Enhancing Recommendations of Composite Refactorings based on the Practice

Ana Carla Bibiano*, Daniel Coutinho*, Anderson Uchôa†, Wesley K. G. Assunção‡, Alessandro Garcia*, Rafael de Mello§, Thelma E. Colanzi¶, Daniel Tenório*, Audrey Vasconcelos‖, Baldoino Fonseca‖, Márcio Ribeiro‖.
*Pontifical Catholic University of Rio de Janeiro, Brazil. †Federal University of Ceará, Brazil.
‡North Carolina State University, USA. §Federal University of Rio de Janeiro, Brazil.
¶State University of Maringá, Brazil. ‖Federal University of Alagoas, Brazil.

Abstract—**Refactoring is a non-trivial maintenance activity. Developers spend time and effort refactoring code to remove structural problems, i.e., code smells. Recent studies indicated that developers often apply *composite refactoring* (composite, for short), i.e., two or more interrelated refactorings. However, prior studies revealed that only 10% of composite refactorings are considered *complete*, i.e., those fully removing code smells. Many incomplete refactorings can even replace or introduce smells, requiring additional effort for their removal later in the project. Moreover, existing refactoring recommendations are not well-detailed and do not alert developers about these possible side effects. To address these gaps, we conducted a large-scale study involving more than 250k refactorings from 42 software projects, including both open-source and closed-source projects. Our goal is to investigate how the most common complete composites are combined and their side effects in the practice. Our results reveal that the current recommendation to apply Extract Method(s) with fine-grained refactoring types needs refinements. We found that certain fine-grained refactorings like *Change Variable Types* and *Change Return Types* can introduce up to 45% of *Brain Methods* when combined with *Extract Method(s)*. Moreover, *Extract Method(s)* and Move Method(s), a common recommendation to remove Feature Envy, may inadvertently introduce about 30% of *Lazy Classes* and approximately *70%* of *Data Classes*. Despite these potential side effects, existing refactoring catalogs and tools' recommenders do not alert developers about these side effects. Finally, we consolidate our findings into a catalog to provide clear guidance for developers and researchers on effectively applying composite refactorings to fully remove code smells.**

Index Terms—**software refactoring, code smells, refactoring recommendations, mining software repositories**

I. Introduction

Developing software projects with high design quality is the goal of every company [1], [2]. However, due to extensive maintenance and evolution in those projects, the internal software quality usually decays and degrades [3], [4], [5]. Internal quality problems are known as code smells [6]. A *code smell* is a symptom of bad design or poor implementation choice in the source code of a software system that possibly indicates a deeper problem [6]. These symptoms have a negative influence on software quality regarding maintainability, understandability, and testability. Thus, developers must identify and remove code smells as soon as possible [6], [7]. A well-known and widely used practice to deal with code smells is code refactoring [8]. *Code refactoring* aims at improving the structure of the software, removing code smells, without changing its external behavior [6]. Developers apply code refactoring intending to fully remove code smells, even when refactorings are applied with non-refactoring changes [9].

Despite its benefits, refactoring is a non-trivial activity. To apply refactoring, developers must: (i) identify where to refactor, (ii) know what refactoring type(s) to apply, and (iii) analyze both the beneficial and harmful effects of refactoring on the source code. In the first step, smelly code is often the target of refactorings [10]. In the second step, developers often have to combine refactorings [11], [12] through composite refactorings. A *composite refactoring* (composite, for short), is formed by two or more interrelated single refactorings [13], [14], [15]. The application of composites is a complex and error-prone task, as the smelly code is often modified in multiple parts by different refactoring types combined [13]. To make matters worse, studies have indicated that composites are generally applied manually [11], [16] and often combined with other code changes [11], [13]. Yet, studies indicate that only 10% of composites could remove code smells entirely [12], [13], even when refactoring is the primary goal. Composites are expected to remove at least the target code smells. For instance, *Extract Method(s)* and *Move Method(s)* are often indicated to remove *Feature Envy* [6], [13]. Thus, the *Feature Envy* is the target smell for a composite formed by these refactoring types. We refer here to a composite able to fully remove target code smells as *complete composite refactorings* [17] (see Section II). In that way, if *Extract Method(s)* and *Move Method(s)* fully remove the target *Feature Envy*, then this composite is complete.

As a way to support developers in performing refactorings, some IDEs, such as Eclipse or IntelliJ, attempt to automate this activity. Unfortunately, when it comes to the application of composite refactorings, developers have to do it manually due to the limitations of existing tools. For instance, Kim *et al.* [16] reported that Microsoft developers needed to create refactoring solutions to support the Windows 7 development. There are two main reasons why developers rarely use IDE functionalities to perform refactorings: (i) the existing tools only recommend isolated refactorings, and (ii) they do not show the possible (side) effects of the refactoring, since previous work shows that refactorings may inadvertently introduce code smells even more severe than those removed ones [18].

Researchers have explored approaches to contribute to refactoring applications for many years [19]. Despite the advances in both the state of the art and the practice, there are still limitations to providing proper support for developers. The first limitation is that existing recommendations are not well-detailed, neglecting some of the refactoring types to be combined and when refactoring may be applied. For instance, Bibiano *et al.* [17] recommend *Extract Methods* and "fine granularity refactorings" or "fine-grained refactorings" (i.e., code transformations on variables and attributes) to remove the *Long Method* smell. However, they do not detail which fine-grained refactorings may be applied and how these refactorings can help to remove a *Long Method* in conjunction with an *Extract Method*, reducing the applicability of their findings.

The second limitation is the lack of information about the side effects of composite refactoring recommendations. Bibiano *et al.* also reported that complete composites may have side effects like the introduction or propagation of code smells. Despite indicating side effects for some complete composites, the authors do not perform an in-depth empirical analysis of side effects for complete composites. Another example of this can be found in the catalog of Brito *et al.* [20], where there is a recommendation for using *Pull Up Methods* to create a single and more general method in the superclass, leveraging code reuse. However, Brito *et al.* do not alert developers about the side effects of *Pull Up Methods*.

To effectively support developers in applying refactorings, our study focuses on gaining knowledge from the practice of composite refactorings. Thus, the goals of this study are three-fold: (i) explore the most common combinations of refactoring types in complete composite refactorings, (ii) discover side effects of existing recommendations of complete composites, and (iii) enhance existing recommendations of complete composite refactorings to overcome our results.

Aimed at study goals, we conducted a large-scale study on 42 open and closed-source Java projects. We collected 31,066 composites (composed of 250,172 single refactorings) from which we identified 1,397 complete composites. These composites are used to address 17 different types of code smells. We also identified the most frequent combinations within complete composites applied in the practice and the side effects of those complete composites. From these results, we created our catalog of composite recommendations. Our results show that (but not limited to):

- *Extract Methods* and *Move Methods* are commonly recommended to remove *Feature Envy*. In fact, about 74% of *Feature Envies* were removed by *Extract Methods* and *Move Methods*. However, about 30% of *Lazy Classes* and about *70%* of *Data Classes* can be introduced after the application of these extractions and motions of methods. However, the current studies found in the literature do not alert about these side effects. In our catalog, we included these side effects and justified them because these side effects can directly benefit developers and researchers [17].
- Developers frequently combine *Change Variable Type* or *Change Parameter Type*, when extracting methods to remove *Long Methods*, *Feature Envies*, and *Duplicated Code*. However, *Brain Methods* are often (45%) introduced due to these composites. In this case, we identified a recommendation of a complete composite composed of *Extract Methods* and *Change Parameter Types* (49%) that can fully remove these smells without introducing *Brain Methods*. This result shows that the existing recommendation of *Extract Method(s)* combined with fine-grained refactorings [17] cannot be considered a general solution for eliminating code smells, as some combinations may negatively impact software quality by introducing side effects, like *Brain Methods*.

Our study contributes to the practice by providing a more complete catalog with concrete recommendations to guide developers on applying complete composites. Also, our catalog clearly describes potential side effects, allowing developers to make more informed decisions on how to refactor their code. Finally, our findings can be a source of information for tool builders and researchers to create tools that adhere to the actual practice. Existing state-of-the-art refactoring recommenders [21], [22] neglect fine-grain refactorings, which were frequently present in complete composite recommendations.

II. Background and Problem Statement

This section describes the main concepts, and existing limitations regarding complete composite refactorings.

A. Composite Refactoring (or Composite)

A single refactoring rarely removes a code smell in practice [18]. Developers very often need to apply composite refactorings to eliminate the incidence of some code smell types [23]. A *composite refactoring* is a set of interrelated refactorings, defined as $c = \{r_1, r_2, ...r_n\}$, where each r is a single refactoring and $_i$ is an identifier for each refactoring applied [13]. A composite c can be formed of the same refactoring type, or a combination of different refactoring types [12], [13], [24], [25], [26], [27].

A recent study proposed a *range-based* heuristic [13] for composite detection. For that, the heuristic considers as composite those refactorings applied by the same developer and affecting the same code elements, known as the refactoring range. The reliability of this heuristic was demonstrated in several studies [13], [14], [17]. This heuristic also enables one to identify the smell(s) in the composite range being targeted by the interrelated refactorings.

Similar studies also indicate that developers often apply composites manually [12], [13], [23]. Additionally, related studies found that composites frequently result in undesirable *side effects* [12], [13]. A side effect is when a code smell is introduced or kept after the application of a composite refactoring. Based on existing literature, we observe that (i) there is a lack of knowledge, and consequently tooling support, on the best alternatives of composites for effectively removing code smells; and (ii) there is a misguidance of automated support for developers applying composites, potentially leading to worse internal quality due to side effect of some refactorings.

Aiming at fulfilling the gaps described above, we investigated the completeness of composite refactorings, a concept described in the next section. Therefore, a deeper understanding of composites is needed to support developers' decisions and tool builders on advancing the practice of refactoring, which is the main motivation of our work.

B. Completeness of Composite Refactorings

Recent studies recommend composites to remove a single code smell type, typically referred to as the target smell of a composite [17]. For instance, a recent study recommends applying *Extract Method(s)* combined with *Move Method(s)* to remove Feature Envy [13]. Thus, a *Feature Envy* is the target smell in such cases. Alternatively, **Composite Completeness** is a characteristic given to those composites able to achieve the full removal of the target code smells [17] as defined below:

Completeness of Composite Refactoring: Considering r_i as a single refactoring, and c is a composite refactoring. For each $r_i \in c$, r_i transforms a code element e, such as a method or/and class. We then have $\forall e$ that has a code smell s, and $SUM_{BEFORE(s)}$ is the sum of all target code smells before the application of a composite refactoring c, $SUM_{AFTER(s)}$ is the sum of all target code smells after the application of a composite refactoring c. A composite refactoring is complete when $SUM_{AFTER(s)} < SUM_{BEFORE(s)}$

Based on the principle that the main goal of refactorings is to improve the overall internal quality of the software system, mainly removing code smells, an in-depth investigation of the completeness of composites is necessary. Developers often spend time and effort applying composite refactorings to combat code smell incidences. However, this improvement (i.e., better internal quality due to code smell removal) is frequently not perceived in the practice [13], [14].

Researchers have investigated the recommendation of complete composites [12], [13], [17], [20]. Even though existing work have contributed to the field of composite refactoring, existing recommendations do not guide developers on how and when to apply them. For instance, Bibiano *et al.* [17] recommend applying *Extract Method* and *Move Method* combined with fine-grained refactorings for removing *Feature Envy*, but they do not recommend which fine-grained refactorings should be applied and applying the recommendations.

C. Fine-Grained and Coarse-Graine Refactorings

A refactoring of fine granularity, or a fine-grained refactoring (FGR), is a minor code transformation directly on variables or attributes. This transformation can be a change of variable type, a merge between two or more variables. Some fine-grained refactorings can indirectly involve multiple classes, like *Pull Up Attribute* or *Push Down Attribute*, but the code transformation is directly on the attribute. A refactoring of large granularity, or coarse-grained refactoring (CGR), is a code transformation that involves directly method(s) or class(es). Common examples of CGR are *Extract Method*, and *Move Method*. In this study, we considered the term "coarse-grained" to better align with the term "fine-grained". Table I shows the refactoring types classified in FGR and CGR. We used this classification of refactoring types because although there are many fine-grained types of refactorings. We observed that FGR were not investigated by previous studies that recommend complete composite refactorings [13], [20].

TABLE I
CLASSIFICATION OF REFACTORING TYPES

Fine-Grained (FGR)		**Coarse-Grained (CGR)**
Move Attribute	Rename Variable	Inline Method
Pull Up Attribute	Rename Parameter	Rename Method
Push Down Attribute	Replace Variable	Move Method
Rename Attribute	Merge Variable	Pull Up Method
Replace Attribute	Change Return Type	Push Down Method
Extract Attribute	Change Parameter Type	Extract Class
Merge Attribute	Change Variable Type	Extract Subclass
Split Attribute	Merge Parameter	Extract Superclass
Extract Variable	Split Variable	Move Class
Inline Variable	Replace Variable With Attribute	Rename Class
Parameterize Variable		Extract Interface
Split Parameter		Extract Method
Total: 22		**Total: 12**

D. Limitations of Existing Recommendations of Complete Composite Refactorings

Table II summarizes existing recommendations for complete composites. The table presents, respectively, the complete composites that may be applied; the existing smells before the composite; the target smell to be removed; the expected effect after applying the composite; and the side effects of complete composites. We argue that these recommendations have two main limitations. The first one is that the existing recommendations are not well-detailed, mainly about which refactoring types may be applied over certain circumstances and when. For instance, Bibiano *et al.* [17] recommend Extract Methods and fine-grained refactoring to remove Long Method. However, they did not detail which fine-grained refactorings should be applied and how the refactorings help to remove a *Long Method* in conjunction with an *Extract Method*.

The second limitation is the side effects of these recommendations. Bibiano *et al.* [17] reveal that complete composites may lead to side effects like introducing or propagating the incidence of code smells. However, the authors discuss only the side effects surrounding just some recommendations of complete composites. Brito *et al.* [20] recommend *Pull Up Methods* to create a single and more general method in the superclass, achieving code reuse. On the other hand, they did not alert developers about the side effects of *Pull Up Methods* in practice. An example of a side effect is methods with *Feature Envy* or *Long Method* that may have these smells propagated to the superclass after the *Pull Up Methods*. Therefore, we must overcome these limitations to guide developers on how and when to apply each recommendation.

III. MOTIVATING EXAMPLE

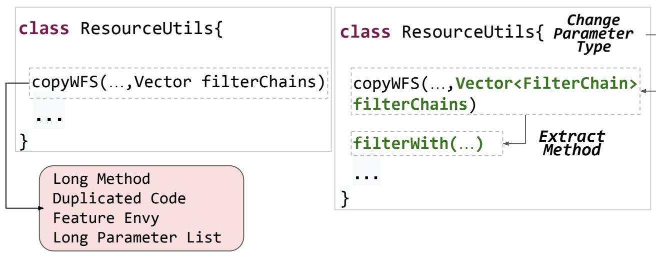
For the motivating example, we rely on a code fragment of the Apache Ant project in which we identified the incidence

TABLE II
EXISTING RECOMMENDATIONS OF COMPLETE COMPOSITES

Complete Composites	Existing Smells	Target Smell	Effect	Side effect
Extract Methods, Move Method(s) and Fine-Grained Refactorings [17]	Long Method, Feature Envy	Feature Envy	Removal of Feature Envy, Long Method	Introduction of Feature Envy(s)
Extract Method and Fine-Grained Refactorings [17]	Long Method, Feature Envy	Long Method	Removal of Long Method	Introduction of Feature Envy and Long Parameter List
Move Method(s) [17]	Feature Envy	Feature Envy	Removal of Feature Envy	-
Move Method(s) [20]	-	-	Improvement of cohesion and coupling	-
Extract Methods, Move Method [20]	-	Duplicated Code	Promotion of reuse and removing duplication	-

of four different code smell types in the same method called `copyWithFilterSets`, shortly `copyWFS`. The method `copyWFS` is responsible for copying a resource based on several filters. The observed smells are *Long Method*, *FeatureEnvy*, *Duplicated Code*, and *Long Parameter List*. This method was refactored in the commit `b7d1e9bde44c` [28], represented on the right in Figure 1.

The *Long Method* and *Duplicated Code* smells are expressed by the several lines of duplicated code addressing the parameter `filterChains`. The method `copyWFS` had some code fragments duplicated with the method `copyWithFilterChainsOrTranscoding` (or `copyWFCT`) from the same class. The excessive duplication resulted in an unnecessarily complex and too-long method. The incidence of *Feature Envy* is due to the recurrent calls to external methods from the class `ChainHeaderHelper`. Finally, the method signature has eight parameters, indicating the incidence of a *Long Parameter List*.



```
class ResourceUtils{

copyWFS(…,Vector filterChains)
...
}
```

```
class ResourceUtils{

copyWFS(…,Vector<FilterChain> filterChains)

filterWith(…)
...
}
```

Fig. 1. Code smells in the commit `af74d1f6b882` [29] (on the left), and the refactoring applied in the commit `b7d1e9bde44c` [28] (on the right).

In the presented commit, the developer opted for applying a composite refactoring formed by *Extract Method* and *Change Parameter Type*. With the *Extract Method*, the developer created a method called `filterWith`. The *Change Parameter Type* was applied over the parameter `filterChains` from the method `copyWFS`. In this case, the developer changed the original data type of `FilterChains` to `Vector<FilterChains>`. Consequently, these refactorings fixed the incidence of *Long Method* and *Duplicated Code* in the class. The change of the parameter type led to the removal of some duplicated lines, while the new method `filterWith` received other duplicated lines. Besides, now both methods `copyWFS` and `copyWFCT` are using the extracted method. However, the composite applied did not fully solve the incidence of *Feature Envy* once this smell was moved to the extracted method. Besides, one may see that the `copyWFS` method remains with a *Long Parameter List*.

From this example, we can observe that the developer applied a composite in a method with multiple instances of code smells, but this composite may be considered incomplete, since it was insufficient to remove all smells previously identified. This phenomenon, frequently observed in the practice during our study, suggests that developers may benefit from hands-on guidelines for supporting the complete removal of multiple code smells. Without that, developers tend to rely only on their intuition and experience for analyzing and deciding which refactoring strategies they should follow. Consequently, developers may have more difficulty identifying the best options for composite refactorings considering their impact on the code structure, including its side effects.

The technical literature proposes some recommendations to guide developers to remove certain smell types [17], [20], [30]. For instance, Bibiano *et al.* [17] and Chavez *et al.* [30] recommend combining *Extract Method* and *Move Method* for removing *Feature Envy* and *Long Method*. However, these studies do not take into account side effects when applying these refactorings in a complex scenario that includes other smell types. For instance, the application of *Extract Method* may be effective to mitigate or remove the *Long Method*. However, the creation of a new method may propagate other smells, such as the *FeatureEnvy* and the *Long Parameter List* once the new method can inherit bad practices, such as the excessive number of parameters and external method calls, from the source method. These side effects especially occur when the developers are unaware of the incidence of other smell types in the source code analyzed.

In summary, we see the need for recommendations aligned with the practice and considering potential effects and side effects. Thus, it is necessary to investigate concrete cases in which code smells are introduced, which is the goal of our study described in the next section.

IV. STUDY SETTINGS

Our study aims to enhance and assess the recommendations of complete composite refactorings to be used in practice. Aiming at conceiving this enhancement, we consolidate the previous empirical-driven recommendations [17], [20] and extract knowledge from complete composites applied in 42 real software projects. Regarding their latter, we collected the frequent combinations forming complete composites applied in practice and their side effects, improving the existing recommendations. In this section, we describe our research questions (RQs) and the steps of our study.

RQ$_1$: *What are the most frequent combinations in complete composite refactorings in practice?* – **RQ$_1$** aims at identifying and analyzing the most frequent refactoring combinations in complete composites. We consider two aspects: (i) the frequency in which each combination appears as a whole; and (ii) the fine-grained refactoring types that appear the most in frequent complete composites (as explained in Section II-D). Additionally, our analysis helps us understand the actual contribution of each refactoring type (e.g., *Extract Method* and *Move Method*) on the complete introduction or removal of code smells. From these observations, we can improve existing recommendations and suggestions derived from practice.

RQ$_2$: *What are the side effects of the most frequent complete composite refactorings?* – Complementary to the previous research question, **RQ$_2$** aims to identify the side effects of the most frequent complete composite refactorings in terms of introduction, removal, and prevalence of code smells. Additionally, we analyze the propagation of code smells, i.e., when an existing code smell is moved to other parts of the source code. By answering **RQ$_2$**, we can derive a clear understanding of the side effects of the most frequent complete composite refactorings. This understanding is of paramount importance to avoid misinforming developers on refactorings that may not remove smells effectively. As composites also aim to improve design quality, the application of a complete composite should ideally remove smells, as intended by developers, without creating smells in other parts of the codebase.

A. Study Steps and Procedures

Figure 2 illustrates our study steps and dataset. Study steps are mainly related to the data collection and analyses. The *replication package* is available in [31].

Step 1: Software Project Selection. We selected 42 software projects from GitHub according to the following criteria: (i) the software project must be implemented in Java due to the availability of robust tools for software analysis; (ii) the software project must use Git as the main version control system because state-of-the-art tools for refactoring detection work on Git projects only; and (iii) the software project must have been investigated by at least one related study regarding refactoring [13], [17], [20], [25]. Previous studies reported software communities that have a refactoring culture in some software projects [13], [17], [20], [25], which is relevant because it shows developers' concerns about refactoring.

Step 2: Single Refactoring Detection. For detecting single refactorings applied in practice, we used the RefMiner 2.0 tool [32] due to its high precision and recall levels (98% and 87%, respectively). Besides, the tool supports a total of 52 refactoring types [33]. In this study, we considered 34 refactoring types that are applied in the code scope of attributes, methods, and classes. A recent study [17] showed that fine-grained refactoring types are often applied in complete composites. Table I summarizes the 34 refactoring types investigated in our work.

TABLE III
CODE SMELL TYPES ANALYZED IN THIS STUDY

Code Smell Type	ID	Definition
Method level		
Brain Method	BrM	Method overloaded with software features
Dispersed Coupling	DsC	Method that calls too many methods
Feature Envy	FeE	Method "envying" other classes' features
Intensive Coupling	InC	Method that depends much on other ones
Long Method	LoM	Too long and complex method
Long Parameter List	LPL	Too many parameters in a method
Message Chain	MeC	Too long chain of method calls
Shotgun Surgery	ShS	Method whose changes affect other ones
Class level		
Brain Class	BrC	Class overloaded with software features
Class Data should be Private	CDSBP	Class that overexposes its attributes
Complex Class	CoC	Too complex software features in a class
Data Class	DaC	Only data management features in a class
God Class	GoC	Too many software features in a class
Lazy Class	LaC	Too short and simple class
Refused Bequest	ReB	Child class rarely uses parent class features
Spaghetti Code	SpC	Too much code deviation and nesting
Speculative Generality	SpG	Useless abstract class

Step 3: Composite Refactoring Computation. For the detection of composite refactorings, we created a script based on the *range-based* heuristic. As mentioned in Section II-A this heuristic captures refactorings that were applied on a common set of code elements (classes and/or methods) and implemented by the same developer. In that way, we can capture the developer's intent to improve the internal software quality of this set of code elements from a composite. Thus, this heuristic best fits our study goal, since it considers multiple code elements. The reliability of this heuristic was demonstrated in [13], [14], [17]. More details about the *range-based* heuristic are available in [13]. Our script was developed in Java, being tested and validated by two authors. This script can detect composites formed by refactorings applied in an isolated class or multiple classes, as expected by the *range-based* heuristic [13]. We better detailed our script on [31].

Step 4: Code Smell Detection. Similarly to Bibiano *et al.* [17] and studies that proposed recommendations of composites [12], [13], we used the Organic tool [34] for detecting code smells in our study. Organic can detect 17 code smell types. The Organic tool uses detection strategies based on code metrics to collect code smells. These detection strategies have already been evaluated by prior studies [18], [34], [35]. Besides that, the *range-based* heuristic captures composites applied on multiple classes. Then, it motivated us to investigate the effect of complete composites on code smells that involve multiple classes. Table III lists the 19 code smell types analyzed in our study.

Step 5: Complete Composite Computation. We focused on the complete composites for removing the 19 collected code smell types (Table III). These code smells are very common and can be removed from refactoring types investigated in this study. We then elaborated Table II that presents the recommended composites for the removal of some code

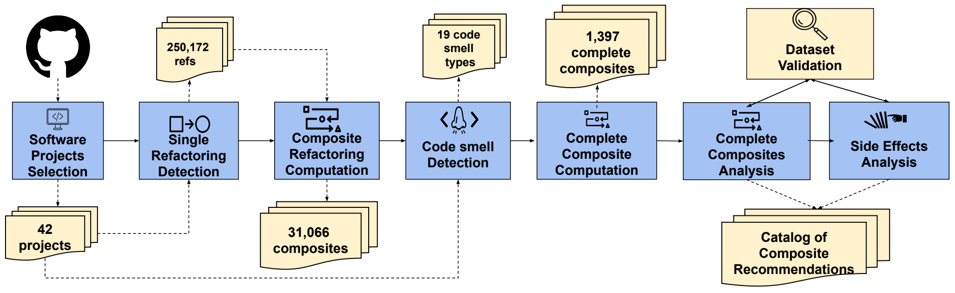

250,172 refs
19 code smell types
1,397 complete composites
Dataset Validation
Software Projects Selection
Single Refactoring Detection
Composite Refactoring Computation
Code smell Detection
Complete Composite Computation
Complete Composites Analysis
Side Effects Analysis
42 projects
31,066 composites
Catalog of Composite Recommendations

Fig. 2. Study Steps

smells according to prior studies [17], [20]. However, these recommendations are limited to the removal of four code smell types only. Thus, we aim to extend these recommendations to other code smell types investigated in our study. For the full removal of code smells, we created a formal definition for completeness based on a related work [17] (see Section II-B). We then elaborated a script to collect complete composites according to our definition. This script was manually tested and validated by the authors (see details in Section VII).

Step 6: Complete Composite Analysis. Aiming to identify frequent combinations for composing complete composites, we created scripts to group complete composites in types. We follow the definition of composite types presented in [17] (see Section II-A). We collected the frequent combinations between groups. An example of that is when we have a group g1 of composite types formed by g1=[*Extract Method(s)*, *Move Method(s)*, *Change Return Type(s)*], and another group g2 = [*Extract Method(s)*, *Change Return Type(s)*]. We can observe that the combination c1=[*Extract Method(s)*, *Change Return Type(s)*] is common between these groups g1 and g2.

Step 7: Side Effects Analysis. We collected the side effects (i.e., code smells introduced, removed, and unaffected) by the most frequent complete composites identified in the previous step. Then, three authors manually analyzed the effect of complete composites. Additionally, we focus on finding the relation between the introduction of code smells and the complete composites that removed the target code smell. For each composite, we analyzed the code snippets refactored, other code changes, the arguments for commit, and pull request discussions in the commits in which the complete composites were applied. This in-depth analysis allowed us to understand whether other code changes could have introduced the code smell, and if developers knew these code smells. The findings of this step helped us to complement our catalog with the side effects of complete composites. We summarized our results from Steps 6 and 7 in a catalog with recommendations of common complete composites and their possible side effects. A summary of this catalog is presented in Section VI.

Step 8: Dataset Validation. We randomly selected a sample formed by 36 complete composites from our dataset for validation. Six developers validated whether the composites were complete for code smells that were detected. For the validation, we provided a table to the developers with the following composite data: refactoring types that form each composite, the project name and the commits where the composite was applied, the code element names that were touched for each composite, and the set of code smells of these code elements before and after the application of composite refactorings. Each developer had one week to evaluate six complete composites according to their availability. After this period, 28 composite refactorings were evaluated: two developers evaluated six composites, and four developers evaluated four composites. According to the developers, 24 composites were complete for at least one code smell that was detected.

V. Results

In this section, we present and discuss our results, highlighting the most frequent combinations observed in complete composite refactorings and their associated side effects.

A. Frequent combinations in Complete Composites (*$\boldsymbol{RQ}_1$*)

Table IV presents the most frequent combinations in refactoring types. We found that three coarse-grained refactoring types, namely *Extract Method*, *Move Method*, and *Move Class*, are commonly applied with fine-grained refactoring types.

This table also shows that 132 (28%) out of 462 complete composites have at least one *Extract Method* combined with *Change Variable Type(s)*. We observed that the *Change Variable Types* and *Change Parameter Types* help to simplify or remove some code statements, decreasing lines of code and minimizing the excessive method calls of external classes. Thus, these combinations of refactoring types are appropriate to remove code smells like *Long Methods* or *Feature Envies*.

However, it is unclear if these two code smells often coexist in the same method and commit. To investigate this, we analyzed the frequency of *Long Methods* and *Feature Envy* that occur in conjunction. We randomly selected 5,000 commits from 13 software projects in our dataset to investigate the frequency of methods with these two code smells. In our sample of 123,100 long methods, our analysis revealed that 60% of the long methods are also envious methods. We conjecture that developers are usually aware of only one smell, and even when both smells are known, our analysis shows that they are not typically addressed together.

Our results also reveal that developers frequently changed the type of the method return (23%) or parameter(s) (22%)

TABLE IV
MOST FREQUENT COMBINATIONS IN COMPLETE COMPOSITES

#CC with at least one Extract Method = 462	
Combination	#CC(%)
[Change Variable Type, Extract Method]	132 (28,57%)
[Change Return Type, Extract Method]	107 (23,16%)
[Change Parameter Type, Extract Method]	102 (22,08%)
[Extract Variable, Extract Method]	96 (20,78%)
[Change Return Type, Change Variable Type, Extract Method]	69 (14,93%)
#CC with at least one Move Method = 183	
[Change Parameter Type, Move Method]	65 (35,52%)
[Extract Class, Move Method]	64 (34,97%)
[Change Variable Type, Move Method]	53 (28,96%)
[Change Attribute Type, Move Method]	52 (28,42%)
[Change Return Type, Move Method]	47 (25,68%)
#CC with at least one Move Class = 317	
[Change Variable Type, Move Class]	91 (28,70%)
[Change Attribute Type, Move Class]	81 (25,55%)
[Change Paramter Type, Move Class]	77(24,30%)
[Change Return Type, Move Class]	63 (19,87%)
[Change Parameter Type, Change Return Type, Move Class]	42 (13,25%)
#CC with at least one Extract Method and Move Method = 62	
[Extract Method, Move Method]	62 (100%)
[Change Variable Type, Extract Method, Move Method]	29 (46,77%)
[Change Parameter Type, Extract Method, Move Method]	24 (38,71%)
[Change Return Type, Extract Method, Move Method]	24 (38,71%)
[Extract Variable, Rename, Extract Method, Move Method]	21 (33,87%)

when extracting methods. Besides, we observed that it is not common to form a composite with *Change Parameter Type* and *Change Return Type* together with the same instance of *Extract Method*. Despite these refactoring types being simple, they can be related to major code changes. In other words, when developers applied a *Change Parameter Type(s)* and *Extract Method(s)* or changed the return of a method, they need to update all methods that were calling extracted ones. Yet, developers need to change the parameter(s) in each call of the method and also adapt the source code to perform the method extraction.

Therefore, we noted that developers often apply a single type of fine-grained refactoring combined with a single coarse-grained refactoring type. Besides, we observe that the more frequent fine-grained refactorings in complete composites address the changing of data types, including attributes, parameters, variables, and returning data. For instance, Table IV shows that the developers frequently extract methods combined with changing variable types. We may interpret this decision as a cautious strategy for avoiding the incidence of side effects and then reducing rework on maintenance. Previous work revealed that performing multiple and simultaneous structural modifications leads to the frequent introduction of new code smell instances in the source code [12].

Finding 1: Developers tend to apply a single type of coarse-grained refactoring with a single type of fine-grained refactoring. The fine-grained ones often address changing data types.

*B. Side Effects of the Frequent Combinations in Complete Composites (**RQ**$_2$)*

Figure 3 presents the smell incidences as side effects over the most frequent combinations. Figure 3(a) shows how the incidence of different smell types was affected by refactorings, combining *Extract Method* with *Change Variable Type*. One would expect this composite refactoring can affect only smells at the method level, since the main refactoring (i.e., *Extract Method*) is a method-level transformation. However, class-level smells can also be affected through this combination when the developer changes the variable type. For instance, a prior study [17] reports that *Extract Methods* and fine grain refactorings frequently (75%) introduce *Long Parameter Lists* (LPLs). Differently, our results reveal that the combination *Extract Methods* and *Change Variable Type* introduced about 38% of LPLs. This inconsistent result indicates the need for a deep understanding of the side effects caused by each combination in complete composites. We understand that this prior related work generalized the side effects for all combinations with at least one fine-grained refactoring [17]. However, each recurring combination has a particular side effect.

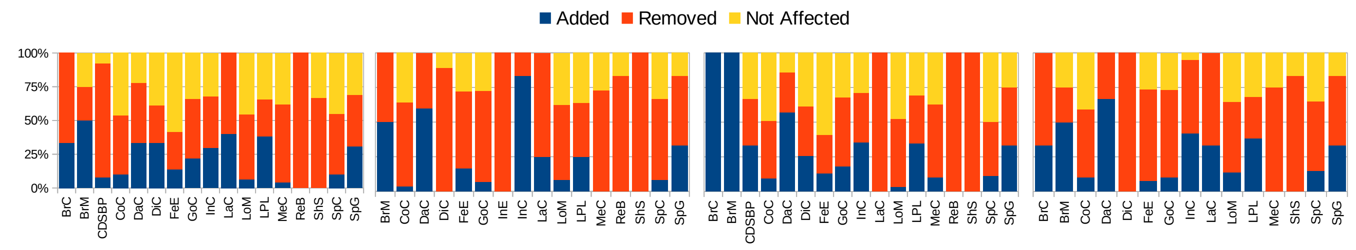
As discussed in the previous RQ, developers apply many code modifications to support a simple combination formed by coarse and fine-grained refactorings. Generally, such combinations are applied with non-refactoring code changes. This fact can explain the introduction (50%) of *Brain Methods* (BrM), and the prevalence (45%) of *Long Methods* (LoM). Despite the developers' aim to reduce the size and complexity of methods when extracting code, *Brain Methods* have been introduced due to other code modifications related to the change of variable type, thereby increasing code complexity. The method size is not reduced as expected, and the *Long Methods* are not affected. This analysis leads to another finding of our study.

Finding 2: The complexity of methods tends to increase when Extract Methods and Change Variable Types are applied together.

Surprisingly, *Move Methods* and *Change Parameter Types* can be related to the introduction (83%) of *Intensive Couplings* (InC), as shown in Figure 3(b). Generally, when a method is moved, developers tend to introduce more calls to the methods from other classes. Consequently, some parameters are also modified because the method uses attributes of other classes, explaining why *Intensive Couplings* are frequently introduced.

Figure 3(c) illustrates the side effects of complete composites formed by *Extract Method(s)* and *Change Parameter Type(s)*. This combination can effectively remove (49%) *Long Methods*, especially those containing duplicated code. As observed in the motivating example (Section III), developers changed parameter types to facilitate code statement removal, and along with *Extract Methods*, this led to the elimination of *Long Methods* and *Duplicated Code*. However, our understanding of the relation between *Duplicated Code* and *Long Methods* is limited, as the Organic tool did not identify Duplicated Methods.

To corroborate our analysis, we utilized PMD CPD [36] and developed Java scripts to extract duplicated methods with at least 30 duplicate statements, as per CPD rules. We analyzed the same set of 123,100 commits analyzed previously to determine the frequency of duplicated methods that were



(a) Extract Method(s), Change Variable Type(s)

(b) Move Method(s) and Change Parameter Type(s)

(c) Extract Method(s) and Change Parameter Type(s)

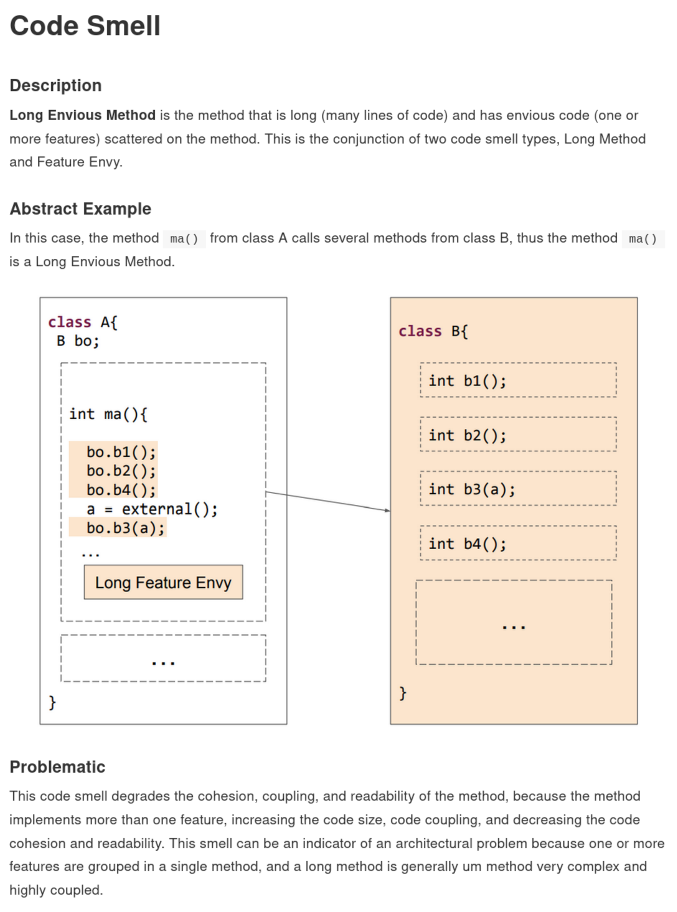
(d) Extract Method(s) and Move Method(s)

Fig. 3. Side Effect of Common Complete Composites

also long in the same commit. However, the occurrence of these two code smells in conjunction is not frequent (4%), but developers often fully remove them using *Extract Method* and *Change Parameter Types*. This empirical knowledge can provide recommendations.

Extract Methods and *Move Methods* are frequently recommended to remove *Feature Envies* [13], [17]. Figure 3(d) shows that this combination can fully indeed remove 66% of Feature Envies (FeE). We also observed that about 26% of *Feature Envies* are not affected when developers extract and move methods. In that case, developers need to be aware of this situation. Composite refactorings formed by extractions and moves of methods can be related to the introduction of *Long Parameter Lists* (38%) and *Intensive Coupling* (42%). Bibiano *et al.* [17] suggested that this combination can introduce long lists of parameters, but they did not report the proportion of this side effect. Our results revealed that this side effect is not very frequent in practice, but it can happen. *Long Parameter Lists* can be introduced because many variables are transformed in parameters when methods are extracted, as suggested by the prior work. Existing recommendations do not alert about the introduction of *Intensive Coupling*. A possible cause of this side effect is the addition of many calls of methods from other classes when the developer moves a method, increasing the coupling of the class. This leads us to the following finding.

Finding 3: The application of Move Methods tends to increase the coupling of classes, regardless of the full removal of Feature Envies.



Code Smell

Description

Long Envious Method is the method that is long (many lines of code) and has envious code (one or more features) scattered on the method. This is the conjunction of two code smell types, Long Method and Feature Envy.

Abstract Example

In this case, the method `ma()` from class A calls several methods from class B, thus the method `ma()` is a Long Envious Method.

```
class A{
 B bo;

  int ma(){

    bo.b1();
    bo.b2();
    bo.b4();
    a = external();
    bo.b3(a);
   ...
    Long Feature Envy

  ...
}
```

```
class B{
  int b1();
  int b2();
  int b3(a);
  int b4();
  ...
}
```

Problematic

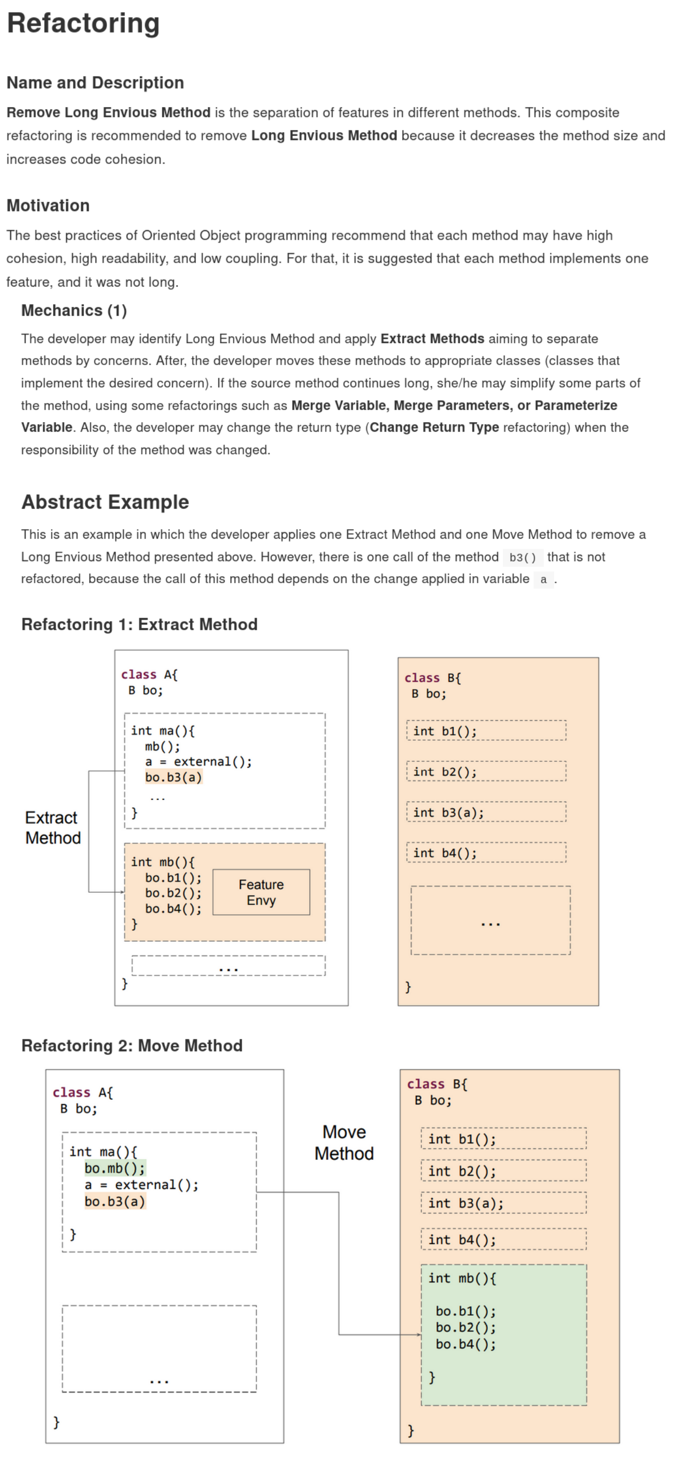
This code smell degrades the cohesion, coupling, and readability of the method, because the method implements more than one feature, increasing the code size, code coupling, and decreasing the code cohesion and readability. This smell can be an indicator of an architectural problem because one or more features are grouped in a single method, and a long method is generally um method very complex and highly coupled.

Fig. 4. Catalog: code smell section.

VI. An Enhanced Catalog of Complete Composites

Based on our results, we created a catalog of composite recommendations. The catalog is available online [37], and examples of its section are presented in Figures 4 to 8.

Previous composite recommendations focused on removing a single code smell, but our catalog takes a different approach. Based on our quantitative analyses, we extracted recommendations by examining code smells that were fully and frequently removed by the same composite refactoring pattern. In this sense, we created two new types of code smell. We called *Long Envious Method* (see Figure 4), which occurs when both *Long Method* and *Feature Envy* are detected on the same method. The second type, *Long-signed Clone* is the junction of *Long Method* and *Duplicated Code*, and denotes when a method is long and duplicated because one or more parameters require the application of many repetitive statements. Subsequently, we retrieved four recommendations that remove these two new code smell types. *Extract Methods* and *Move Methods* are commonly applied for the removal of *Long Envious Methods*. Our catalog suggests two mechanics: the first involves extraction followed by moving (see Figure 5), while the second involves extraction and moving at the same time (see online [37]). Although both mechanics yield similar



Refactoring

Name and Description

Remove Long Envious Method is the separation of features in different methods. This composite refactoring is recommended to remove **Long Envious Method** because it decreases the method size and increases code cohesion.

Motivation

The best practices of Oriented Object programming recommend that each method may have high cohesion, high readability, and low coupling. For that, it is suggested that each method implements one feature, and it was not long.

Mechanics (1)

The developer may identify Long Envious Method and apply **Extract Methods** aiming to separate methods by concerns. After, the developer moves these methods to appropriate classes (classes that implement the desired concern). If the source method continues long, she/he may simplify some parts of the method, using some refactorings such as **Merge Variable, Merge Parameters, or Parameterize Variable**. Also, the developer may change the return type (**Change Return Type** refactoring) when the responsibility of the method was changed.

Abstract Example

This is an example in which the developer applies one Extract Method and one Move Method to remove a Long Envious Method presented above. However, there is one call of the method `b3()` that is not refactored, because the call of this method depends on the change applied in variable `a`.

Refactoring 1: Extract Method

```
class A{
 B bo;

  int ma(){
    mb();
    a = external();
    bo.b3(a)
    ...
  }

  int mb(){
    bo.b1();
    bo.b2();
    bo.b4();
  }

  ...
}
```

```
class B{
 B bo;

  int b1();

  int b2();

  int b3(a);

  int b4();

  ...

}
```

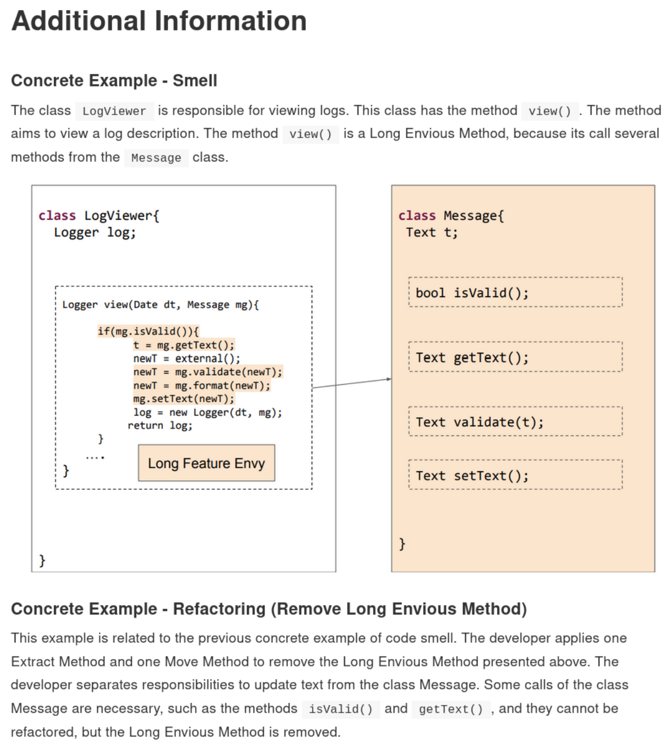
Refactoring 2: Move Method

```
class A{
 B bo;

  int ma(){
    bo.mb();
    a = external();
    bo.b3(a)

  }

  ...

}
```

```
class B{
 B bo;

  int b1();
  int b2();
  int b3(a);
  int b4();

  int mb(){

    bo.b1();
    bo.b2();
    bo.b4();

  }
}
```

Fig. 5. Catalog: refactoring section.

refactoring outcomes, we proposed these mechanics because developers might need to move two or more methods that were extracted to different classes using the first mechanics. We believe that the second mechanics is more complex to use. For the removal of *Long-signed Clone*, we observed that *Extract Method(s)* is one alternative to remove this smell. Another frequent alternative is the application of *Extract Method(s)* and *Change Parameter Type(s)*.



Additional Information

Concrete Example - Smell

The class `LogViewer` is responsible for viewing logs. This class has the method `view()`. The method aims to view a log description. The method `view()` is a Long Envious Method, because its call several methods from the `Message` class.

```
class LogViewer{
  Logger log;

  Logger view(Date dt, Message mg){

      if(mg.isValid()){
          t = mg.getText();
          newT = external();
          newT = mg.validate(newT);
          newT = mg.format(newT);
          mg.setText(newT);
          log = new Logger(dt, mg);
         return log;
      }
    ....
  }

}
```

```
class Message{
 Text t;

  bool isValid();

  Text getText();

  Text validate(t);

  Text setText();

}
```

Concrete Example - Refactoring (Remove Long Envious Method)

This example is related to the previous concrete example of code smell. The developer applies one Extract Method and one Move Method to remove the Long Envious Method presented above. The developer separates responsibilities to update text from the class Message. Some calls of the class Message are necessary, such as the methods `isValid()` and `getText()`, and they cannot be refactored, but the Long Envious Method is removed.

Fig. 6. Catalog: concrete smell example section.

Our catalog also provides additional information with concrete examples of the code smell (see Figure 6) and the refactoring (see Figure 7). Furthermore, our catalog describes the potential side effects of such composites (see Figure 8).

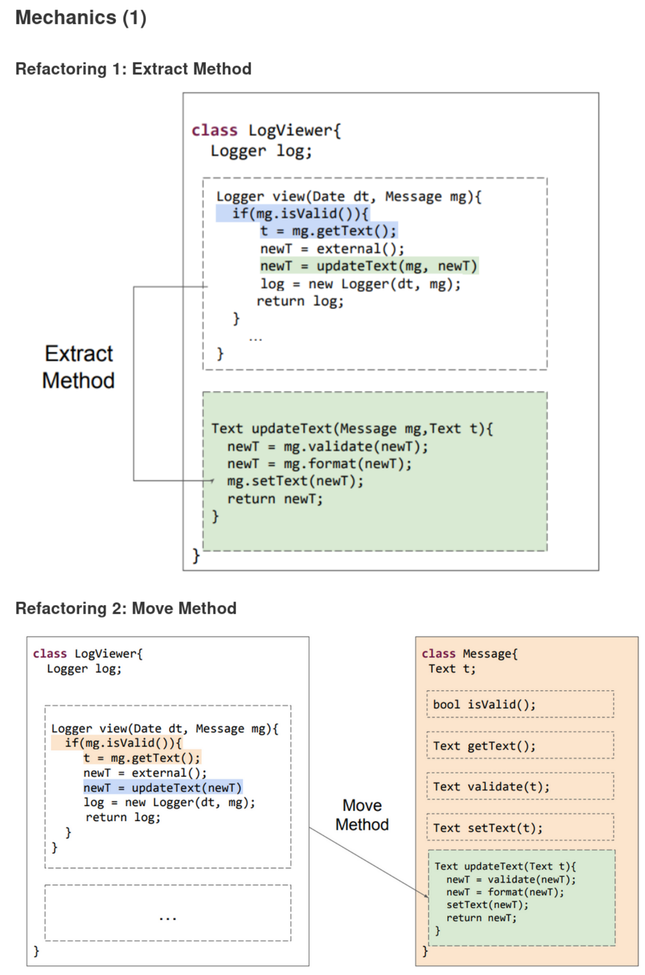
VII. Threats to Validity

This section presents and discusses threats to validity [38].

Construct Validity. Relying purely on automated detection tools may be risky for identifying code smells and refactorings [39]. However, manually validating large-scale samples is unfeasible, especially for different projects. To mitigate this threat, we carefully selected the tools employed: RefMiner 2.0 and Organic. Both tools are highly accurate for refactoring detection and code smell identification (see Section IV).

Another threat addresses the risk of developers merging one or more commits into a single commit (squash commits). These commits would address independent and disconnected refactorings, which is incompatible with the definition of composite refactorings. In this sense, one common symptom of squash commits is the large time gap between the changes performed. Again, our decision to use RefMiner 2.0 is beneficial once this tool was designed to ignore squash commits [40]. As a result, the time interval between commits analyzed in our study is short, i.e., two weeks on average.

The heuristics to detect complete composites might bias the results. To mitigate this threat, we employed the heuristics proposed by Sousa *et al.* [13] for detecting composite refactorings, combined with the definition of complete composites


Mechanics (1)

Refactoring 1: Extract Method

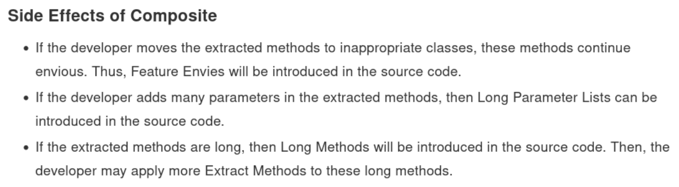
```
class LogViewer{
  Logger log;

  Logger view(Date dt, Message mg){
    if(mg.isValid()){
        t = mg.getText();
        newT = external();
        newT = updateText(mg, newT)
        log = new Logger(dt, mg);
       return log;
    }
      ...
  }

  Text updateText(Message mg,Text t){
    newT = mg.validate(newT);
    newT = mg.format(newT);
    mg.setText(newT);
    return newT;
  }
}
```

Refactoring 2: Move Method

```
class LogViewer{
  Logger log;

  Logger view(Date dt, Message mg){
    if(mg.isValid()){
        t = mg.getText();
        newT = external();
        newT = updateText(newT)
        log = new Logger(dt, mg);
        return log;
    }
  }

        ...
}
```

```
class Message{
 Text t;

 bool isValid();

 Text getText();

 Text validate(t);

 Text setText(t);

 Text updateText(Text t){
   newT = validate(newT);
   newT = format(newT);
   setText(newT);
   return newT;
 }
}
```

Fig. 7. Catalog: concrete refactoring example section.


Side Effects of Composite

- If the developer moves the extracted methods to inappropriate classes, these methods continue envious. Thus, Feature Envies will be introduced in the source code.
- If the developer adds many parameters in the extracted methods, then Long Parameter Lists can be introduced in the source code.
- If the extracted methods are long, then Long Methods will be introduced in the source code. Then, the developer may apply more Extract Methods to these long methods.

Fig. 8. Catalog: side effects section.

proposed in previous work [17]. The concept of completeness used in our study is based on these works (see Section II-B).

Internal Validity. The complete composites used in our studies were detected by scripts written by the authors of this paper. We implemented unit tests to assert that all scripts would perform the expected behavior. Besides, two authors double-checked the scripts and results of the unit tests, mitigating the risk of validation bias.

Conclusion Validity. Our definition of "completeness" for classifying composite refactorings [17] is based on rigid thresholds established by code smell detection tools [34]. Therefore, this definition may lead to misclassification. Besides the already reported robustness of Organic, we also relied on asking developers about their agreement with the thresholds employed for code smell detection (see Section IV). To identify the most frequent combinations in complete composites (RQ$_1$), we should keep in mind that some sequences of refactorings would not be performed to intentionally remove code smells. To mitigate this threat, the authors manually assessed which refactoring instances actually contributed to partially or completely eliminating the code smells detected.

To mitigate threats related to the automated identification of side effects (RQ$_2$), two authors manually analyzed the severity and intensity of samples of smells propagated and introduced by complete composite refactorings.

External Validity. Considering the nature of this study, we do not intend to claim the generalization of our findings. However, we made efforts to employ heterogeneous samples of projects and participants. We analyzed projects having different sizes and addressing different domains. Besides, we found consistent results for different subsets.

VIII. Conclusion

Given the limitations of existing catalogs of refactorings, we presented in this work an enhanced catalog of complete composite refactorings. We conducted a large-scale study on 42 software projects, collecting 1,397 complete composites that are used to remove 17 smell types. The main findings of our study include (i) the identification of the most frequent combinations in complete composites applied in the practice, and (ii) the side effects of complete composites.

Our main contribution is the recommendations derived from the practice, which includes four complete composites to remove code smells. These recommendations can guide developers to perform composite refactorings, while alerting developers about the possible side effects. In future work, we intend to extend our recommendations, explaining possible motivations in which each composite can be applied to fully remove two or more code smells.

Acknowledgment

This work is supported by CNPq grants 404027/2023-7, 404406/2023-8, 434969/2018-4, 427787/2018-1, 409536/2017-2, 141180/2021-8, 141276/2020-7, and 312149/2016-6; CAPES grants 175956 and 88887.900069/2023-00, FAPERJ grant 22520-7/2016, FAPERJ PDR-10 program 202073/2020, FAPEAL grant 60030.0000002357/2022, and FUNCAP grant BP5-00197-00042.01.00/22.

References

[1] D. Galin, *Software quality: concepts and practice*. John Wiley & Sons, 2018.

[2] C. Y. Laporte and A. April, *Software quality assurance*. John Wiley & Sons, 2018.

[3] A. Uchôa, C. Barbosa, W. Oizumi, P. Blenilio, R. Lima, A. Garcia, and C. Bezerra, "How does modern code review impact software design degradation? an in-depth empirical study," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 511–522.

[4] A. Baabad, H. B. Zulzalil, S. Hassan, and S. B. Baharom, “Software architecture degradation in open source software: A systematic literature review,” *IEEE Access*, vol. 8, pp. 173 681–173 709, 2020.

[5] W. Oizumi, L. Sousa, A. Oliveira, L. Carvalho, A. Garcia, T. Colanzi, and R. Oliveira, “On the density and diversity of degradation symptoms in refactored classes: A multi-case study,” in *IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, 2019, pp. 346–357.

[6] M. Fowler, *Refactoring: Improving the Design of Existing Code*, 1st ed. Addison-Wesley Professional, 1999.

[7] D. Oliveira, W. K. G. Assunção, A. Garcia, B. Fonseca, and M. Ribeiro, “Developers’ perception matters: machine learning to detect developer-sensitive smells,” *Empirical Software Engineering*, vol. 27, no. 7.

[8] N. Yoshida, T. Saika, E. Choi, A. Ouni, and K. Inoue, “Revisiting the relationship between code smells and refactoring,” in *24th International Conference on Program Comprehension (ICPC)*, 2016, pp. 1–4.

[9] D. Silva, N. Tsantalis, and M. T. Valente, “Why we refactor? Confessions of GitHub contributors,” in *24th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2016, pp. 858–870.

[10] E. Fernandes, A. Chávez, A. Garcia, I. Ferreira, D. Cedrim, L. Sousa, and W. Oizumi, “Refactoring effect on internal quality attributes: What haven’t they told you yet?” *Information and Software Technology*, vol. 126, p. 106347, 2020.

[11] E. Murphy-Hill, C. Parnin, and A. Black, “How we refactor, and how we know it,” *IEEE Transactions on Software Engineering (TSE)*, vol. 38, no. 1, pp. 5–18, 2012.

[12] A. C. Bibiano, E. Fernandes, D. Oliveira, A. Garcia, M. Kalinowski, B. Fonseca, R. Oliveira, A. Oliveira, and D. Cedrim, “A quantitative study on characteristics and effect of batch refactoring on code smells,” in *13th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2019, pp. 1–11.

[13] L. Sousa, D. Cedrim, A. Garcia, W. Oizumi, A. C. Bibiano, D. Tenorio, M. Kim, and A. Oliveira, “Characterizing and identifying composite refactorings: Concepts, heuristics and patterns,” in *17th International Conference on Mining Software Repositories (MSR)*, 2020.

[14] A. C. Bibiano, V. Soares, D. Coutinho, E. Fernandes, J. Correia, K. Santos, A. Oliveira, A. Garcia, R. Gheyi, B. Fonseca, M. Ribeiro, C. Barbosa, and D. Oliveira, “How does incomplete composite refactoring affect internal quality attributes?” in *28th International Conference on Program Comprehension (ICPC)*, 2020.

[15] A. C. Bibiano, A. Uchôa, W. K. Assunção, D. Tenório, T. E. Colanzi, S. R. Vergilio, and A. Garcia, “Composite refactoring: Representations, characteristics and effects on software projects,” *Information and Software Technology*, vol. 156, p. 107134, 2023.

[16] M. Kim, T. Zimmermann, and N. Nagappan, “An empirical study of refactoring: Challenges and benefits at Microsoft,” *IEEE Transactions on Software Engineering (TSE)*, vol. 40, no. 7, pp. 633–649, 2014.

[17] A. C. Bibiano, W. Assunçao, D. Coutinho, K. Santos, V. Soares, R. Gheyi, A. Garcia, B. Fonseca, M. Ribeiro, D. Oliveira *et al.*, “Look ahead! revealing complete composite refactorings and their smelliness effects,” in *37th International Conference on Software Maintenance and Evolution (ICSME)*, 2021.

[18] 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 ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2017, pp. 465–475.

[19] G. Lacerda, F. Petrillo, M. Pimenta, and Y. G. Guéhéneuc, “Code smells and refactoring: A tertiary systematic review of challenges and observations,” *Journal of Systems and Software (JSS)*, vol. 167, p. 110610, 2020.

[20] A. Brito, A. Hora, and M. Tulio Valente, “Towards a catalog of composite refactorings,” *Journal of Software: Evolution and Process*, vol. 36, no. 4, p. e2530, 2024.

[21] Y. Lin, X. Peng, Y. Cai, D. Dig, D. Zheng, and W. Zhao, “Interactive and guided architectural refactoring with search-based recommendation,” in *24th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2016, pp. 535–546.

[22] W. Oizumi, A. C. Bibiano, D. Cedrim, A. Oliveira, L. Sousa, A. Garcia, and D. Oliveira, “Recommending composite refactorings for smell removal: Heuristics and evaluation,” in *34th Brazilian Symposium on Software Engineering (SBES)*, 2020, pp. 72–81.

[23] G. Szőke, G. Antal, C. Nagy, R. Ferenc, and T. Gyimóthy, “Empirical study on refactoring large-scale industrial systems and its effects on maintainability,” *Journal of Systems and Software (JSS)*, vol. 129, pp. 107–126, 2017.

[24] A. C. Bibiano and A. Garcia, “On the characterization, detection and impact of batch refactoring in practice,” in *34th Brazilian Symposium on Software Engineering Software Engineering - Doctoral and Master Theses Competition (SBES-CTD)*. Porto Alegre, RS, Brasil: SBC, 2020, pp. 165–179. [Online]. Available: https://sol.sbc.org.br/index.php/cbsoft_estendido/article/view/14626

[25] A. Brito, A. Hora, and M. T. Valente, “Refactoring graphs: Assessing refactoring over time,” in *26th Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2019, pp. 504–507.

[26] D. Tenorio, A. C. Bibiano, and A. Garcia, “On the customization of batch refactoring,” in *3rd International Workshop on Refactoring, co-alocated International Conference on Software Engineering (IWoR-ICSE)*. IEEE Press, 2019, pp. 13–16.

[27] M. Cinnéide and P. Nixon, “Composite refactorings for java programs,” in *14th ECOOP (2000)*, 2000, pp. 129–135.

[28] Ant. (2017) Apache ant. Available at: https://github.com/apache/ant/commit/b7d1e9bde44c.

[29] ——. (2017) Apache ant. Available at: https://github.com/apache/ant/commit/af74d1f6b882.

[30] 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.

[31] A. C. Bibiano. (2022) Complete composite website. [Online]. Available: https://compositerefactoring.github.io/site/

[32] N. Tsantalis, A. Ketkar, and D. Dig, “Refactoringminer 2.0,” *IEEE Transactions on Software Engineering*, 2020.

[33] N. Tsantalis, M. Mansouri, L. Eshkevari, D. Mazinanian, and D. Dig, “Accurate and efficient refactoring detection in commit history,” in *40th International Conference on Software Engineering (ICSE)*, 2018, pp. 483–494.

[34] 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.

[35] 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.

[36] PMD. (2024) An extensible cross-language static code analyzer. Available at: https://pmd.github.io/latest/pmd_userdocs_cpd.html.

[37] A. C. Bibiano, D. Coutinho, A. Uchôa, W. Assunçao, , A. Garcia, D. Oliveira, R. de Mello, T. Colanzi, B. Fonseca, and A. Vasconcelos. (2023) Catalog of complete composites. [Online]. Available: https://compositerefactoring.github.io/catalog

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

[39] R. de Mello, R. Oliveira, A. Uchôa, W. Oizumi, A. Garcia, B. Fonseca, and F. de Mello, “Recommendations for developers identifying code smells,” *IEEE Software*, 2022.

[40] N. Tsantalis, A. Ketkar, and D. Dig, “Refactoringminer 2.0,” *IEEE Transactions on Software Engineering (TSE)*, 2020.