How Does Incomplete Composite Refactoring Affect Internal Quality Attributes?

Ana Carla Bibiano, Vinicius Soares
PUC-Rio, Rio de Janeiro, Brazil
[abibiano,vsoares]@inf.puc-rio.br

Daniel Coutinho, Eduardo Fernandes
PUC-Rio, Rio de Janeiro, Brazil
[dcoutinho,emfernandes]@inf.puc-rio.br

João Lucas Correia
UFAL, Maceió, Brazil
jlmc@ic.ufal.br

Kleber Santos
UFCG, Campina Grande, Brazil
klebersantos@ufcg.edu.br

Anderson Oliveira, Alessandro Garcia
PUC-Rio, Rio de Janeiro, Brazil
[aoliveira,afgarcia]@inf.puc-rio.br

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

Baldoino Fonseca, Márcio Ribeiro
UFAL, Maceió, Brazil
[baldoino,marcio]@ic.ufal.br

Caio Barbosa, Daniel Oliveira
PUC-Rio, Rio de Janeiro, Brazil
[csilva,doliveira]@inf.puc-rio.br

ABSTRACT

Program refactoring consists of code changes applied to improve the internal structure of a program and, as a consequence, its comprehensibility. Recent studies indicate that developers often perform composite refactorings, i.e., a set of two or more interrelated single refactorings. Recent studies also recommend certain patterns of composite refactorings to fully remove poor code structures, *i.e*, code smells, thus further improving the program comprehension. However, other recent studies report that composite refactorings often fail to fully remove code smells. Given their failure to achieve this purpose, these composite refactorings are considered incomplete, *i.e*, they are not able to entirely remove a smelly structure. Unfortunately, there is no study providing an in-depth analysis of the incompleteness nature of many composites and their possibly partial impact on improving, maybe decreasing, internal quality attributes. This paper identifies the most common forms of incomplete composites, and their effect on quality attributes, such as coupling and cohesion, which are known to have an impact on program comprehension. We analyzed 353 incomplete composite refactorings in 5 software projects, two common code smells (Feature Envy and God Class), and four internal quality attributes. Our results reveal that incomplete composite refactorings with at least one Extract Method are often (71%) applied without Move Methods on smelly classes. We have also found that most incomplete composite refactorings (58%) tended to at least maintain the internal structural quality of smelly classes, thereby not causing more harm to program comprehension. We also discuss the implications of our findings to the research and practice of composite refactoring.

CCS CONCEPTS

• **Software and its engineering** → **Software maintenance tools**; **Software evolution**.

KEYWORDS

Code refactoring, composite refactoring, incomplete composite, code smell, internal quality attribute, code metric, quantitative study

ACM Reference Format:
Ana Carla Bibiano, Vinicius Soares, Daniel Coutinho, Eduardo Fernandes, João Lucas Correia, Kleber Santos, Anderson Oliveira, Alessandro Garcia, Rohit Gheyi, Baldoino Fonseca, Márcio Ribeiro, and Caio Barbosa, Daniel Oliveira. 2020. How Does Incomplete Composite Refactoring Affect Internal Quality Attributes?. In *28th International Conference on Program Comprehension (ICPC '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3387904.3389264

1 INTRODUCTION

Code refactoring [21] is one of the most popular techniques to improve the internal code structure and, consequently, the comprehensibility of a program [21, 25]. Each single *refactoring* is an instance of a refactoring *type*. Each type determines the changes required to produce an expected enhancement of a certain code structure [21]. Examples of popular refactoring types include Extract Method and Move Method [27, 37]. Like other types, they are expected to contribute to fully remove poor code structures [5, 7], such as *code smells* [20, 47].

However, given its fine-grained nature, a single refactoring rarely suffices to assist developers in achieving their intents, *e.g.* to fully remove a poor code structure [5, 7] such as *code smells* [20, 47]. The removal of some code smells are considered highly relevant by developers and practitioners. This is the case of *God Class* and *Feature*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICPC '20, October 5–6, 2020, Seoul, Republic of Korea
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7958-8/20/05…$15.00
https://doi.org/10.1145/3387904.3389264

Envy [23, 37, 39, 46] smells as they have a wide, harmful impact in the program structure; both affect two or more classes.

Notwithstanding, previous studies have reported that single refactorings often do not remove or even introduce these code smells [4, 7]. Nevertheless, they provide little or no insight on a wider and more complex phenomenon called *composite refactoring* [5, 9, 38]. Composite refactoring occurs when two or more interrelated single refactorings are applied on one or multiple code elements [5, 18, 33, 38, 41]. This phenomenon happens frequently in software projects [5, 27], and each set of interrelated single refactorings is called a *composite*.

Previous studies recommend specific patterns of composites to remove certain code smells [5, 21]. For example, Fowler recommends the application of various *Move Methods* together to fully remove a *God Class*. However, empirical studies report that developers often fall short in fully removing those code smells through composites. Indeed, most composites either introduce or not fully remove code smell instances [5]. This can be related to the fact that developers often apply composites alongside other code changes [16, 27], and frequently these composites are applied to perform development activities that do not purely affect the code structure, *e.g* a feature addition. Besides that, developers may not be applying the recommended composites to remove the code smells. The literature suggests that developers fail in removing code smells because some of the recommended refactorings are missing within the composites [5, 7]. The lack of one or more refactorings in a composite, to remove a particular smell type, constitutes an *incomplete composite refactoring* (shortly called incomplete composite). It is expected that incomplete composites can gradually improve the internal structure quality, improving also the program comprehension.

The existing refactoring tools offer little help in assisting the completion of incomplete composites by providing the refactorings needed to fully remove remaining code smells [24, 26, 40, 42]. Designing tools for providing such assistance requires proper empirical investigation. This investigation includes characterizing the most frequent types of incomplete composites applied to real programs. It also includes understanding how incomplete composites gradually affect the internal quality attributes when compared to code smells. The internal quality attributes are often used to detect problematic microstructures of source code, which are known to harm program comprehensibility [1, 8]. For example, increasing code complexity is highly related to low program comprehensibility. However, previous studies on composites in real projects did not investigate the effect of incomplete composites on internal quality attributes [5, 6].

Based on these limitations, this paper presents a quantitative study aimed at addressing the aforementioned literature gap. Our goal is to understand *the most common incomplete composites* and *how incomplete composite refactoring affects internal quality attributes*. We selected five software projects of different domains and targeted 34 popular refactoring types [27, 37]. We then collected 353 (47%) incomplete composites for *Feature Envy* removal or *God Class* removal. We then computed the frequency of incomplete composites according to the refactoring types constituting each composite. We evaluated the effect of those incomplete composites on 11 code metrics that are used to capture four internal quality attributes [8]. Hereafter we present our main findings and an overview of possible implications:

1. Composites often affect the structure of two or more classes; most of the incomplete composites with such a wide scope (82%) are composed of multiple refactoring types. This observation contradicts findings from recent studies [5, 6], which analyzed a much smaller set of refactoring types than the one considered in our study. Given the type heterogeneity and the wide scope of the aforementioned composites, one could expect they would often have a positive effect on multiple internal attributes, including the cohesion and coupling of the affected classes. However, this scenario was often not the case possibly because such heterogeneous composites are hard to apply properly.

2. Incomplete composites with at least one Extract Method often (71%) and without Move Methods are often the reason why Feature Envies and God Classes are not resolved. These results may indicate that the classes affected by such incomplete refactorings are likely still hard to comprehend as the key underlying problem (i.e., lack of separation of concerns) remains. In fact, we observed that most of these cases did not result in coupling and cohesion improvement. This implies that automated refactoring tools could be extended to identify opportunities for recommending the completion of otherwise harmful incomplete composites.

3. Most incomplete composites (58%) tend to not change the internal quality attributes on smelly classes. In a way, one could consider this fact problematic. However, this finding reveals that despite the incomplete composites not fully removing code smells, they maintain the structural internal quality of the affected classes. At least, the incomplete nature of composites has possibly not harmed even further the program comprehensibility and other related quality attributes. This observation suggests that certain developers may be keen to maintain the structural quality of their programs through refactoring, even when they do not have the explicit intent of doing so. Thus, they might also be open to receive additional refactoring recommendations to help them improve the program structure even further, while achieving their primary goals. Recommender systems could be designed, then, to assist developers in "completing" their composite refactorings, while also favoring the achievement of their other goals.

2 BACKGROUND

This section presents the main concepts for this study. Program refactoring is the process of performing changes that aim to improve the internal code structure of a program [21]. The literature presents catalogs of *refactoring types* [21], and an example of one refactoring type is *Extract Method*, which is when a part of the source code is extracted from an existing method to a new method.

2.1 Composite Refactoring (or Composite)

In the context of this work, a *composite refactoring* (or shortly, a composite) consists of two or more interrelated single refactorings applied by the same developer to one or more code elements. In fact, Bibiano *et.* al. [5] have shown that interrelated single refactorings usually are applied by the same developer [5]. The literature presents some heuristics to identify composites. A recent study proposed a *range-based* heuristic, in order to detect each composite

formed by single refactorings that are structurally interrelated [38]. It groups single refactorings that have the following characteristics: i) at least one code element was affected by all refactorings in the composite, and; ii) they were applied by the same software developer, and relies on the fact that composite refactorings often have those two characteristics [5]. This heuristic is different from the *element-based* heuristic proposed by another previous study [5]. The *range-based* heuristic captures the source and target classes to which the single refactorings were applied in the composites. However, the *element-based* heuristic limits its scope to the source class only. The *range-based* heuristic was designed according to developers' practices, as a previous study [6] has shown that composites are often applied to multiple classes.

2.2 Incomplete Composite Refactoring: A Smell Removal Perspective

Recommended Composite Refactoring for Code Smell Removal. The literature recommends the application of certain composites to remove specific code smell types [5, 21]. Fowler [21] recommended composites for code smells such as *Feature Envy* and *God Class*. Also, Bibiano *et* al. [5] have presented recommendations of composites for those code smells, observing that Fowler's suggestions are not usually applied. They also found that developers often combine the recommended refactoring types with other refactoring types in composites to remove code smells. For example, to remove Feature Envy, developers combined *Extract Method*, *Move Method* and *Move Attribute*.

Incomplete Composite Refactoring for Code Smell Removal. Bibiano *et* al. [5] observed that composites often failed to remove code smells, especially *Feature Envy* and *God Class*. This is expected because these code smells are regularly in classes in which other code changes happen frequently, such as a feature addition [34, 44, 46], thus, making the removal of the code smell more challenging. We call *incomplete composite refactorings* (or shortly *incomplete composites*), composites that contain at least one recommended refactoring type that is used to remove a particular code smell, but failed to remove that code smell after its application. For example, Fowler recommended composites consisting of *Extract Methods* and *Move Methods* to remove *Feature Envy* [21]. In that case, a composite is an incomplete composite if: (i) the composite has at least one refactoring type is recommended to remove Feature Envy [5, 21] (at least one *Extract Method* or one *Move Method*), and (ii) the composite did not remove the *Feature Envy*.

2.3 Motivating Example

This section describes an example of an incomplete composite for Feature Envy removal in a real software project. Figure 1 presents an incomplete composite that was applied in the commit `66fbd3202a` [15] from the Dubbo project. In this commit, the `ServiceConfig` class has an envious method called `getExportedUrl`. This method calls several times methods of the `AbstractInterfaceConfig` and the `ReferenceConfig` classes. Possibly aiming to solve this, the developer applied a `Move Attribute`, moving the `url` attribute to the `ReferenceConfig` class. Then, the developer moved the envious `getExportedUrl` method to the `AbstractInterfaceConfig` class. However, the `getExportedUrl` method continues envious, because this method has several calls to the `ReferenceConfig` class.

The developer applied a composite refactoring composed of one *Move Attribute*, and one *Move Method*. This composite was applied to a class that has an envious method (the `getExportedUrl` method). However, this composite did not remove the *Feature Envy* code smell completely, because this method continues to have calls to `ReferenceConfig`. Previous studies presented recommendations of composites to remove this code smell [5, 7, 21].

This composite from the Dubbo project is an *incomplete composite refactoring* for this case of *Feature Envy* removal, because this composite has at least one recommended refactoring type to remove this code smell (the *Move Method* refactoring type by Fowler's recommendation) [21], and yet this composite did not entirely remove the code smell (see Section 2.2). The developer could have "completed" this composite applying more *Extract Methods* and *Move Methods* on the `getExportedUrl` method.

Through this example, we observed the existing limitations on the effect of incomplete composites on the internal structure quality. A recent study on the effect of composites was limited on evaluating the effect of this example on the internal structure quality, because they only observed the effect on the code smell removal [5]. This study would also have concluded that this composite does not remove the code smell, however, this incomplete composite has affected some code metrics that capture one or more internal quality attributes, independently if the code smell was not removed. For example, the number of lines of code in the `ServiceConfig` has decreased, improving the coupling and cohesion of this class. The code metrics related to the cohesion and coupling in the `ReferenceConfig` and `ServiceConfig` are not changed significantly. Thus, the incomplete composite remains the internal structure quality of the `ReferenceConfig` and `ServiceConfig` classes. This can be considered a positive result on the effect of this incomplete composite because regardless the non-removal of the *Feature Envy*, the incomplete composite does not worsen the internal quality attributes of the classes.

This example motivates that existing studies offer a limited knowledge on the effect of incomplete composites on internal quality attributes.The existing refactoring tools provide little help in assisting the completion of incomplete composites by providing those refactorings needed to fully remove remaining code smells [24, 26, 40, 42]. Designing tools for providing such assistance still requires further empirical investigation. This investigation includes characterizing the most frequent types of incomplete composites applied to real programs. More importantly, it includes understanding as to how incomplete composites gradually affect the internal quality attributes when compared to code smells. Although there are several works that study code smells [10, 22, 29–32, 45], which are intrinsically based on these internal attributes, they do not address the association of such smells and attributes with incomplete refactoring.

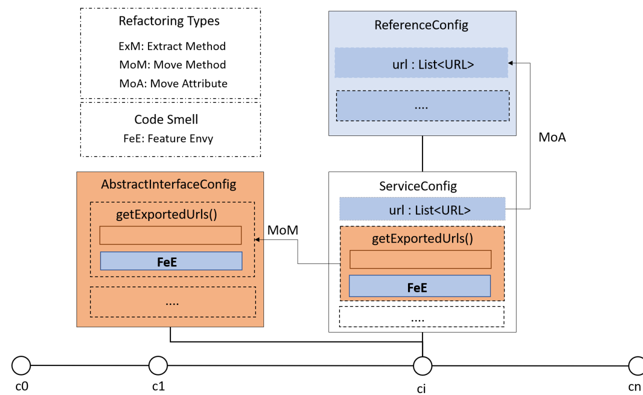

Figure 1: Incomplete composite for Feature Envy removal

3 STUDY SETTINGS

This section summarizes our empirical study settings as follows. Our companion research website provides the study artifacts for the complementary information: https://researcher-icpc-104.github.io/icpc2020_incomplete_composite.

3.1 Goal and Research Questions

Our study goal is based on the GQM methodology [3] to *analyze* incomplete composite refactorings applied to software projects by their developers; *for the purpose of* revealing the effect of incomplete composites on internal quality attributes; *with respect to* i) the most common compositions of single refactorings that constitute each incomplete composite instance, and ii) how frequently incomplete composites either improve, do not affect, or worsen each internal quality attribute; *in the context of* the life cycle of five Java open source software projects with active code refactoring practices. We carefully designed two Research Questions (RQs) aimed at achieving our study goal.

RQ$_1$: *What are the most common incomplete composites applied in real programs?* The literature reports that many types of composites can support the removal of the same particular smell type [5, 7, 21]. However, the literature related to the characterization of incomplete composites is scarce (Section 2). Thus, one needs to investigate and characterize recurring incomplete composites which, albeit they may often fail to remove a certain smell type, have the potential to remove it if complemented with other single refactorings. By doing so, we can understand how varied and frequent the manifestation of incomplete composites is in practice. Along with this, we also expect to reveal, even if partially, the required support for the completion of these composites, thus enabling the complete removal of the targeted smells. The results of this **RQ** may reveal common practices of composite refactoring that are likely to: (i) hamper full smell removals, or (ii) gradually improve the internal structural quality.

RQ$_2$: *How does incomplete composite refactoring affect internal quality attributes?* Most strategies for detecting code smells rely on the combination of code metrics [17], which capture the current state of various internal quality attributes. Thus, the degradation of these values may imply a degradation in the code's quality itself. Some early studies [4, 7] have already attempted to understand the effect of refactorings on code smells. However, due to the fine-grained code change caused by each single refactoring, it was expected that single refactorings rarely suffice to fully remove code smells [4]. Certain smell types, *e.g. God Class*, are too coarse-grained to be removed with a single refactoring, *e.g. Extract Method*. Surprisingly, a recent study [5] shows that, much like single refactorings, composite refactorings also rarely remove their targeted code smells.

Therefore, recent studies [1, 8] shifted from code smells to a more fine-grained perspective: internal quality attributes. These studies concluded that, although single refactorings rarely remove code smells, they can still have a positive effect on the internal quality attributes. For instance, *Extract Method* reduces the class' *complexity*, which is considered improvements. Nevertheless, no previous work has assessed the effect of composite refactorings on internal quality attributes. Thus, the answers to this research question can reveal if incomplete composites gradually improve the internal structure quality as expected, and what incomplete composites usually affect each internal quality attribute. These observations can generate insights for future studies and help designers of existing refactoring tools on improving their approaches for the removal of code smells and the improvement of internal quality attributes, based on real development practices.

3.2 Study Steps

Figure 2 illustrates our six study steps. We describe below each study step.

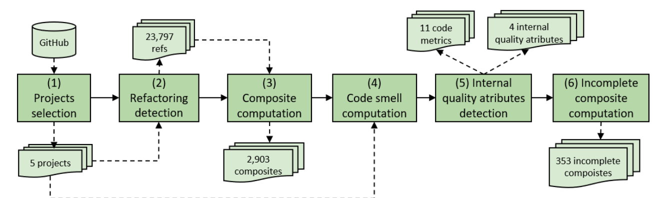

Figure 2: Study steps

Step 1: *Software Project Selection* – We relied on previous studies [1, 5–8, 37, 38] to derive three criteria for selecting software projects for analysis. (i) The software project must be open source and implemented using the Java programming language. Java is one of the most popular languages worldwide (https://www.tiobe.com/tiobe-index/). Open source projects were selected to support the study replication. (ii): The software project must use Git as the main version control system. This criterion aimed at supporting the use of state-of-the-art tools for refactoring detection that work on Git projects only. (iii) The software project must have been analyzed by one or more related studies regarding the refactoring [6, 8] and code smells [5, 7]. Thus, we could select projects that are known to have a considerable amount of refactorings and smell instances.

Step 2: *Single Refactoring Detection* – We used the RMiner tool for detecting single refactorings applied to each software project [1, 5, 7, 8] and this tool is available for the study replication. A recent

study presents that this tool presents a very high precision (98%) with a recall of (87%) [43]. We investigated 34 of the refactoring types detectable by the tool, by prioritizing the refactoring types related to our study scope. The complete list of the investigated refactoring types is available on our web site.

Step 3: *Composite Refactoring Computation* – We collected the composites using the *range-based* heuristic discussed in Section 2.1. This heuristic allowed us to analyze the composite refactoring effect encompassing from the source class to the target class associated with each single refactoring. Thus, we could identify those cases in which a smell instance was simply moved from one class to another rather than actually removed from the source code, and how the internal quality attributes are affected among these classes. Besides that, this heuristic is more conservative when capturing single refactorings that were applied on code elements that have interrelated code structures [38]. Thus, the composites detected by the *range-based* heuristic are more likely to encompass composites applied to remove a code smell that involved multiple classes. Therefore, this decision partially reduces the threat that composites may not have been applied with the intent of removing code smells.

Step 4: *Code Smell Detection* – We selected two code smell types: *Feature Envy* and *God Class* (as described in Table 2). These types were selected because the recommended composite refactorings for their removal is already known [5, 21]. They are also code smells that involve multiple classes, and can be related to various internal quality attributes such as cohesion and coupling. Thus, the incomplete composites applied on classes that have these smells (smelly classes) may have an effect on these internal quality attributes. Besides that, the *range-based* heuristic motivated us to investigate the effect of incomplete composites on code smells that also usually involve multiple classes. Moreover, previous studies have also investigated the effect of refactoring on these types of code smells [5, 7]. In terms of the tool used to detect these code smells, we selected the Organic tool [29, 30, 45], which uses strategies based on software metrics to collect the smells. This tool was selected due to its detection strategies, that use the code metrics we analyzed for evaluating the effect of incomplete composites. In addition, these detection strategies were already evaluated by previous studies [7, 19, 28].

Step 5: *Internal Quality Attribute Computation* – Table 1 presents the 11 code metrics [1, 8] that were investigated in this study. The columns present, respectively, the internal quality attributes related to each metric, the code metrics are collected, and their descriptions. These code metrics were selected due to them having been already evaluated for another study [8] for analyzing the effects of single refactorings on internal quality attributes. Thus, these code metrics can reveal the effect of incomplete composites on these internal quality attributes for the classes in which these incomplete composites were applied, because these code metrics are of class-level scope. We chose to perform the analysis in a class-level scope due to a recent study about the effect of composites on code smells presenting that composites are often applied at class level [5]. We then aimed to analyze the effect of composites on internal quality attributes for each class in which a composite was applied. We used an automated tool called SciTools Understand (https://scitools.com/) to collect these code metrics, as this tool also is used by other studies about refactoring and internal quality attributes [7, 8].

Table 1: Code Metrics by Internal Quality Attributes

Internal Quality Attribute	Code Metric	Description
Cohesion	LCOM2	Number of pairs of methods that do not share attributes, minus the number of pairs of methods that share attributes
Coupling	CBO	The number of classes to which a class is coupled
Complexity	MAxNEST	Maximum nesting level of control constructs
	CC	Measure of the complexity of a module's decision structure
	WMC	The sum of Cyclomatic Complexity of all methods declared in the given class
Size	LOC	The number of lines of code in the class excluding whitespaces and comments
	CLOC	Number of lines in the class containing code comments
	STMTC	Number of statements in the class's code
	NIV	Number of instance variables in the class
	NIM	Number of instance methods in the class

Step 6: *Incomplete Composite Computation* – We investigated the incomplete composites for the *Feature Envy* removal and *God Class* removal. For this study, a composite was considered incomplete according to the following criteria: (i) composites that have at least one refactoring type that is recommended to remove a *Feature Envy* or a *God Class* [5, 21], and (ii) composites that did not remove a *Feature Envy* and a *God Class*. We have filtered the incomplete composites through composites that have at least one *Extract Method* or one *Move Method* (refactoring types recommended to remove these code smells). These composites are the candidates (Candidate is a composite that have at least one Extract Method or Move Method.) of incomplete composite. We then elaborated Table 2 that presents the recommended composites for *Feature Envy* and *God Class* removal according to the literature, the code smell type and their description, and the incomplete composites for each recommended composites.

Table 2: Inc. Comp. for Feature Envy and God Class Removal

Recommended Composite	Code Smell	Description	Incomplete Composite
Extract Methods and Move Methods [5] [21]	Feature Envy	A method more interested in other class(es)	Extract Method{n} Move Method{n} Extract Method{n}, Move Method{n}
Move Methods [21]	God Class	A class that implements too many software features	Move Methods{n}

4 DATASET OVERVIEW

This section presents an overview of our dataset of incomplete composites.

4.1 Incomplete Composite Dataset

Our dataset provides 23,797 single refactorings, and 2,903 composite refactorings collected from five software projects. Table 3 summarizes our data on these software projects. The columns show the software project's name, followed by the number of commits, classes, candidates of incomplete composite and incomplete composites for each software project. Notably, these projects present diversity about domains, the number of commits, and the number of classes. It is relevant since it allows for an investigation of the incomplete composite practices applied to software projects with different sizes and domains. We then found 747 candidates of incomplete composites, of which 353 (47%) are incomplete composites. Our data shows that 276 (78.19%) incomplete composites were applied to classes that have at least one *Feature Envy*, while

Table 3: General Data Analyzed in this Study

Software Project	**Commits**	**Classes**	**Cand. Inc. Comp.**	**Inc. Comp.**
couchbase-java-client	1,023	656	34	10
dubbo	3,961	1,971	202	94
fresco	2,207	994	115	47
jgit	7,513	1566	264	156
okhttp	4,319	167	132	46
Total	19,023	5,354	**747**	**353**

81 (22.95%) incomplete composites were used on *God Classes*. Note that, an incomplete composite can have been applied to a class that is a *God Class* and also it has *Feature Envies*.

4.2 Dataset Validation

We performed two manual validations with developers for our dataset, which helped us check if our identification of incomplete composites was correct, and understand the context of these incomplete composites.

First validation. We performed a manual validation with nine developers not associated with the implementation of the incomplete composites. Their development experience varied from two to five years. Due to the time limitations of the developers, they evaluated only 30 composites. These composites were randomly selected, where we presented 26 composites that could be incomplete composites for *Feature Envy* removal and 4 composites that could be incomplete composites for *God Class* removal.

We asked developers if: (i) the composites were incomplete composites; and; (ii) what were the development activities done while the composite was applied. This final question allowed us to mitigate composites not applied for code smell removal. To answer those questions, developers evaluated classes and commits before and after of each composite was applied.

They pointed that 13 (50%) out of 26 incomplete composites were for *Feature Envy* removal, 7 (27%) out of 26 composites were not incomplete composites for *Feature Envy* removal, developers did not find the classes of six composites (23%) of them. For *God Class* removal, all composites were confirmed incomplete composites. Therefore, 17 (56%) out of 30 incomplete composites were confirmed by the manual process.

On the intents of developers during the application of the composites, we concluded through commit messages [33] and code changes, that developers applied changes in which: 7 (41%) out of 17 were applied with the intent of refactoring only; 3 (18%) composites had the intents of a feature addition and refactoring; 6 (35%) were applied for a feature addition only, and; 1 (6.5%) was applied for a bug fix only. We observed that 10 (59%) out of 17 incomplete composites were applied in commits in which developers explicitly mentioned the intent of refactoring. This suggests that these composites may have been applied to remove a code smell, and they were incomplete to remove them. These results also allowed us to measure that a significant percentage of the composites in our data set might truly be incomplete composites.

Second validation. In the second step of the validation, we aimed to ask developers related to the implementation and application of the incomplete composites. At first, we submitted three pull requests to validate if the composites are incomplete composites for the Feature Envy removal [12–14]. They were submitted one month after that the incomplete composites were applied. It would be easier for developers to remember which and why the incomplete composites were applied. Currently, one pull request was accepted, while the two other pull requests are open. The accepted pull request improved the code structure by removing an instance of *Feature Envy* from the Dubbo project. The developer answered that our composite recommendation caused the code to become clearer, the developer told: *"Hi, thanks. I think this patch makes the code cleaner."*

5 COMMON INCOMPLETE COMPOSITES

This section answers our RQ_1 on most common incomplete composites across software projects.

5.1 Procedures

We collected the frequency of each incomplete composite for *Feature Envy* and for *God Class* removal. We counted the incomplete composites according to their compositions, though we did not consider the order of the single refactorings in each composite, since a recent study observed that incomplete composites often are applied in the single commit [5] and, in the context of a single commit, it is not possible to know the order of the single refactorings. We then created a ranking for the frequency of each composition of incomplete composites for each project, regardless of the order of single refactorings in composites.

5.2 Results

Most of the incomplete composites were common to all analyzed projects. Thus, we created a ranking of the incomplete composites for all software projects. Table 4 presents a ranking of the 5 most common incomplete composites across projects. The first column indicates the position of the incomplete composite in the frequency-based ranking. The second column then presents the single refactorings that compose each incomplete composite. The last column presents the frequency of each incomplete composite.

Our results show that incomplete composites with only *Extract Methods* were the most common for all software projects. We observed that developers applied 30 (8.50%) out of 353 incomplete composites with only two *Extract Methods*, while 23 (6.51%) out of 353 incomplete composites had three or more *Extract Methods*. Thus, 53 (15.01%) out of 353 incomplete composites had only method extraction. Based on that, we grouped the incomplete composites based on frequent compositions, and by compositions that can be strongly related to a common proposal. For instance, incomplete composites containing only *Extract Methods* are in one group because developers certainly aim to extract methods only, it can be to remove a code smell or improve the cohesion between methods, while incomplete composites containing at least one *Extract Method* and one *Move Method* were grouped in another group because developers aim to improve the coupling and cohesion on multiple classes, *Extract Methods* and unusual refactoring types were grouped because it can indicate that developers can be interested to apply other code changes that are not strictly related to the code structure improvements. In total, we found seven groups of common compositions of incomplete composites.

Grouping of Incomplete Composites. Table 5 presents the groups of incomplete composites across the software projects. The first group, G_1, is composed by composites that contain at least one *Extract Method* and one single *Rename* refactoring: *Rename Parameter*, *Rename Variable*, *Rename Attribute*, *Rename Method* or *Rename Class*. G_2, the second group, is composed by composites that contained *Extract Methods* and unusual refactoring types. G_3 is composed by composites that contained only *Extract Methods*, as previously discussed. G_4 has composites that contained all refactoring types except for *Extract Methods* and *Move Methods*. G_5 is composed by incomplete composites that have at last one *Extract Method* and one *Move Method*. G_6 consists of incomplete composites that have at least one *Move Method*, but do not have *Extract Methods*. Finally, G_7 has incomplete composites that are composed by only *Move Methods*.

Considering the incomplete composites in all groups, our first finding is that 291 (82.44%) out of 353 incomplete composites have more than one refactoring type. This result is different from an existing study about composites since this previous study has reported that incomplete composites are often composed of a single refactoring type [5], and another study only investigated refactoring types of the method-scope [6]. Our results are different because they analyzed a much smaller set of refactoring types than the one considered in our study. Given the type heterogeneity and the wide scope of the aforementioned composites, one could expect they would often have a positive effect on multiple internal attributes, including the cohesion and coupling of the affected classes

Finding 1: Incomplete composites often have more than one refactoring type. It implies that existing refactoring recommendation systems may support more refactoring types that are not commonly investigated by previous studies.

Analysis of the Groups of Incomplete Composites. We observed that the composites in G_1 – *Extract Methods* and *Renames* – were the ones applied most often by developers in composites, not in isolation as reported by previous studies [7, 37]. Also, this group was often applied to smelly classes with at least one *Feature Envy* or a *God Class*, but these code smells were not removed by these refactorings. This is an interesting result because it presented that, on smelly classes, developers often extract methods and apply several renames in composites, potentially to improve code comprehension in these classes to some extent, but not fully remove code smells. In groups G_1 and G_3, it is expected that developers were to improve the cohesion of the class, since developers are separating the code in different methods, regardless of the removal of *Feature Envy* or *God Class*.

Surprisingly, incomplete composites composed by at least one *Extract Method* are often (71%) applied without *Move Methods* (groups G_1, G_2, and G_3). One such reason for this would be that developers may be reluctant to move methods if they do not know which class(es) in the program should receive the moved method(s). This result suggests that the lack of *Move Methods* in incomplete composites that have *Extract Methods* can be related to code smells that were not removed. It can also suggest that if *Extract Methods* were applied with *Move Methods* in composites, they would be able to remove code smells. Developers may be reluctant to move methods if they do not know (or need to spend time) which class(es) in the program should receive the moved methods.

Finding 2: Incomplete composites with at least one *Extract Method* are often (71%) applied without Move Methods on smelly classes. This implies that automated refactoring tools could be extended to identify opportunities for recommending the completion of incomplete composites with extractions, mainly.

In G_5, we observed that *Extract Methods* and *Move Methods* are not often applied together in incomplete composites. It can suggest that composites with these two refactoring types could have successfully removed *Feature Envies* and *God Classes*. We noticed that developers did not apply the necessary amount of *Extract Methods* and *Move Methods* for the removal of the code smells. However, it is expected that developers improve cohesion and coupling through these incomplete composites, since they are separating methods and moving it to other classes.

Even though *Move Method* is a common refactoring type [37], G_7 was not often applied in incomplete composites, representing less than 2% of the incomplete composites. Developers have mostly applied *Move Methods* with other refactoring types. Thus the existing recommendations composed of using only *Move Methods* to remove *God Classes* are not applied in practice. With this result, we suggested that future studies might recommend composites that have more than one refactoring type for the removal of this smell. On the context of internal quality attributes, it is expected for these *Move Methods* to improve the cohesion and coupling, since they are moving methods and decreasing the dependency between the classes.

Table 4: Five Most Common Incomplete Composites

Rank	Incomplete Composite	Frequency
1	{Extract Method, Extract Method}	30 (8.50%)
2	{Extract Method, Extract Method, Extract Method}	12 (3.40%)
3	{Extract Variable, Extract Method}	8 (2.27%)
4	{Rename Variable, Extract Method}	8 (2.27%)
5	{Rename Parameter, Extract Method}	5 (1.42%)
Total		**63 (17.86%)**

Table 5: Groups of Incomplete Composites Across Projects

Id	Groups	Frequency
G_1	Extract Method and Rename	145 (41.07%)
G_2	Extract Method and Unusual Refactoring Types	53 (15.01%)
G_3	Extract Method	53 (15.01%)
G_4	Other types	52 (14.73%)
G_5	Extract Method and Move Method	26 (7.37%)
G_6	Move Method and Other types	18 (5.10%)
G_7	Move Method	6 (1.70%)
Total		**353 (100.00%)**

6 EFFECT OF INCOMPLETE COMPOSITES

This section answers our RQ_2 on the effect of incomplete composites on internal quality attributes.

6.1 Procedures

We classified the effect of incomplete composites on the internal quality attributes as (i) positive, (ii) neutral, and (iii) negative. This

classification also appears in previous studies [1, 2, 8, 11] as a comprehensive mechanism to capture the overall refactoring effect on internal quality attributes. This classification relies on three premises. First, each code metric (associated with a particular attribute) can either increase, be unaffected, or decrease after the refactoring application. Second, certain code metrics improve when their values decrease (e.g., CBO), while others improve when their values increase (e.g. TCC). Third, an internal quality attribute improves when the code metrics that capture this attribute improve as well; the attribute worsens when the corresponding metrics worsen. Similarly, we consider that: (i) an incomplete composite has a positive effect when at least one code metric that captures an internal quality attribute has improved; (ii) a neutral effect when none of the code metrics that capture the attribute have changed, and; (iii) a negative effect occurs in the other remaining cases.

Then, we aimed to combine the incomplete composite data with the collected code metrics to detect the former's impact on the latter. For that purpose, we designed and executed a set of three steps, tailored for accuracy in that detection, described as follows:

1. Collecting metric thresholds from significant time periods in the projects' development. To determine a baseline for comparing the classes' code metrics to, in order to ascertain if they may contain a problematic structure due to the values of the measured metrics, we collected metric thresholds from significant time periods in the project. We started with yearly thresholds, but further analysis proved that the changes between consecutive years were too significant, so we narrowed it down to 6-month periods.

These "significant changes" were defined as changes that modified *over 25% of a metric's value* in *two or more internal quality attributes*, for higher or lower, except for Size (due to the tendency of Size changing frequently with code changes [27, 35]). These thresholds were defined based on quartiles, with a metric with values within the 25% smallest values (Q_1) being considered *good*, a metric between the 25% smallest and 25% largest values (Q_2 and Q_3) being considered *average*, and a metric within the 25% largest values (Q_4) being considered *problematic*, except for the CLOC metric, of which definitions are inverted, due to its nature of higher values meaning an improvement [8].

2. Defining the frequency of significant changes to code metrics. We then attempted to determine their impact, by analyzing how incomplete composites affected the refactored classes, using the following criteria: (i) each analysis looked at the metrics in two states: the commit *immediately before* the composite and the *last* commit in the composite; (ii) a significant change was defined in the same way as in 1., i.e., a change was considered significant if it caused a variance of over 25%. Thus, to understand the composites' impact, looked at how each composite changed the classes they affected, by analyzing each of the affected class's metrics before and after such composites.

3. Defining the state of the classes' metrics related to their changes. To analyze the quality of the code in the classes affected by the composites (before and after their application), we compared their metrics to the thresholds defined in **step 1**, thus defining the class as *problematic*, *average* or *good* in terms of their metrics' values. With this, and with the information from **step 1**, it is possible to determine if the composites caused an improvement, did not affect a class or worsened its state.

6.2 Results

Table 6 displays a summarized comparison of the before-after states of the classes in the project, by presenting, for each internal quality attribute (Cohesion, Coupling, Complexity, Size), the % of individual metrics changed for the better (i.e., had their values reduced, except for CLOC), that changed for the worse, or that did not change for each composite.

By analyzing each composite individually, we determined that out of 416 composites that only modified a class's contents (i.e., did not delete nor rename the original class), 58% (239) changed no metric to a value within a threshold different than the one they were before the composite; 22% (92) only increased the metrics' values to a higher threshold, 13% (55) only decreased the metrics' values to a lower threshold, and 7% (30) both increased and decreased the metrics' values to higher and lower thresholds, respectively. However, out of those that did change one or more metrics' values to different thresholds, over half only changed a single metric's value enough to change its threshold.

Thus, this analysis' results can be summarized as follows: in a general sense, the majority (58%) of changes tended to keep the metric within the same threshold as it was before the composite. Most changes that did impact the metrics caused an increase in their values, which, in most cases, causes a negative impact in the resulting source code. Third, most changes that did impact the metrics mostly impacted only a single metric at a time. Fourth, the majority of times a positive change happened in the code, it was because of an increase in the amount of CLOC (Comment Lines of Code). This means that, while the quality might have improved, the actual smells were either not fixed or even subtly ameliorated by the composites; Thus, this can be summarized in the following finding, which corroborates with a similar one found by a previous work [7] for single refactorings:

> **Finding 3**: Most incomplete composites tend to not change the state of the code structure, with respect to its internal attributes – i.e. well-maintained code often remains well-maintained, while smelly code often remains smelly. This may motivate refactoring tools to improve their recommendations to maintain the internal structural quality of the program in composites that do not successfully remove code smells.

Nonetheless, by taking a closer look at the absolute values of the metrics changed by each incomplete composite, we discovered that over half of their changes (52%) worsened at least one of the internal quality attributes' metrics. The majority of this worsening, however, was related to size metrics. Out of the other changes, 27% did not change metrics' values at all, and 21% improved their values.

However, by looking at the intensity of these changes, it is possible to see that 70% did not significantly increase or decrease the quality of the code (>25% change in the measurements), while a small, but still significant, set of incomplete composites significantly increased the internal quality attributes' measurements (22%), which in most cases, indicates a decrease in the code quality, and a very small amount of incomplete composites actually significantly decreased these metrics (8%). This means that, even if some of the composites tended to modify the values of the internal quality attributes' metrics, they did so in small increments. This confirms the

result discussed in the Section 5.2, which shows that incomplete composites have often not improved the program comprehension significantly. We also observed that the majority of those that do make significant changes tend to worsen these attributes. But, as a major finding, we concluded that:

> **Finding 4**: The majority of incomplete composite refactorings tend to not make significant changes to the class-level internal quality attributes of the code. This indicates that developers often apply composites to minimally maintain the level of structural quality while achieving their other primary goals. Existing refactoring recommenders could make developers aware on how to apply (complete) composites while also facilitating their goal achievements.

This finding contradicts previous works (e.g. [4]) that did not find a relation between refactorings and the values of different metrics. This could be caused by the fact that they only analyzed single refactorings or due to their analysis focusing on a single metric at a time instead of looking at the internal quality attributes.

We then analyzed the effects of incomplete composites in terms of the groups presented in Section 5. For that purpose, we chose the 4 most common groups that contain refactorings recommended by the literature to remove *Feature Envy* and *God Class*. Therefore, groups G1, G3, G5 and G7 were chosen as they only contain *Extract Methods*, *Move Methods* and *Renames* (renaming is not necessarily recommended to remove these code smells, but is recommended when a method is extracted). This was done in an attempt to mitigate the threat of analyzing composites that were not intentionally applied to remove those smells. In the composites pertaining to those groups, developers only applied refactoring types that are recommended to remove them, thus, reducing the likelihood of the composite not being applied for that purpose.

Therefore, Figure 3 presents the effects of these groups on the four internal quality attributes. We can observe that, once again, size-related metrics are the ones that change the most with incomplete composites, followed by complexity metrics. It is also notable that: (i) composites in G_7 (only *Move Methods*) often fail to improve code metrics; (ii) no composite in G_3 (only *Extract Methods*) changed coupling-related metrics and; (iii) G_5 (*Extract and Move Method*) had the most overall improvement in their composites. This is in accordance to the discussion presented in Section 5.2, in which it was speculated that several extractions and renames (the group G_1) in composites should have reduced the code complexity, improving the program comprehension. Besides that, the positive results of the G_3 group then confirmed that when developers apply *Extract Methods* and *Move Methods* in composites, they often improve the most internal quality attributes, and thus the program comprehension, regardless of the presence of smelly classes.

Finally, by correlating these findings to the fact that these incomplete composites are composed by single refactorings that were also frequently applied alongside other code changes, keeping the non-size related metrics within acceptable parameters can be considered a good effort in preventing code quality decay, since feature additions and other non-refactoring related changes might have happened in the code (due to the majority of the worsened metrics being size-related). Thus, by keeping the code quality from decreasing due to those other changes, some of these incomplete composites could have acted as preventive measures, not allowing code quality to degrade because of these new additions. This can be summarized as the following finding:

> **Finding 5**: Incomplete composites rarely increase or decrease code quality, but, when performed alongside other code changes, they can prevent the quality decay that could happen because of these additions. This implies that even throughout the application of incomplete composites, the developers are putting effort into attempting to increase the code structures' quality.

This strengthens the conclusion that, even though incomplete composites aim to improve the internal structure quality of certain code elements, they do not bring about significant changes to the smelly class' state, by mostly keeping it in the same state as it was before the composite – though they do mostly prevent quality decay from other non-refactoring changes, much like what was described in Section 2.3. However, a non-insignificant percentage of incomplete composites actually worsens the affected class's problems, which could be solved by the completion of the used composite.

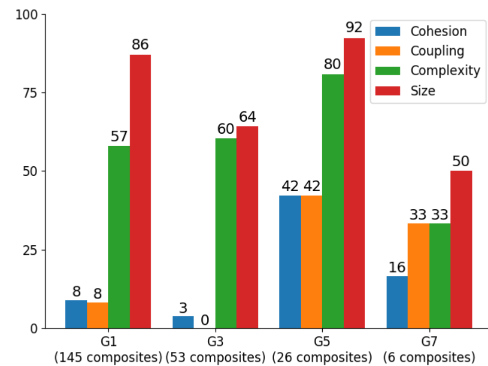

Figure 3: % of positive changes in IQA per Inc. Comp. Group

7 THREATS TO VALIDITY

Construct and Internal Validity: We reused criteria for selecting *software projects* from previous studies [5–8, 37]. We aimed at preventing a biased project selection that could favor our study results. We used RMiner [43] to perform *single refactoring detection*, since it has a high accuracy. Similarly to previous studies [5, 7, 8], we performed *code metric computation* via the SciTools Understand tool that computes class-level code metrics that capture the four internal quality attributes analyzed in this study (Table 1). Inspired by the literature [7, 28], we used the Organic tool to detect *code smell instances*. The smell detection strategies used by this tool have a high accuracy: 72% precision and 81% recall in average [4]. We validated the associated smell instances of *Feature Envy* and *God Class* (Section 4.2). By doing that, we confirmed the tool's accuracy.

We reused an heuristic from the literature [38] for detecting *composite refactorings*. By reusing this heuristic, we could prevent manual biases while supporting large-scale composite computation. We could also analyze the effect on internal quality attributes in a

Table 6: % of changes which significantly improved each metric for each the four internal quality attributes, per project

Project	# of Composites	Cohesion: 1 metric			Coupling: 1 metric			Complexity: 4 metrics			Size: 9 metrics		
		Positive	Neutral	Negative	Positive	Neutral	Negative	Positive	Neutral	Negative	Positive	Neutral	Negative
couchbase-java-core	10	10%	90%	0%	0%	100%	0%	2%	96%	2%	4%	96%	0%
dubbo	122	5%	90%	5%	4%	94%	2%	6%	90%	4%	4%	92%	4%
fresco	50	10%	88%	2%	2%	98%	0%	5%	91%	4%	2%	97%	1%
jgit	178	6%	92%	2%	4%	93%	3%	3%	93%	4%	6%	92%	2%
okhttp	56	7%	90%	3%	4%	94%	2%	2%	94%	4%	1%	96%	3%

wide scope ranging from the source class to the target class of each refactoring. We carefully designed an heuristic for characterizing those incomplete composites [22, 38]. The definition of incomplete composite is considerably subjective, once it depended on our body of knowledge on what composites target a particular smell type. Although there is such subjectivity, we strongly relied on Fowler's refactoring catalog [21] and empirical evidence derived from recent studies, e.g. [5, 7]. One author wrote scripts for computing incomplete composites, and two authors double-checked these scripts, thereby reducing the manual bias and errors.

A previous study presented that developers do not necessarily consider a code smell like a problem in the source code [36]. Thus, we can assume that developers often do not have the explicit intent to remove code smell when applying composites to be a threat to this work's soundness. To mitigate it, then, we submitted pull requests and performed a manual validation aimed to determine the developers' intent to apply composites. The manual validation of the composites and incomplete composites (Section 4.2) was performed by developers that are familiar with refactoring. This was important for demonstrating the accuracy of our heuristics and the meaningfulness of our set of incomplete composites, from the perspective of experienced developers [10, 32]. The validated sample of composites is quite small, but we distributed this sample between nine developers for a careful analysis.

Conclusion and External Validity: We reused a three-fold classification of the refactoring effect on internal quality attributes from previous studies [1, 8] (Section 6.1). We assumed that this classification could be successfully adapted to the context of incomplete composites. One could criticize our approach that classifies an incomplete composite as positive when it improves at least one associated code metric is too loose. However, a recent study [8] showed no considerable difference between this approach and stricter ones, such as considering the improvement of most metrics as a positive effect. Besides that, a study [8] used this approach to investigate the effect of single refactoring on internal quality attributes. We then reused this approach to compare the results of single refactoring with the results of composites.

We applied traditional techniques of descriptive analysis on the quantitative data [1, 5–8]. We computed percentages of the most common refactoring combinations that constitute incomplete composites ($\mathbf{RQ_1}$) as well as the effect classification of incomplete composites ($\mathbf{RQ_2}$). These techniques allowed us a detailed comprehension of the incomplete composites' effects in different scenarios. For classifying thse effects, we relied on quartiles, as discussed in Section 6.1, similarly to related studies [1, 8].

Regarding $\mathbf{RQ_1}$, we were unable to compute the order of the refactorings within a composite. There is empirical evidence that most composites are fully applied in a single commit, so we cannot assure the precedence of one refactoring over another refactoring. This limitation has also affected previous studies [5, 38] but they still did not prevent interesting insights of the effects of refactoring on internal quality attributes from being derived. With respect to $\mathbf{RQ_2}$, previous studies [7, 8] show that single refactorings are very often applied alongside other types of code change. The same reasoning applies by extension to composite refactorings.

The scope of our study is quite limited for allowing a wide generalization of our findings and their implications. Although we aimed at a certain diversity in terms of project size and commit history (Table 3). Our preference for open source Java projects may support the study's replication, but they may not cover all refactoring practices worldwide. Nevertheless, these projects have been successfully analyzed by related studies [1, 5–8].

8 CONCLUSION

This paper presents a quantitative study, in which we investigated the incomplete composites in-depth in five software projects of different domains. Our findings reveal that developers often (58%) apply composites to minimally maintain the level of structural quality while achieving their other primary goals. It implies that automated refactoring tools could be extended to identify opportunities for recommending the completion of otherwise harmful incomplete composites. As future works, we aim (i) to investigate the incomplete composites for the removal of more code smells, (ii) to classify manually more incomplete composites that are applied only with the development activity of refactoring and incomplete composites that are applied for other development activities, and (iii) to evaluate if composites with the recommended refactoring types have removed the code smells.

ACKNOWLEDGEMENTS

This study was in part financed by CNPq (434969/2018-4), (427787/2018-1), (409536/2017-2), and (312149/2016-6), CAPES (175956), and FAPERJ (22520-7/2016).

REFERENCES

[1] Eman AlOmar, Mohamed Mkaouer, Ali Ouni, and Marouane Kessentini. 2019. On the impact of refactoring on the relationship between quality attributes and design metrics. In *13th ESEM (2019)*. 1–11.

[2] Mohammad Alshayeb. 2009. Empirical investigation of refactoring effect on software quality. *IST (2009)* 51, 9 (2009), 1319–1326.

[3] Victor Basili and Dieter Rombach. 1988. The TAME project: Towards improvement-oriented software environments. *TSE (1988)* 14, 6 (1988), 758–773.

[4] 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. *JSS (2015)* 107 (2015), 1–14.

[5] 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 (2019)*. 1–11.

[6] Aline Brito, Andre Hora, and Marco Tulio Valente. 2019. Refactoring graphs: Assessing refactoring over time. In *26th SANER (2019)*. 504–507.

[7] 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 *11th FSE (2017)*.

[8] Alexander Chávez, Isabella Ferreira, Eduardo Fernandes, Diego Cedrim, and Alessandro Garcia. 2017. How does refactoring affect internal quality attributes? A multi-project study. In *31st SBES (2017)*. 74–83.

[9] Mel Cinnéide and Paddy Nixon. 2000. Composite refactorings for Java programs. In *14th ECOOP (2000)*. 129–135.

[10] Rafael Maiani de Mello, Anderson G. Uchôa, Roberto Felicio Oliveira, Willian Nalepa Oizumi, Jairo Souza, Kleyson Mendes, Daniel Oliveira, Baldoino Fonseca, and Alessandro Garcia. 2019. Do Research and Practice of Code Smell Identification Walk Together? A Social Representations Analysis. In *ESEM (2019)*. IEEE, 1–6.

[11] Bart Du Bois, Serge Demeyer, and Jan Verelst. 2004. Refactoring: Improving coupling and cohesion of existing code. In *11th WCRE (2004)*. 144–151.

[12] Dubbo. 2019. Refactoring to remove duplicate methods and feature envy. (2019). Available at: https://github.com/apache/dubbo/pull/5506.

[13] Dubbo. 2019. Refactoring to remove feature envy. (2019). Available at: https://github.com/apache/dubbo/pull/5559.

[14] Dubbo. 2019. refactoring to remove feature envy. (2019). Available at: https://github.com/apache/dubbo/pull/5529.

[15] Dubbo. 2019. Rewrite UTs. (2019). Available at: https://github.com/apache/dubbo/commit/66fbd320.

[16] Eduardo Fernandes. 2019. Stuck in the middle: Removing obstacles to new program features through batch refactoring. In *41st ICSE: Doctoral Symposium (2019)*. 1–4.

[17] Eduardo Fernandes, Johnatan Oliveira, Gustavo Vale, Thanis Paiva, and Eduardo Figueiredo. 2016. A review-based comparative study of bad smell detection tools. In *20th EASE (2016)*. 18:1–18:12.

[18] Eduardo Fernandes, Anderson Uchôa, Ana Carla Bibiano, and Alessandro Garcia. 2019. On the alternatives for composing batch refactoring. In *3rd IWoR, co-located: 41st ICSE (2019)*. 1–4.

[19] Eduardo Fernandes, Gustavo Vale, Leonardo Sousa, Eduardo Figueiredo, Alessandro Garcia, and Jaejoon Lee. 2017. No code anomaly is an island: Anomaly agglomeration as sign of product line instabilities. In *16th ICSR (2017)*. 48–64.

[20] Francesca Fontana, Marco Mangiacavalli, Domenico Pochiero, and Marco Zanoni. 2015. On experimenting refactoring tools to remove code smells. In *16th XP, Scientific Workshops (2015)*. 1–7.

[21] Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code* (1 ed.). Addison-Wesley Professional.

[22] Everton T. Guimarães, Alessandro F. Garcia, and Yuanfang Cai. 2015. Architecture-sensitive heuristics for prioritizing critical code anomalies. In *14th International Conference on Modularity (2015)*, Robert B. France, Sudipto Ghosh, and Gary T. Leavens (Eds.). ACM, 68–80.

[23] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2014. An empirical study of refactoring: Challenges and benefits at Microsoft. *TSE (2014)* 40, 7 (2014), 633–649.

[24] Yun Lin, Xin Peng, Yuanfang Cai, Danny Dig, Diwen Zheng, and Wenyun Zhao. 2016. Interactive and guided architectural refactoring with search-based recommendation. In *FSE (2016)*. 535–546.

[25] Tom Mens and Tom Tourwé. 2004. A survey of software refactoring. *TSE (2004)* 30, 2 (2004), 126–139.

[26] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, Kalyanmoy Deb, and Mel Ó Cinnéide. 2014. Recommendation system for software refactoring using innovization and interactive dynamic optimization. In *29th ASE (2014)*. 331–336.

[27] Emerson Murphy-Hill, Chris Parnin, and Andrew Black. 2012. How we refactor, and how we know it. *TSE (2012)* 38, 1 (2012), 5–18.

[28] Willian Oizumi, Alessandro Garcia, Leonardo Sousa, Bruno Cafeo, and Yixue Zhao. 2016. Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems. In *38th ICSE (2016)*. 440–451.

[29] Willian Nalepa Oizumi, Leonardo da Silva Sousa, Anderson Oliveira, Alessandro Garcia, O. I. Anne Benedicte Agbachi, Roberto Felicio Oliveira, and Carlos Lucena. 2018. On the identification of design problems in stinky code: experiences and tool support. *J. Braz. Comp. Soc. (2018)* 24, 1 (2018), 13:1–13:30.

[30] Willian Nalepa Oizumi, Alessandro F. Garcia, Leonardo da Silva Sousa, Bruno Barbieri Pontes Cafeo, and Yixue Zhao. 2016. Code anomalies flock together: exploring code anomaly agglomerations for locating design problems. In *38th ICSE (2016)*, Laura K. Dillon, Willem Visser, and Laurie Williams (Eds.). ACM, 440–451.

[31] 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 ICSE-SEIP (2017)*. IEEE Computer Society, 33–42.

[32] 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. *IST (2020)* 120 (2020).

[33] Matheus Paixão, Anderson Uchôa, Ana Carla Bibiano, Daniel Oliveira, Alessandro Garcia, Jens Krinke, and Emilio Arvonio. 2020. Behind the Intents: An In-depth Empirical Study on Software Refactoring in Modern Code Review. In *17th MSR (2020)*.

[34] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. 2018. A large-scale empirical study on the lifecycle of code smell co-occurrences. *IST (2018)* 99 (2018), 1–10.

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

[36] José Amancio M Santos, João B Rocha-Junior, Luciana Carla Lins Prates, Rogeres Santos do Nascimento, Mydiã Falcão Freitas, and Manoel Gomes de Mendonça. 2018. A systematic review on the code smell effect. *JSS (2018)* 144 (2018), 450–477.

[37] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. 2016. Why we refactor? Confessions of GitHub contributors. In *24th FSE (2016)*. 858–870.

[38] 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 MSR (2020)*.

[39] Leonardo Sousa, Anderson Oliveira, Willian Oizumi, Simone Barbosa, Alessandro Garcia, Jaejoon Lee, Marcos Kalinowski, Rafael de Mello, Baldoino Fonseca, Roberto Oliveira, Carlos Lucena, and Rodrigo Paes. 2018. Identifying design problems in the source code: A Grounded Theory. In *40th ICSE (2018)*. 921–931.

[40] Gábor Szőke, Csaba Nagy, Lajos Fülöp, Rudolf Ferenc, and Tibor Gyimóthy. 2015. FaultBuster: An automatic code smell refactoring toolset. In *15th SCAM (2015)*. 253–258.

[41] Daniel Tenorio, Ana Carla Bibiano, and Alessandro Garcia. 2019. On the customization of batch refactoring. In *3rd IWoR, co-alocated ICSE (2019)*. IEEE Press, 13–16.

[42] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. 2018. Ten years of JDeodorant: Lessons learned from the hunt for smells. In *P25th SANER (2018)*. 4–14.

[43] Nikolaos Tsantalis, Matin Mansouri, Laleh Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and efficient refactoring detection in commit history. In *40th ICSE (2018)*. 483–494.

[44] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. 2017. When and why your code starts to smell bad (and whether the smells go away). *TSE (2017)* 43, 11 (2017), 1063–1088.

[45] Santiago A. Vidal, Willian Nalepa Oizumi, Alessandro Garcia, J. Andres Diaz-Pace, and Claudia Marcos. 2019. Ranking architecturally critical agglomerations of code smells. *Sci. Comput. Program. (2019)* 182 (2019), 64–85.

[46] Aiko Yamashita and Leon Moonen. 2013. Do developers care about code smells? An exploratory survey. In *20th WCRE (2013)*. 242–251.

[47] Norihiro Yoshida, Tsubasa Saika, Eunjong Choi, Ali Ouni, and Katsuro Inoue. 2016. Revisiting the relationship between code smells and refactoring. In *24th ICPC (2016)*. 1–4.