A Large-Scale Study on Recommendations of Composite Refactorings in Practice

Ana Carla Bibiano[1], **Alessandro Garcia**[1], **Wesley K. G. Assunção**[2]

[1]Department of Informatics – Pontifical Catholic University of Rio de Janeiro (PUC-Rio)
Rio de Janeiro, Brazil.

[2]Department of Computer Science – North Carolina State University (NCSU)
Raleigh, North Carolina, USA.

abibiano@inf.puc-rio.br, afgarcia@inf.puc-rio.br, wguezas@ncsu.edu

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, previous studies revealed that only 10% of composite refactorings in practice are considered complete, i.e., those fully removing code smells. However, the literature is scarce to catalog the most common complete composites and their side effects. Aim: To fill this gap, we performed a mixed-method study to create an enhanced catalog of complete composites derived from practice knowledge. Method: We empirically collected the most common complete composites and their side effects applied in 42 open and closed-source projects, generating a catalog of composite recommendations. Then, our catalog was evaluated by 21 software developers. Results: The derived catalog includes four recommendations to fully remove multiple code smells, minimizing possible side effects. The evaluation indicates that 85% of developers reported that their own solutions could have worse side effects without our recommendations, increasing the effort to remove smells critical to them. Conclusion: Our catalog can guide both developers and researchers on how to better combine refactorings in composites that effectively reduce side effects in the practice.*

Resumo. *A refatoração é uma atividade não-trivial de manutenção de software. Desenvolvedores gastam muito tempo e esforço refatorando o código para remover problemas estruturais, tais como anomalias de código. Estudos recentes indicaram que desenvolvedores frequentemente aplicam refatoração composta, ou seja, duas ou mais refatorações inter-relacionadas. No entanto, estudos anteriores revelaram que apenas 10% das refatorações compostas em projetos de software são consideradas completas, ou seja, aquelas que removem completamente as anomalies de código. Ademais, a literatura é escassa quanto ao catálogo de refatorações completas mais comuns e seus efeitos colaterais. Objetivo: Para preencher essas lacunas, realizamos um estudo de métodos mistos para criar um catálogo de refatorações compostas completas derivadas do conhecimento prático. Método: Coletamos empiricamente as refatorações completas mais comuns e seus efeitos colaterais aplicados em 42 projetos de código aberto e fechado, gerando um catálogo de recomendações de refatorações com-*

postas completas. Em seguida, as recomendações do nosso catálogo foram avaliadas por 21 desenvolvedores de software. Resultados: O catálogo derivado inclui quatro recomendações para remover completamente múltiplas anomalias de código, minimizando possíveis efeitos colaterais. A avaliação indica que 85% dos desenvolvedores relataram que suas próprias soluções poderiam ter efeitos colaterais piores sem nossas recomendações, aumentando o esforço para remover anomalias de código. Conclusão: Nosso catálogo pode orientar desenvolvedores e pesquisadores sobre como combinar melhor refatorações compostas que reduzam efetivamente os efeitos colaterais na prática.

1. Introduction

Developing software projects with high quality is the goal of every company [Galin 2018, Laporte and April 2018]. However, due to extensive maintenance and evolution in those projects, the internal software quality usually degrades [Uchôa et al. 2020, Baabad et al. 2020, Oizumi et al. 2019], generating poor code structures, like code smells [Fowler 1999]. *Code refactoring* is a single code transformation that intends to remove code smells [Fowler 1999]. A *composite refactoring* (composite, for short), consists of two or more interrelated single refactorings [Sousa et al. 2020, Bibiano et al. 2020, Bibiano et al. 2023]. The application of composites is a complex and error-prone task, as the smelly code must often be modified in multiple parts by different refactoring types combined [Sousa et al. 2020]. To make matters worse, studies have indicated that composites are generally applied manually [Murphy-Hill et al. 2012, Kim et al. 2014] and eventually combined with other code changes, e.g, the addition of features. [Murphy-Hill et al. 2012, Sousa et al. 2020]. Yet, studies indicate that only 10% of composites could remove code smells entirely [Sousa et al. 2020, Bibiano et al. 2019]. In the other words, developers spend time and effort applying composite refactorings, but rarely composites fully remove code smells. We refer here to a composite that fully remove smells as a *complete composite refactoring* [Bibiano et al. 2021, Bibiano et al. 2023].

However, the existing knowledge about complete composites is limited. The first major limitation is that albeit some studies have recommended a few complete composites [Sousa et al. 2020, Brito et al. 2022], these recommendations are both not known to be effective in practice and not structured in comprehensive catalogs. Effective, well-structured refactoring catalogs are essential to guide developers (e.g., to effectively remove smells) and tool builders (e.g., to build tools that recommend effective refactorings). The second major limitation is that previous studies have only relied on a restricted number of refactoring types to propose their recommendations. Finally, the third major limitation is that previous studies did not investigate and alert developers about the side effects of the (existing and new) refactoring recommendations.

The goal of our study (reported here and not published elsewhere) is to enhance recommendations of composite refactoring based on the practice and properly support developers when applying refactorings. This paper is a shortened version of Chapter 5 of the PhD thesis of the first author. In this study, we aimed at extending and adding to existing catalogs of complete composite refactorings that overcome the limitations mentioned above. We also aimed at systematically assessing our enhanced recommendations with developers. To achieve this, we conducted a mixed-method (i.e., repository mining + interviews) and large-scale study on 42 open and closed-source Java projects. From the

projects, we mined 31,066 composites (composed of 250,172 single refactorings) from which we identified 1,397 complete composites that are used to remove 19 different types of code smells. Then, our enhanced catalog was shown and used by 21 developers during interviews. We identified the most frequent combinations within complete composites applied in the practice and the side effects of those complete composites. Our main results are summarized as follows.

1. Developers frequently combine *Extract Method* with fine-grained refactorings, such as changes on parameter types to fully remove *Long Methods*, *Feature Envies*, and *Duplicated Code*. Interestingly, we observed that 45% of these composites introduced *Brain Methods* as a side effect because many variables were modified. Based on that, we empirically identified that complete composites formed of *Extract Methods* and *Change Parameter Types* can fully solve 49% of those same smells without introducing *Brain Methods*.

2. *Extract Methods* and *Move Methods* are commonly recommended to remove *Feature Envy*. However, we observed that 42% of those complete composites tend to introduce *Intensive Couplings*, which is not reported in existing recommendations. Thus, we identified two recommendations to alleviate the introduction of *Intensive Couplings*, while explicitly describing this side effect in our enhanced catalog.

3. All 21 developers agreed with our recommendations. We observed that seven (33%) developers were unaware of side effects while proposing their solutions to remove code smells. After the proposal of their solutions, 18 (85%) developers reported that their refactoring solutions could lead to worse side effects without our recommendations. These results confirm the need to alert developers about the side effects of composite refactorings.

Our study contributes to the practice by providing a catalog with concrete recommendations to guide developers to apply complete composites. Also, our catalog 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.

2. Background and Problem Statement

This section describes the main concepts and existing limitations regarding the completeness of composite refactorings.

2.1. Composite Refactoring (or Composite)

A single refactoring rarely removes a code smell [Cedrim et al. 2017]. Developers need to apply composite refactorings to eliminate several code smell types [Szőke et al. 2017]. 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 [Sousa et al. 2020]. A composite c can be formed of the same refactoring type, or a combination of different refactoring types [Bibiano et al. 2019, Bibiano and Garcia 2020, Sousa et al. 2020, Brito et al. 2019, Tenorio et al. 2019, Cinnéide and Nixon 2000].

Due to the complexity of identifying whether a refactoring is part of a composite, recent studies proposed heuristics to detect composites [Bibiano et al. 2019, Sousa et al.

2020]. Several studies [Bibiano et al. 2020, Oizumi et al. 2020, Brito et al. 2019] have used the *range-based* heuristic [Sousa et al. 2020] for composite detection. The heuristic considers a composite as those refactorings applied by the same developer and affecting the same code elements (i.e., the refactoring range). In that way, as the same developer refactor the same code elements, we can consider that this developer has a common goal to refactor that set of code elements. The reliability of this heuristic was demonstrated in [Sousa et al. 2020, Bibiano et al. 2020, Bibiano et al. 2021].

Different studies indicate that developers often apply composites manually [Bibiano et al. 2019, Sousa et al. 2020, Szőke et al. 2017]. Besides, we found evidence that composites frequently result in undesirable side effects [Bibiano et al. 2019, Sousa et al. 2020]. Therefore, having in-depth knowledge about composites is needed to support developers when applying refactoring. However, we observe that (i) there is little knowledge on the best alternatives of composites for effectively removing code smells, and (ii) there is a misguidance of automated support for developers applying composites. Aiming at fulfilling these gaps, we investigated the completeness of composite refactorings.

2.2. Completeness of Composite Refactorings

A previous study recommend composites to remove a particular type of code smell [Sousa et al. 2020]. When a composite is recommended to remove one or more code smell types, each code smell is considered a "target" of this composite. For example, a recent study recommends applying *Extract Method(s)* combined with *Move Method(s)* to remove Feature Envy [Sousa et al. 2020]. Thus, the *Feature Envy* type is the target smell in such cases. When the target smell is fully removed, we can consider that the composite refactoring is complete on the removal of the smell, otherwise the composite is incomplete. Completeness of a composite refactoring (i.e., composite completeness) is a characteristic given to those composites able to achieve the full removal of code smells [Bibiano et al. 2023], as defined as follows:

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 touches in a code element e, such as a method or/and class. We then have $\forall e$ that has a target code smell s, and $TOTAL_{BEFORE(s)}$ is the number of all target code smells before the application of a composite refactoring c, $TOTAL_{AFTER(s)}$ is the number of all target code smells after the application of c. A composite refactoring is complete when $TOTAL_{AFTER(s)} < TOTAL_{BEFORE(s)}$. Otherwise, an incomplete composite refactoring is when none code smell target was removed, thus $TOTAL_{AFTER(s)} >= TOTAL_{BEFORE(s)}$.

2.3. Fine-Grained Refactorings

A fine-grained refactoring (FGR), is a minor code transformation on variables or attributes. This transformation can be a change of variable type, a merge between two or more variables. A refactoring of large granularity, or coarse-grained refactoring (CGR), is a code transformation that involve method(s) or class(es). Common examples of CGR are *Extract Method*, *Move Method*. In this study, we considered the term "coarse-grained" to better align with the term "fine-grained". Table 1 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 fine-grained refactorings were not investigated by previous studies that recommend complete composite refactorings [Sousa et al. 2020, Brito et al. 2022].

Tabela 1. Classification of Refactoring Types

Fine-Grained Refactoring (FGR)		Coarse-Grained Refactoring (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**

2.4. Limitations of Existing Recommendations of Complete Composite Refactorings

Existing recommendations [Sousa et al. 2020, Brito et al. 2019, Bibiano et al. 2021] are quite limited. First limitation, studies have recommended some complete composites [Sousa et al. 2020, Brito et al. 2022], but these recommendations are not structured in comprehensive catalogs. Refactoring catalogs are useful to guide developers (e.g., to effectively remove smells) and tool builders (e.g., to build tools that recommend effective refactorings). Thus, it is important that a catalog describes complete composites that are successful in practice. The knowledge empirically extracted from existing software projects can demonstrate to developers that it is possible to perform successful composites in their routine. In addition, existing recommendations of complete composite refactorings are limited [Sousa et al. 2020, Brito et al. 2022]. Second limitation, previous studies investigated a restricted number of refactoring types to propose their recommendations, but in the practice, developers can combine not previously investigated types of refactorings. Third limitation, previous studies did not alert developers about the side effects of the refactoring recommendations. Existing studies [Sousa et al. 2020, Brito et al. 2022] recommend refactoring types like *Extract Methods* to remove *Long Methods*, but they did not alert about possible side effects of this recommendation.

3. Study Settings

Our mixed-method study aims at assessing and enhancing recommendations of complete composite refactorings. To accomplish such goal, we first propose the following research questions.

RQ_1: *What are the most frequent refactorings in complete composites 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 most appear in frequent complete composites, once these refactoring types were not investigated by [Sousa et al. 2020, Brito et al. 2022].

RQ$_2$: *What are the side effects of the most frequent complete composites?* – Complementary to the previous research question, **RQ$_2$** aims at identifying the side effects of the most frequent complete composites 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 understand the side effects of the most frequent complete composites. This understanding is of paramount importance, otherwise, we may misguide developers on refactoring.

RQ$_3$: *To what extent are the recommendations of complete composite refactorings perceived as useful for developers in practice?* – **RQ$_3$** aims at providing and documenting a catalog of recommendations of complete composites to guide the developers in their preventive maintenance activities. To this end, we combine the empirical recommendations provided by prior studies [Sousa et al. 2020, Brito et al. 2022] and new ones extracted from the knowledge obtained in the previous RQs. For that, we interviewed developers to know how and when they would use those recommendations.

3.1. Study Steps and Procedures

Figure 1 illustrates our study steps and dataset. The **replication package** is available in [Bibiano 2023b].

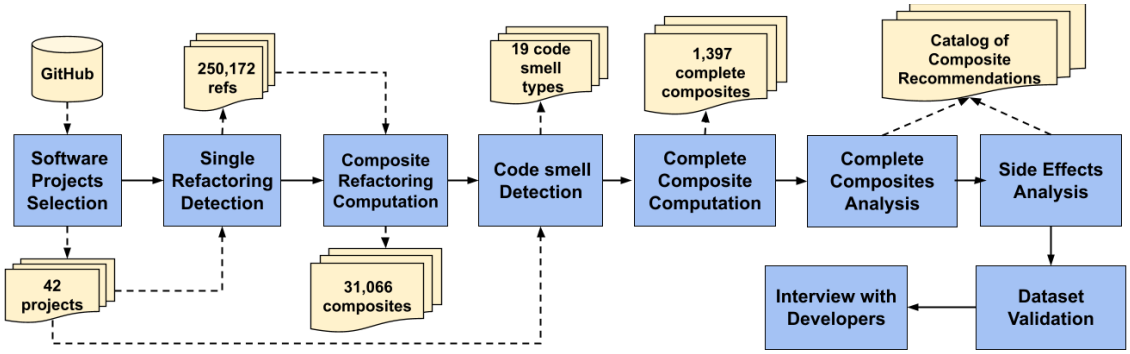


Figura 1. Study Steps

Step 1: Software Project Selection. We selected 42 software projects according to the following criteria: (i) the software projects must be implemented using Java due to the availability of robust tools for software analysis; (ii) the software projects 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 projects must have been investigated by at least one related study regarding refactoring [Brito et al. 2022, Brito et al. 2019, Sousa et al. 2020]. This last criterion was considered because related studies have already confirmed those software projects have occurrences of composite refactorings and code smells.

Step 2: Single Refactoring Detection. For detecting single refactorings applied on software projects, we used the RefMiner 2.0 tool [Tsantalis et al. 2020] due to its high precision and recall levels (98% and 87%, respectively). This tool supports a total of 52 refactoring types [Tsantalis et al. 2018]. The tool identifies 52 refactoring types. We focused on the 34 ones that are in the scope of our study (Table 1). These 34 refactoring types are applied in the code scope of attributes, methods, classes, and the code smells analyzed in this study can happen in that code scope.

Step 3: Composite Refactoring Computation. For the detection of composite refactorings, we created a script written in Java to implement the *range-based* heuristic [Sousa et al. 2020], described in Section 2.1. This heuristic fits well our study goal, as it considers multiple code elements. Besides, the reliability of this heuristic was demonstrated in [Bibiano et al. 2020, Sousa et al. 2020]. The heuristic script was developed in Java. Two researchers tested and validated our script.

Step 4: Code Smell Detection. Similarly to the studies that proposed recommendations of composites [Bibiano et al. 2019, Sousa et al. 2020], we used the Organic tool [Oizumi et al. 2016] for detecting code smells in our study. Organic is able to detect 19 code smell types. The Organic tool uses detection strategies based on code metrics to identify each type of code smell. These detection strategies have already been validated by prior studies [Cedrim et al. 2017, Fernandes et al. 2017, Oizumi et al. 2016]. Besides, this tool identifies types of code smells that involve multiple classes. The investigation of the effect on code smells that are related to multiple classes is interesting because code smells can be propagated between these classes while the existing code smells are removed.

Step 5: Complete Composite Computation. We focused on the complete composites for removing 19 code smell types. We elaborated scripts written in Java to identify when composites are complete, according to our definition of composite completeness (Section 2.2). These scripts were tested and validated as detailed in Section 7.

Step 6: Complete Composite Analysis. Aiming to answer our RQ_1, we need to identify frequent combinations for composing complete composites. This identification helps to collect combinations of refactoring types in composites that commonly obtained success to fully remove code smells. For the detection of frequent combinations, we created scripts to group complete composites in types. We follow the definition of composite types presented in [Bibiano et al. 2021]. We collected the frequent combinations between groups. An example of that is when we have a group g1 of composite types formed of 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. In other words, the combination of *Extract Method(s)*, *Change Return Type(s)* are commonly applied when certain code smell types are fully removed. We then collected the frequent combinations and what code smell types are often removed when applied. We then extracted recommendations for our catalog [Bibiano 2023a] based on these recurring combinations of complete composites.

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 aimed to find the relation between the introduction of code smells and the complete composites that removed the target code smell. We analyzed the code, other code changes, commit messages, and pull request discussions for each complete composite. This in-depth analysis allowed us to understand whether other code changes could have introduced the code smell, and if developers are aware of these code smells.

Step 8: Dataset Validation. We randomly selected a sample of 36 complete composites from our dataset for validation. Six developers validated whether the composites are complete for detected code smells. We provided a table for the developers with composite data: refactoring types that form each composite, the project name that was applied to the composite, the commits in which the composite was applied, the code element names that were touched for each composite, and 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 that, two developers evaluated six composites (the total of 12 composites), and four developers only had time to evaluate four composites (the total of 16 composites), resulting in the evaluation of 28 composites. The developers identified that 24 out of 28 composites are complete for at least one code smell detected.

Step 9: Interviews with Developers. We designed a semi-structured interview to be conducted with the developers. We conducted a pilot of the interview with two developers, aiming to identify opportunities to improve the interview design. After the pilot, we interviewed 21 developers from different countries. These developers have a median of nine years of development experience. We detail the interview procedures in what follows.

Activity 1: *Characterization and training session.* We asked the participants to fill out a *Characterization Form* to collect data about their refactoring development profile. Next, the participants watched a *training video* (15 minutes) about the main concepts about refactoring and code smells. The goal was reduce the bias by focusing on main concepts used in this study.

Activity 2: *Smell identification task.* We asked participants to perform a code smell identification task. We then presented a source code that contains one or more code smell types from our dataset. Next, we explain the domain of the source code, since the participants are not contributors to the source code under analysis (5 minutes). We instructed participants to think-aloud about their code smell identification task, and also asked participants to share their screens for observation purposes. Finally, we asked participants to mention the code smells that were identified and justified why the code is smelly (10 minutes).

Activity 3: *Presentation of the code smells.* We presented the smell definition and examples of our catalog (5 minutes). We asked participants if they agree with the definition of the proposed code smells, and if the previous analyzed code elements (Activity 2) contain these smells.

Activity 4: *Presentation of the refactoring recommendations.* We asked participants to talk about solutions they would apply to remove code smell(s) identified in Activity 2 (10 minutes). After that, we presented the refactoring recommendations of our catalog. We explained the definition, mechanics, and examples of composite refactorings that can be applied to remove these code smells. The developers could choose between their solution or the solution of our catalog. Finally, we presented the possible side effects of our refactoring recommendations, asking the participants to explain if they thought about them when they selected the refactoring recommendations.

Activity 5: *General evaluation of the interview.* The subjects answered a ques-

tionnaire to check if our examples address common situations observed in the industry. Finally, we asked about the positive and negative points of our interviews.

4. Most Frequent Combinations in Complete Composites (RQ_1)

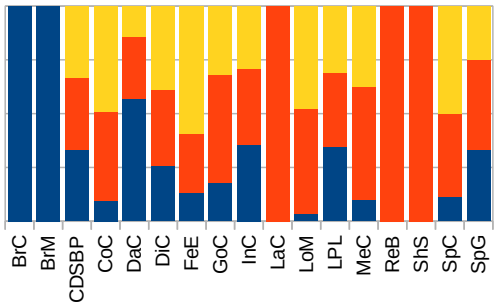
This section describes the most frequent combinations in complete composite refactorings, and their side effects. Table 2 presents the most frequent combinations of refactoring types. We found that three CGRs, namely *Extract Method*, *Move Method* and *Move Class*, are commonly applied with FGRs. We also found that 132 (28%) out of 462 complete composites have at least one *Extract Method* combined with *Change Variable Type(s)*. These combinations of refactoring types helped to remove code smells, such as *Long Methods* or *Feature Envies*. We observed that the *Change Variable Types* and *Change Parameter Types* might help to simplify or remove some code statements, decreasing lines of code and minimizing the excessive method calls to external classes.

Tabela 2. Most Frequent Combinations of Refactorings in Complete Composites

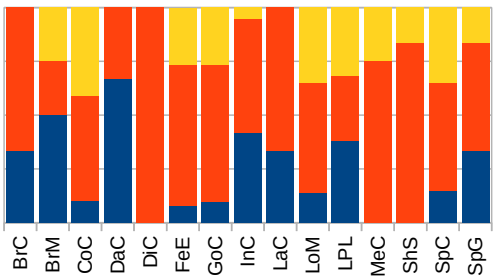
#CC with at least one Extract Method = 462	
Combination	#CC(%)
Change Variable Type, Extract Method	132 (28,57%)
Change Return Type, Extract Method	107 (23,16%)
Change Parameter Type, Extract Method	102 (22,08%)
Extract Variable, Extract Method	96 (20,78%)
Change Return Type, Change Variable Type, Extract 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%)

In summary, our results reveal that developers frequently changed the type of the method return (23%) or parameter(s) (22%) when extracting methods. Despite type changes being simple, they can be related to more complex code changes. In other words, when developers apply a *Change Parameter Type(s)* and *Extract Method(s)* or change the return of a method, they need to update all methods that were calling the original methods. Developers changed the parameter(s) in each call of the method and also adapted the source code to perform the method extraction.

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



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



(b) Extract Method(s) and Move Method(s)

Figura 2. Side Effect of Common Complete Composites

5. Side Effects of the Frequent combinations in Complete Composites (RQ_2)

As described in the previous section, developers apply many code modifications to support a simple combination including coarse and fine-grained refactorings, generally, such a combination is applied with non-refactoring code changes. Figure 2(b) shows interesting situations in which developers applied *Extract Method(s)* and *Change Parameter Type(s)*. In that case, a high incidence of *Brain Methods* was introduced. This behavior can be due to the introduction of refactoring code changes while the developer intends to decrease the size when extracting code. On the other hand, we can see these refactorings helped to remove mainly *Long Method(s)* (60%), and God Classes (58%).

Finding 2: The method extractions and changes in the type parameters helped to improve the code size and coupling, regardless of the increase in the method complexity

This previous finding caught our attention because it suggests that *Long Methods* might implement two or more features from external classes (due to the increase of complexity method). Then, these methods can also be *Feature Envies* [Bibiano et al. 2019]. However, there is no empirical evidence about the frequency of methods that are *Long Methods* and *Feature Envy* in conjunction. To investigate the frequency of methods that have these two code smells at the same commit, we randomly selected 13 software projects, and randomly selected nearly 5,000 commits from each project. We found that 61% of methods have *Long Methods* and *Feature Envy* in conjunction. This high frequency indicates it is necessary to have a new code smell type to represent this recurring poor structure (long and envious code). In that way, we defined a *Long Envious Method* as a method that has excessive lines of code and implements one or more features from external classes. This anomaly seems to occur frequently because the developers are unlikely aware of the joint occurrence of these smells on those methods. Moreover, they might focus on the removal of a single smell only, due to the high complexity of removing two or more smells through the same composite.

Extract Methods and *Move Methods* are frequently recommended to remove *Feature Envies* [Sousa et al. 2020]. Figure 2(b) shows that this combination can indeed fully remove 66% of *Feature Envies*. We also observed that about 26% of *Feature Envies* are not affected when developers extract and move methods. In that case, the developer needs to be alerted. Composite refactorings formed of extractions and method moves can be related to the introduction of *Long Parameter Lists* (38%) and *Intensive Coupling* (42%).

Our results revealed that this side effect is not so frequent in the practice, but it can happen. *Long Parameter Lists* can be introduced because many variables are transformed in parameters when methods are extracted. Existing recommendations do not alert developers 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.

Along our manual validation (Section 3), we observed that *Long Methods* are often removed with *Duplicated Code*. However, we did not collect data about duplicated methods. In that way, to better investigate the possible relation between *Long Methods* and *Duplicated Methods*, we used the tool CPD from the PMD Source Code Analyzer [PMD 2022]. We created Java scripts to mine duplicated methods through CPD output. Based on the CPD rules, we considered duplicated those methods containing 30 or more duplicated statements. We used the sample of commits that was investigated in *Long Envious Method* analysis. We then analyzed the frequency of duplicated methods that are also long at the same commit. We called these methods of *Long-Signed Clone*. We observed that the methods are long and duplicated because one or more parameter cause the repetitive and excessive lines of code. In addition, composites constituted of *Extract Method* and *Change Parameter Types* often to fully remove *Long-Signed Clones*.

6. Evaluation of the Proposed Catalog (RQ_3)

Based on the common combinations in complete composites found (Section 4), we created a catalog of composite recommendations [Bibiano 2023a]. We provide four recommendations to remove two new smell types, *Long Envious Method* and *Long-Signed Clone*. In that way, we created composite recommendations for the removal of these two smells in our catalog.

To enhance our catalog, we interviewed twenty-one developers. Eleven developers analyzed methods with *Long Envious Method*, and ten developers evaluated methods with *Long-signed Clone*.

Code Smell Analysis: As explained in Section 3.1, the developers analyzed the source code, without being aware of the smell types affecting the code. After the analysis, we mentioned the code smell types. Twelve (57%) developers detected the two code smell types, while nine (43%) developers identified only one code smell type. One developer argued that "*This method is a Feature Envy but, in my opinion, it is not long as the method size fits to my screen*". The developer that detected the two code smell types mentioned that it was hard knowing if the method has envy since it has five calls to external classes. The developer was not sure if five calls is an adequate threshold for *Long Envious Method*. Thus, we can note thresholds to detect these new smell types may be different from existing detection strategies because they are composed by at least two code structural problems. We then presented to developers the definition of the code smells and asked whether they agree with that definition. All developers agreed with our definition. For both new smells, we noted that our definition is aligned with the developers' perceptions.

Refactoring Recommendation: All developers agreed to our refactoring recommendations. For the *Long-signed Clone* removal, developers perceived that only extractions are not sufficient to remove this code smell, causing side effects such as the propagation of the duplicated code. Eight (80%) out of 10 developers opted for *Extract*

Methods and *Change Parameter Types* to remove *Long-signed Clones*. For the *Long Envious Method*, four (36%) out of 11 developers agreed to apply the first mechanics. These developers reported that the first mechanics are more interesting for junior developers once they probably have low familiarity with the source code and more difficulty to identify the code elements modified by composites. Other developers reported that applying two or more code transformations at the same time is the most common for senior developers, mainly because they have high familiarity with the source code. According to the developers' answers, a high familiarity with the source code increases the awareness of the composite refactorings' side effects. Based on that, we have our next finding.

Finding 3: Applying each refactoring by time is better for junior developers because it facilitates the analysis of side effects.

Side Effects: Seven (33%) out of 21 developers were unaware of side effects while proposing their solutions to remove code smells. After the proposal of their solutions, surprisingly, when we mentioned the side effects, 18 (85%) out of 21 developers reported that their own solutions could have worse side effects than our solution. From these answers, we can observe that some developers do not have awareness about side effects while apply composite refactorings. In addition, we confirm that developers should be guided to identify the side effects of composites once their solutions may not the best ones to fully remove code smells.

Finding 4: Developers that are unaware of the side effects of composites tend to perform composites inducing side effects.

7. Threats to Validity

Construct Validity: Relying purely on automated detection tools may be risky for identifying code smells and refactorings [de Mello et al. 2022]. However, performing manual validation in large-scale samples is unfeasible. To mitigate this threat, we carefully selected the tools employed: RefMiner 2.0 and Organic. Both tools are highly accurate for, respectively, refactoring detection and code smell identification (see Section 3.1). RefMiner is also beneficial because it was designed to ignore squash commits [Tsantalis et al. 2020]. One common symptom of squash commits is the large time gap between the changes performed, what is incompatible with the definition of composite refactorings. As a result, the time interval between commits analyzed in our study is short, i.e., two weeks on average. Some of our results may be biased due to the detection of RefMiner. For instance, when classes are renamed, RefMiner identifies *Change Parameter type* for each parameter in which its type was renamed. To mitigate this threat, we performed manual validations to detect when the refactorings were indeed applied, independently of the possible bias of RefMiner. The heuristics followed for detecting complete composites may bias the results. To mitigate this threat, we employed the heuristics proposed by Sousa *et al.* [Sousa et al. 2020] for detecting composite refactorings, combined with the definition of complete composites proposed in [Bibiano et al. 2020].

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 validate all scripts. Besides, two authors double-checked the scripts and results of the unit tests, mitigating the risk of validation bias. We conducted pilots involving two developers,

we then manually analyzed the data from these pilots to mitigate the threat related to possible issues in the interviews. Besides, the authors followed standard guidelines to manually analyze the developers' answers. The interviews were recorded and transcribed, which reached sufficient quality, without the need for contacting interviewees to solve misunderstandings.

Conclusion Validity: Our definition of "completeness" for classifying composite refactorings is based on fixed thresholds established by code smell detection tools [Oizumi et al. 2016]. Therefore, this definition may lead to misclassification. Besides the already reported quality of Organic, we also relied on asking developers about their agreement with the thresholds employed for supporting the code smell detection (see Section 3). To identify the most frequent combinations in complete composites (RQ_1), we should have in mind that some sequences of refactorings would not be performed to intentionally remove code smells. To mitigate this threat, specialists manually assessed which refactoring instances actually contributed to partially or completely eliminating the code smells detected. To mitigate threats addressing 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. To evaluate the proposed catalog, we interviewed developers from different software projects addressing different domains. Besides, we designed the interview to evaluate the catalog from different perspectives, including clarification, adaptability, and usability.

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. The catalog was evaluated by developers playing different roles at companies in several countries with distinct cultures.

8. Conclusion

In the context of this doctoral research, this catalog summarizes the research on composite refactoring. This innovative catalog uses systematized concepts on the characterization of composite refactoring and completeness proposed in [Bibiano et al. 2023]. Given the limitations of existing recommendations for composite refactorings. We built a dataset of 42 software projects, collecting 1,397 complete composites that were the base to create our catalog. The first results that were extracted from this dataset were published in [Bibiano et al. 2021]. We then extended our quantitative analysis and assessed our catalog with 21 developers to have a practical view of our enhanced recommendations, generating this study. Our main findings include (i) the identification of the most frequent combinations in complete composites applied in the practice, and (ii) the side effects of complete composites. In future work, we intend to extend our recommendations, explaining possible motivations in which each composite can be applied to fully solve two or more code smells.

Referências

Baabad, A., Zulzalil, H. B., Hassan, S., and Baharom, S. B. (2020). Software architecture degradation in open source software: A systematic literature review. *IEEE Access*, 8:173681–173709.

Bibiano, A. C. (2023a). Catalog of complete composites. page https://compositerefactoring.github.io/catalog.

Bibiano, A. C. (2023b). Composite refactoring - replication package. page https://compositerefactoring.github.io/site/.

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

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

Bibiano, A. C. and Garcia, A. (2020). 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)*, pages 165–179, Porto Alegre, RS, Brasil. SBC.

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

Bibiano, A. C., Uchôa, A., Assunção, W. K., Tenório, D., Colanzi, T. E., Vergilio, S. R., and Garcia, A. (2023). Composite refactoring: Representations, characteristics and effects on software projects. *Information and Software Technology*, 156:107134.

Brito, A., Hora, A., and Valente, M. T. (2019). Refactoring graphs: Assessing refactoring over time. In *26th Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 504–507.

Brito, A., Hora, A., and Valente, M. T. (2022). Towards a catalog of composite refactorings. *Journal of Software: Evolution and Process*.

Cedrim, D., Garcia, A., Mongiovi, M., Gheyi, R., Sousa, L., de Mello, R., Fonseca, B., Ribeiro, M., and Chávez, A. (2017). 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)*, pages 465–475.

Cinnéide, M. and Nixon, P. (2000). Composite refactorings for java programs. In *14th ECOOP (2000)*, pages 129–135.

de Mello, R., Oliveira, R., Uchôa, A., Oizumi, W., Garcia, A., Fonseca, B., and de Mello, F. (2022). Recommendations for developers identifying code smells. *IEEE Software*.

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

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1 edition.

Galin, D. (2018). *Software quality: concepts and practice*. John Wiley & Sons.

Kim, M., Zimmermann, T., and Nagappan, N. (2014). An empirical study of refactoring: Challenges and benefits at Microsoft. *IEEE Transactions on Software Engineering (TSE)*, 40(7):633–649.

Laporte, C. Y. and April, A. (2018). *Software quality assurance*. John Wiley & Sons.

Murphy-Hill, E., Parnin, C., and Black, A. (2012). How we refactor, and how we know it. *IEEE Transactions on Software Engineering (TSE)*, 38(1):5–18.

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

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

Oizumi, W., Sousa, L., Oliveira, A., Carvalho, L., Garcia, A., Colanzi, T., and Oliveira, R. (2019). 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)*, pages 346–357.

PMD, W. (2022). Pmd website. page https://pmd.github.io/latest/pmd_userdocs_cpd.html.

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

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

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

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

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

Uchôa, A., Barbosa, C., Oizumi, W., Blenilio, P., Lima, R., Garcia, A., and Bezerra, C. (2020). 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)*, pages 511–522. IEEE.