious") methods; (2) many "envious" methods were created via Extract Method but not all of them were moved. Note that batches with only Extract Method introduced 841 Feature Envy instances; it reinforces how important is to combine Extract Method with other transformation type to fully remove smells.

Regarding **Message Chain**, batches of four natures (*Motion*, *Pull Up*, *Extraction/Motion*, and *Motion/Rename*) removed rather than introduced smells. Our results are quite revealing, once Fowler's recommendation to fully remove Message Chain instances is limited to *Extract/Method* batches [1]. In the case reported by Fowler, the recommended batch can reduce a too long chain of method calls by moving methods that are closely related to the same class. As expected, batches of this nature were successful in fully removing Message Chain instances ($I(g) = 37.5$). As a complement, our results indicate alternative batches, such as those of *Motion/Renaming* nature. In this particular case, batches were able to remove 36 Message Chain instances; it is possible that methods within a chain of method calls was moved across classes, thereby reducing the chain size. Similar reasoning applies to *Pull Up* batches, which removed 5 Message Chain instances. This particular case is tricky: moving a method from subclass to superclass may have removed the smell instance from the subclass but introduced a smell instance in the superclass.

The success of some Fowler-recommended batches in removing smells. Besides the aforementioned batches recommended by Fowler's catalog [1] to fully remove Feature Envy and Message Chain instances, there are some other recommendations proposed in the same catalog. These recommendations target two code smell types: Data Class and Long Method. There are two batches recommended to remove Data Class: one of the Motion nature and other of the *Extraction/Motion* nature. Four out of the eight Data Class instances (50%) were removed through *Motion* batches, while one out of the eight instances (12%) was removed via a *Extraction/Motion* batch. However, it is worth mentioning that Data Class was too rare in the 57 analyzed projects for us to generalize

TABLE VIII
BATCH EFFECT ON CODE SMELLS ACCORDING TO THE NATURE OF CODE TRANSFORMATIONS

Nature of Code Transformations	**Total**		**Code Smell Type**									
	Abs.	**%**	**Class Data s/b Private**	**Complex Class**	**God Class**	**Intensive Coupling**	**Spaghetti Code**	**Speculative Generality**	**Feature Envy**	**Long Method**	**Long Param. List**	**Message Chain**
Extraction	1,449	31	100	78.7	92.3	87.1	83.2	100	83.8	61.4	95.1	82.2
Motion	1,142	25	98.5	85.5	89.3	*100	88.5	86.8	82.4	*100	*100	41.7
Rename	580	13	100	100	91.7	100	80	100	98.7	94.1	100	97.8
Pull Up	352	8	87.5	88.9	77.8	*100	57.1	95	47.8	*100	*33.3	28.6
Extraction/Inline	233	5	100	87.5	92.3	87.5	81	*100	84.8	88.2	81.3	80
Inline	221	5	100	73.9	*66.6	88.9	75.0	*100	45.1	60	72.2	76.5
Extraction/Motion	153	3	100	78.4	88.9	*100	94.4	*100	98.1	73.7	100	37.5
Extraction/Rename	105	2	100	92.6	100	*100	100	*100	87.5	87.5	100	100
Motion/Pull Up	84	2	*50	85.7	*50	*0	*50	100	*0	*0	*0	*0
Motion/Rename	59	1	100	100	*100	*50	*50	*0	77.3	*66.6	*0	37.2
Push Down	58	1	*100	75.0	*0	*0	*0	100	*0	*0	*0	*0
Inline/Motion	52	1	*100	75.0	*50	*100	*33.3	*0	35	66.7	40	*50
Extraction/Inline/Motion/Rename	28	1	100	100	*100	*100	*100	*0	100	100	*100	*100
Inline/Rename	25	1	*100	100	100	*100	*100	*100	*100	*100	*100	*0
Extraction/Pull Up	18	0	*0	*0	*100	*0	*0	*0	50.0	*0	*100	*0
Extraction/Push Down	11	0	*100	*66.6	*0	*0	*100	*100	80	*0	*0	*100
Extraction/Inline/Motion/Pull Up/Push Down/Rename	11	0	*0	*0	*0	*0	*0	*66.6	*66.6	*0	*100	*0
Inline/Pull Up	8	0	*0	*0	*0	*0	*0	*0	*0	*0	*0	*0
Pull Up/Rename	7	0	*0	*100	*0	*0	*0	*0	*0	*0	*0	*0
Pull Up/Push Down	6	0	*100	*0	*0	*0	*0	*100	*100	*0	*0	*0
Inline/Push Down	3	0	*0	*0	*0	*0	*0	*0	*20	*0	*0	*100
Push Down/Rename	2	0	*0	*0	*0	*0	*0	*0	*0	*0	*0	*0
All	**4,607**	**100**	**98.3**	**84.2**	**89.4**	**88.6**	**83.0**	**94.4**	**81.9**	**69.9**	**91.9**	**74.6**

the positive effect of both recommended batches on code smells. *Extraction* batches were recommended to remove Long Method. These batches were able to remove 95 out of the 124 Long Method instances (77%). Nevertheless, as pointed out by Table VIII, the majority of *Extraction* batches introduce rather than remove Long Method instances. This result may have been found because some Long Method instances are too long to fully remove via a few code transformations – note that most batches range from 2 to 10 transformations (Table V).

Finding 4: A half of batches are constituted of code transformations of either Extraction (31%) or Motion (25%) natures. Only 16% of batches combine transformations of different natures. We have found that batches of certain natures (such as *Pull Up*) usually remove two code smells: Feature Envy and Message Chain. These batches were not previously reported by catalogs such as Fowler's book [1]. Nevertheless, batches often end up introducing rather than removing smells, regardless the combination of natures.

VI. THREATS TO VALIDITY

Construct Validity. We carefully designed the study artifacts required to perform our quantitative data analysis (Sections IV and V). We collected five batch characteristics and manifestations often discussed by various studies [3] [5] [17] [18] [21]. Thus, we expected to understand how batches have been applied in practice from a considerable number of (eventually complementary) perspectives. We followed strict guidelines for collecting and filtering our 57 software projects before extracting data from them (Section III-B). Inspired by previous work [4] [6] [8] [30], we selected highly popular projects whose refactoring and maintenance activities are expected to reflect the current software industry.

We detected code transformation instances via the Refactoring Miner tool whose accuracy is high [6] [25]. Thus, we mitigated possible errors derived from a manual detection. The tool enabled us to detect 13 types that affect code structure at different levels (Table I). We computed software metrics from the Understand tool, which has been largely employed for research purposes [4] [8] [37]. The metrics were combined to detect code smells based on strategies proposed by the literature [10] [15] [29]; these strategies showed a high accuracy [29]. Thus, we expected to avoid the incorrect detection of code smell instances. The strategies enabled us to detect 19 different code smell types (Table II).

The accuracy of our batch computation heuristic (Section III-B) can be contested due to the lack of developer validation. Unfortunately, it is hard to capture developers' motivations behind each transformation and draw interrelations that form valid batches, specially in retrospective studies as ours. We overcame this difficulty by interrelating transformations applied by the same developer on the same code element. Our heuristic favors cases when each developer maintains a particular code part [38] [39]. Thus, the heuristic may suffice to capture batches at method and class scope but overlook two cases in special: when developers collaborate to compose a batch [20], and coarse-grained batches designed at the architecture level [22] [34]. Nevertheless, our heuristic sounds reasonable especially to understand the batch effect on code smells whose definitions target one method or class (Table II). We wrote Python scripts to automate the batch computation based on a heuristic cherry-picked from a previous work [17]. These scripts were validated by two of the paper authors.

Internal Validity. We payed special attention while collecting data based on the artifacts and instruments mentioned above. Two paper authors carefully read the full text of five studies [3] [5] [17] [18] [21] in order to extract the batch characteristics and manifestations. We discussed any divergences in a pair aimed to reach a consensus about which characteristics and manifestations were feasible to compute and analyze. The data of code transformations, batches, and code smells by software project were collected through a task force that counted on more than four paper authors. The batch instances were tabulated and doubled-validated by two paper authors. Thus, we aimed at mitigating threats regarding missing, invalid, and duplicated data; our procedures were essential to discard duplicates, for instance.

Conclusion Validity. We carefully performed our quantitative data analysis in two fronts: descriptive and statistical analysis. Regarding the descriptive analysis, two paper authors contributed to the frequency computation of batch characteristics by manifestations (Section IV describes our results). We then tabulated and plotted all data based on descriptive analysis guidelines for Software Engineering [40] [41]. Thus, we expected to apply the most appropriate techniques for aggregating and visually representing our data. For the statistical analysis, we relied on Fisher's test [35] plus an Odd Ration interpretation guideline from the literature [36]. All analysis results were double-checked by two paper authors aimed to mitigate biases and the misapplication of analysis procedures. We analyzed the batch effect on code smells by computing the number of smells before and after the batch application, similarly to a previous work [17]. This approach is limited because it overlooks eventual changes of code smell types caused by the application of neutral batches (Step 5 of Section III-B). However, it was discussed that only a few neutral batches (less than 1%) changed the smell types [17].

External Validity. In this work, we have investigated the code refactoring practices only on Java software projects. Because of that, our study results may be biased by the underlying code structure of Java-based projects. Although this threat remained not addressed, we highlight that Java is one of the most popular programming languages in both industry and academia (Section III-B). Additionally, although we have assessed both open and closed source projects, the number of closed source projects is quite low (only 3 projects) when compared to the number of open projects (the other 54 projects) and we did not conduct additional analyses applying the blocking principal to this aspect yet. Hence, collecting data from more (in particular closed source) projects and conducting such additional analyses is part of future work. Finally, the element-based heuristic [17] chosen to compute batches for analysis is quite limited. By considering code transformations performed by the same developers on the same code element (Step 3 of Section III-B), various refactoring practices may have been overlooked. This is the case of developers working collaboratively aimed to compose and apply batches [20] [34]. Although this threat was not address, we expect that the computed heuristics reflect at least partially the current refactoring practices in the industry.

VII. Final Remarks

Do our results discourage the application of complex batches? We found that batches at the method scope are more beneficial than those at the class scope, once they often remove smells. However, removing certain smell types, e.g., Feature Envy, requires applying several interrelated code transformations affecting much more than just one method [1] [4] [17]. Thus, developer may have to apply batches that are far more complex than those at the method level. Similar reasoning about the batch complexity may also apply to the batch cardinality, variety, and so forth. Although future work is still required to deeply understand the role of batch characteristics in removing code smells, our results (specially in Section V) suggest that slightly more complex batches could be able to fully remove code smell types such as Feature Envy.

Current refactoring tools enhanced to better support code smell removal. Our large-scale study derived some insights on how to improve current refactoring tools, e.g., Eclipse IDE, JDeodorant [23] and JMove [42]. Most tools guide the application of isolated transformations only. Thus, they end up providing little support to the batch application, especially with respect to (1) reasoning about how two or more transformations interrelate towards improving code structures and (2) supports the combination of multiple transformations aimed at the same refactoring motivation (e.g., fully removing a code smell instance). Batches are quite common in practice [3] [5] [17]; however, batches have been subexplored by developers according to our study findings. Thus, automated batch guidance is desired. Section V documents some batches that could be recommended to support the full removal of Feature Envy and Long Method, for instance. Further studies are required to understand the current batch refactoring practices. Thus, automated tools can cope with different developers' motivations behind batches, such as the addition of new program features [34].

Enhancing software architectures through batches. Some studies [22] [43] proposed approaches for enhancing software architectures through code refactoring. One particular work [22] introduced an interactive mechanism for composing batches so that a target, idealized architecture can be reached from an existing architecture. One could wonder that achieving such a target architecture may require the application of several interrelated code transformations (or batches). However, our study shows that most batches end up introducing or not fully removing code smells. This result becomes even more critical when considering that 91% of batches affect the *class* scope (Table V). This means that these batches may be often affecting the interfaces of those classes. Because various classes play a key role in a software architecture, and the unguided batch application may lead to poor code structures, we recommend researchers to investigate the batch effect on software architecture. Thus, the current enhancement of software architectures via refactoring could be boosted.

References

[1] M. Fowler, *Refactoring: Improving the design of existing code*, 1st ed. Addison-Wesley Professional, 1999.

[2] R. Potvin and J. Levenberg, "Why google stores billions of lines of code in a single repository," *Comm. ACM*, vol. 59, no. 7, pp. 78–87, 2016.

[3] M. Kim, T. Zimmermann, and N. Nagappan, "An empirical study of refactoring: Challenges and benefits at Microsoft," *IEEE Trans. Softw. Eng. (TSE)*, vol. 40, no. 7, pp. 633–649, 2014.

[4] D. Cedrim, A. Garcia, M. Mongiovi, R. Gheyi, L. Sousa, R. de Mello, B. Fonseca, M. Ribeiro, and A. Chávez, "Understanding the impact of refactoring on smells: A longitudinal study of 23 software projects," in *11th Symposium on the Foundations of Software Engineering (FSE)*, 2017, pp. 465–475.

[5] E. Murphy-Hill, C. Parnin, and A. Black, "How we refactor, and how we know it," *IEEE Trans. Softw. Eng. (TSE)*, vol. 38, no. 1, pp. 5–18, 2012.

[6] D. Silva, N. Tsantalis, and M. T. Valente, "Why we refactor? Confessions of GitHub contributors," in *24th International Symposium on the Foundations of Software Engineering (FSE)*, 2016, pp. 858–870.

[7] N. Tsantalis and A. Chatzigeorgiou, "Identification of extract method refactoring opportunities for the decomposition of methods," *J. Syst. Softw. (JSS)*, vol. 84, no. 10, pp. 1757–1782, 2011.

[8] A. Chávez, I. Ferreira, E. Fernandes, D. Cedrim, and A. Garcia, "How does refactoring affect internal quality attributes? A multi-project study," in *31st Brazilian Symposium on Software Engineering (SBES)*, 2017, pp. 74–83.

[9] R. Peters and A. Zaidman, "Evaluating the lifespan of code smells using software repository mining," in *16th European Conference on Software Maintenance and Reengineering (CSMR)*, 2012, pp. 411–416.

[10] I. Macia, R. Arcoverde, A. Garcia, C. Chavez, and A. von Staa, "On the relevance of code anomalies for identifying architecture degradation symptoms," in *16th European Conference on Software Maintenance and Reengineering (CSMR)*, 2012, pp. 277–286.

[11] W. Oizumi, A. Garcia, L. Sousa, B. Cafeo, and Y. Zhao, "Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems," in *38th International Conference on Software Engineering (ICSE)*, 2016, pp. 440–451.

[12] E. Fernandes, G. Vale, L. Sousa, E. Figueiredo, A. Garcia, and J. Lee, "No code anomaly is an island: Anomaly agglomeration as sign of product line instabilities," in *16th International Conference on Software Reuse (ICSR)*, 2017, pp. 48–64.

[13] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, and A. D. Lucia, "Do they really smell bad? A study on developers' perception of bad code smells," in *30th International Conference on Software Maintenance and Evolution (ICSME)*, 2014, pp. 101–110.

[14] A. Yamashita and L. Moonen, "Exploring the impact of inter-smell relations on software maintainability: An empirical study," in *35th International Conference on Software Engineering (ICSE)*, 2013, pp. 682–691.

[15] M. Lanza and R. Marinescu, *Object-oriented Metrics in Practice*, 1st ed. Springer Science & Business Media, 2006.

[16] G. Bavota, B. De Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, and O. Strollo, "When does a refactoring induce bugs? An empirical study," in *12th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2012, pp. 104–113.

[17] D. Cedrim, "Understanding and improving batch refactoring in software systems," Ph.D. dissertation, DI/PUC-Rio, Brazil, 2018.

[18] P. Meananeatra, "Identifying refactoring sequences for improving software maintainability," in *27th International Conference on Automated Software Engineering (ASE)*, 2012, pp. 406–409.

[19] G. Szőke, C. Nagy, L. Fülöp, R. Ferenc, and T. Gyimóthy, "FaultBuster: An automatic code smell refactoring toolset," in *15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2015, pp. 253–258.

[20] E. Fernandes, A. Uchôa, A. C. Bibiano, and A. Garcia, "On the alternatives for composing batch refactoring," in *3rd International Workshop on Refactoring (IWoR)*, 2019, pp. 1–8.

[21] G. Szoke, C. Nagy, R. Ferenc, and T. Gyimóthy, "Designing and developing automated refactoring transformations: An experience report," in *23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2016, pp. 693–697.

[22] Y. Lin, X. Peng, Y. Cai, D. Dig, D. Zheng, and W. Zhao, "Interactive and guided architectural refactoring with search-based recommendation," in *24th International Symposium on the Foundations of Software Engineering (FSE)*, 2016, pp. 535–546.

[23] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "Ten years of JDeodorant: Lessons learned from the hunt for smells," in *25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 4–14.

[24] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Trans. Softw. Eng. (TSE)*, vol. 35, no. 3, pp. 347–367, 2009.

[25] N. Tsantalis, V. Guana, E. Stroulia, and A. Hindle, "A multidimensional empirical study on refactoring activity," in *23rd Annual International Conference on Computer Science and Software Engineering (CASCON)*, 2013, pp. 132–146.

[26] L. Sousa, A. Oliveira, W. Oizumi, S. Barbosa, A. Garcia, J. Lee, M. Kalinowski, R. de Mello, B. Fonseca, R. Oliveira, C. Lucena, and R. Paes, "Identifying design problems in the source code: A grounded theory," in *40th International Conference on Software Engineering (ICSE)*, 2018, pp. 921–931.

[27] A. Yamashita and L. Moonen, "Do developers care about code smells? An exploratory survey," in *20th Working Conference on Reverse Engineering (WCRE)*, 2013, pp. 242–251.

[28] V. Basili and D. Rombach, "The TAME project: Towards improvement-oriented software environments," *IEEE Trans. Softw. Eng. (TSE)*, vol. 14, no. 6, pp. 758–773, 1988.

[29] G. Bavota, A. D. Lucia, M. D. Penta, R. Oliveto, and F. Palomba, "An experimental investigation on the innate relationship between quality and refactoring," *J. Syst. Softw. (JSS)*, vol. 107, pp. 1–14, 2015.

[30] H. Borges and M. T. Valente, "What's in a GitHub star? Understanding repository starring practices in a social coding platform," *J. Syst. Softw. (JSS)*, vol. 146, pp. 112–129, 2018.

[31] T. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng. (TSE)*, vol. SE-2, no. 4, pp. 308–320, 1976.

[32] E. Fernandes, P. Souza, K. Ferreira, M. Bigonha, and E. Figueiredo, "Detection strategies for modularity anomalies: An evaluation with software product lines," in *14th International Conference on Information Technology: New Generations (ITNG)*, 2018, pp. 565–570.

[33] C. Goutte and E. Gaussier, "A probabilistic interpretation of precision, recall and F-score, with implication for evaluation," in *27th European Conference on Information Retrieval (ECIR)*, 2005, pp. 345–359.

[34] E. Fernandes, "Stuck in the middle: Removing obstacles to new program features through batch refactoring," in *41st International Conference on Software Engineereng (ICSE): Doctoral Symposium*, 2019, pp. 1–4.

[35] R. Fisher, "On the interpretation of χ^2 from contingency tables, and the calculation of p," *J. Royal Stat. Soc.*, vol. 85, no. 1, pp. 87–94, 1922.

[36] J. Bland and D. Altman, "The odds ratio," *Br. Med. J. (BMJ)*, vol. 320, no. 7247, p. 1468, 2000.

[37] I. Ferreira, E. Fernandes, D. Cedrim, A. Uchôa, A. C. Bibiano, A. Garcia, J. L. Correia, F. Santos, G. Nunes, C. Barbosa, B. Fonseca, and R. de Mello, "The buggy side of code refactoring: Understanding the relationship between refactorings and bugs," in *40th International Conference on Software Engineering (ICSE): Poster Track*, 2018, pp. 406–407.

[38] C. Bird, N. Nagappan, P. Devanbu, H. Gall, and B. Murphy, "Does distributed development affect software quality? An empirical case study of Windows Vista," in *31st International Conference on Software Engineering (ICSE)*, 2009, pp. 518–528.

[39] F. Rahman and P. Devanbu, "Ownership, experience and defects: A fine-grained study of authorship," in *33rd International Conference on Software Engineering (ICSE)*, 2011, pp. 491–500.

[40] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*, 1st ed. Springer Science & Business Media, 2012.

[41] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empir. Softw. Eng. (EMSE)*, vol. 14, no. 2, p. 131, 2009.

[42] R. Terra, M. T. Valente, S. Miranda, and V. Sales, "JMove: A novel heuristic and tool to detect move method refactoring opportunities," *J. Syst. Softw. (JSS)*, vol. 138, pp. 19–36, 2018.

[43] R. Terra, M. T. Valente, K. Czarnecki, and R. Bigonha, "Recommending refactorings to reverse software architecture erosion," in *16th European Conference on Software Maintenance and Reengineering (CSMR)*, 2012, pp. 335–340.