PONTIFÍCIA UNIVERSIDADE CATÓLICA
DO RIO DE JANEIRO

Ana Carla Gomes Bibiano

On the Completeness of Composite Code Refactorings for Beneficial Smell Removal

Tese de Doutorado

Thesis presented to the Programa de Pós–graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Doutor em Informática.

Advisor: Prof. Alessandro Fabricio Garcia

Rio de Janeiro
April 2023

Ana Carla Gomes Bibiano

On the Completeness of Composite Code Refactorings for Beneficial Smell Removal

Thesis presented to the Programa de Pós–graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Doutor em Informática. Approved by the Examination Committee:

Rio de Janeiro, April 24th, 2023

Ana Carla Gomes Bibiano

Ana Carla Bibiano is a PhD candidate in Software Engineering in the Informatics Department at PUC-Rio. Currently, Ana is Back-end Developer at Ivve Tech Company in São Paulo, Brazil. She was Substitute Professor of Informatics' Center at Federal University of Pernambuco (UFPE) in 2021. She received her Master Degree in Informatics at Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Brazil in 2019. She holds a Bachelor's degree in Computer Science from Federal University of Alagoas (2017). Ana published 12 papers (2017-2022) in conferences with high reputation in Software Engineering. One of these papers was recognized as one of the best papers of the conference. Her Master's dissertation was the second best dissertation of Brazil in Software Engineering in 2020. She earned, in 2014, the Academic Excellence award by the Brazilian National Council for Scientific and Technological Development (CNPq) for her scientific initiation's work. Ana Carla has 13 years of experience as a software developer in Brazilian software companies. She has worked as a software developer for companies from Maceió (AL), Blumenau (SC), and Rio de Janeiro (RJ). Ana has research collaborations with international universities located in Austria, EUA, United Kingdom, and Portugal. Ana Carla is currently a research scholar in Software Engineering for the OPUS Research Group at PUC-Rio. Her main research interests are (not limited to): software refactoring, maintenance, and evolution.

Bibliographic data

Gomes Bibiano, Ana Carla

On the Completeness of Composite Code Refactorings for Beneficial Smell Removal / Ana Carla Gomes Bibiano; advisor: Alessandro Fabricio Garcia. – 2023.
194 f: il. color. ; 30 cm

Tese (doutorado)–Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2023.
Inclui bibliografia

1. Informática – Teses.
2. Refatoração Composta;. 3. Manutenção de Software;. 4. Anomalia de Código-Fonte;. 5. Atributo de Qualidade Interna;. 6. Mapeamento Sistemático.. I. Fabricio Garcia, Alessandro. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Acknowledgments

I would like to take this opportunity to express my heartfelt gratitude to the individuals and entities who have been instrumental in the successful completion of my doctoral thesis. Firstly, I would like to thank God, the heavenly court, and Our Lady for being with me every step of the way, and for providing me with the strength and perseverance to overcome all obstacles.

I am also deeply grateful to my family, especially my mother Lourinete Bibiano, father Antonio Bibiano (in memoriam), and grandparents (in memoriam), who have been my pillars of support throughout this journey and inspired me to be a strong and happy person. Their unwavering love, encouragement, and sacrifices have been the driving force behind my success. I can not forget to express my gratitude to my siblings to be my support in happy and hard times of this journey.

All thanks to my amazing advisor Alessandro Garcia. He became my dear friend and father over the years. We have become better people, and I am happy person for having contributed to this personal growth. Wesley Assunção, my dear co-advisor is a wonderful human. I learned a lot from his humility, generosity, and comprehension, having your research collaboration made this doctorate lighter. I have lucky to have the best research colleagues, you provide me with all friendship, support, and guidance along this journey. Anderson Uchôa, Willian Oizumi, Anderson Oliveira, Caio Barbosa, Daniel Coutinho, Vinicius Soares, Kleber Santos, João Lucas Correia, Daniel Tenório, and all OPUS research members, you're amazing and thank you so much to support me in all moments. Their insights, feedback, and encouragement have been instrumental in shaping my research and helping me to achieve my goals. Special thanks for all professors that collaborated with me along these years, professors Baldoino Fonseca, Marcio Ribeiro, Rohit Gheyi, Leonardo Murta, Rafael de Mello, Thelma Colanzi and Silvia Regina.

I would like to thank my fiancé Anderson Santos and his family. He is my best friend over ten years and thank for putting up with me through the best and worst of times. Thanks to my dear friend Jakson Leao. Special gratitude to my medical doctors who accompanied me along this hard pandemic period, doctors Juliana Oliveira, Maria Julia and Yuri Toledo.

I extend my heartfelt appreciation to my thesis committee members professors Marcos Kalinowski, Alberto Sardinha, Juliana Pereira, Marcio Ribeiro, Rohit Gheyi and Leonardo Murta. I really appreciate their valuable feedback, constructive criticism, and expert guidance, which have been instrumental in shaping the direction and scope of my research.

My gratitude to the funding agencies for their financial support, which enabled me to conduct my research and bring it to fruition. Their support has been crucial in enabling me to realize my dreams and aspirations, especially the Brazilian National Council for Scientific and Technological Development (CNPq), plus the partner companies of the Software Engineering Laboratory (LES/PUC-Rio) for the student scholarship. This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001. Once again, I extend my heartfelt gratitude to each and every one of you for your unwavering support, encouragement, and guidance throughout this journey.

Abstract

Gomes Bibiano, Ana Carla; Fabricio Garcia, Alessandro (Advisor). **On the Completeness of Composite Code Refactorings for Beneficial Smell Removal**. Rio de Janeiro, 2023. 194p. Tese de doutorado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Code refactoring is a code transformation that aims to enhance the internal code structure. A single refactoring is rarely sufficient to achieve the full removal of a poor code structure, such as a code smell. Developers then apply *composite refactorings* to fully remove a code smell. A composite refactoring (or, simply, *composite*) consists of two or more interrelated single refactorings. A composite is considered "complete" when it fully eliminates the target smell. However, studies report that developers often fail in completely removing target code smells through composites. Even when composite refactorings are complete they may still not be entirely beneficial to the code structure. They may induce side effects, such as the introduction of new smells or the propagation of existing ones. There is a limited understanding of the completeness of composite refactorings and their possible effects on structural quality. This thesis investigates whether and how composite refactorings fully remove smells without inducing side effects. We found that 64% of complete composites in several software projects are formed of refactoring types not previously recommended in the literature. Based on this study, we derived a catalog of recommendations for supporting developers in applying composite refactorings. Out of twenty one developers evaluating our catalog, 85% reported that they would use the catalog recommendations and that their own refactoring solutions would have induced side effects. We also qualitatively evaluated three existing approaches to automatically recommend composite refactorings. In our study with ten developers, most (80%) developers reported that existing approaches frequently induce side effects. Overall, the findings and the proposed catalog can help developers to perform complete composite refactorings with better awareness of possible side effects.

Keywords

Software Maintenance; Code Smell; Internal Quality Attribute; Systematic Mapping; Quantitative Study.

Resumo

Gomes Bibiano, Ana Carla; Fabricio Garcia, Alessandro. **Sobre a Completude de Refatorações Compostas de Código-Fonte para a Remoção Benéfica de Anomalias de Código**. Rio de Janeiro, 2023. 194p. Tese de Doutorado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

A refatoração de código é uma transformação de código que visa aprimorar a estrutura interna do código. Uma refatoração isolada raramente é suficiente para remover completamente uma estrutura de código ruim, como uma anomalia de código. Os desenvolvedores então aplicam *refatorações compostas* para remover totalmente uma anomalia de código. Uma refatoração composta consiste em duas ou mais refatorações inter-relacionadas. Um refatoração composta é considerada "completa" quando elimina totalmente a anomalia de código alvo. Estudos relatam que os desenvolvedores geralmente falham em remover completamente as anomalias de código alvo por meio de refatorações compostas. Refatorações compostas concluídas podem não ser totalmente benéficas para a estrutura do código. Pois, estas podem induzir efeitos colaterais, como a introdução de anomalias de código ou a propagação de anomalias existentes. Há uma compreensão limitada sobre a completude das refatorações compostas e seus possíveis efeitos colaterais. Esta tese investiga como as refatorações compostas removem totalmente as anomalias de código sem induzir efeitos colaterais. Descobrimos que 64% das refatorações compostas completas são formadas por tipos de refatoração não recomendados anteriormente. Dessa forma, derivamos um catálogo de recomendações para apoiar os desenvolvedores na aplicação de refatorações compostas. Na avaliação do catálogo, 85% de 21 desenvolvedores relataram que usariam as recomendações do catálogo e que suas próprias soluções de refatoração teriam induzido efeitos colaterais. Também avaliamos qualitativamente três abordagens existentes para recomendar automaticamente refatorações compostas. Nesse estudo, a maioria (80%) dos 10 desenvolvedores relatou que as abordagens existentes frequentemente induzem efeitos colaterais. No geral, as descobertas e o catálogo proposto podem ajudar os desenvolvedores a realizar refatorações compostas completas.

Palavras-chave

Refatoração Composta; Manutenção de Software; Anomalia de Código-Fonte; Atributo de Qualidade Interna; Mapeamento Sistemático.

Table of Contents

List of Figures

List of Tables

Tuus totus ego sum, et omnia mea tua sunt

Saint Louis-Marie Grignion de Montfort, *A Treatise on the True Devotion to the Blessed Virgin.*

1
Introduction

Code refactoring is a code transformation that aims to enhance the internal code structure [253,263]. Code refactoring is cataloged in types [25,253, 263], and each type determines the program modifications required to produce an expected enhancement of a certain code structure [25, 253]. Examples of popular refactoring types include *Extract Method* and *Move Method* [46, 60]. Each single *refactoring* is an instance of a refactoring *type*.

Over the last thirty years, code refactoring has been one of the most popular topics investigated in the Software Engineering community [168]. The interest in refactoring research is also a reflection of its practical importance. Developers apply refactoring aiming to improve the internal software quality and, consequently, the comprehensibility of a program [25,199,253]. A common way to improve the internal software quality is to fully remove poor code structures [8, 25, 60], such as *code smells* [197, 198]. The removal of some code smells is considered highly relevant by developers and practitioners [35, 60, 62, 73]. For instance, the smells *God Class* and *Feature Envy* are considered quite relevant given their wide and harmful impact on the program structure [62,73]. Both smells can somehow affect two or even more classes.

However, a single refactoring rarely suffices to assist developers in fully removing a code smell [8], mainly when the code smell involves two or more classes. Recent studies then observed developers often apply composite refactorings on smelly source code [7, 143]. A *composite refactoring*, or, simply, *composite* consists of two or more interrelated single refactorings [143, 171]. Previous work has recommended specific composites to remove certain types of code smells [25, 143]. However, the empirical knowledge about whether, which, and how composite refactorings fully remove code smells is scarce. As a consequence, proper guidance to support developers on completely removing smells is quite limited. In that way, developers may spend time and effort applying composite refactorings that do not fully remove code smells.

Based on that, the (in)completeness of composite refactorings needs to be better investigated. In our research, completeness is a characteristic of a composite refactoring that concerns the full removal of target code smells [77]. We consider it incomplete when a composite does not fully remove the target code smell(s) [76,171]. Ideally, the application of a composite refactoring should be entirely beneficial to the code structure. In addition to fully remove the target code smell(s), the refactoring modifications should not induce side effects in the code, such as the introduction of new smells or even the propagation of existing ones.

In spite of its importance, the literature about the completeness of composite refactorings from both theoretical and practical perspectives is quite scarce. From a theoretical perspective, there is a lack of an overall conceptualization of composite refactorings. From a practical perspective, empirical studies of (in)complete composites and their effects in software projects are rare. Moreover, some studies propose approaches to recommend composite refactorings [145, 222, 242, 244], but these studies did not evaluate whether these approaches support code smell removal without inducing side effects. Given these limitations, developers and researchers may be misguided about how to perform the beneficial removal of code smells through composite refactorings. In this thesis, we aim to investigate how better to formulate these problems (Section 1.2) and tackle them (Section 1.3).

1.1 Motivating Example

This section details a real example in the `Dubbo` [181] project, in which a developer applied a composite refactoring for removing the target *Long Method* and *Feature Envy* smells. We identified the target smells from the messages of pull request [184] related to the commit in which the composite refactoring was applied. In this pull request, developers discussed why the refactorings were applied. A developer mentioned: *"I have moved the access log creation..."*, adding *"Refactored code to separate our and group related tasks in separate methods and have enhanced the readability by using: Method renaming, Reducing big methods to small..."*. In another words, the developers aim to separate responsibilities because the method `invoke()` was too long and implemented functionalities of the `AccessLogData` class. As presented in Figure 1.1, at some point in time (commit c_i) the class `AccessLogFilter` had a method called `invoke()`, having two code smells, namely a *Long Method* and a *Feature Envy*. In order to remove these smells, a developer applied a composite formed of two *Extract Methods* (ExM_1 and ExM_2) and one *Move Method* (MoM) in the commit c_{i+1} [180].

According to existing studies [25, 171], two *Extract Methods* can remove the *Long Method* and at least one *Extract Method* with one *Move Method* are expected to remove the *Feature Envy*. In the commit $\text{c}_{(i+1)}$, by applying

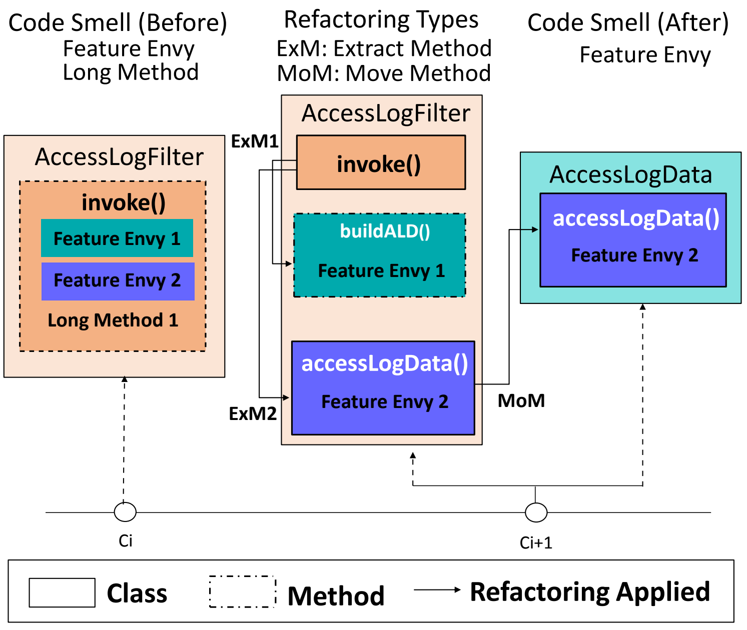


Figure 1.1: An Example of Complete Composite and its Side-effect

the two *Extract Methods* on the `invoke()` method, creating `buildAdd()` and `accessLogData()` methods, the *Long Method* was removed and the *Feature Envy* ended up being scattered across the two extracted methods. Then, the developer applied one *Move Method*, moving the `accessLogData()` method to the `AccessLogData` class, aiming to remove one of the *Feature Envies*. However, the code smell was not removed from the `accessLogData()` method as the method was actually more interested in data from another class.

The composite [*Extract Method*, *Extract Method*, and *Move Method*] could have removed both *Long Method* and *Feature Envy*, as indicated by previous studies [25, 143, 171]. We observed that this commit applied only changes on code related to these refactorings. Besides, the goal of this commit is refactoring to remove the *Long Method* and *Feature Envy*, as indicated in the commit message“Acesslog dateformat enhancement”. However, the composite refactoring did not solve totally the problem. Despite having removed the *Long Method* (in the `invoke()` method), the composite did not fully remove the *Feature Envy* (in the `accessLogData()` method). On the contrary, although the composite is considered complete as it removes the *Long Method*, according to [143, 171], it induced side effects. The composite induced the harmful propagation of the *Feature Envy* smell to additional methods and an additional class affected by the refactorings. We can see that existing recommendations of composites can lead to side effects, such as: (i) the prevalence of a smell in the program elements touched by a composite refactoring, or (ii) even the introduction of code smells.

Existing descriptions of complete composite types do not indicate at all what are their possible recurring side effects. This limitation is also not addressed by previous empirical studies [143, 171]. Perhaps, those developers applied this recommendation without being aware of the side effects of these complete composites. Because the existing descriptions do not have information about that, limiting the decisions of developers regarding these side effects. Thus, the case discussed above illustrates why existing descriptions of composite refactorings have to be extended to cover recurring complete composite types and their side effects. This advance can also further motivate developers to use and trust such descriptions while enabling them to make more informed decisions through the selection and application of composite refactorings.

1.2 Problem Statement

As aforementioned, specific composite refactorings may be performed by developers with the goal of completely removing code smell(s) [143, 171]. Despite this goal, existing studies found that some smells remain after the application of composites, i.e., the completeness of composites to remove code smells is not achieved [143, 171]. This means that developers should be more cautious in avoiding to perform incomplete composites. If not avoided, the structure of the smelly program may get even worse. Moreover, at least, the incompleteness of a composite refactoring should be brought to the developers' attention. Then, developers can decide whether to complete it. Despite its practical importance, the literature is scarce on investigating the effect of (in)complete composites in the overall internal software quality. As consequence, developers are misinformed about the effect of (in)complete composites on internal structural quality.

In the practice, developers assume that once a complete composite is performed, the improvement of the software quality is directly achieved. There is no guarantee that a complete composite will improve the code quality, as discussed in the previous example (Section 1.1). In fact, existing studies reported that even a single refactoring can degrade the internal structural quality frequently [8, 10, 168]. As a composite refactoring usually is formed of several refactorings [172], the transformation combinations may degrade even further the structural quality of the program. Thus, we hypothesize that even a complete composite, which is focused in a particular smell, can introduce other types of smells as a side effect, thereby also decreasing the software quality.

Currently, there is a lack of in-depth studies about the completeness of refactorings. Existing studies do not only ignore the influence of composite incompleteness, but also fail in revealing possible side effects of complete composites. This lack of empirical knowledge hinders assisting developers with recommendations of how to successfully apply a composite without introducing any harm. As consequence, developers and researchers do not have information on whether (in)complete composites can have negative or positive effects. Those limit the support for composite recommendations for beneficial removal of code smells, i.e, recommendations of composite refactoring that fully remove target code smells, minimizing their side effects. This motivation leads us to the general problem of this doctoral research, presented below and detailed in what follows.

General Research Problem: Developers and researchers are misinformed or misguided about the (in)completeness of composite refactorings and their positive or negative effects.

For an in-depth understanding of composite completeness, it is necessary to derive a systematic conceptualization of composite refactorings. However, the current knowledge about composite refactoring is fragmented [169, 203]. Most studies do not provide a comprehensive description of composites grounded on systematic knowledge [111, 143, 171, 203]. Consequently, each study represents composites through different representation models. There are several composite representation models, e.g., a composite represented by a refactoring list [94,98,111] or a set of refactorings [104,109,116]. This myriad of options makes it hard to choose which representation model to adopt for different scenarios or specific designs of solutions, such as catalogs of refactorings or refactoring recommenders.

Previous studies partially describe some characteristics of composites. A comprehensive collection of composite characteristics help us to identify and classify composites. However, the current knowledge on the characteristics of composites is sparse and contradictory. An example of a composite characteristic is code scope [171], which is not always explicitly and clearly defined in previous research on composite refactoring [35,65]. Moreover, some studies assume that a code scope of a composite is a single code element, e.g., a method or a class [43, 143]. Other studies suggest that code scope consists of many code elements [35, 50, 171]. These contradictory conceptualizations hinder the selection of characteristics to use in empirical studies and tools for composite refactorings.

As far as the knowledge on composite effects are concerned, the literature is also limited and often conflicting. Most studies only superficially report that composites can have different effects on software quality. For instance, studies found that composites often introduce code smells [143], but other studies conclude that composites more often remove code smells [79,171]. These conflicting observations increase the misunderstanding of practitioners and developers on composite effects. Furthermore, these contradictory conclusions hinder proper recommendations of composites for code smell removal. The existing recommendations indicate composites to remove code smells, but they do not alert developers on the possible introduction of other code smells. Consequently, these recommendations might lead to the degradation of internal quality attributes.

Due to the absence of a systematic conceptualization of composites, there are several problems for researchers and developers. These issues are: (i) it is not clear how composites are formed; (ii) the lack of knowledge on what are gaps concerning composites with certain representations, characteristics, and types of effects; and (iii) the literature conflicts on definitions, characterizations and effects of composites remain unrevealed and unresolved. To advance the research and practice, it is necessary to create a systematic and comprehensive body of knowledge on composite refactorings. This systematic knowledge basis will help researchers to investigate composites properly, as also practitioners to improve solutions for composite refactorings. Based on that, we can summarize our first research problem as follows.

Research Problem 1: The lack of a systematic conceptualization of composite refactorings.

Our recent studies also reported the fact that composites are frequently incomplete to remove code smells [143, 171]. For example, the report of one of these studies shows that composites that include multiple *Extract Methods* often fail to remove *Long Methods* [143]. This same study makes theoretical assumptions that composites formed of *Extract Methods* can improve some internal quality attributes such as code size, regardless of a full smell removal or not [143]. However, existing studies do not provide empirical evidence about the positive and negative effects of incomplete composites on internal quality attributes. Studies provide some complete composite recommendations to remove specific smells [143, 171, 204]. Despite that, based on Bibiano *et* al. [143], we have assumptions that complete composites can have side effects. Also, complete composites are formed of several refactorings, as mentioned in the previous discussion of our general research problem.

There is no empirical evidence about how (in)complete composites can affect the internal structural quality. Therefore, researchers and developers might not notice that even composites recommended by the literature are decreasing the internal quality of a program. Based on that, we should revisit the effectiveness of existing complete composite recommendations. Furthermore, we must find proper ways of guiding developers on how to perform composite refactorings that effectively result in software with superior internal quality.

Research Problem 2: There is no empirical knowledge about the positive, negative, and side effects of composite (in)completeness.

A few studies recommend composites to remove a single code smell [143, 171, 204], but their effectiveness has not been assessed. Previous pieces of work also recommend composites to remove multiple code smells simultaneously [79, 157]. Nonetheless, these recommendations are automatically obtained without considering the knowledge or preferences of actual developers of a subject system [79, 80, 156, 157]. In other words, these recommendations are neither based on observed successful practices, not supported by proper empirical evidence. If these recommendations proposed in the literature are not effective, we are facing a misalignment between theory and practice of composite refactoring.

This misalignment is already evident in recent studies reporting that developers are reluctant to use existing approaches to recommend complete composites [35, 211]. If empirical knowledge about the effectiveness of (in)complete composites is missing, researchers and practitioners are not able to understand how to advance the state of the art and state of the practice. In particular, this lack of knowledge will make it difficult to properly recommend composites to fully remove multiple code smells according to the developers' needs. Without such support, developers are increasingly avoiding to perform more complex refactorings in their projects.

Research Problem 3: Lack of recommendations to properly support developers on applying effective complete composites.

1.3 Study Goal and Research Questions

According to the research problems presented above, the main goal of this doctoral thesis is to *provide solutions of composite refactorings for beneficial code smell removal.* For that, we have the following specific goals: (i) systematically conceptualize composite refactorings; (ii) identify the negative, positive, and side effects of composite (in)completeness on internal structural quality; and (iii) recommend complete composites to support the full removal of target code smells. We then have performed empirical studies to solve the research problems described previously (Section 1.2) and our study goals presented above.

1.3.1 Research Questions

As mentioned previously in Section 1.2, the knowledge on the effect of composite (in)completeness is still scarce, limiting the use of composite refactorings both in research and practice. We aim to cover this limitation by answering this research question.

General Research Question: How to tackle the misinformation and misguidance of developers and researchers about the (in)completeness of composite refactorings and their effects?

Guided by this general research question, we have been conducting empirical studies to describe composite refactorings, know the positive, negative effects, and side effects of (in)complete composites, and recommend composites that can improve the internal structural quality. Our results will be reported for developers and researchers, increasing the knowledge on composite (in)completeness. In addition to the general research question, we have one specific research question related to each problem described in the previous section.

RQ$_1$. What are the representation models, characteristics and effects of composite refactorings? As aforementioned in Research Problem 1, an in-depth understanding of composite completeness is not feasible without a better understanding of composite refactorings. Based on that, to address this RQ$_1$, we performed a systematic mapping about composites. Systematic mapping is a broad review of primary studies in a specific topic area that aims to identify what evidence is available on the topic [36, 167]. Our systematic mapping is an extension of a literature review [203] that was elaborated and described in the Master's dissertation [220] (Chapter 3). Along the Doctoral research, we additionally investigated representation models and explored primary studies of the last 12 years (2010-2022). The mapping study of the Master's dissertation only covered studies published until 2018. We also extended the initial mapping by revealing new characteristics of composite refactorings such as their completeness and complexity, as well as other types of composite effects. A conceptual framework is a design artifact that provides a general view of a particular phenomenon [32]. We then have a systematic conceptualization of composites organized as a conceptual framework. Our conceptual framework is aimed at revealing representation models, characteristics, and effects of composites. Besides that, we mapped existing conflicts, and gaps in the literature about composites. Our conceptual framework can motivate future studies to investigate fields not explored in-depth on composites. More specifically, this work focus on answering the research questions presented in what follows.

RQ_2. How does the composite (in)completeness affect the internal structural quality? The systematic knowledge on composite refactorings is the basis to help us to understand the composite (in)completeness. From this basis, we can deeply investigate the (in)completeness of composites. Our RQ_2 is addressed to solve Research Problem 2. Aiming at the construction of knowledge to answer this research question, we observed studies presenting that single refactorings can negatively affect the internal structural quality, in particular, the degradation of internal quality attributes [10, 168] and the frequent introduction of additional smells [8]. A previous study also suggests that incomplete composites can affect internal quality attributes, such as code cohesion [143]. Thus, we investigated: (i) how incomplete composites affect internal quality attributes (positive and negative effects), and (ii) whether complete composites can introduce code smells while the target code smell is removed (side effects). With this investigation, we can have empirical evidence on how (in)complete composites affect the internal structural quality in practice. We can also understand whether and to what extent such composites introduce or not worse internal quality problems. As the main contribution, based on this obtained knowledge, we can guide developers and researchers on which and why the (in)complete composites can degrade or not the internal software quality.

RQ_3. How to recommend complete composites that better support developers in practice? The empirical knowledge on the effect of (in)complete composites is the starting point to guide developers and researchers on composite recommendations (the answer of RQ_2). The understanding of how complete composites are applied can improve this guidance. By answering our RQ_3, using the empirical knowledge obtained in RQ_2, we can provide a better guidance on how to recommend complete composites, thus addressing Research Problem 3. Thus, we aim to aid developers by recommending complete composites using observations derived from the practice. We created a catalog of complete composite recommendations. In addition, we explored how existing automated approaches for recommendations of composite refactorings can improve in terms of completeness based on the perspective of developers.

1.3.2 Methodology

Figure 1.2 presents the activities that were performed in this doctoral research.

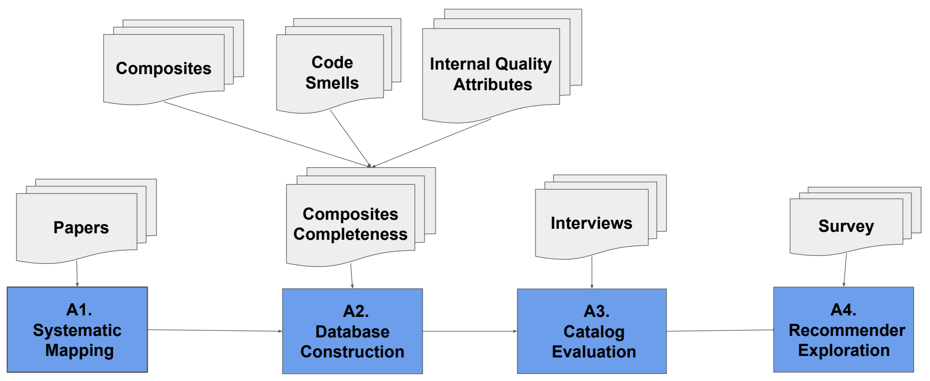


Figure 1.2: Thesis Activities

- **A1: Systematic Mapping.** This activity answers our RQ_1. The goal of this activity is to perform a systematic mapping study to define a conceptual framework of composite refactorings according to the literature. This systematic mapping was an extension of a preliminary systematic mapping [220], as previously mentioned in Section 1.3.1. We retrieved 140 primary empirical studies from three search engines: ACM Digital Library, IEEE Xplore, and Elsevier Scopus. These studies empirically presented approaches to composite refactorings of the last 12 years (2010-2022), differently of our previous study [203]. Some approaches consist of either tool support or methods to detect or recommend composites. Other studies, which were focused on investigating single refactorings, mentioned the occurrence of composite refactorings, but only superficially. Our systematic mapping is detailed in Chapter 2.

- **A2: Dataset Construction.** This activity consists in creating a dataset of composite refactorings applied in existing software projects. From this dataset, we can answer our RQ_2, analyzing the effect of (in)complete composites on the internal software quality. We created the dataset of composite refactorings. We performed two quantitative studies to create a database of composite refactorings, complete composites, and incomplete composites [76, 77]. For that, we collected data from 20 Java software projects according to the criteria detailed in [77]. We collected refactorings using Refactoring Miner 2.0 because this tool has a high accuracy [173]. We used the Organic tool to collect the code smells because this tool has a high accuracy [223]. For both studies, we created scripts to collect composites that are (in)complete to remove four smell types, namely *God Class*, *Complex Class*, *Long Method*, and *Feature Envy*. These smell types are the most common, and they can involve more than one class [8, 25]. According to previous studies [143, 171], a composite is classified as incomplete when it has at least one refactoring type recommended to remove the target smell type, but the smell was not removed. Similarly, our script detected a complete composite when the target smell was removed. In our first quantitative study, we collected 353 incomplete composites. We investigated the effect of incomplete composites on four internal quality attributes (cohesion, coupling, size, and complexity) using 10 code metrics based in [10]. Chapter 3 presents details about this study. In our second study, we collected 618 complete composites. They can remove the four smell types mentioned previously, but they can also remove the other 15 smell types detailed in [77]. We collected if these complete composites can also introduce these smell types while removing the target code smell. We also detected 18 refactorings types in complete composites that are not investigated previously [203, 204] to know if these refactoring types are often applied in practice. This second study is detailed in Chapter 4. Our dataset is available on websites of our published studies [76, 77].

- **A3: Catalog Evaluation.** Our RQ$_3$ is partially answered through this activity, in which we provided a catalog of composite recommendations using the results obtained from studies of the activity A2. Our catalog presents the five most common types of complete composites applied in the practice. Currently, this catalog describes the following details for each recommended complete composite: (i) the type of complete composite, (ii) the code smell to be removed, (iii) the code smells that can be introduced, (iv) an explaining about why these smells can be introduced, and (v) the description about how to fully avoid and/or remove these smells altogether. Our catalog is shortly presented in Chapter 4, and it is available on the website of our recent study in its extended form [193]. We detailed the improvement and evaluation of the catalog as follows.

– **A3.1: Catalog Improvement.** We created an website with our recommendations. On the catalog website, we summarized the four most common recommendations. We provided four recommendations that remove up to three common code smell types. According to quantitative results, the code smells that are commonly found together, they are *Long Method*, *Feature Envy*, and *Duplicated Code*. We then created two new code smell types that represent the conjunction of each pair of these smells. We called *Long Envious Method* when a *Long Method* and *Feature Envy* are detected on the same method. *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 statement statements. Our four recommendations are complete composite types that frequently removed these code smell types. We provide two complete composite types to remove *Long Envious Methods*, and two complete composite types for removal of *Long Signed Clone*. Our catalog presents the (i) definition of each code smell type, (ii) abstract and concrete examples of each smell type, (iii) definition of each complete composite type for smell removal, (iv) abstract and concrete example of each composite application, and (v) the side effects of each complete composite.

– **A3.2: Interview Execution.** This activity regards the application of a qualitative study to evaluate the catalog of composite recommendations from developers' and researchers' perspectives. We conducted interviews with 21 developers to evaluate the recommendations of our catalog. We selected smelly classes from three real software projects. The catalog evaluation was divided in four parts: (i) the developers evaluated what code smells are in that classes; (ii) we presented what code smell types were found in that classes according to our results and our catalog; (iii) the developers agreed or not with our code smells' identification; (iv) the developers proposed a solution to remove the code smell; (v) we showed our recommendations to remove the code smells according to our catalog; (vi) the developers accepted or not our recommendations; and (vii) the developers evaluated the catalog and study in general.

- **A4: Recommender Exploration.** The complementary response to our RQ$_3$ is answered in this activity. We explored an existing recommender of composite refactorings, OrganicRef [244]. We called this extension of REComposite. We investigated how this recommender can improve its recommendations to provide beneficial removal of code smells according to the perception of developers. We needed to extend OrganicRef because it does not recommend *Extract Method*. In addition, the tool does not detect common code smells, such as *Long Method*. We then implemented the recommendation of *Extract Method*, the identification of *Long Method*, and *Long Envious Method* [259]. We then performed a survey with 10 developers to evaluate the recommendations of composite refactorings in terms of completeness and side effects. Our findings can guide researchers and refactoring tool builders on how to recommend complete composite refactoring aiming for the beneficial removal of code smells.

1.4 Main Contributions

The main contributions of our research are described in this section. Our studies respectively resulted in a conceptual framework of composite refactoring, a list of findings about composite (in)completeness, a summary of lessons we learned with the evaluation of our catalog, and with the exploration of search-based algorithms for composite refactoring recommendations.

– **Conceptualization of Composite Refactorings.** We generated a conceptual framework of composite refactorings through the 140 primary studies analyzed in our systematic Mapping (Chapter 2), answering our RQ_1. The conceptual framework reveals that the literature represents a composite refactoring using seven representation models, nine characteristics, and thirty effects of composites. Studies often use multidimensional representation models, such as graphs or matrix, to automatically generate composite refactorings. Some studies also represent a composite as a sequence using vectors, or an arrays, for the recommendation of composite refactorings. These representations can be interesting to support step-wise, incremental composite refactorings because they help to know the order of each refactoring to be recommended. Those results can help future studies to decide what representation model is appropriate according to authors' approaches to supporting composites. The studies also mentioned the characteristic of completeness of composite refactorings and indicated that this characteristic can be used to properly recommend composite refactorings. However, existing studies did not present a formation definition of completeness and did not investigate in-depth refactoring (in)completeness. On composite effects, existing studies are often limited to empirically investigating the effect on code smells, they have little empirical evidence on how composites affect internal quality attributes in practice. In summary, our conceptual framework can guide researchers and refactoring tool builders on how to solve composite refactoring limitations and what is the appropriate characterization of composites according to each approach, like identification of composite refactorings or recommendation of composites.

– **Empirical Evidences on (In)completeness of Composite Refactorings.** After the conceptualization of composite refactorings, we confirmed that completeness is a characteristic of composite refactorings mentioned by previous studies and this characteristic can guide researchers and refactoring tool builders on how to recommend composite refactorings. In the literature, there is no in-depth empirically driven research about refactoring (in)completeness. We then performed two quantitative studies about (in)complete composites.

In the Chapter 3, we investigated how incomplete composites are commonly applied. These results can help to partially answer the RQ_2 of this proposal thesis. In our dataset with 353 incomplete composites, we confirm that composites are incomplete frequently to remove code smells. Our results reveal that incomplete composite refactorings with at least one *Extract Method* are often (71%) applied without *Move Methods* on smelly classes. However, surprisingly, incomplete composites maintain the internal quality attributes values, such as the code complexity. We have found that most incomplete composite refactorings (58%) tended to at least maintain the internal structural quality of smelly classes, thereby not causing more harm to program comprehension. We then can observe that the incomplete nature of composites has possibly not harmed even further the program comprehensibility and other related quality attributes.

The study of Chapter 3 focused on incomplete composites. A better understanding of complee composites is also needed. We performed a quantitative study (Chapter 4) to investigate complete composites and fully answer our RQ_2. In our dataset with 618 complete composites from 20 software projects, we have found (i) almost half (48%) of *Feature Envies* were removed when the composite *Move Methods* were applied. This information is not documented by existing composite recommendations. Since the occurrence of *Feature Envy* is a common situation [8], knowing about the usage of the *Move Methods* composite in advance can ease refactoring tasks, and (ii) nearly 36% of complete composites include *Extract Methods* to remove *Long Methods* have introduced *Feature Envies* and *Intensive Couplings* as side effects. Surprisingly, with the goal of improving readability, by removing *Long Methods*, developers end up degrading the software internal quality by creating unnecessary high coupling.

– **Catalog for Recommendations of Composite Refactorings.** With the knowledge obtained in our previous studies about (in)complete composites applied in the practice, we extracted the common (in)complete composites and created a catalog to recommend composite refactorings and alert developers about their possible side effects. Chapter 5 describes how we created and evaluated this catalog. Our catalog evaluation helps to partially answer our RQ_3. Our catalog recommends four composite refactoring types to remove two new types of code smells, and describes the possible side effects of our recommendations. Our recommendations are based on common complete composites applied in software projects. We interviewed 21 developers to evaluate out catalog. The most (85%) of developers reported that their solutions could have the worse side effects without our catalog recommendations. Besides, we observed a significant number (33%) of developers were unaware of side effects when proposing solutions to remove code smells. These findings reinforce the need of recommendations to at least caution developers about the side effects of composite refactorings.

– **Empirical Evidences of (In)completeness and Side Effects of Existing Automated Recommendations of Composite Refactorings.** In our catalog evaluation, we observed that developers often elaborate composite refactorings that could have the worse side effects, and many times developers were unaware of them.

Extension of an Existing Recommender. We then extended an existing recommender of composite refactorings to explore the perceptions of developers on automated recommendations of composite refactorings, mainly in terms of completeness and side effects. From the recommender OrganicRef [244], we generated REComposite, a recommender system of composite refactorings. REComposite generates composite recommendations using search-based techniques [30, 234], using three search-based algorithms: SA [240], MOSA [240–242], and NSGA-II [239]. The recommender identifies nine common types of code smells and recommends four refactoring types. REComposite indicates (i) the code smells that were identified, (ii) the smelly code elements, (iii) the composite that may be applied, and (iv) the side effects that may be minimized or removed.

Empirical Exploration of Search-based Algorithms for Automated Recommendations of Complete Composites. We performed a survey with ten developers to assess REComposite, exploring which search-based algorithm provides the best recommendations in terms of meaningfulness, completeness, and side effects. Our results reveal the most (80%) developers considered that NSGA-II recommendations are complete frequently. We then observed that NSGA-II is a search-based algorithm that better explore the search space. Thus, this algorithm generally finds new opportunities of composite refactoring. However, NSGA-II recommendations often can lead to side effects, according to 70% of developers. Based on this result, we perceived that existing search-based algorithms need to be improved to recommend complete composites without inducing side effects. We then provided a list of lessons learned for researchers and tool builders of recommenders of composite refactorings based in search, lessons such as (i) search-based algorithms need to better explore the search space aiming composite refactorings with different levels of completeness and (ii) the minimization of side effects may be a parameter to find composite refactoring for the beneficial removal of code smells.

In summary, our findings reveal that developers tend not to solve structural problems completely when they apply composite refactorings. These composites can remove one target code smell, but potentially introduce or do not remove other ones. This may be alert regarding the in-practice use of existing recommendations. Our results suggest that existing recommendations of complete composites should be either revisited or enhanced to explicitly include possible side effects. We then present a catalog that can help developers in practice to fully remove code smells, minimizing side effects. Our catalog can inspire future research to improve existing refactoring tools, helping developers to achieve code structure improvement, based on the code context of their development activities. In addition, we provide a list of lessons learned for researchers and tool builder of automated recommenders of composite refactorings that uses search-based algorithms.

1.5 Thesis Outline

This thesis is structured as a set of papers either published or under submission. Each chapter contains a paper. The remainder of this thesis is organized as follows. Chapter 2 shows our systematic mapping on composite refactorings. Our exploratory studies on (in)complete composite refactorings are reported in Chapters 3 and 4. Chapter 5 reports the presentation and evaluation of our catalog of composite recommendations. The description and assessement of our recommender system is described in Chapter 6. Finally, we concluded this thesis in Chapter 7.

Due to this thesis structure, then some sections become repetitive and some concepts have evolved throughout the doctoral research. Based on that, we recommend that this thesis be read in the following sequence to make the reading of this more fluid. Section 2.2.1 conceptualizes refactoring, Section 2.2.2 defines composite refactoring and describes the current state-of-the-art about composite refactoring. Section 2.3 reports the protocol of our systematic mapping about the conceptualization of composite refactorings. The results of our systematic mapping are detailed in Section 2.4. The related work and threats to the validity of our systematic mapping are in Sections 2.5 and 2.6, respectively. Section 2.7 concludes our systematic mapping and indicates our future steps.

After the conceptualization of composite refactoring, we formally defined composite refactoring in Section 4.2.1, and composite completeness in Section 5.2.2. The reader can read about our quantitative studies about (in)complete composites. Section 3.2.2 indicates the current limitations of incomplete composites. Section 3.3 describes the settings of the study about incomplete composites. Section 3.4 reports the overview of our dataset. The results about incomplete composites are detailed in Sections 3.5 and 3.6. The threats of validity in the study of incomplete composites are in Section 3.7. After the investigation of incomplete composites, the next sections are related to the study of complete composites. Section 4.2.3 focuses on the limitations of the literature on complete composites. Section 4.3.2 details the protocol of this study. The results on the complete composites are described in Sections 4.4 and 4.5. Section 4.6 indicates the threats to the validity of this study.

In a sequence, we suggest the read about our composite solutions. Section 5.4 details the study steps to build and evaluate our catalog of composite recommendations. Section 5.5 describes the results of the evaluation of the catalog. The threats to the validity of the catalog evaluation were detailed in Section 5.6. After that, the reader can read about the exploration of an existing recommender to support our catalog recommendations. Sections 6.2 and 6.2.4 show an overview of the search-based approaches to recommend composite refactorings. Sections 6.3 and 6.3.4 describe the study proceedings about the empirical exploration of the existing recommender. We explain the results of this study in Section 6.4. The threats to the validity of this empirical exploration are presented in Section 6.5. Finally, Chapter 7 concludes our thesis.

2
Composite Refactoring: Representations, Characteristics and Effects on Software Projects

This chapter contains a paper published in
the Information and Software Technology (IST) journal [260].

This chapter reports our systematic mapping on composites aiming to answer the RQ_1 (Section 1.3.1) of this thesis. A recent study reveals developers often apply *composite refactorings*. Two or more single refactorings form a composite refactoring. The knowledge about composites is fragmented and limited [203]. Previous studies mention the frequent occurrence of composites in software projects, but there is not a systematic conceptualization about what is and how to form a composite refactoring. Firstly, for that, it is necessary an understanding of how to conceptualize a phenomenon. We then performed a systematic mapping from 140 out of 454 empirical primary studies. These studies empirically suggest approaches to composite refactorings. Some approaches can be tools or methods to detect or recommend composites. Other studies investigate single refactorings, and they mentioned composite refactorings superficially.

We then created a conceptual framework of composites. Our conceptual framework includes the representation models, characteristics, and the effect of composites, revealing how existing studies investigate composites, existent conflicts, and gaps in the literature about composites. Thus, our conceptual framework and our findings answer our RQ_1 (Section 1.3.1). The next sections are the content of the published paper in IST [260]. This paper was an extension of a preliminary systematic mapping [203] that was published during the Master's degree and also described in the Master's dissertation [220] (Chapter 3). Along the Doctoral research, we additionally investigated representation models and explored primary studies of the last 12 years (2010-2022). The mapping study of the Master's dissertation only covered studies published until 2018. We also extended the initial mapping by revealing new characteristics of composite refactorings such as their completeness and complexity, as well as other types of composite effects.

2.1 Introduction

A code refactoring is a code transformation with the goal of improving the internal software quality [25]. Every single refactoring has a type that describes how and where each single refactoring may be applied on specific code elements (attributes, methods, classes, etc.). An example of refactoring type is an *Extract Method*. This refactoring type is applied when the developer creates a new method and extract pieces of source code from an existing method to the new method. Fowler's book presents a catalog of refactoring types [25].

A recent study indicated that developers often apply *composite refactorings* (formerly batch refactoring [143]), that are two or more interrelated single refactorings [143, 171]. Each instance of a composite refactoring is hereafter called composite. As an example, a composite can be formed by an *Extract Method* and a *Move Method*. The basic elements of composite refactorings are a representation model, characteristics, and effects [203]. A representation model is structural modeling, like a list or a graph. A representation is a representation model's instance of a composite. A characteristic identifies or classifies a composite, such as code scope or size. Composites also have effects, which refer to how a composite impacts the software quality. An example of effect is the removal of poor code structures, such as code smells.

Despite wide use in practice [46, 143] the knowledge about composite refactorings is still limited and fragmented. Previous studies used different representation models for composites, like a set [104, 109] or a graph of single refactorings [95, 105]. Given the diverging rules of these representation models, each representation model imposes important constraints on the characteristics of composites. Therefore, it is necessary to understand what are the representation models used by literature and why each study used these composite models. This understanding helps us to know what model is more appropriate for each refactoring approach.

The knowledge about composite characteristics is fragmented because studies have different definitions for the same characteristic. For example, *code scope* is a composite characteristic that describes where the composite was applied to source code. Nevertheless, previous studies use varying ways of determining the code scope of composites. Some studies considered that composites may be applied on a *single code element* [43, 143], whereas other studies consider that composites may be applied on *many code elements*. Thus, a summary of different descriptions of each characteristic is necessary to align each characteristic definition and also reveal composite characteristics that are not investigated in practice. Also, an in-depth understanding of composite characteristics is necessary to know how composites are applied in practice. For instance, what is the size of composites applied frequently in practice, what are the most common types of composites?

The knowledge on the composite effects is conflicting and superficial. An example of conflict is the effect on code smells. A study concluded that composites can remove code smells [50], but another study indicated that composites can introduce code smells [43]. These conflicts increase the misunderstanding of practitioners and developers on composite effects. Therefore, a summary of knowledge on composite effects is necessary to (i) know what are the possible effects of composites on the software quality, (ii) what composite effects are interesting to study in practice, and (iii) investigate the effect of composites in practice, minimizing or removing the existing conflicts of the literature.

Based on those limitations, we performed a systematic mapping to build a conceptual framework including the representation models, characteristics, and effects of composites. Systematic mapping is a broad review of primary studies in a specific topic area that aims to identify what evidence is available on the topic [36, 167]. We performed this empirical method based on strict guidelines of systematic mappings [36, 167]. We applied this method because it allows us to (i) summarize the current knowledge from the literature, and also (ii) create our conceptual framework of composites. A conceptual framework is a design artifact that provides a general view of the similarities and variabilities of a particular phenomenon (a conceptual framework is a type of feature model, and this definition was based in [32]). Recent studies [216–218] used feature models, like conceptual frameworks, to represent phenomena of software engineering. Based on those studies, we observed that a conceptual framework is an appropriate method to represent composite refactorings. This is because the conceptual framework allows us to represent graphically the phenomenon according to the literature, and guide researchers and practitioners about how composites can be investigated. As results, we retrieved 140 out of 454 empirical primary studies from three search engines: ACM Digital Library, IEEE Xplore, and Elsevier Scopus. These studies suggested approaches for composite refactorings. Some approaches propose tools or methods to detect or recommend composites. Other studies investigated single refactorings, and they mentioned composite refactorings superficially. Based on the analysis of these studies, we found the following results:

– On representation models, a composite can be modeled through seven representations. Existing studies used multidimensional representations (e.g., graphs or matrix) when they generated composites automatically [S25, S77, S112, S158]. These representations were adopted because their approaches suggested one or more refactorings. Therefore, multidimensional representations allow them to generate many "paths" of refactorings and know what refactoring may be suggested. These results can help practitioners or researchers to decide what representation model is appropriate according to their approaches to supporting composites. Studies usually represented a composite as a sequence using models as a vector [S29, S43, S79], or an array [S11, S33], when the authors aimed to recommend composites. These representations help them to know the order of each refactoring to be recommended.

– We found out nine characteristics. Examples of composite characteristics are *time* and *completeness*. *Time* regards the time window by developers for applying a composite in a software project. We have found out that time can be measured by *minutes* [PS11], *days* [PS10], *weeks* [S04]. A recent study indicated that 32% of composites took one day to be completed, and 46% took more than one day and less than 30 days (one month) [S02]. An overview of this characteristic is interesting to motivate the investigation of why composites are applied into one month. Also, studies can explore approaches to reduce the time to apply a composite. The composite *completeness* is related to fully achieve a specific goal, such as bug fixing. A recent study investigated this characteristic [S259]. However, some gaps need to be filled in the literature, for example, how many composites need to be applied to achieve a specific goal? Thus, our conceptual framework can help future studies mitigate existing gaps in composite characteristics.

– On composite effect, we observed that studies suggested that composites can have thirteen effects. However, existing studies are limited to understand these effect types empirically. For example, studies often investigated the effect on code smells, but they have little evidence on how composites affect internal quality attributes in practice. On *External Quality Attributes*, studies also mentioned composites can affect positively these attributes, such as the *correctness* (e.g., composites can help the bug removal or the correction of syntax errors) [PS22, PS23]. On the other hand, a paper suggested composites can introduce bugs (*Bug Introduction*) [S137]. However, these studies did not present empirical evidence about it. Our conceptual framework revealed the possible effects of composites, and existing conflicts in the literature on composite effects. Our results can motivate researchers to know what are the effects of composites that need to be explored deeply. Besides that, our results can motivate practitioners to create tools for applying composites to improve the software quality, such as the recommendations of composites to help the bug fixing.

In summary, the main contributions of our study are: (i) a summary of the body of knowledge on composites, and (ii) a conceptual framework on the representation models, characteristics, and the effect of composites. Our contributions can be used by future studies that aim to investigate how composites are applied in practice and to provide insights to build automated support tools for composite refactoring applications, such as a recommender system of composites. The remainder of this article is structured as follows. Section 2.2 summarizes background information aiming at supporting the understanding of our work. Section 2.3 describes our systematic mapping protocol based on the Goal Question Metric template [1]. Section 2.4 presents and discusses our study results. Section 2.5 overviews related work. Section 2.6 discusses the major threats to the study validity. We conclude our paper and present future study steps in Section 2.7. Finally, we presented as an appendix our primary studies.

2.2 Background

This section discusses code refactoring in terms of single refactoring (Section 2.2.1), composite refactoring (Section 2.2.2), and the use of conceptual framework to represent the knowledge of Software Engineering contexts (Section 2.4.1).

2.2.1 Refactoring

In the seminal book of Fowler et al. [25], refactoring is defined as "*the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs. In essence, when you refactor, you are improving the design of the code after it has been written.*". Code refactoring was designed for supporting the improvement of code structures in software projects [25]. Refactoring is done by successive code transformations affecting one or more code elements, e.g., methods and classes, in which developers can enhance the code maintainability [8, 46, 67]. Code transformations vary by type, which defines how the code structure should be modified[1] [25].

[1]Catalogs of refactorings are available at: <https://refactoring.guru/refactoring/catalog> and <https://refactoring.com/catalog/>

A recurring example of refactoring type in real software projects at the **method-level** is Extract Method [25]. This is a typical refactoring type from which specific code statements are extracted from a method in order to compose a new one [67]. This refactoring type can be summarized in four transformations [25]: (i) extract code statements from an existing method, (ii) create a new method; (iii) copy the extracted code statements into the new method, and (iv) replace the extracted code statements with a call to the new method.

An example at the **class-level** that is also frequent in real projects is Extract Class. This typical refactoring type aims to split a too long and complex class into two [25]. It consists of three modifications, namely: (i) decide how to split the features realized by a given class, (ii) create a new class, and (iii) move methods and attributes from the existing class to the new one. A typical refactoring type that affects the code structure at the **attribute-level** is Pull Up Attribute. This refactoring occurs when developers move an attribute that is common to two or more children in the class hierarchy to the parent class [25]. It consists of two steps: (i) inspect the attribute that is a candidate to move to the superclass, and (ii) move this attribute to the superclass.

According to Fowler [25], each single refactoring does little, but a sequence of these refactorings can produce a significant impact on restructuring software projects. Although this sequence of refactorings can be unrelated to each other, in practice we have seen that they direct or indirectly end up touching similar code elements [263]. In addition, developers tend to use a sequence of refactorings to achieve a wider goal than only restructuring the code, such as preparing the software to include a new feature [167]. Refactorings of this nature are described in the next section.

2.2.2
Composite Refactoring or Composite

Composite refactoring is composed by two or more interrelated refactorings [171]. The technical literature has referred to the concept of composite refactoring in many ways: composite refactoring [171], batch refactoring [143], refactoring sequence [43], and so forth. Unfortunately, the current knowledge of composite refactoring is quite scattered in the literature. Moreover, the previous studies provide only a partial viewpoint on the topic, with many speculations on what a composite refactoring is indeed. More critically, a summary of studies on composite refactorings is still missing in the literature. Consequently, little is known about the extent in which the current knowledge has an empirical ground. All the aforementioned issues make it difficult to design solutions for recommending composite refactoring and it limits the understanding of developers on how to apply composites and the effect of composites in practice.

Lin et al. [39] highlighted the importance of a correct sequence of refactorings (composites) to architectural restructuring. Also, they point out that architectural refactorings are a costly, risky, and challenging task. To find proper composites, the authors proposed a search-based algorithm. This algorithm focuses on (i) minimizing the inconsistencies between the target and source; (ii) improving OO design quality, such as coupling and cohesion; and (ii) maximizing the lexical similarity between the target and source. This shows how refactoring is a multi-facet task. Murphy-Hill et al. [46] performed an in-depth study about refactoring tool usage. They observed that 40% of refactorings performed using a tool occur in composites. These authors considered refactorings of the same kind that execute within 60 seconds of each another to identify composites. However, the authors acknowledge that this identification based on time window is not the best way to infer how refactorings are related.

On the basis that a single refactoring rarely suffices to fully remove code smells, Bibiano et al. [143] conducted an empirical study with 57 open and closed software projects to understand composite refactoring application from two perspectives: characteristics that typically constitute a composite (e.g., the variety of refactoring types employed), and the composite effect on code smells. As a result, the authors identified that although most composites are applied on more than one method, they are usually composed of the same refactoring type. Surprisingly, it was found out that composites mostly ended up introducing or not fully removing smells.

A recent study provided an investigation from different viewpoints on how composite refactoring manifests in practice [171]. In this study, the authors analyzed how different kinds of composite refactorings affect the removal, prevalence or introduction of smells. For that, they provided two heuristics to respectively characterize and identify composite refactorings within and across commits. An interesting result is that many smells are introduced in a program due to “incomplete” composite refactorings.

2.3 Systematic Mapping Protocol

A systematic mapping can bring us a systematic conceptualization of a phenomenon based on knowledge provided by the literature. Systematic mapping is a broad review of primary studies in a specific topic area that aims to identify what evidence is available on the topic [36,167]. We relied on well-known literature guidelines aimed at carefully designing our systematic mapping protocol [31, 36, 70, 167]. Section 2.3.1 introduces the study goal and research questions. Finally, Section 2.3.2 describes the protocol steps and procedures.

2.3.1 Goal and Research Questions

Our study goal is: *to analyze* the technical literature of composite refactoring; *for the purpose of* building a conceptual framework of the representation models, characteristics, and the effect of composite refactoring according to previous studies; *from the viewpoint of* Software Engineering researchers specialized in code refactoring-related research; *in the context of* studies published from last 12 years (2010 to 2021). Aimed at addressing our study goal, we have defined three Research Questions (RQs) described as follows.

RQ$_1$: *How has composite refactoring been modeled?* The literature reveals that previous studies represent composite refactoring in many ways. Examples of representation models are graphs, sets, and so forth. A summary of representation models used in the composite refactoring context could reveal the most appropriate model according to the purpose. For instance, graphs could fit the strategies to optimize the number of refactorings that constitute a composite refactoring, e.g., [43,55]. Other representation models like sequences could fit other purposes, such as a sequence of refactorings to remove a code smell, e.g., [132]. Through RQ$_1$, we expected to summarize the representation models of composite refactoring employed so far, especially to drive future research on recommender systems.

RQ$_2$: *What characteristics constitute a composite refactoring instance?* The current knowledge of composite refactoring is unfortunately scattered in the literature. Each previous study assumes some characteristics as representative of composite refactoring. Certain characteristics are intuitive, e.g., the number of refactorings within a composite refactoring instance [43, 46, 143]. Others are more specific to the context of each study, e.g., the time between the refactorings of a composite refactoring [46]. As one could expect, these characteristics vary across studies. We highlight that many characteristics are mere assumptions of the study authors [143]. Besides that, the literature is conflicting on the characteristics. For example, some studies assumed that a composite refactoring may affect a single code element [43,143], and other studies motivated that a composite refactoring can affect many code elements [35,46].

Thus, an understanding of the characteristics of composite refactoring without a mapping of the literature is unfeasible. RQ_2 is addressed to organize the current knowledge, thereby clarifying convergences and conflicts on what really characterizes a composite refactoring.

RQ_3: *Which are the effects reported in the literature of applying composite refactoring on software projects?* Code refactoring was originally designed to enhance the maintainability of software projects. Since then, studies empirically investigated the refactoring effect on various software quality attributes, including the pure maintainability enhancement. Examples of attributes are the bug-proneness of code elements [2, 21] and the improvement of internal quality attributes [10,168], e.g., cohesion and coupling. A recent study [143] investigated empirically the composite refactoring effect on software projects. In this case, they focused on the introduction and removal of code smell instances. Contrary to expectations, they found out that some composite refactoring recommendations of Fowler [25] rarely remove poor code structures [143]. There may be other types of composite refactoring effects assumed by previous studies that deserve empirical validation. RQ_3 aims at eliciting these types of effects, thereby driving future work.

2.3.2 Steps and Procedures

Figure 2.1 depicts the sequence of study steps and their respective artifacts according to the existing guidelines for systematic mapping [36]. Below, we described and justified each step. Our results and artifacts are available on [219].

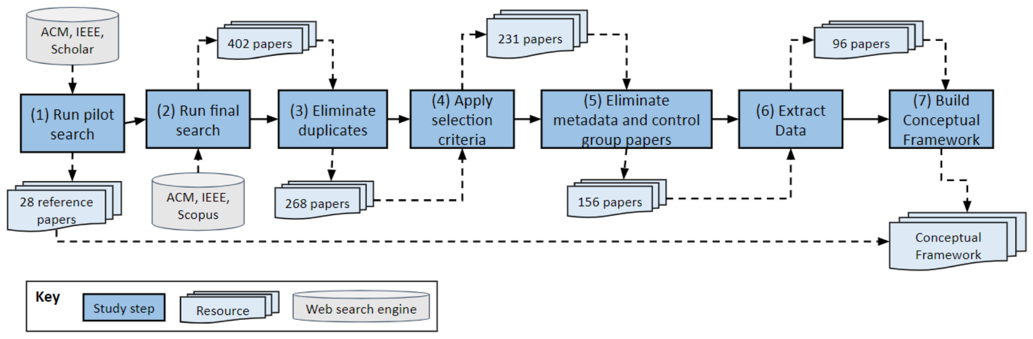


Figure 2.1: Study Steps and Artifacts

Step 1: *Run Pilot Search.* This step aimed at designing a pilot search that could drive us in the design of a systematic mapping. We then created an initial search string via a pilot search based on our current knowledge of composites. Our very first search string was: (*batch refactoring** OR *continuous* *refactoring** OR *sequence of refactoring**). At that time, we were unfamiliar with the *composite refactoring* term. The "*" character indicates the inclusion of any variant words whose prefix precedes this character. *Batch refactoring** was found in our initial exploration of the matter, in studies such as [46, 65]. *Sequence of refactoring** was also quite common in our exploration and appeared in studies such as [43, 55, 64, 254]. *Continuous refactoring* has also appeared in a previous study [65]. We then performed the designed pilot search.

First Round of Pilot Search. In July 2017, we performed a first round of pilot search [169] on two Web search engines: ACM Digital Library[2] and Google Scholar[3]. We took our very first search string (**Step 1**) as a basis. We considered the 20 first studies listed for each engine by relevance (default criteria of the engines). After performing backward snowballing procedures [31], we additionally collected a total of eight primary studies.

Second Round of Pilot Search. Later in February 2018, we ran a second round of pilot search [169] by (i) adding the IEEE Xplore[4] engine and (ii) discarding the *continuous refactoring* term. The term discarding was motivated by a large amount of out-of-scope search results. Indeed, continuous refactoring often targets an unceasing refactoring process rather than the combination of code transformations. The search results consisted of 15 additional primary studies. We performed backward snowballing procedures on these 15 studies. Five additional studies were added to the set, totaling 20 new primary studies. As a result, we selected 28 papers listed in Table 2.1. These papers were considered in our final set of papers for analysis because, after reading them, we noticed the papers were explicitly addressing composite refactoring. Most of them were published in journals or conferences with high reputation, including IEEE Transactions on Software Engineering (TSE) and International Conference on Automated Software Engineering (ASE).

Step 2: *Run Final Search.* This step consisted of running the definitive search for primary sources. Aimed at more refined search results, we kept ACM Digital Library and Xplore, but replaced Google Scholar with Elsevier's Scopus[5]. Scopus has been largely used for systematic mappings in Software Engineering by its large dataset of papers [170]. We learned a lot from reading the metadata (title, keywords, and abstract mostly) of the 28 primary studies collected in **Step 1**. Table 2.2 presents the search string we designed from the acquired knowledge. As a result, this search string retrieved a total of 454 primary studies.

[2]https://dl.acm.org/
[3]http://scholar.google.br/
[4]https://ieeexplore.ieee.org/
[5]https://www.scopus.com/

Table 2.1: Papers Selected through the Pilot Search

Paper Title and Reference
A methodology for the automated introduction of design patterns [PS1]
A new software maintenance scenario based on refactoring techniques [PS2]
Algebraic and cost-based optimization of refactoring sequences [PS3]
An empirical study of refactoring: Challenges and benefits at Microsoft [S04]
Composite refactorings for Java projects [PS5]
Designing and developing automated refactoring transformations [PS6]
DRACO: Discovering refactorings that improve architecture using fine-grained co-change dependencies [PS7]
Evolving transformation sequences using genetic algorithms [PS8]
Experimental assessment of software metrics using automated refactoring [PS9]
FaultBuster: An automatic code smell refactoring toolset [PS10]
How we refactor, and how we know it [PS11]
Identifying refactoring sequences for improving software maintainability [PS12]
Improving refactoring speed by 10x [PS13]
Interactive and guided architectural refactoring with search-based recommendation [PS14]
Pareto optimal search based refactoring at the design level [PS15]
Recommendation system for software refactoring using innovization and interactive dynamic optimization [PS16]
Refactoring with synthesis [PS17]
Scripting parametric refactorings in Java to retrofit design patterns [PS18]
Search-based detection of high-level model changes [PS19]
Search-based refactoring based on unfolding of graph transformation systems [PS20]
Search-based refactoring detection [PS21]
Search-based refactoring using recorded code changes [PS22]
Search-based refactoring: Towards semantics preservation [PS23]
Searching for opportunities of refactoring sequences: Reducing the search space [PS24]
Template-based reconstruction of complex refactorings [PS25]
The use of development history in software refactoring using a multi-objective evolutionary algorithm [PS26]
TrueRefactor: An automated refactoring tool to improve legacy system and application comprehensibility [PS27]
WitchDoctor: IDE support for real-time auto-completion of refactorings [PS28]

Table 2.2: Final Search String

(*batch refactoring** OR *chain of refactoring** OR *combination of refactoring** OR *complex refactoring** OR *composite refactoring** OR *list of refactoring** OR *refactoring combination* OR *refactoring sequence** OR *refactoring set** OR *sequence of refactoring** OR *sequence of transformations* OR *set of refactoring** OR *transformation sequence**) AND (*code structure** OR *software evolution* OR *software maintenance* OR *software project** OR *source code*)

Step 3: *Eliminate duplicates.* We considered as duplicated papers, studies that have the same title, list of authors, avenue, and year of publication. In that way, we eliminated 71 papers, remaining 383 (84%) out of 454 papers.

Step 4: *Apply Selection Criteria.* This step consisted of defining and applying selection criteria, to either keep or discard primary studies for the analysis. After rounds of discussion among the paper authors, we defined our set of selection criteria. Table 2.3 summarizes these criteria. We included studies that have been published for the last 12 years (2010-2021) (Section 2.3.1), because we observed in this period there was an increase of publications related to refactoring [168]; in either an international conference, journal; in English; in an online repository so that we can download the full study version. We eliminated papers published in workshops because they are generally short papers or position papers. Studies that neither theoretically nor empirically mention composite refactoring were discarded. We also discarded gray literature: short studies (3 pages or fewer), poster summaries, and “white papers”. To secure the relevance of papers, we only included venues that are known for publishing high-quality software engineering research; this strategy is also followed by other authors (e.g., [255]). By applying the selection criteria, we included the 28 papers from Pilot Search with the 383 primary studies to double validation.

After following our elimination criteria, a total of 295 studies remained.

Table 2.3: Criteria Selection of our Systematic Mapping

Criteria Selection	**Justification**
Papers published between 2010-2021	This period there was an increase of publications related to refactoring.
Papers published in English	To facilitate the reproducibility of our study because papers written in English are more accessible by the international Software Engineering Community.
Papers that are not published in workshops	They are generally short papers or position papers.
Papers published in International conferences, and journals	To facilitate the reproducible of our study because international conferences and journals are more accessible by the international Software Engineering Community.
Papers published that are not from grey literature	To ensure that our results were extracted from venues that are known for publishing high-quality software engineering research

Step 5: *Filter based on the Metadata.* We then evaluated the metadata of each paper, i.e, the title, abstract, and keywords, evaluating if the paper investigated composites indirectly or not. We observed that 81 papers considered only single refactoring or did not investigate code refactoring, remaining 214 papers. We also removed 18 studies that were found in the pilot search of this list because we already had read the metadata of those papers. We then aggregated the data of these studies in **Step 6** to build the conceptual framework. In that way, we obtained our final set of 196 primary studies for analysis.

Step 6: *Extract Data.* Table 2.4 presents the metadata extracted of each paper, the extracted data helps to validate if these studies investigate or mention composites. Two authors extracted this metadata. Thereafter, we divided a set of papers for each author to read, following a protocol based on existing guidelines. We divided the papers among authors, aiming to eliminate a misunderstanding on each paper. All authors read the papers and answered a web form to fill the data types extracted for each selected paper. Two authors evaluated the answers to this form and evaluated the papers that help to address our RQs. We have that 84 papers were discarded, remaining 112. Finally, after we retrieved 28 papers from the pilot search that were mentioned in **Step 5**, our dataset consisted of 140 selected papers.

Step 7: *Build Conceptual Framework.* Two authors aggregated the data according to each question of the form. For example, all answers to the question related to the characteristics of composites were aggregated in a single document. We then evaluated if each answer is related to what was asked. For example, if the answer is a characteristic, effect, or representation model of a composite according to the definition of each of these topics presented in the form. When we had doubts about the answers, we then contacted the authors

Table 2.4: Metadata extracted of Each Paper

RQ	Data Type	Description
N/A	Paper Title	Full paper title
	Authors	Full authors list
	Year	Year of publication
	Type	Paper type, e.g., conference and journal
	Venue	Full name of the publication venue
	Abstract	Full abstract as constant in the paper
	Keywords	Paper keywords
	Methodology	Methodology of previous studies
RQ_1	Representation Model	Representation model of composite refactorings
RQ_2	Characteristics	Any characteristics that constitute a composite refactoring
RQ_3	Effect	Any effects of composite refactoring

that answered them to understand better each answer. If necessary, we reread the paper that inspired each answer to validate it.

We have applied some basic Grounded Theory procedures [13, 63] on the extracted data. We performed both *coding* and *classification* on excerpts extracted for each paper in order to identify the composite characteristics and types of effect. We executed the following steps of our protocol: (i) we tabulated sentences that mention composite characteristics or effect; (ii) three authors validated the tabulated sentences in order to assure they mention characteristics and types of effect; (iii) three authors together grouped the sentences by semantics, thereby extracting final representation models, characteristics and types of effect; (iv) we then created a list of representation models of composites based on the literature (RQ_1); (v) we grouped composite characteristics based on the current knowledge (RQ_2); and (vi) we listed the types of effect of composites in our feature model of composites (RQ_3).

2.4 Results and Discussion

The Conceptual Framework. Figure 2.2 introduces our conceptual framework of composite refactoring based on the literature. A composite has at least a representation and characteristics as mandatory features, and a composite can have an effect, thus an effect is an optional feature of our conceptual framework. We also observed the characteristics of a composite are related to the structure or the application of the composite, and these characteristics are concrete features of our framework. We describe details of the conceptual framework content in Sections 2.4.1, 2.4.2, 2.4.3, 2.4.4, and 2.4.5.

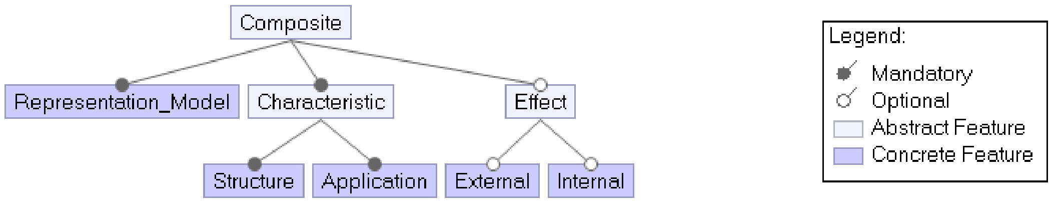


Figure 2.2: Overview of Conceptual Framework

2.4.1 The Use of Conceptual Model for Composite Refactoring Characterization

A strategy commonly used to represent the knowledge of a particular domain is the use of conceptual frameworks [213], such as feature models [32,221]. A feature model is a design artifact that provides a general view of the similarities and variabilities between features that may compose the possible configurations or characterizations of a particular domain [32]. Through the conceptual frameworks (like feature models), software engineers can characterize all possible configurations of a software family, in which each configuration represents a fully formalized specification of a software product [214].

Recent studies [216–218] have been using the feature model notation not only to model the different configurations of a software product, but also to summarize the knowledge extracted from primary or secondary empirical studies. Oliveira et al. [216] used the feature model notation to represent a summary of qualitative results on the activities performed by developers, either individually or collaboratively, for the activity of identifying code smells. Each feature then represents these activities, in which the six mandatory activities for smell identification are presented by the top-level features. Moreover, for each activity, the authors represented alternative activities that may exist. These alternative activities were represented through Or and XOr relations based on the feature model notation.

In our study, we used the feature model notation to provide a conceptual framework of composite refactorings based on the knowledge extracted from the literature. Similar to the aforementioned study, we used the notation of optional and mandatory features, as well as the Or and XOr relations, to characterize complex knowledge. In our case, different aspects of composite refactorings.

2.4.2 Representation Models of Composite Refactorings (RQ$_1$)

For our RQ$_1$, we elicited the representation models of a composite. Figure 2.3 presents the representation models of a composite. Our results revealed that a composite can be represented by seven representation models according to the needs of each study (first nodes from root), as described in what follows.

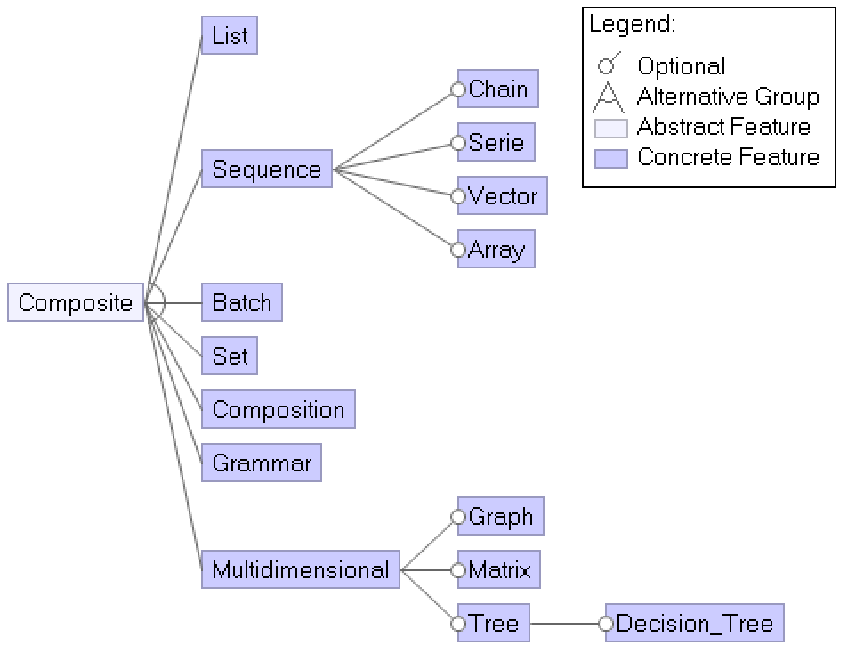


Figure 2.3: Representation Model of Composites

Studies represent composites through different ways. Some studies represented a composite as a *list of refactorings* [S04,S38,S127]. This representation model is very limited because it does not specify if this list may be applied on single or multiple code element(s). For instance, by using this representation model, it is possible to represent a composite as a list of all refactorings applied in a software project. Thus, this kind of representation requires the specification of constraints to avoid a limited definition.

Table 2.5 presents the primary studies that mention each representation model. The first column shows the id of the paper according to our artifacts. The second column presents the representation model that was cited for the primary studies. The last column indicates the reference number of each study. We can observe that the majority of the studies consider a composite as a *sequence of refactorings* [S30, S32, S69]. This kind of representation, in which the order is important, is typically used by approaches to recommend composites. Therefore, for these studies, it is necessary to know the order of refactorings in a composite for a better recommendation. These studies mentioned a composite using a **Vector** [S29, S43, S79], an **Array** [S11, S33], a **Chain** [S46,S68,S126], or a **Serie** [S23]. However, the composite representation through sequence can have some limitations in practice. For example, if a composite considers all refactorings applied in a single commit only, then it is not always possible to know the order of each refactoring within the current commit. Thus, these representations can be used when it is possible to detect the order of refactorings, such as multiple commits, multiple revisions, or in real-time (exactly when each refactoring was applied, for instance, in the code review context).

A composite can be represented as a **set of refactorings** [S71,S72,S141]. This representation model allows analyzing each refactoring of a composite as a unique instance of refactoring, independently of refactoring types. For instance, a composite can have two refactorings *Extract Methods*, in which each *Extract Method* is considered unique (*Extract Method*$_1$ and *Extract Method*$_2$). This representation model is used generally to represent a composite as a set of refactorings that has the same type only. There are studies that use the term **Composition** to indicate that a composite may have more than one refactoring type [S89, S148]. Some studies are concerned with the order of the refactorings in their composites, thus, they use a set ordered to represent composites [S11, S33, S45, S59, S110, S119].

Some pieces of work used multidimensional representation models to represent a composite. These representations can be **Graph** [S77, S112, S158], **Matrix** [S25], and **Tree** [S158]. Such studies proposed these representations because they introduced approaches to generate composites according to a certain goal. Then, they needed to use these representations because their approaches suggested one or more refactorings, generating many "paths" of refactorings. Some studies represented graphs with refactorings as edges, and nodes as code elements [S59, S112, S125]. These graphs can be directed, in which the source nodes are the code elements before the refactoring, and the target nodes are the code elements after the refactoring [S125]. There are graphs in which each edge represents step by step of the refactoring [S112]. Jensen et al. [S158] represented each state of code elements as a tree, each state represented how the code elements were after the application of each refactoring of the composite. Another study [S148] represented a composite using the Hierarchic Task Network, there is a graph in which each node is a task, and some tasks are more relevant than others. In that way, the root node represents the main refactoring of the composite, and other nodes represent other refactorings that are less relevant for the composite.

Other study [S57] represented a composite through a **Grammar**, indicating that refactorings of a composite may follow rules of this grammar. For example, a rule that states some refactoring types may not be applied after the application of other refactoring types. In that case, *Extract Method* cannot be applied after the application of *Inline Method* on the same method, because an inline method cannot be extracted, once this method was removed from *Inline Method.*

Summary 1: Composites are usually represented as a sequence when authors aim to recommend composites. Multidimensional representations (e.g., graphs and matrix) are generally adopted when authors aim to generate composites automatically.

Table 2.5: List of Papers that mention Representation Models

Paper ID	**Representation Model**	**Reference**
11, 33	Array	[S11, S33]
9	Batch	[S9]
46, 68, 126	Chain	[S46, S68, S126]
15,55,103,123,149	Composite	[S15, S55, S103, S123, S149]
89,148	Composition	[S89, S148]
59	Decision Three	[S59]
57	Grammar	[S57]
77,112,125,149,154,158,250,261,280	Graph	[PS152, S77, S112, S125, S149, S154, S158, S250, S261, S280]
4,38,42,119,127,138,155,248	List	[S04,S38,S42,S119,S127,S138, S155, S248]
25	Matrix	[S25]
90	Queue	[S30]
8,10,11,15,17,30,32,33,39,45,46,49, 53,55,59,62,67,69, 74,75,76,79,86,90,93,99, 101,110,118,123,143,148,266, 273,274,275	Sequence	[S8, S10, S11, S15, S17, S30, S32, S33, S39, S45, S46, S49, S53, S55, S59, S62, S67, S69, S74–S76, S79, S86, S90, S93, S99, S101, S110, S118, S123, S143, S148, S266, S273–S275]
23	Serie	[S23]
2,9,11,15,33,34,45,52,71,72,85,101,110,119, 129,141,247,259,265,276,280	Set	[S02, S9, S11, S15, S33, S34, S45, S52, S71, S72, S85, S94, S101, S110, S119, S129, S136, S141, S247, S259, S265, S276, S280]
158, 258	Three	[S158, S258]
29,31,33,38,43,60,79,95,99,124,155,248	Vector	[S29, S31, S33, S38, S43, S60, S79, S95, S99, S124, S155, S248]

2.4.3 Characteristics of Composite Refactorings (RQ_2)

Figures 2.4 and 2.5 answer our RQ_2 about composite characteristics. We classified the composite characteristics as **Application Characteristics** and **Structure Characteristics**. These classifications are represented as abstract features of a composite in our conceptual framework. The Application characteristics define how a composite was applied. For example: Who applied it? Which time the composite was applied? The Structure characteristics specify the structure of a composite, i.e., how the composite is formed. For instance: What do refactoring types form a composite? How are refactorings of a composite ordered? A characteristic can have different manifestations. An example is the characteristic related to how many commits have a composite. This characteristic can have two manifestations: a single commit only or many commits. In our conceptual framework, the characteristics are represented as mandatory features, and their manifestations are represented as alternative features. Table 2.6 presents the primary studies that mention each characteristic. The first column shows the id of the paper according to our artifacts. The second columns presents the characteristic that was cited for the primary studies. Last column indicates the reference number of each study.

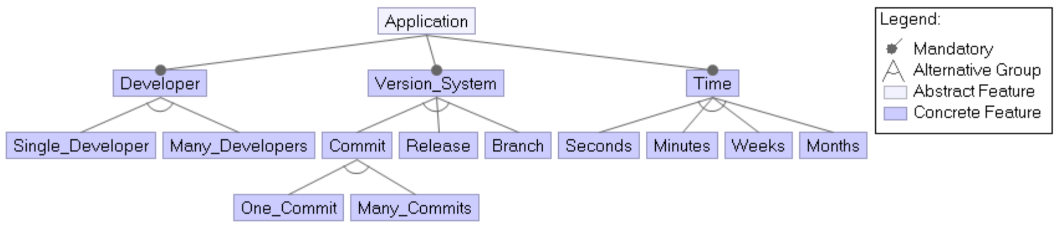


Figure 2.4: Characteristics related to Composite Application

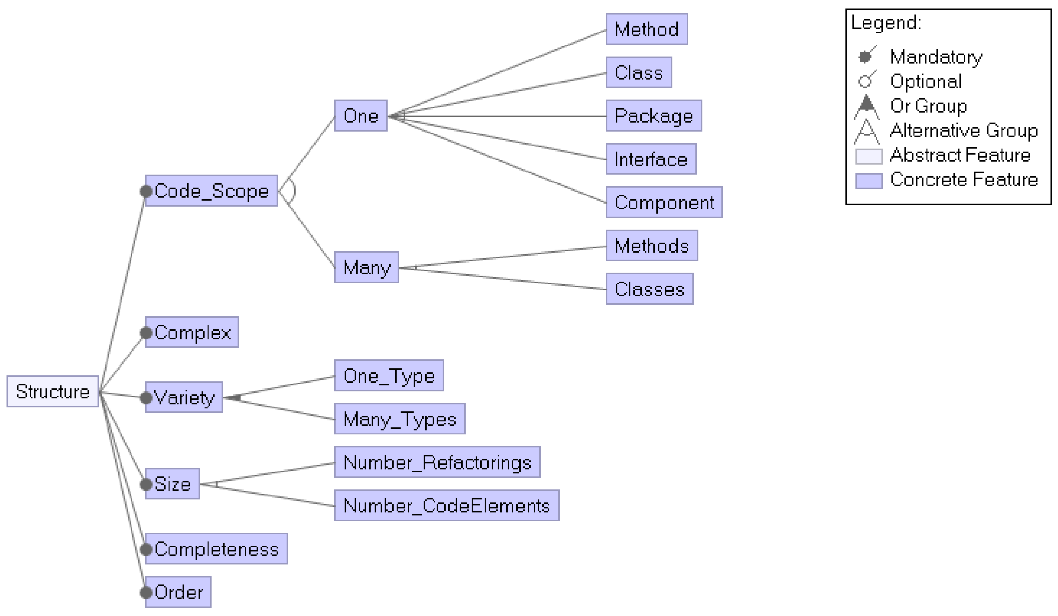


Figure 2.5: Characteristics related to Composite Structure

The Application group is composed of three composite characteristics: *developer*, *time* and *version system*, as depicted in Figure 2.4. These characteristics regard the way, who, and how to apply composites. Thus, these characteristics say something about the developer practices along with the composite application. These characteristics have two or more manifestations described by previous studies. We discussed each characteristic and its respective manifestations as follows.

Table 2.6: List of Papers that mention Characteristics

Paper ID	**Characteristics**	**Reference**
129	branch	[S129]
247, 248,259	completeness	[S247, S248, S259]
37, 117, 156	complex	[S37, S117, S156]
2, 111, 45, 55, 250, 261	developer	[S02,S45,S55,S111,S250,S261]
11,15,19,25,29,30,31,38,43,45,53,55,59,60, 62,67,68,75,76, 79,86,93,95,118,120,123, 124,125,126,127,138,143,146,148,154,158	order	[S11, S15, S19, S25, S29–S31, S38,S43,S45,S53,S55,S59,S60, S62,S67,S68,S75,S76,S79,S86, S93,S95,S118,S120,S123–S127, S138, S143, S146, S148, S154, S158]
2,10,15,29,34,43,46,50,52,54,55,57,60,62, 66,67,75,76,79,86, 89,89,93,93,95,110,112, 118,124,126,129,137,140,141,148, 149,247, 266,274	scope	[S02, S10, S15, S29, S34, S43, S46,S50,S52,S54,S55,S57,S60, S62,S66,S67,S75,S76,S79,S86, S89, S93, S95, S110, S112, S118, S124, S126, S129, S137, S140, S141, S148, S149, S247, S266, S274]
2,11,15,30,32,34,37,47,49,50,60,74,76,79,85, 91,101,103,111, 135,158,247,248,248,250, 266,280	size	[S02, S11, S15, S30, S32, S34, S37,S47,S49,S50,S60,S74,S76, S79, S85, S91, S101, S103, S111, S135, S158, S247, S248, S250, S266, S280]
30,54,111,112,247,261	time	[S30, S54, S111, S112, S247, S261]
2,11,16,25,29,43,49,50,52,54,60,62,66,68, 75,76,79,86,93,110, 111,124,129,131,247, 250,259,273,274,275,280	variety	[S02, S11, S16, S25, S29, S43, S49, S50, S52, S54, S60, S62, S66,S68,S75,S76,S79,S86,S93, S110, S111, S124, S129, S131, S136, S247, S250, S259, S273–S275, S280]
129, 137	version system	[S129, S137]

Developer regards the number of software developers that contribute with the composite application on a software project. Various studies [PS22, PS11, PS17] suggested that each composite is applied by only *one developer*. This view is quite reasonable when considering that motivations behind refactoring are often associated with a problem faced by a particular developer [S02]. However, a study [S04] assumed that *two or more developers* can work together in order to fully apply a certain composite. This approach works as a response to cases in which developers join forces to plan and perform a complex composite.

The way how this characteristic manifests in practice strongly depends on some organizational aspects. On the one hand, large software projects have entire development teams allocated to refactor code structures, which increases the chances of composites being composed and applied by two or more developers in conjunction. Conversely, small projects may have only a few (or just one) developers allocated to perform code refactoring. By investigating the *developer* characteristic, researchers can better understand how the allocation of developers to refactor the code may eventually affect the quality of the resulting code.

On the *Version System* characteristic, composite can be applied into a **commit**, **release**, or **branch**. *Commit* regards how many commits were performed by developers to apply a composite. A study indicated that a composite can be completed in *one commit* [S02]. However, other studies have mentioned that a composite may take *many commits* to be completely applied [PS22,S04]. Studies also suggested that there are software companies that need to create branches or releases only to apply a composite refactoring [S04]. These are scenarios in which, the software is highly restructured, then developers need to apply many refactorings that can affect the software behavior [S04]. Therefore, developers need to apply these refactorings separated from other changes to facilitate the restructuring of the system to support a future migration of technology [S52] or a feature addition [S9].

Time regards the time window by developers for applying a composite in a software project. Only a few studies refer to this composite characteristic, but still, the authors have varied viewpoints. One particular study assumed that each refactoring into a composite should be applied in up to 60 seconds after the previous refactoring [PS11], then the time spent for applying a composite can be *minutes*. Other studies [PS22,PS10,S04,PS6] suggested *to not constrain the time* so that consecutive refactorings can be applied at any time.

We found out different manifestations of the *time* characteristic across the studies. They suggest the time computation by measuring working *seconds* [S112], *minutes* [PS11], *days* [PS10], *weeks* [S04], or *months* [S54]. In this case, the manifestations of this composite characteristic also depend on specific team or organization practices. It may be the case of developers having too little time to complete their composites due to high demands for other maintenance tasks than code refactoring. Thus, the time span for the composite application can be shorter than in teams or organizations, in which developers can spend entire weeks with code refactoring. In summary, we observed a clear opportunity for future research in order to: (i) identify ways to compute *time* in real settings, and (ii) understand the relationship between the resulting quality of a composite and the time spent on it.

The Structure Group has four composite characteristics: *code scope*, *variety*, *size*, and *order*. These characteristics regard internal aspects of the composite. In other words, these characteristics reflect the internal composite structure. Three identified characteristics had at least two manifestations mentioned by previous studies. We provided below a detailed discussion about each characteristic and its respective manifestations.

Code Scope regards the scope of code elements affected by refactorings that constitute a composite. This characteristic is also called of *Coverage*, a level of the coverage of a composite is indicated by the number of code elements touched by the composite. Certain studies assume that a composite is performed on a single *method* [PS12, PS24] or *class* [PS3, PS7, PS24]. Conversely, other studies such as [PS22, S04] considered that each composite may affect *multiple code elements* together. Certain limitations may emerge depending on the *scope of code elements* considered for computing a composite. For instance, by assuming that composites are constituted by refactorings exclusively applied to methods, the refactorings at attribute and class levels, such as *Pull Up Attribute* and *Extract Class*, are overlooked. Thus, the understanding of how composites affect the code structure of a software project is limited to the method scope only. Even though our work focuses on refactoring at the code level, composite refactorings may realize changes that end up being relevant to the level of Software Architecture or/and Software Design. For supporting these major changes, composite refactorings are often applied on *packages* [S75, S76, S86], *interfaces* [S43, S67], and *components* [S67]. In this context, we observe that composite refactorings are often applied while adding or altering design patterns [S122, S158]. The combinations of refactorings such as *Extract Interfaces* and *Extract Classes* can help to separate concerns and include design patterns such as Strategy and Factory Methods [S122, S158].

Complex is a characteristic because studies suggested that composites have different levels of “complexity”. These complexity levels can change according to the “number of refactorings”, “number of refactoring types” or “number of affected code elements” of each composite. This complexity of composites can help us to measure the effort to apply each composite. On the recommendation of composites, it is possible to evaluate composites that have high complexity and a non-effective result, they may not be recommended, for example.

Variety regards the diversity of refactoring types applied along with a composite. Some studies [PS11, PS16, PS26] considered the *number of occurrences by refactoring type* as a means to differentiate composites. Other studies discuss the so-called *refactoring patterns*, i.e., the varied combinations of different refactoring types in order to compose a composite [PS12,PS24,PS20,PS17]. The *variety* of refactoring types within a composite depends on the assumed *scope* of composites. In fact, if composites strictly affect methods, all refactorings at attribute and class levels could be ignored in the composition. We highlighted that the approach used by a specific study [PS11] considers only refactorings of the *same type*. In this case, certain composites suggested by previous pieces of work [PS12, PS20, PS16, PS17], which combine refactorings at different levels to remove code smells, would be ignored. Future work could empirically validate if the variation of types has different effects on code maintenance.

Size regards the extension of a composite applied to the code structure. Previous studies used different approaches for measuring the *composite size*. Many studies [PS11, PS10, PS24, PS20, PS7, PS9, PS13, PS16, PS28] rely on *the number of refactorings* that constitute a composite. Other studies like [PS12, S02] count how many code elements are modified by the refactorings that constitute the composite. These studies do not present empirical evidence about the common *size* of composites in practice. Besides, only one study has speculated an upper limit to the *composite size* [PS17]. However, we understand that the lack of consensus about which code elements are usually affected by composites (see *Code Scope*) makes it hard to compute the *size* of composites.

Completeness is related to achieve some goal fully. For example, a composite can be complete to remove a specific code smell, i.e, this composite is able to remove this code smell fully. In another hand, some composites can be incomplete to remove other code smell, i.e, this composite did not remove this code smell. Composites can also have a "completeness" in relation to internal or external quality attributes improvement, the removal of a bug, and others. This characteristic can help us to reveal and evaluate what refactorings are necessary "to complete" a composite to achieve a specific goal. Studies revealed in the practice that composites are frequently incomplete (did not fully remove) to remove code smells [S02, S247]. However, the literature is limited to know what are the most common composites that are incomplete, and what code smell types are not removed fully in practice. This knowledge is necessary to guide developers about how to complete composites aiming at the full removal of code smells.

Order regards how the refactorings are organized in a composite. Previous studies considered composites as two or more refactorings whose order does matter to distinguish one composite from another [PS12,PS20,PS3,PS27]. Certain studies indicated that refactorings within a composite are *ordered*. These studies referred to composites as *chains* [PS5], *ordered lists* [PS12,PS22,PS23], and *ordered sets* of refactoring [PS2, PS5, PS17, PS19]. The refactoring order can be influenced by the developer's motivation behind the application of a composite. For instance, applying an *Extract Method* before a *Move Method* can be used to remove a *Feature Envy* smell [25]. The application of these refactorings in the reverse order would not remove such a smell. Other studies like [S04] considered that the refactoring order within a composite does not matter. These studies that do not consider the *order* to composite computation suggest that this approach facilitates the computation of applying composites from software projects that use platforms of version control such as GitHub. In fact, project version data provided by platforms like GitHub are commit-focused and provide little or no data about what happens within a commit. Thus, it is quite hard to precisely characterize the order of refactorings within a commit.

Summary 2: Composites can have nine characteristics, and these characteristics can be related to the application and the structure of a composite, and they can have different manifestations.

2.4.4 Composite Effect on Software Projects (RQ$_3$)

Figure 2.6 presents the answers of RQ$_3$. We have found 13 effect types of a composite (last nodes from root) reported by the literature. Studies reported composites can affect *Internal Quality Attributes*, *External Quality Attributes*, *Design Pattern Introduction*, and *Effort Minimization*. Table 2.7 presents the primary studies that mention each effect. The first column shows the ID of paper according to our artifacts. The second column presents the effect that was cited for the primary studies. The last column indicates the reference number of each study.

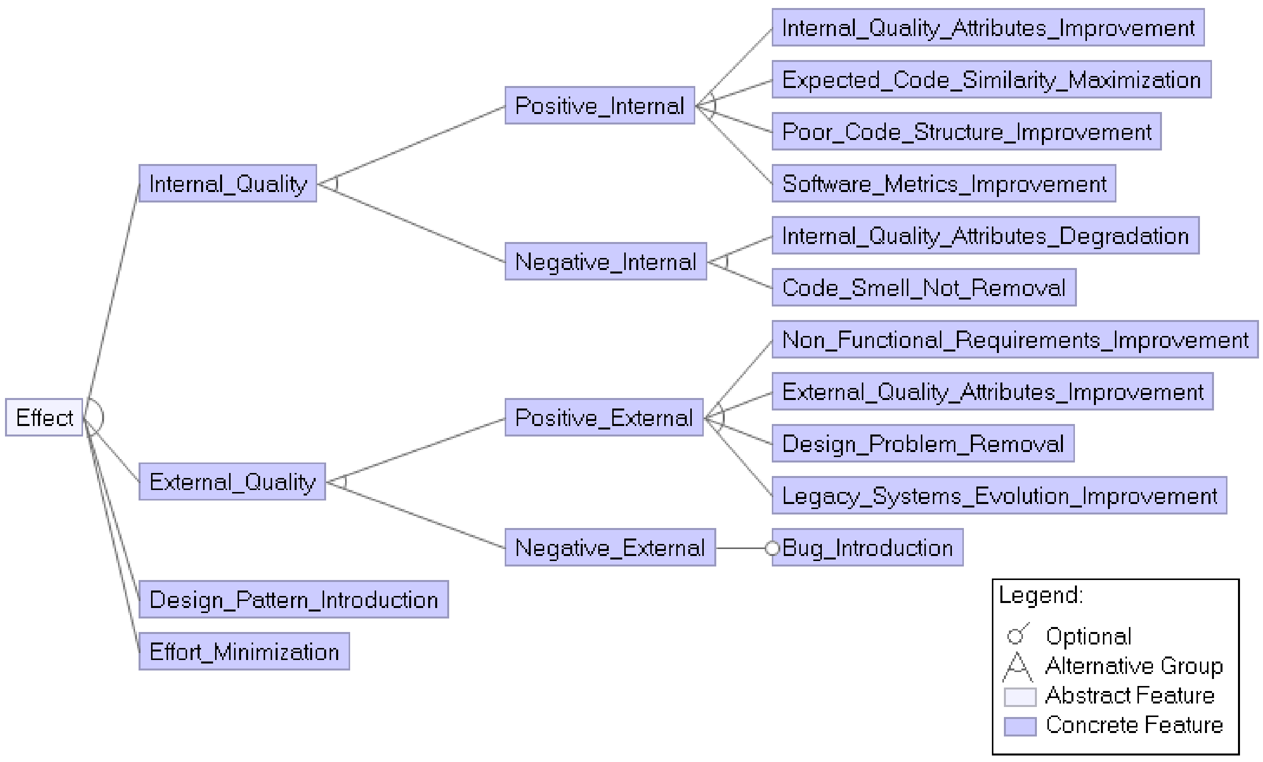

Effect
Internal_Quality
Positive_Internal
Internal_Quality_Attributes_Improvement
Expected_Code_Similarity_Maximization
Poor_Code_Structure_Improvement
Software_Metrics_Improvement
Negative_Internal
Internal_Quality_Attributes_Degradation
Code_Smell_Not_Removal
External_Quality
Positive_External
Non_Functional_Requirements_Improvement
External_Quality_Attributes_Improvement
Design_Problem_Removal
Legacy_Systems_Evolution_Improvement
Negative_External
Bug_Introduction
Design_Pattern_Introduction
Effort_Minimization
Legend:
Optional
Alternative Group
Abstract Feature
Concrete Feature

Figure 2.6: Effect Types of a Composite

Table 2.7: List of Papers that mention Effect

Paper ID	**Effect**	**Reference**
137	Bug Introduction	[S137]
2,9,247,250,259,274	Code Smell Not Removal	[S02, S9, S247, S250, S259, S274]
2,8,9,23,30,31,32,33,42,45, 53,54,62,66,67,75,79,86,90,93,95, 110,115,118,122, 127,129,150,247, 248,258,259,265,266, 273,276	Code Smell Removal	[S02, S8, S9, S23, S30–S33, S42, S45, S53, S54, S62, S66, S67, S75, S79, S86, S90, S93, S95, S110, S115, S118, S122, S127, S129, S150, S247, S248, S258, S259, S265, S266, S273, S276]
122,158	Design Pattern Introduction	[S122, S158]
122	Design Problem Removal	[S122]
8,33,49,118,138,146,248,258	Effort Minimization	[S8, S33, S49, S118, S138, S146, S248, S258]
8, 38, 248	Expected Code Similarity Maximization	[S8, S38, S145, S248]
8,11,23,30,32,39,42,49,50,52,62, 69,75,76,96,115, 117,122,125,126, 129,135,155,156,261,273,274	External Quality Attributes Improvement	[S8, S11, S22, S23, S30, S32, S39, S42, S49, S50, S52, S62, S69, S75, S76, S96, S115, S117, S122, S125, S126, S129, S135, S155, S156, S261, S273, S274]
9	Internal Quality Degradation	[S9]
15,34	Internal Quality Improvement	[S15, S34]
8	Internal Quality Attributes Degradation	[S8]
8,10,25,29,32,37,42,43,49,52,54, 55,57,60,66,67,68,71,74,75,76, 81,91,93,99,103,146,147,156,158, 261,273,274,276	Internal Quality Attributes Improvement	[S8, S10, S25, S29, S32, S37, S42, S43, S49, S52, S54, S55, S57, S60, S63, S66–S68, S71, S74–S76, S81, S91, S93, S99, S103, S146, S147, S156, S158, S261, S273, S274, S276]
52	Legacy System Evolution Improvement	[S52]
96, 133,275	Non-Functional Requirements Improvement	[S96, S133, S275]
72,59,123,266	Poor Code Structure Improvement	[S02, S59, S123, S266]
60,68,86,93,99,118,126,151,158,266	External Quality Improvement	[S60, S68, S86, S93, S99, S118, S126, S151, S158, S266]
9,10,11,25,34,37,39,53,62,66,76, 86,99,110,140	Software Metrics Improvement	[S9–S11, S25, S34, S37, S39, S53, S62, S66, S76, S86, S99, S110, S140, S145]
32, 119, 248	Software Quality Degradation	[S32, S119, S248]
8,32,33,45,45,59,131,155,265	Software Quality Improvement	[S8, S32, S33, S45, S45, S59, S131, S155, S265]

For *Internal Quality Attributes*, previous pieces of work mentioned composites can have a positive effect such as the *Internal Quality Attributes Improvement*, *Expected Code Similarity Maximization*, *Poor Code Structure Improvement*, and *Software Metrics Improvement*. *Internal Quality Improvement* regards the possible effects of composites on the internal code structures. Some studies [S8, S10] assumed that applying composites can positively affect the code elements that constitute a project, thereby improving software metrics such as code size and code coupling, which can be indicators of a maintainability improvement [10]. This assumption is quite expected due to the traditional expectation that single code refactoring can affect the indicators of code maintainability as well. Such improvement can be assessed by means of *code metrics* [PS22, S04, PS23, PS7, PS16] or special indicators of poor code structures, like *code smells* [PS12, PS3, PS16, PS26, PS27]. Certain studies assumed composites can remove code smells, even without empirical evidence [PS12,PS3,PS16,PS27]. In fact, an existing study observed that single refactorings [8] can remove certain types of code smells. However, other studies [S02, PS11] reported that most of the refactorings are applied in composites. Thus, the effect that was observed for single refactorings can be different from the effect observed from the composite perspective. Other pieces of work mentioned composites can have a negative effect such as the *Internal Quality Attributes Degradation*, and *Code Smell Not Removal* [S02, S247]. *Internal Quality Degradation* regards how composites may negatively affect the internal code structure of software projects. Previous studies [PS12, PS9] discussed that composites may negatively affect certain parts of the code structure, especially in terms of basic code metrics. *Code smell not removal* regards the negative effect of composites on the code structure of software projects. In fact, one study [PS12] warned that an undisciplined application of composite refactoring might degrade the internal code structure of a project, especially by introducing code smells.

On *External Quality Attributes*, studies also mentioned composites can affect positively these attributes [PS20, S27]. These effects can have a *Non-Functional Requirements Improvement*, an *External Quality Attributes Improvement*, a *Design Problem Removal* or a *Legacy System Evolution Improvement. External Quality Improvement* regards the effect that composites may have on external attributes of software projects, such as evolvability. We found out a considerable number of studies that assumed that composites can leverage the project *evolvability*, for example, composites can help a feature addition or a migration to new technology [PS12, S04, PS26]; and *correctness* (e.g., composites can help the bug removal or the correction of syntax errors) [PS22,PS23]. Refactorings that constitute a composite directly affect the code structure and, therefore, it is expected an internal quality improvement of software projects. However, studies assumed that each refactoring can also affect external quality attributes such as correctness through refactorings that aim to prevent code elements from becoming buggy in the future [S27, S137]. Thus, it is reasonable that some authors assumed an external effect of composites. Studies suggest composites can introduce bugs (*Bug Introduction*), but these studies do not report empirical evidence about it [S27, S137].

Some studies [S122,S158] indicated that composites can help to introduce Design Patterns because some patterns, e.g., Abstract Factory and Adapter, need to restructure many classes involving the motion of methods, and code extractions, to facilitate the separation of concerns and creation of hierarchies. However, these studies are limited to explore which composites are applied frequently to introduce Design Patterns and to recommend composites for Design Pattern introduction.

Effort Minimization. Previous pieces of work [S115, S118] expected that composites can minimize the effort of developers in terms of time and costs of development. By definition of refactoring, it is expected that refactoring can minimize the effort of development, but it depends on many factors, for example, the complexity and number of refactorings that were applied, the number of code elements that needed to be modified by refactorings. Other studies [S04, S46] also indicated that composites can increase the effort of development, forcing developers to create separated branches only for refactoring. Interesting future research about it could be to investigate what are the scenarios of development that composites can minimize or increase the effort, and what composite types can be recommended to minimize the effort of development according to each scenario. On the *Legacy System Evolution Improvement*, some studies suggested that composites can help the evolution of legacy systems, this evolution can be a migration to a new technology or an adaption for a new system environment [S52].

Summary 3: Studies suggested that composites can affect the internal and external software quality positively and negatively, but they also can help to introduce Design Patterns. However, existing studies are limited to understand these effect types empirically because they did not investigate these effect types in-depth.

2.4.5 Conflicting Composite Characteristics and Types of Effect

We found out some cases of conflict among studies with respect to the characteristics and effect types of a composite. These conflicts are interesting to reveal opportunities for future studies to improve the current knowledge about composites. We discuss below each of these cases. The conflicts regarding characteristics and effects are referred to, respectively, as C-n and E-n, where n is the conflict number.

Conflict C-1: Scope of a composite: one code element or several code elements? The current knowledge about what code elements are affected by a composite refactoring is ultimately conflicting. Some studies considered that the refactorings constituting a composite should be constrained to the *same code element*, e.g., a method or a class [PS12, PS24, PS3, PS7]. Conversely, other studies assumed that a composite refactoring can affect *multiple classes* [PS22, S04]. Each study adopted a different manifestation of the scope characteristic because it could facilitate their study goals. However, these studies did not explain why they did not use other manifestations. There is a need for a proper understanding of what boundaries determine the code elements affected by composite refactoring. Otherwise, it is hard to elaborate or choose a heuristic to identify existing composites in a software project.

Conflict C-2: Who is responsible for applying a composite refactoring? There is also a conflict on the current knowledge about who is usually responsible for applying the refactorings that constitute a composite refactoring. Some studies assumed that a composite refactoring is usually applied by a *single developer* [PS22, PS11, PS17]. However, another study assumed that a composite refactoring can be started by one developer and complemented by other developers [S04]. We found out that each study used different manifestations of the developer characteristic to facilitate the investigations of their study goals. For instance, Kim et al. [S04] investigated the refactoring practices of a single *team of developers* in a specific software company. Murphy-Hill et al. [PS11] considered composites that were applied in a specific time span using the Eclipse IDE[6]. This work only took into consideration the composites applied by a single developer. Bibiano et al. [S02] investigated composites as a set of interrelated refactorings performed by a single developer. In summary, there is a lack of consensus about how many developers are responsible for applying composite refactoring. This lack of consensus makes it difficult to identify composites in existing projects.

Conflict C-3: How long is a composite applied? We also identified a conflict regarding the *time* spent by developers to apply a composite refactoring. For convenience, one study assumed that two subsequent refactorings that form a composite refactoring should be applied in up to 60 seconds one from another [PS11]. One might question whether this conservative threshold is reasonable. For instance, this study unlikely captured composites that last for more than an hour, as a developer often has to intertwine refactoring edits with his other routine activities. Conversely, other studies *did not constrain the time spent* to apply these refactorings, thereby making it possible to compute composite refactorings that last *days*, *weeks*, and even *months* to be completed [PS22, PS10, S04, PS6]. In these cases, the lack of a threshold may lead one to the consideration of non-cohesive composites. Examples of it are cases where two groups of interrelated refactorings, because they affected

[6]IDE - Integrated Development Environment

the same program element. However, the time separating the two commits is higher than one year, once each group of refactorings took place. If there is no time-related constraint, one would consider these two groups of refactorings as a composite. In any case, existing studies lack empirical evidence about how long should the application of a composite refactoring last in practice. Thus, there is insufficient information to constrain the set of refactorings that constitute a composite refactoring. Consequently, it makes it difficult to identify "cohesive" composites applied to existing projects.

Conflict C-4: Is a composite constituted by a single or various refactoring types? We observed some conflicts in the way previous studies consider the *variety* of refactoring types that may constitute a composite refactoring. Some papers suggested that *multiple refactoring types* can co-occur into a composite refactoring [PS24, PS16, PS17]. Conversely, another study assumed that the refactorings that constitute a composite refactoring should have a *single type* [PS26]. This conflict can affect the analysis of the effect of composites in a program. If a heuristic is used in an empirical study to detect only composites with a *single refactoring type*, the conclusions will be constrained to only these composite types. Composites of this kind might be not effective to remove various types of structural problems in the source code.

We also found out cases of conflict among studies with respect to the expected effect of composites on software projects. They were labeled as E-n, where n is the conflict number. We discuss these cases as follows.

Conflict E-1: Do composites make a software project easier to maintain? Some studies assume that composites can effectively improve the maintainability of software projects [PS12, S04, PS26]. They mainly expect an *internal code structure improvement* through the *removal* of code smells [PS12]. On the other hand, one study [S02] indicates composites frequently can either *introduce* or *end up not removing* code smells. In order to better support developers in applying composites that are effective in making the code easier to maintain, future work should empirically investigate these positive and negative types of composite effects in-depth. In fact, the study of Meananeatra [PS12] suggested that certain forms of composites can remove code smells, thereby improving the program maintainability. However, Sousa et al. [S247] reported that composites often have a negative effect on code smell removal. Thus, the current knowledge is limited and conflicting.

Conflict E-2: Are composites more likely to improve internal code structures rather than degrade these structures? We found out studies, such as [S04, PS16, PS26], that point out composites as means for *improving the internal quality of software projects.* Conversely, a particular study [PS9] discussed that undisciplined composite application can lead to the *degradation of code structures.* Similarly to Conflict E-1, we expect that future work can address this particular issue in order to draw more assertive conclusions about the composite effect on the internal quality of software projects.

Conflict E-3: What is the actual composite effect at the architectural level? Some studies [PS7, PS13, PS20] assumed that composites likely affect *negatively the current architecture of software projects.* These studies suggested that composites can *increase the coupling* between modules of software architecture. On the other hand, a particular paper [PS14] proposed that composites can be applied *to improve the software architecture* by improving the cohesion and the coupling of the software's components. Thus, there is a lack of consensus on how composites affect the architecture of software projects.

2.5 Related Work

The topic of software refactoring has been receiving attention from academia and industry for the past two decades. Due to this wide interest, we can find many surveys, systematic reviews, and mapping studies taking into account the topic of refactoring from different points of view. For instance, refactoring of sequential code to parallel computing [264], refactoring of system variants to systematize software reuse [265], and UML model refactoring [266, 267]. In addition to different topics, there are also secondary studies focusing different characteristics, such as industrial perspectives of refactoring [280], refactoring tools [268], software refactoring in the context of modern code review [269], refactoring in continuous integration [270], methods to automate the software refactoring process [271], and search-based techniques to support software refactoring [272, 273].

We can still find more secondary studies since from 2004 investigating the advances in the broad research topic of refactoring [274–279]. The majority of these and the aforementioned studies has appeared in the past five years. Taking into account the existence of many secondary studies, Lacerda et al. [251] presented a tertiary systematic review describing challenges and observations of refactoring and code smells. Among all these studies, next we described those more related to our work, pointing out the main differences and the gaps filled by our contribution.

The seminal work of Opdyke defined 23 refactoring types and presented only three examples of composite refactoring, composed of those primitive refactorings [263]. These composite refatorings usually deal with macro or architectural design changes [251]. Mens and Tourwe highlighted it is useful to determine a sequence in which refactorings have to be applied and which refactorings are mutually independent [274]. Two secondary studies only touched the topic of composite refactorings by mentioning the OBEY, an Eclipse plug-in that executes composites plan on Java source code, analyzing their impact on cohesion and coupling quality attributes [277, 279]. Another tool is Refactoring Scheduler that identifies refactoring sequences to maximize software quality and remove software clones [279].

Two studies mapped the literature on search-based software refactoring [272, 273]. They mentioned how genetic algorithms, genetic programming, and multi-objective algorithms are used to find and optimize sequences of refactorings to defects or bad smells. Curiously, some secondary studies on software refactoring simple discard from their analysis the pieces of work based on refactoring sequences [267, 277]. On the other hand, a paper from Sharma et al. described challenges to and solutions for refactoring adoption in industrial settings [280]. On the topic of composite refactorings, they pointed out that tools mostly rely on supporting primitive refactorings. These authors also indicated that IDEs and their extensions need to support non-trivial composite refactorings to encourage refactoring adoption in practice.

In summary, only few systematic mappings, literature reviews, or surveys explicitly described the nature of composite refactorings. Furthermore, when they mentioned composites, they only briefly provide information or rely on very specific characteristics. Our study provides a comprehensive analysis of composite refactorings, also describing and organizing the existing literature. We performed an in-depth investigation of composites from the perspectives of representation models, their characteristics, and their possible effects. We provided more than a summary of current state-of-art. Our study contributes with a conceptual framework of composite refactoring derived from the literature knowledge.

Our results can help practitioners and researchers on further developing the topic by deciding what representation model is appropriate according to their composite refactoring approaches. Certain representations are more advantageous than others, depending on the goal of the composite refactoring approach being formulated. For instance, certain types of representations, albeit more complex, are more useful in situations where the order of each refactoring in a composite is important.

Our conceptual framework reveals characteristics and manifestations of composite refactoring. These characteristics and manifestations can be used on the design of tools, such as recommendation systems for composite refactoring. Examples of characteristics that may be used in the decision making of tool designs are *time* and *variety*. For instance, tools can recommend a composite formed by *one refactoring type* (e.g., repetitive renames) or *many refactoring types* (e.g. extracting and moving methods). These recommendations can be offered before each commit, pull request or at the end of the day. We also revealed the possible effects of composites and how existing studies have conflicting views about such effects. These conflicts can further motivate researchers to replicate studies by examining these contractions in more depth.

2.6 Threats to Validity

We discuss threats to the study validity [70] as follows.

Construct Validity. We carefully defined our study protocol prior to the conduction of the systematic mapping. We defined the study goal and research questions according to the Goal Question Metric framework [1]. Thus, we expected to minimize the chances of changing the focus of our study while it was performed and the systematic mapping data were analyzed.

Internal Validity. All data collection procedures were performed by a pair of authors in order to mitigate problems with missing, duplicated, and invalid data. These procedures include running our search string in the web search engines, for instance. Similarly, we paired two authors in order to tabulate the data so that we could easily perform the Grounded Theory (GT) procedures of open and axial coding [13]. By strictly following the GT procedures, we expected to reduce problems with the identification of representation models, composite characteristics, and types of effect from the selected papers through systematic mapping. We did not conduct a quality assessment of the studies found, but to ensure the relevance of our results, we only included papers that were published at venues that are widely recognized for publishing high-quality software engineering research.

Conclusion Validity. We carefully analyzed all tabulated data in order to (i) aggregate the representation models, composite characteristics, and types of effect extracted from the literature in our conceptual framework, and (ii) validate all analyzed data in a pair. Thus, we expected to minimize biases in the identification of characteristics and types of effect, but especially in the characterization of conflicts among previous studies. In this particular case, we promoted meetings to discuss which characteristics and effects are conflicting and deserved explicit consideration.

External Validity. We performed two rounds of pilot search, aiming to improve the search string and the control dataset. We included the Scopus, a search tool largely recommended performing systematic mapping [170], and followed the existing guidelines for systematic mapping [36, 167].

2.7 Conclusion and Future Work

We summarized the current knowledge on composites for (i) identifying representations of composites, (ii) eliciting the characteristics that previous studies consider as constitutive of a composite refactoring, (iii) eliciting the types of composite effect on software projects as assumed by previous studies, and (iv) identifying eventual conflicts in the literature about composite characteristics and types of effect. Our conceptual framework presents seven representations, nine characteristics, and thirty effects of composites. Our results can motivate future studies to investigate in-depth about application of composite refactorings. We aim to add constraints between the characteristics and effects of our conceptual framework.

3
How Does Incomplete Composite Refactoring Affect Internal Quality Attributes?

This chapter contains a paper published in
the Proceedings of the 28th International Conference on Program Comprehension (ICPC) [76].

In the previous chapter (Chapter 2), we conceptualized composite refactoring through our systematic mapping. In our conceptual model, we indicated characteristics and effects of composite refactorings according to the literature. Completeness is one of the interesting characteristics of composite refactorings. Some studies [171, 222] indicate that the notion of completeness can help one to reveal what refactorings are needed "to complete" a composite to achieve a specific structural goal. However, existing studies assume that composites often completely remove code smells even without empirical evidence [28, 37, 43, 44]. On the other hand, other studies indicate that certain composites may be incomplete to eliminate code smells [43, 47]. These contradicting views call for empirical studies about the (in)completeness of composite refactorings.

Aiming to mitigate this literature gap and address our RQ_2 (Section 1.3.1), we performed two empirical studies reported in this chapter and in the next one. In this chapter, specifically, we focused on analyzing incomplete composite refactorings. The next sections are the content of a published paper [76]. Sections 3.1 and 3.2 can have repetitive content, thus these sections can be ignored.

3.1 Introduction

Code refactoring [25] is one of the most popular techniques to improve the internal code structure and, consequently, the comprehensibility of a program [25, 199]. Each single *refactoring* is an instance of a refactoring *type*. Each type determines the changes required to produce an expected enhancement of a certain code structure [25]. Examples of popular refactoring types include Extract Method and Move Method [46, 60]. Like other types, they are expected to contribute to fully remove poor code structures [8, 143], such as *code smells* [197, 198].

However, given its fine-grained nature, a single refactoring rarely suffices to assist developers in achieving their intents, *e.g.* to fully remove a poor code structure [8,143] such as *code smells* [197,198]. The removal of some code smells are considered highly relevant by developers and practitioners. This is the case of *God Class* and *Feature Envy* [35,60,62,73] smells as they have a wide, harmful impact in the program structure; both affect two or more classes.

Notwithstanding, previous studies have reported that single refactorings often do not remove or even introduce these code smells [4,8]. Nevertheless, they provide little or no insight on a wider and more complex phenomenon called *composite refactoring* [12,143,171]. Composite refactoring occurs when two or more interrelated single refactorings are applied on one or multiple code elements [20,143,167,171,226]. This phenomenon happens frequently in software projects [46,143], and each set of interrelated single refactorings is called a *composite*.

Previous studies recommend specific patterns of composites to remove certain code smells [25,143]. For example, Fowler recommends the application of various *Move Methods* together to fully remove a *God Class*. However, empirical studies report that developers often fall short in fully removing those code smells through composites. Indeed, most composites either introduce or not fully remove code smell instances [143]. This can be related to the fact that developers often apply composites alongside other code changes [19, 46], and frequently these composites are applied to perform development activities that do not purely affect the code structure, *e.g* a feature addition. Besides that, developers may not be applying the recommended composites to remove the code smells. The literature suggests that developers fail in removing code smells because some of the recommended refactorings are missing within the composites [8,143]. The lack of one or more refactorings in a composite, to remove a particular smell type, constitutes an *incomplete composite refactoring* (shortly called incomplete composite). It is expected that incomplete composites can gradually improve the internal structure quality, improving also the program comprehension.

The existing refactoring tools offer little help in assisting the completion of incomplete composites by providing the refactorings needed to fully remove remaining code smells [39, 44, 64, 201]. Designing tools for providing such assistance requires proper empirical investigation. This investigation includes characterizing the most frequent types of incomplete composites applied to real programs. It also includes understanding how incomplete composites gradually affect the internal quality attributes when compared to code smells. The internal quality attributes are often used to detect problematic microstructures of source code, which are known to harm program comprehensibility [10,168]. For example, increasing code complexity is highly related to low program comprehensibility. However, previous studies on composites in real projects did not investigate the effect of incomplete composites on internal quality attributes [143,172].

Based on these limitations, this paper presents a quantitative study aimed at addressing the aforementioned literature gap. Our goal is to understand *the most common incomplete composites* and *how incomplete composite refactoring affects internal quality attributes.* We selected five software projects of different domains and targeted 34 popular refactoring types [46,60]. We then collected 353 (47%) incomplete composites for *Feature Envy* removal or *God Class* removal. We then computed the frequency of incomplete composites according to the refactoring types constituting each composite. We evaluated the effect of those incomplete composites on 11 code metrics that are used to capture four internal quality attributes [10]. Hereafter we present our main findings and an overview of possible implications:

1. Composites often affect the structure of two or more classes; most of the incomplete composites with such a wide scope (82%) are composed of multiple refactoring types. This observation contradicts findings from recent studies [143,172], which analyzed a much smaller set of refactoring types than the one considered in our study. Given the type heterogeneity and the wide scope of the aforementioned composites, one could expect they would often have a positive effect on multiple internal attributes, including the cohesion and coupling of the affected classes. However, this scenario was often not the case possibly because such heterogeneous composites are hard to apply properly.

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

3. Most incomplete composites (58%) tend to not change the internal quality attributes on smelly classes. In a way, one could consider this fact problematic. However, this finding reveals that despite the incomplete composites not fully removing code smells, they maintain the structural internal quality of the affected classes. At least, the incomplete nature of composites has possi-

bly not harmed even further the program comprehensibility and other related quality attributes. This observation suggests that certain developers may be keen to maintain the structural quality of their programs through refactoring, even when they do not have the explicit intent of doing so. Thus, they might also be open to receive additional refactoring recommendations to help them improve the program structure even further, while achieving their primary goals. Recommender systems could be designed, then, to assist developers in "completing" their composite refactorings, while also favoring the achievement of their other goals.

3.2 Background

This section presents the main concepts for this study. Program refactoring is the process of performing changes that aim to improve the internal code structure of a program [25]. The literature presents catalogs of *refactoring types* [25], and an example of one refactoring type is *Extract Method*, which is when a part of the source code is extracted from an existing method to a new method.

3.2.1 Composite Refactoring (or Composite)

In the context of this work, a *composite refactoring* (or shortly, a composite) consists of two or more interrelated single refactorings applied by the same developer to one or more code elements. In fact, Bibiano *et.* al. [143] have shown that interrelated single refactorings usually are applied by the same developer [143]. The literature presents some heuristics to identify composites. A recent study proposed a *range-based* heuristic, in order to detect each composite formed by single refactorings that are structurally interrelated [171]. It groups single refactorings that have the following characteristics: i) at least one code element was affected by all refactorings in the composite, and; ii) they were applied by the same software developer, and relies on the fact that composite refactorings often have those two characteristics [143]. This heuristic is different from the *element-based* heuristic proposed by another previous study [143]. The *range-based* heuristic captures the source and target classes to which the single refactorings were applied in the composites. However, the *element-based* heuristic limits its scope to the source class only. The *range-based* heuristic was designed according to developers' practices, as a previous study [172] has shown that composites are often applied to multiple classes.

3.2.2 Incomplete Composite Refactoring: A Smell Removal Perspective

Recommended Composite Refactoring for Code Smell Removal. The literature recommends the application of certain composites to remove specific code smell types [25, 143]. Fowler [25] recommended composites for code smells such as *Feature Envy* and *God Class*. Also, Bibiano *et* al. [143] have presented recommendations of composites for those code smells, observing that Fowler‘s suggestions are not usually applied. They also found that developers often combine the recommended refactoring types with other refactoring types in composites to remove code smells. For example, to remove Feature Envy, developers combined *Extract Method*, *Move Method* and *Move Attribute*.

Incomplete Composite Refactoring for Code Smell Removal. Bibiano *et* al. [143] observed that composites often failed to remove code smells, especially *Feature Envy* and *God Class*. This is expected because these code smells are regularly in classes in which other code changes happen frequently, such as a feature addition [73, 261, 262], thus, making the removal of the code smell more challenging. We call *incomplete composite refactorings* (or shortly *incomplete composites*), composites that contain at least one recommended refactoring type that is used to remove a particular code smell, but failed to remove that code smell after its application. For example, Fowler recommended composites consisting of *Extract Methods* and *Move Methods* to remove *Feature Envy* [25]. In that case, a composite is an incomplete composite if: (i) the composite has at least one refactoring type is recommended to remove Feature Envy [25, 143] (at least one *Extract Method* or one *Move Method*), and (ii) the composite did not remove the *Feature Envy*.

3.2.3 Motivating Example

This section describes an example of an incomplete composite for Feature Envy removal in a real software project. Figure 3.1 presents an incomplete composite that was applied in the commit `66fbd3202a` [187] from the Dubbo project. In this commit, the `ServiceConfig` class has an envious method called `getExportedUrl`. This method calls several times methods of the `AbstractInterfaceConfig` and the `ReferenceConfig` classes. Possibly aiming to solve this, the developer applied a `Move Attribute`, moving the `url` attribute to the `ReferenceConfig` class. Then, the developer moved the envious `getExportedUrl` method to the `AbstractInterfaceConfig` class. However, the `getExportedUrl` method continues envious, because this method has several calls to the `ReferenceConfig` class.

The developer applied a composite refactoring composed of one *Move Attribute*, and one *Move Method*. This composite was applied to a class that has an envious method (the `getExportedUrl` method). However, this composite did not remove the *Feature Envy* code smell completely, because this method continues to have calls to `ReferenceConfig`. Previous studies presented recommendations of composites to remove this code smell [8,25,143].

This composite from the Dubbo project is an *incomplete composite refactoring* for this case of *Feature Envy* removal, because this composite has at least one recommended refactoring type to remove this code smell (the *Move Method* refactoring type by Fowler's recommendation) [25], and yet this composite did not entirely remove the code smell (see Section 3.2.2). The developer could have "completed" this composite applying more *Extract Methods* and *Move Methods* on the `getExportedUrl` method.

Through this example, we observed the existing limitations on the effect of incomplete composites on the internal structure quality. A recent study on the effect of composites was limited on evaluating the effect of this example on the internal structure quality, because they only observed the effect on the code smell removal [143]. This study would also have concluded that this composite does not remove the code smell, however, this incomplete composite has affected some code metrics that capture one or more internal quality attributes, independently if the code smell was not removed. For example, the number of lines of code in the `ServiceConfig` has decreased, improving the coupling and cohesion of this class. The code metrics related to the cohesion and coupling in the `ReferenceConfig` and `ServiceConfig` are not changed significantly. Thus, the incomplete composite remains the internal structure quality of the `ReferenceConfig` and `ServiceConfig` classes. This can be considered a positive result on the effect of this incomplete composite because regardless the non-removal of the *Feature Envy*, the incomplete composite does not worsen the internal quality attributes of the classes.

This example motivates that existing studies offer a limited knowledge on the effect of incomplete composites on internal quality attributes.The existing refactoring tools provide little help in assisting the completion of incomplete composites by providing those refactorings needed to fully remove remaining code smells [39, 44, 64, 201]. Designing tools for providing such assistance still requires further empirical investigation. This investigation includes characterizing the most frequent types of incomplete composites applied to real programs. More importantly, it includes understanding as to how incomplete composites gradually affect the internal quality attributes when compared to code smells. Although there are several works that study code smells [174–176, 178, 195, 196, 202], which are intrinsically based on these internal attributes, they do not address the association of such smells and attributes with incomplete refactoring.

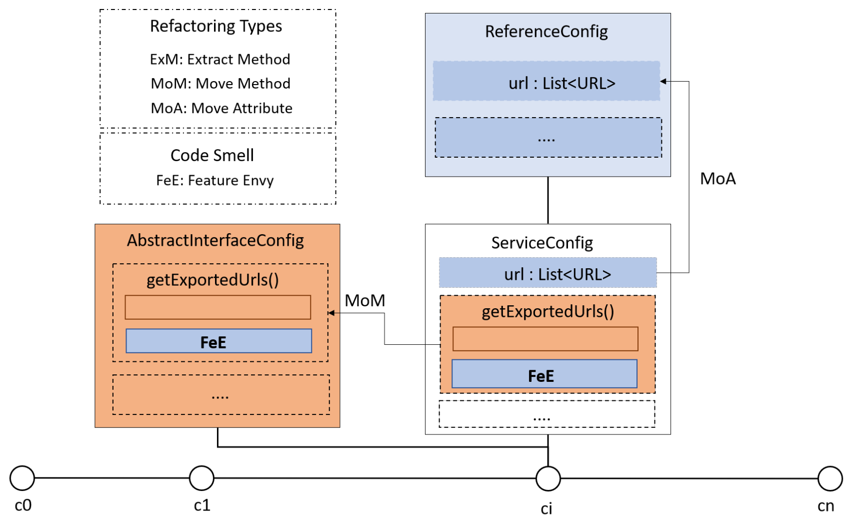


Figure 3.1: Incomplete Composite for Feature Envy Removal

3.3 Study Settings

This section summarizes our empirical study settings as follows. Our companion research website provides the study artifacts for the complementary information [224].

3.3.1 Goal and Research Questions

Our study goal is based on the GQM methodology [1] to *analyze* incomplete composite refactorings applied to software projects by their developers; *for the purpose of* revealing the effect of incomplete composites on internal quality attributes; *with respect to* i) the most common compositions of single refactorings that constitute each incomplete composite instance, and ii) how frequently incomplete composites either improve, do not affect, or worsen each internal quality attribute; *in the context of* the life cycle of five Java open source software projects with active code refactoring practices. We carefully designed two Research Questions (RQs) aimed at achieving our study goal.

RQ$_1$: *What are the most common incomplete composites applied in real programs?* The literature reports that many types of composites can support the removal of the same particular smell type [8, 25, 143]. However, the literature related to the characterization of incomplete composites is scarce (Section 3.2). Thus, one needs to investigate and characterize recurring incomplete composites which, albeit they may often fail to remove a certain smell type, have the potential to remove it if complemented with other single refactorings. By doing so, we can understand how varied and frequent the manifestation of incomplete composites is in practice. Along with this, we also expect to reveal, even if partially, the required support for the completion of these composites, thus enabling the complete removal of the targeted smells. The results of this **RQ** may reveal common practices of composite refactoring that are likely to: (i) hamper full smell removals, or (ii) gradually improve the internal structural quality.

RQ$_2$: *How does incomplete composite refactoring affect internal quality attributes?* Most strategies for detecting code smells rely on the combination of code metrics [16], which capture the current state of various internal quality attributes. Thus, the degradation of these values may imply a degradation in the code's quality itself. Some early studies [4, 8] have already attempted to understand the effect of refactorings on code smells. However, due to the fine-grained code change caused by each single refactoring, it was expected that single refactorings rarely suffice to fully remove code smells [4]. Certain smell types, e.g. *God Class*, are too coarse-grained to be removed with a single refactoring, e.g., *Extract Method*. Surprisingly, a recent study [143] shows that, much like single refactorings, composite refactorings also rarely remove their targeted code smells.

Therefore, recent studies [10,168] shifted from code smells to a more fine-grained perspective: internal quality attributes. These studies concluded that, although single refactorings rarely remove code smells, they can still have a positive effect on the internal quality attributes. For instance, *Extract Method* reduces the class' *complexity*, which is considered improvements. Nevertheless, no previous work has assessed the effect of composite refactorings on internal quality attributes. Thus, the answers to this research question can reveal if incomplete composites gradually improve the internal structure quality as expected, and what incomplete composites usually affect each internal quality attribute. These observations can generate insights for future studies and help designers of existing refactoring tools on improving their approaches for the removal of code smells and the improvement of internal quality attributes, based on real development practices.

3.3.2 Study Steps

Figure 3.2 illustrates our six study steps. We describe below each study step.

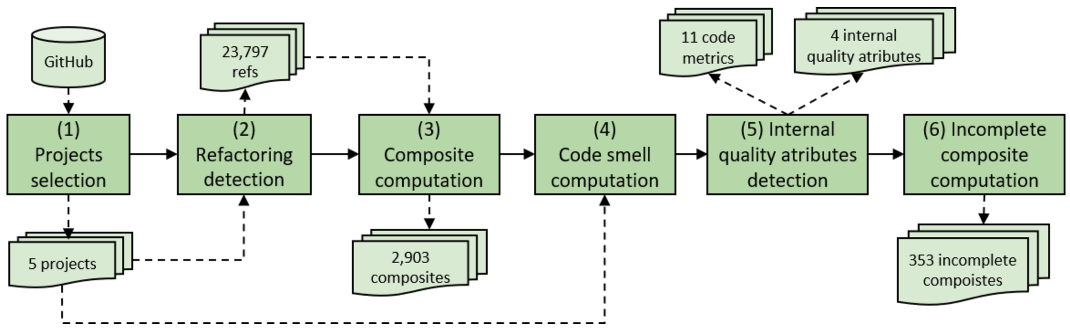

GitHub
23,797 refs
11 code metrics
4 internal quality atributes
(1) Projects selection
(2) Refactoring detection
(3) Composite computation
(4) Code smell computation
(5) Internal quality atributes detection
(6) Incomplete composite computation
5 projects
2,903 composites
353 incomplete compoistes

Figure 3.2: Study Steps

Step 1: *Software Project Selection.* – We relied on previous studies [8, 10, 60, 143, 168, 171, 172] to derive three criteria for selecting software projects for analysis. (i) The software project must be open source and implemented using the Java programming language. Java is one of the most popular languages worldwide [1]. Open source projects were selected to support the study replication. (ii): The software project must use Git as the main version control system. This criterion aimed at supporting the use of state-of-the-art tools for refactoring detection that work on Git projects only. (iii) The software project must have been analyzed by one or more related studies regarding the refactoring [10,172] and code smells [8,143]. Thus, we could select projects that are known to have a considerable amount of refactorings and smell instances.

Step 2: *Single Refactoring Detection.* – We used the RMiner tool for detecting single refactorings applied to each software project [8, 10, 143, 168] and this tool is available for the study replication. A recent study indicates that this tool presents a very high precision (98%) with a recall of (87%) [173]. We investigated 34 of the refactoring types detectable by the tool, by prioritizing the refactoring types related to our study scope. The complete list of the investigated refactoring types is available on our website.

Step 3: *Composite Refactoring Computation.* – We collected the composites using the *range-based* heuristic discussed in Section 3.2. This heuristic allowed us to analyze the composite refactoring effect encompassing from the source class to the target class associated with each single refactoring. Thus, we could identify those cases in which a smell instance was simply moved from one class to another rather than actually removed from the source code, and

[1]https://www.tiobe.com/tiobe-index/

how the internal quality attributes are affected among these classes. Besides, this heuristic is more conservative when capturing single refactorings that were applied on code elements that have interrelated code structures [171]. Thus, the composites detected by the *range-based* heuristic are more likely to encompass composites applied to remove a code smell that involved multiple classes. Therefore, this decision partially reduces the threat that composites may not have been applied with the intent of removing code smells.

Step 4: *Code Smell Detection.* – We selected two code smell types: *Feature Envy* and *God Class* (as described in Table 3.2). These types were selected because the recommended composite refactorings for their removal are already known [25,143]. They are also code smells that involve multiple classes, and can be related to various internal quality attributes such as cohesion and coupling. Thus, the incomplete composites applied on classes that have these smells (smelly classes) may have an effect on these internal quality attributes. Besides that, the *range-based* heuristic motivated us to investigate the effect of incomplete composites on code smells that also usually involve multiple classes. Moreover, previous studies also investigate the effect of refactoring on these types of code smells [8, 143]. In terms of the tool used to detect these code smells, we selected the Organic tool [174–176], which uses strategies based on software metrics to collect the smells. This tool was selected due to its detection strategies, that use the code metrics we analyzed for evaluating the effect of incomplete composites. In addition, these detection strategies were already evaluated by previous studies [8,17,48].

Step 5: *Internal Quality Attribute Computation.* – Table 3.1 presents the 11 code metrics [10,168] that were investigated in this study. The columns present, respectively, the internal quality attributes related to each metric, the code metrics are collected, and their descriptions. These code metrics were selected due to them having been already evaluated for another study [10] for analyzing the effects of single refactorings on internal quality attributes. Thus, these code metrics can reveal the effect of incomplete composites on these internal quality attributes for the classes in which these incomplete composites were applied, because these code metrics are of class-level scope. We chose to perform the analysis in a class-level scope due to a recent study about the effect of composites on code smells presenting that composites are often applied at class level [143]. We then aimed to analyze the effect of composites on internal quality attributes for each class in which a composite was applied. We used an automated tool called SciTools Understand[2] to collect these code metrics, as this tool also is used by other studies about refactoring and internal quality

[2]https://scitools.com/

attributes [8, 10].

Table 3.1: Code Metrics by Internal Quality Attributes

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

Step 6: *Incomplete Composite Computation.* – We investigated the incomplete composites for the *Feature Envy* removal and *God Class* removal. For this study, a composite was considered incomplete according to the following criteria: (i) composites that have at least one refactoring type that is recommended to remove a *Feature Envy* or a *God Class* [25, 143], and (ii) composites that did not remove a *Feature Envy* and a *God Class*. We have filtered the incomplete composites through composites that have at least one *Extract Method* or one *Move Method* (refactoring types recommended to remove these code smells). These composites are the candidates of incomplete composite. Candidate is a composite that has at least one Extract Method or Move Method. We then elaborated Table 3.2 that presents the recommended composites for *Feature Envy* and *God Class* removal according to the literature, the code smell type, their description, and the incomplete composites for each recommended composites.

Table 3.2: Incomplete Composites for Feature Envy and God Class Removal

Recommended Composite	**Code Smell**	**Description**	**Incomplete Composite**

Extract Methods and Move Methods [143] [25]	Feature Envy	A method more interested in other class(es)	Extract Method{n}
			Move Method{n}
			Extract Method{n}, Move Method{n}

Move Methods [25]	God Class	A class that implements too many software features	Move Methods{n}

3.4 Dataset Overview

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

Table 3.3: General Data Analyzed in this Study

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

3.4.1 Incomplete Composite Dataset

Our dataset provides 23,797 single refactorings, and 2,903 composite refactorings collected from five software projects. Table 3.3 summarizes our data on these software projects. The columns show the software project's name, followed by the number of commits, classes, candidates of incomplete composite and incomplete composites for each software project. Notably, these projects present diversity about domains, the number of commits, and the number of classes. It is relevant since it allows for an investigation of the incomplete composite practices applied to software projects with different sizes and domains. We then found 747 candidates of incomplete composites, of which 353 (47%) are incomplete composites. Our data shows that 276 (78.19%) incomplete composites were applied to classes that have at least one *Feature Envy*, while 81 (22.95%) incomplete composites were used on *God Classes*. Note that, an incomplete composite can have been applied to a class that is a *God Class* and also it has *Feature Envies*.

3.4.2 Dataset Validation

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

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

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

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

On the intents of developers during the application of the composites, we concluded through commit messages [167] and code changes, that developers applied changes in which: 7 (41%) out of 17 were applied with the intent of refactoring only; 3 (18%) composites had the intents of a feature addition and refactoring; 6 (35%) were applied for a feature addition only, and; 1 (6.5%) was applied for a bug fix only. We observed that 10 (59%) out of 17 incomplete composites were applied in commits in which developers explicitly mentioned the intent of refactoring. This suggests that these composites may have been applied to remove a code smell, and they were incomplete to remove them. These results also allowed us to measure that a significant percentage of the composites in our data set might truly be incomplete composites.

Second validation. In the second step of the validation, we aimed to ask developers related to the implementation and application of the incomplete composites. At first, we submitted three pull requests to validate if the composites are incomplete composites for the Feature Envy removal [188–190]. They were submitted one month after that the incomplete composites were applied. It would be easier for developers to remember which and why the incomplete composites were applied. Currently, one pull request was accepted, while the two other pull requests are open. The accepted pull request improved the code structure by removing an instance of *Feature Envy* from the Dubbo project. The developer answered that our composite recommendation caused the code to become clearer, the developer told: *"Hi, thanks. I think this patch makes the code cleaner."*

3.5 Common Incomplete Composites

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

3.5.1 Procedures

We collected the frequency of each incomplete composite for *Feature Envy* and for *God Class* removal. We counted the incomplete composites according to their compositions, though we did not consider the order of the single refactorings in each composite, since a recent study observed that incomplete composites often are applied in the single commit [143] and, in the context of a single commit, it is not possible to know the order of the single refactorings. We then created a ranking for the frequency of each composition of incomplete composites for each project, regardless of the order of single refactorings in composites.

3.5.2 Results

Most of the incomplete composites were common to all analyzed projects. Thus, we created a ranking of the incomplete composites for all software projects. Table 3.4 presents a ranking of the 5 most common incomplete composites across projects. The first column indicates the position of the incomplete composite in the frequency-based ranking. The second column then presents the single refactorings that compose each incomplete composite. The last column presents the frequency of each incomplete composite.

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

Grouping of Incomplete Composites. Table 3.5 presents the groups of incomplete composites across the software projects. The first group, G_1, is composed by composites that contain at least one *Extract Method* and one single *Rename* refactoring: *Rename Parameter*, *Rename Variable*, *Rename Attribute*, *Rename Method* or *Rename Class*. G_2, the second group, is composed by composites that contained *Extract Methods* and unusual refactoring types. G_3 is composed by composites that contained only *Extract Methods*, as previously discussed. G_4 has composites that contained all refactoring types except for *Extract Methods* and *Move Methods*. G_5 is composed by incomplete composites that have at last one *Extract Method* and one *Move Method*. G_6 consists of incomplete composites that have at least one *Move Method*, but do not have *Extract Methods*. Finally, G_7 has incomplete composites that are composed by only *Move Methods*.

Considering the incomplete composites in all groups, our first finding is that 291 (82.44%) out of 353 incomplete composites have more than one refactoring type. This result is different from an existing study about composites since this previous study has reported that incomplete composites are often composed of a single refactoring type [143], and another study only investigated refactoring types of the method-scope [172]. Our results are different because they analyzed a much smaller set of refactoring types than the one considered in our study. Given the type heterogeneity and the wide scope of the aforementioned composites, one could expect they would often have a positive effect on multiple internal attributes, including the cohesion and coupling of the affected classes

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

Analysis of the Groups of Incomplete Composites. We observed that the composites in G_1, *Extract Methods* and *Renames*, were the ones applied most often by developers in composites, not in isolation as reported by previous studies [8, 60]. Also, this group was often applied to smelly classes with at least one *Feature Envy* or a *God Class*, but these code smells were not removed by these refactorings. This is an interesting result because it presented that, on smelly classes, developers often extract methods and apply several renames in composites, potentially to improve code comprehension in these classes to some extent, but not fully remove code smells. In groups G_1 and G_3, it is expected that developers were to improve the cohesion of the class, since developers are separating the code in different methods, regardless of the removal of *Feature Envy* or *God Class*.

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

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

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

Even though *Move Method* is a common refactoring type [60], G_7 was not often applied in incomplete composites, representing less than 2% of the incomplete composites. Developers have mostly applied *Move Methods* with other refactoring types. Thus the existing recommendations composed of using only *Move Methods* to remove *God Classes* are not applied in practice. With this result, we suggested that future studies might recommend composites that have more than one refactoring type for the removal of this smell. On the context of internal quality attributes, it is expected for these *Move Methods* to improve the cohesion and coupling, since they are moving methods and decreasing the dependency between the classes.

3.6 Effect of Incomplete Composites

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

Table 3.4: Five Most Common Incomplete Composites

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

Table 3.5: Groups of Incomplete Composites Across Projects

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

3.6.1 Procedures

We classified the effect of incomplete composites on the internal quality attributes as (i) positive, (ii) neutral, and (iii) negative. This classification also appears in previous studies [10, 168, 191, 192] as a comprehensive mechanism to capture the overall refactoring effect on internal quality attributes. This classification relies on three premises. First, each code metric (associated with a particular attribute) can either increase, be unaffected, or decrease after the refactoring application. Second, certain code metrics improve when their values decrease (e.g., CBO), while others improve when their values increase (e.g. TCC). Third, an internal quality attribute improves when the code metrics that capture this attribute improve as well; the attribute worsens when the corresponding metrics worsen. Similarly, we consider that: (i) an incomplete composite has a positive effect when at least one code metric that captures an internal quality attribute has improved; (ii) a neutral effect when none of the code metrics that capture the attribute have changed, and; (iii) a negative effect occurs in the other remaining cases.

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

1. Collecting metric thresholds from significant time periods

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

These "significant changes" were defined as changes that modified *over 25% of a metric's value* in *two or more internal quality attributes*, for higher or lower, except for Size (due to the tendency of Size changing frequently with code changes [46,177]). These thresholds were defined based on quartiles, with a metric with values within the 25% smallest values (Q_1) being considered *good*, a metric between the 25% smallest and 25% largest values (Q_2 and Q_3) being considered *average*, and a metric within the 25% largest values (Q_4) being considered *problematic*, except for the CLOC metric, of which definitions are inverted, due to its nature of higher values meaning an improvement [10].

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

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

3.6.2 Results

Tables 3.6 and 3.7 display a summarized comparison of the before-after states of the classes in the project, by presenting, for each internal quality attribute (Cohesion, Coupling, Complexity, Size), the % of individual metrics changed for the better (i.e., had their values reduced, except for CLOC), that changed for the worse, or that did not change for each composite.

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

Thus, these analysis' results can be summarized as follows: in a general sense, the majority (58%) of changes tended to keep the metric within the same threshold as it was before the composite. Most changes that did impact the metrics caused an increase in their values, which, in most cases, causes a negative impact in the resulting source code. Third, most changes that did impact the metrics mostly impacted only a single metric at a time. Fourth, the majority of times a positive change happened in the code, it was because of an increase in the amount of CLOC (Comment Lines of Code). This means that, while the quality might have improved, the actual smells were either not fixed or even subtly ameliorated by the composites; Thus, this can be summarized in the following finding, which corroborates with a similar one found by a previous work [8] for single refactorings:

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

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

However, by looking at the intensity of these changes, it is possible to see that 70% did not significantly increase or decrease the quality of the code (>25% change in the measurements), while a small, but still significant, set of incomplete composites significantly increased the internal quality attributes' measurements (22%), which in most cases, indicates a decrease in the code quality, and a very small amount of incomplete composites actually significantly decreased these metrics (8%). This means that, even if some of the composites tended to modify the values of the internal quality attributes' metrics, they did so in small increments. This confirms the result discussed in the Section 3.5.2, which shows that incomplete composites have often not improved the program comprehension significantly. We also observed that the majority of those that do make significant changes tend to worsen these attributes. But, as a major finding, we concluded that:

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

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

We then analyzed the effects of incomplete composites in terms of the groups presented in Section 3.5. For that purpose, we chose the 4 most common groups that contain refactorings recommended by the literature to remove *Feature Envy* and *God Class*. Therefore, groups G1, G3, G5 and G7 were chosen as they only contain *Extract Methods*, *Move Methods* and *Renames* (renaming is not necessarily recommended to remove these code smells, but is recommended when a method is extracted). This was done in an attempt to mitigate the threat of analyzing composites that were not intentionally applied to remove those smells. In the composites pertaining to those groups, developers only applied refactoring types that are recommended to remove them, thus, reducing the likelihood of the composite not being applied for that purpose.

Therefore, Figure 3.3 presents the effects of these groups on the four internal quality attributes. We can observe that, once again, size-related metrics are the ones that change the most with incomplete composites, followed by complexity metrics. It is also notable that: (i) composites in G_7 (only *Move Methods*) often fail to improve code metrics; (ii) no composite in G_3 (only *Extract Methods*) changed coupling-related metrics and; (iii)

G_5 (*Extract and Move Method*) had the most overall improvement in their composites. This is in accordance to the discussion presented in Section 3.5.2, in which it was speculated that several extractions and renames (the group G_1) in composites should have reduced the code complexity, improving the program comprehension. Besides that, the positive results of the G_3 group then confirmed that when developers apply *Extract Methods* and *Move Methods* in composites, they often improve the most internal quality attributes, and thus the program comprehension, regardless of the presence of smelly classes.

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

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

This strengthens the conclusion that, even though incomplete composites aim to improve the internal structure quality of certain code elements, they do not bring about significant changes to the smelly class' state, by mostly keeping it in the same state as it was before the composite – though they do mostly prevent quality decay from other non-refactoring changes, much like what was described in Section 3.2.3. However, a non-insignificant percentage of incomplete composites actually worsens the affected class's problems, which could be solved by the completion of the used composite.

3.7 Threats to Validity

Construct and Internal Validity: We reused criteria for selecting *software projects* from previous studies [8, 10, 60, 143, 172]. We aimed at preventing a biased project selection that could favor our study results. We

Table 3.6: % of Changes which Significantly improved Each Metric for Cohesion and Coupling, per Project

Project	**# Composites**	**Cohesion: metric**			**Coupling: metric**		
		Positive	Neutral	Negative	Positive	Neutral	Negative
couchbase-java-core	10	10%	90%	0%	0%	100%	0%
dubbo	122	5%	90%	5%	4%	94%	2%
fresco	50	10%	88%	2%	2%	98%	0%
jgit	178	6%	92%	2%	4%	93%	3%
okhttp	56	7%	90%	3%	4%	94%	2%

Table 3.7: % of Changes which Significantly improved Each Metric for Complexity and Size, per Project

Project	**# Composites**	**Complexity: metric**			**Size: metric**		
		Positive	Neutral	Negative	Positive	Neutral	Negative
couchbase-java-core	10	2%	96%	2%	4%	96%	0%
dubbo	122	6%	90%	4%	4%	92%	4%
fresco	50	5%	91%	4%	2%	97%	1%
jgit	178	3%	93%	4%	6%	92%	2%
okhttp	56	2%	94%	4%	1%	96%	3%

used RMiner [173] to perform *single refactoring detection*, since it has a high accuracy. Similarly to previous studies [8, 10, 143], we performed *code metric computation* via the SciTools Understand tool that computes class-level code metrics that capture the four internal quality attributes analyzed in this study (Table 3.1). Inspired by the literature [8,48], we used the Organic tool to detect *code smell instances*. The smell detection strategies used by this tool have a high accuracy: 72% precision and 81% recall in average [4]. We validated the associated smell instances of *Feature Envy* and *God Class* (Section 3.4.2). By doing that, we confirmed the tool's accuracy.

We reused an heuristic from the literature [171] for detecting *composite refactorings*. By reusing this heuristic, we could prevent manual biases while supporting large-scale composite computation. We could also analyze the effect on internal quality attributes in a wide scope ranging from the source class to the target class of each refactoring. We carefully designed an heuristic for characterizing those incomplete composites [171, 178]. The definition of incomplete composite is considerably subjective, once it depended on our body of knowledge on what composites target a particular smell type. Although there is such subjectivity, we strongly relied on Fowler's refactoring catalog [25] and empirical evidence derived from recent studies, e.g. [8, 143]. One author wrote scripts for computing incomplete composites, and two authors double-checked these scripts, thereby reducing the manual bias and errors.

A previous study presented that developers do not necessarily consider a code smell like a problem in the source code [179]. Thus, we can assume that developers often do not have the explicit intent to remove code smell when applying composites to be a threat to this work's soundness. To mitigate

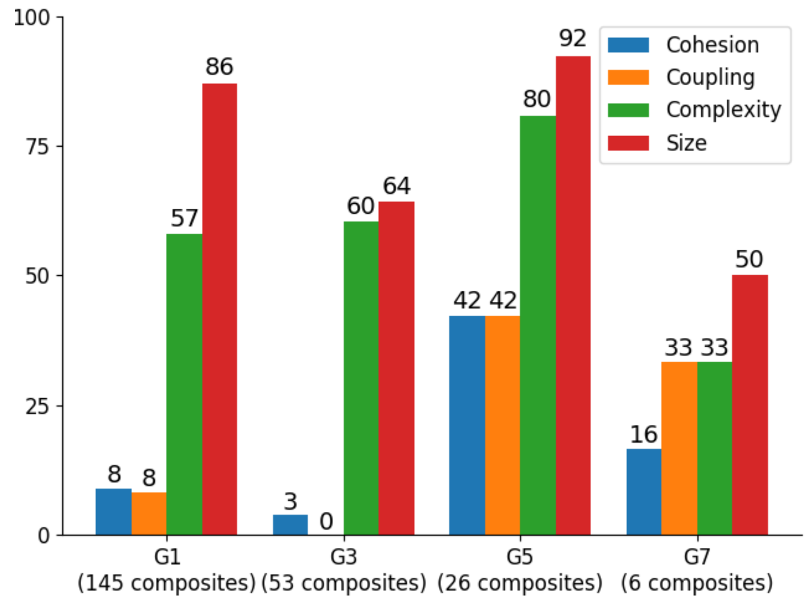


Figure 3.3: % of Positive Changes in Internal Quality Attributes per Incomplete Composites Group

it, then, we submitted pull requests and performed a manual validation aimed to determine the developers' intent to apply composites. The manual validation of the composites and incomplete composites (Section 3.4.2) was performed by developers that are familiar with refactoring. This was important for demonstrating the accuracy of our heuristics and the meaningfulness of our set of incomplete composites, from the perspective of experienced developers [195, 196]. The validated sample of composites is quite small, but we distributed this sample between nine developers for a careful analysis.

Conclusion and External Validity: We reused a three-fold classification of the refactoring effect on internal quality attributes from previous studies [10, 168] (Section 3.6.1). We assumed that this classification could be successfully adapted to the context of incomplete composites. One could criticize our approach that classifies an incomplete composite as positive when it improves at least one associated code metric is too loose. However, a recent study [10] showed no considerable difference between this approach and stricter ones, such as considering the improvement of most metrics as a positive effect. Besides that, a study [10] used this approach to investigate the effect of single refactoring on internal quality attributes. We then reused this approach to compare the results of single refactoring with the results of composites.

We applied traditional techniques of descriptive analysis on the quantitative data [8, 10, 143, 168, 172]. We computed percentages of the most common refactoring combinations that constitute incomplete composites ($\mathbf{RQ_1}$) as well as the effect classification of incomplete composites ($\mathbf{RQ_2}$). These techniques allowed us a detailed comprehension of the incomplete composites' effects in different scenarios. For classifying these effects, we relied on quartiles, as discussed in Section 3.6.1, similarly to related studies [10, 168].

Regarding $\mathbf{RQ_1}$, we were unable to compute the order of the refactorings within a composite. There is empirical evidence that most composites are fully applied in a single commit, so we cannot assure the precedence of one refactoring over another refactoring. This limitation has also affected previous studies [143,171] but they still did not prevent interesting insights of the effects of refactoring on internal quality attributes from being derived. With respect to $\mathbf{RQ_2}$, previous studies [8, 10] show that single refactorings are very often applied alongside other types of code change. The same reasoning applies by extension to composite refactorings.

The scope of our study is quite limited for allowing a wide generalization of our findings and their implications. Although we aimed at a certain diversity in terms of project size and commit history (Table 3.3). Our preference for open source Java projects may support the study's replication, but they may not cover all refactoring practices worldwide. Nevertheless, these projects have been successfully analyzed by related studies [8, 10, 143, 168, 172].

3.8 Related Work

Previous studies have investigated the effect of single refactorings on the software program [8,10,168]. Cedrim *et* al. [8] presented that single refactorings often introduced or did not fully remove poor code structures such as code smells. This study presented that single refactorings often did not remove certain code smell types such as *Feature Envy*, which occurs when a method is more interested in attributes and methods of other class(es) rather than its class [25]. Other studies have investigated the effect of single refactorings on code metrics [4, 10, 168]. These studies presented that single refactorings usually improved code metrics that captured one or more internal quality attributes, e.g. cohesion and coupling. However, these studies were limited to an understanding of the effects of the code refactorings because they investigated single refactorings only, and developers often applied composite refactorings.

The effect of Incomplete Composite Refactoring on Internal Quality Attributes. A recent study suggested that incomplete composites can improve internal quality attributes, regardless of code smell removal [143]. For example, incomplete composites that have *Extract Methods* only. These composites do not remove Feature Envies, but they can increase the code size through the number of lines of code in methods, improving code metrics related to the coupling and cohesion. However, existing studies only present assumptions on the effect of incomplete composites on internal quality attributes, providing limited knowledge about how incomplete composites can affect the internal code structure. Our study is the first study that investigated empirically the effect of incomplete composites on internal quality attributes.

3.9 Conclusion

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

4
Look Ahead! Revealing Complete Composite Refactorings and their Smelliness Effects

This chapter contains a paper published in in
the Proceedings of the 37th International Conference on Software Maintenance and Evolution (ICSME) [77].

In Chapter 3, we investigated the practical application of incomplete composite refactorings on five software projects of different domains and code sizes. We investigated the incompleteness of composites for two code smells, *God Class* and *Feature Envy*, because the recommended composite refactorings for their removal are already documented somewhere [25, 171]. The selection of these smell types was made as they involve multiple classes and can be related to the occurrence of other code smells [200].

Our results revealed that developers often (58%) apply incomplete composites. We observed that most (83%) of incomplete composites are formed by more than one refactoring type. This observation indicates that developers spend time and effort formulating different combinations of refactoring types in composites, but the resulting composites fail in fully removing of code smells. Based on these results, we need to (i) evaluate what refactoring types in composites frequently remove code smells, and (ii) assess whether pre-existing documented recommendations of composites have removed the code smells in the practice [143, 171]. In other words, we need to understand how complete composites are applied by developers in their routine. A better empirical understanding of complete composites can guide us in recommending effective composite refactorings.

In this chapter, we then empirically investigated the nature of complete composites as a means to address our RQ$_2$ (Section 1.3.1). For this study, we expanded our dataset of composite refactorings, increasing the number of software projects and code smells because recent studies found that complete composites are rarely (about 10%) applied [143, 171]. In addition, aiming to extract refactoring recommendations from the practical application of complete composites, we investigated the possible side effects of complete composites in real projects. The exploration of their side effects can provide us the knowledge on how to recommend composite refactorings for beneficial removal of code smells, i.e., composite refactorings that fully remove target code smells, minimizing or removing side effects. The next sections are the content of the published paper [77]. Section 4.2 can have repetitive content; thus, this section can be ignored.

4.1 Introduction

Software companies apply refactoring as a strategy to remove poor code structures such as code smells, ease maintenance, and keep design quality [25, 35]. A refactoring or single refactoring is a code transformation that aims to improve the internal quality of a piece of software [25]. Fowler presents a catalog of refactoring types [25]. Previous studies have revealed that single refactorings are not sufficient to improve structural problems [8, 10, 168, 225]. Developers need to apply multiple refactorings to tackle a structural problem, in instances such as the removal of a code smell [143]. Therefore, in practice, large-scale refactorings are applied [225]. A common practice in this context is the application of composite refactorings [46,143,171]. A *composite refactoring (or, simply, composite)* is a set of two or more interrelated refactorings [143, 171].

Existing studies identified several *composites types*, which are refactoring combinations that constitute a composite [76, 143, 171, 203]. These composite types help practitioners understand which composite configurations are commonly applied in practice. Those studies also revealed that composites affect software quality by introducing or removing code smells [143, 171, 203, 204]. However, they do not study their side effects. A *side effect* is when a composite removes a target code smell, but introduces other smells. Previous studies also presented descriptions recommending certain composite types to completely remove a certain target code smell [143, 171]. In our study, we called these composites as *"complete" composites.* For example, *Extract Method* and *Move Method* form a composite type recommended to completely remove the Feature Envy smell [25, 171]. However, these descriptions are limited and insufficiently explored. Their limitations are detailed as follows:

1. These descriptions are formed by a small subset of Fowler's refactoring types (14 out of 72 types) [25, 143, 171]. Also, they may only partially reflect how complete composites are applied in practice. This limits how common complete composite types used by developers are described. Thus, developers might not be aware of the existence of some complete composite types when relying on existing descriptions.

2. Previous studies presented composite recommendations targeting a single smell, but in practice, a piece of code, i.e., a method or class, can have

multiple code smells [143, 171]. The developer, then, focuses on removing a single code smell, but is not alerted about the existence of other code smells. For example, a study presents a recommendation to remove a *Long Method*, but this long method can also have a *Feature Envy* [143]. When the developer applies this recommendation, they may not be aware of the existence of that *Feature Envy* and/or know what to do with that code smell.

3. Existing descriptions of complete composites do not present side effects, their possible causes, and recommendations to remove or minimize these side effects [143, 171]. Previous studies also present multiple recommendations to remove the same code smell type, without detailing when each recommendation can be applied. This limits developers' knowledge about which complete composites should be applied in a given situation.

Revealing the limitations of these existing descriptions of complete composites is the starting point to reason on how they may be improved and for them to be used to aid practitioners and tool builders [39, 44, 64, 201]. Exploring and revisiting these descriptions is relevant since their limitations can interfere negatively on the developers' decisions to apply composites when they aim to remove code smells.

To address these limitations, our study aims at investigating what are the most common types of complete composites. We used an extended subset composed of 26 refactoring types presented by Fowler [25] together with eight other types widely observed in practice. Then, we are able to analyze if the existing descriptions of complete composites are aligned with the most common types. We also explored the (side) effects of complete composites on a wider set of code smells. We investigated 618 complete composites from 20 open source projects, and the effect of four widely popular code smell types [8,60,73], namely *Long Method*, *Feature Envy*, *God Class*, and *Complex Class*. Differently from previous studies [143, 171], we performed an in-depth analysis on the effects of complete composites, focused on understanding their side effects. Our results are summarized as follows.

1. We observed that 64% of complete composites have refactoring types not covered by previous studies [143, 171]. For example, developers applied composites with *Extract Method(s)*, *Change Return Type*, and *Move Method(s)* on envious methods. These refactoring types aid the removal of the *Feature Envy*. However, existing descriptions recommend only *Extract Method(s)* and *Move Method(s)* to remove this code smell. In other words, the current body of knowledge on composite refactoring is incomplete. In this sense, developers relying on existing studies may be misguided or unsuccessful when trying to remove a code smell. Instead of providing recommendations that ease the

practice of applying composites to improve software quality, existing studies might be leading developers into making wrong decisions.

2. Almost half (48%) of *Feature Envy* smell instances were removed when the composite *Move Methods* were applied. This information is not documented by existing descriptions. Since the occurrence of *Feature Envy* is a common situation [8], knowing about the usage of the *Move Methods* composite in advance can ease refactoring tasks.

3. About 36% of complete composites formed by *Extract Methods* to remove *Long Methods* have introduced *Feature Envies* and *Intensive Couplings* as side effects. Surprisingly, with the goal of improving readability, by removing *Long Methods*, developers degrade the software internal quality by creating unnecessary high coupling. This goes in the opposite direction to the purpose of refactoring, which generally is to improve overall software quality [25].

Our findings reveal that developers tend not to solve structural problems completely when they apply large-scale refactorings, such as composites. These composites can remove one target code smell, but potentially introduce or do not remove other ones. This may be an alert regarding the in practice use of existing descriptions. In summary, our results provide the following contributions.

1. A dataset with 618 complete composites, and their common types and (side) effects. This dataset can be used by other studies and improved, motivating future studies to investigate other topics on complete composites.

2. Our results suggest that existing descriptions of complete composites should be either revisited or enhanced to explicitly include possible side effects. They can (i) include *Change Return Types* in composite recommendations to remove *Feature Envy*, since developers need to change the return type of these methods due to the separation of the concerns; (ii) alert developers about alternatives to remove *Feature Envy*, mainly in methods that are fully envious; and (iii) guide developers to avoid mistakes that can introduce code smells.

3. We present a catalog of complete composites based on existing descriptions and on our results, showing detailed specifications about side effects, recommendations to remove or minimize them, and some scenarios in which recommendations can be applied. Besides that, we discuss a real example in which developers accepted a recommendation from our catalog and removed the side-effects of a real problem. Our catalog also can be useful to improve existing tooling support for refactorings [39,222]. These tools can inform about the side effects and allow the developer to choose if (i) the complete composite may be applied, or (ii) the side effects may be minimized or removed, or (iii) if the complete composite may be ignored. These decisions should take into con-

sideration why and how developers are aiming to improve the code structure, i.e. the context of the problem and their goals.

4.2 Background and Problem Statement

In this section, we presented definitions and the existing limitations on complete composites.

4.2.1 Composite Refactoring (or Composite)

As is already known, in practice, a code smell is rarely removed by applying single refactorings [8]. Mainly, because some code smell types need the application of more than one single refactoring to remove them. Therefore, developers need to apply large-scale refactorings in practice [225]. A common practice to apply large-scale refactorings is applying composite refactorings [46, 143, 171].

A composite 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. A composite c can be formed of many instances of the same refactoring type, or a combination of different types [12, 143, 171, 172, 203, 226].

Due to the complexity of identifying whether a refactoring is part of a composite, a recent study proposed a *range-based* heuristic [171]. This heuristic detects the structurally interrelated refactorings that together form a composite. For that, the heuristic limits a composite to refactorings that share at least a code element with a structural relation, and are applied by the same developer. The reliability of this heuristic was demonstrated in [171]. The *range-based* heuristic was designed taking into account the practices of developers, as a previous study has shown that composites are often applied to multiple classes [172]. This heuristic captures refactorings that were applied on a common set of code elements (classes and/or methods), indicating that the developer could have the intent to improve the internal software quality of this set of code elements from a composite. Thus, this improvement can be a code smell removal. In summary, the range-based heuristic indicates pieces of code where the developer made a set of refactorings, potentially for removing a code smell or improving the source code structure. This heuristic has been used in studies that investigate the effect of composites on multiple classes [76, 171].

A composite can be classified by types, which are refactoring combinations that constitute a composite. For example, composites constituted by

Extract Methods and *Move Methods* are of the type [*Extract Method, Move Method*] [143,203]. Knowing the composite types helps practitioners to understand which ones are most commonly applied in practice [143,171]. Previous studies also revealed that composites have an effect on software quality. For example, a composite can affect a code smell in different ways, such as introducing, removing, or not affecting a code smell [143,171,203]. Thus, the effect of a composite can be a removal, introduction, or unaffected. Unfortunately, the existing studies investigated the effect of composites in a general way. They did not detail that a composite can have side effects. In our study, we consider a side effect when a composite removes a specific code smell, but also introduces other code smells.

4.2.2 Completeness of Composite Refactorings

Some studies have explored the completeness of composites to remove code smells [76,143,171,203]. The completeness of a composite is a characteristic that can be seen from different viewpoints, *e.g.*, an entire code smell removal [143,171] or an improvement to a internal quality attribute [76]. In this work, we rely on the definition that a composite is complete when it completely removes one code smell instance. Although the topic of composite refactoring has been receiving great attention in recent studies [19,143,171,172], existing pieces of work do not investigate the property of completeness in-depth. There is a gap in the existing literature on composite refactorings and the current practice of completely removing code smells. We detailed these limitations next.

Descriptions of complete composites are found in the literature [25,143,171]. Table 4.1 presents such descriptions of complete composites recommended to remove code smells [25,143,171,204]. These studies propose more than one recommendation to remove the same smell type. This happens because studies [143,171,204] presented alternative complete composites applied in practice, explaining why there are multiple recommendations for the same smell. Despite the existence of these descriptions, they fail in describing the specifics of each complete composite, as described in the next subsection.

In this study, the definition of completeness focuses on code smell removal, independently from developers' intents. We do not relate refactoring completeness with its motivation, as refactorings often emerge (>70% of the cases) along with other small modifications and they cannot be captured from issue descriptions, commit messages, or developers' discussions [167]. We are aware that developers do not fully remove smells due to many factors, but

Table 4.1: Descriptions of Complete Composites

Complete Composites	Code Smell
Inline Methods [143] Pull Up Methods [143] Pull Up Attributes, Pull Up Methods [143] Inline Method, Move Method [143] Inline Method, Move Attribute [143] Extract Methods, Move Methods [25] [171] Inline Methods, Extract Methods [171] Extract Methods, Move Attributes [171]	Feature Envy (FeE)
Extract Methods [25, 143]	Long Method (LoM)
Move Methods [25, 143]	Complex Class (CoC)
Pull Up Methods, Move Methods, Pull Up Methods [171] Inline Methods, Extract Methods [171] Move Methods [171] Inline Methods [171] Extract Methods [171] Pull Up Methods [171] Pull Up Attributes, Pull Up Methods [171] Pull Up Attributes, Pull Up Methods, Move Methods, Pull Up Methods [171]	God Class (GoC)

our study advances in an important direction: characterizing which side effects exist and their possible implications. Developers are concerned with improving software quality [167, 205], and many of these concerns are related to smell removal, even if indirectly [73,206–209]. For example, developers can reorganize a method that is hard to read with a composite refactoring even without being concerned with its size, so indirectly they remove a "*Long Method*" smell.

4.2.3
Limitations of Existing Complete Composites Descriptions

Previous pieces of work present descriptions of complete composites [25, 143,171]. These descriptions are not summarized and detailed. We then created Table 4.1 to summarize the existing descriptions of complete composites for developers. However, the existing descriptions do not represent an extensive list of complete composites, as they are formed by a small subset (14 out of 72) of refactoring types defined by Fowler. This is a limitation, as developers can apply other refactoring types in practice frequently, not covered in these descriptions.

Another limitation is that existing studies do not investigate whether during the application of complete composites focusing on a specific code smell, other code smell types are affected or new smells are introduced, resulting in side effects [143, 171]. For example, composites that are complete to remove a *Long Method* can introduce *Feature Envy* (see Section 4.2.4). As we can see, these studies present many recommendations to remove the same code smell type, but they did not detail when each recommendation may be applied. Thus, an in-depth study on the (side) effect of complete composites on other code smells is of paramount importance, since the application of these documented complete composites could lead to the introduction of other code

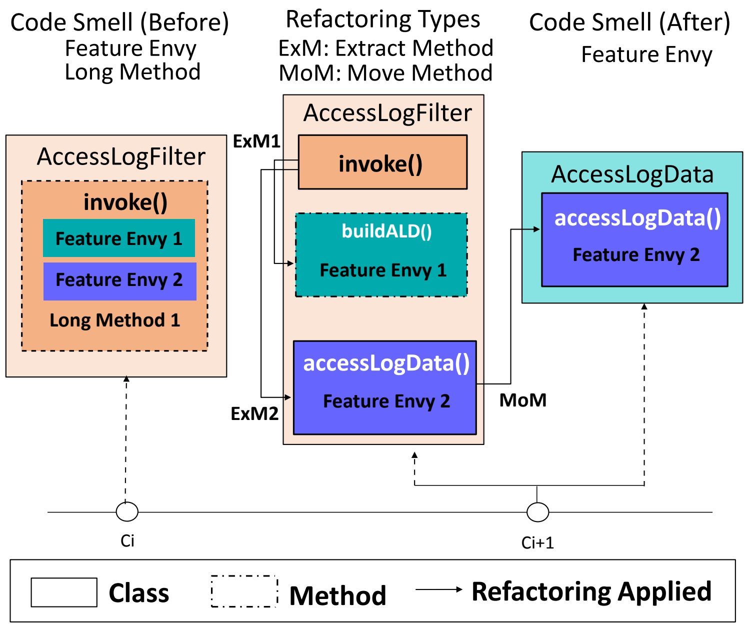


Figure 4.1: An Example of Complete Composite and its Side-effect

smells, impacting negatively the quality of software. These descriptions also need to alert developers about the possible causes of these side effects and recommendations to remove or minimize them, and when each recommendation may be applied. As a consequence of these limitations, developers may be trusting that existing descriptions recommend complete composites that effectively remove problems in their code. However, they are unconsciously increasing deterioration on critical parts of the code.

4.2.4 A Real Example of Complete Composite

This section details a real example in the `Dubbo` [181] project, in which a developer applied a composite refactoring for removing some smells. As presented in Figure 4.1, at some point in time (commit c_i) the class `AccessLogFilter` had a method called `invoke()`, having two code smells, namely a *Long Method* and a *Feature Envy*. In order to remove these smells, a developer applied a composite formed by two *Extract Methods* (ExM_1 and ExM_2) and one *Move Method* (MoM) in the commit c_{i+1} [180].

According to existing descriptions [25, 171], two *Extract Methods* can remove the *Long Method* and at least one *Extract Method* with one *Move Method* are expected to remove the *Feature Envy*. In the commit $c_{(i+1)}$, by applying the two *Extract Methods* on the `invoke()` method, creating `buildAdd()` and `accessLogData()` methods, the *Long Method* was removed and the *Feature Envy* ended up being scattered across the two extracted methods. Then, the developer applied one *Move Method*, moving the `accessLogData()` method to the `AccessLogData` class, aiming to remove one of the *Feature Envies*. However, the code smell was not removed from the `accessLogData()` method as the method was actually more interested in data from another class.

In conclusion, the composite [*Extract Method*, *Extract Method*, and *Move Method*] could have removed both *Long Method* and *Feature Envy*, as indicated by previous studies [25, 143, 171]. We observed that this commit applied only changes on code related to these refactorings. Besides, the goal of this commit is refactoring to remove the *Long Method* and *Feature Envy*. The commit message is "Acesslog dateformat enhancement". This commit is related to a pull request [184] in which developers discussed what refactorings were applied. A developer mentioned *"I have moved the access log creation..."*, adding *"Refactored code to separate our and group related tasks in separate methods and have enhanced the readability by using: Method renaming, Reducing big methods to small..."*.

Despite having removed the *Long Method* (in the `invoke()` method), the composite did not fully remove the *Feature Envy* (in the `accessLogData()` method). On the contrary, although the composite is considered complete as it removes the *Long Method*, according to [143, 171], it induced side effects. The composite induced the harmful propagation of the *Feature Envy* smell to additional methods and an additional class affected by the refactorings. We can see that existing recommendations of composites can lead to side effects, such as: (i) the prevalence of a smell in the program elements touched by a composite refactoring, or (ii) even the introduction of code smells.

Existing descriptions of complete composite types do not indicate at all what are their possible recurring side effects. This limitation is also not addressed by previous empirical studies [143, 171]. Perhaps, these developers applied this recommendation without being aware of the side effects of these complete composites. Because the existing descriptions do not have information about that, limiting the decisions of developers regarding these side effects. Thus, the case discussed above illustrates why existing descriptions of composite refactorings have to be extended to cover recurring complete composite types and their side effects. This advance can also further motivate developers to use and trust such descriptions while enabling them to make more informed decisions through the selection and application of composite refactorings.

4.3 Study Settings

This section presents our research goal, research questions, and the six steps of our study. More details about the results and artifacts of this study are available on our website [193].

4.3.1 Goal, Research Questions, and Metrics

The goal of our study is to analyze composite refactorings applied in practice based on two aspects: (i) the most common types of complete composites, and (ii) their side effect on code smells in the context of 20 software projects. To achieve the goal of our study, we carefully designed two Research Questions (RQs), as follows.

RQ$_1$: *What are the most common types of complete composites?* This RQ aims at investigating what are the types of complete composite most frequently applied by developers. We analyzed 34 refactoring types, of which 26 are present in Fowler's refactoring catalog and 8 refactoring types that are often applied in practice [173]. This number is considerably higher than previous studies that investigated only 14 refactoring types [143,171]. The investigation of complete composite types is relevant to identify if the existing descriptions of complete composites are aligned to development practice. Moreover, we can reveal other types of complete composites that are not documented in existing catalogs.

RQ$_2$: *What are the side effects of complete composites?* This RQ aims at exploring if complete composites introduce other code smells (the side effects, as detailed in Section 4.2.2) while removing one target code smell. We then collected complete composites for 4 common code smells: *Feature Envy*, *Long Method*, *God Class*, and *Complex Class* [8]. Also, we investigated possible causes of these side effects and how to remove or minimize them based on refactorings applied in practice. In that way, we aimed to create a detailed catalog of complete composites and their side effects. This catalog can improve the existing descriptions of complete composites and guide developers and refactoring tools about how to apply complete composites, avoiding or minimizing side effects.

4.3.2 Study Steps

Step 1: *Software Project Selection.* – We adopted three criteria similar to previous studies to select software projects [8,10,60,143,168,171,172]: (i) the software project must be open source and implemented using Java due to the support to the study replication and tools limitations; (ii) the software project must use Git as the main version control system, aiming at supporting the use of state-of-the-art tools for refactoring detection that only work on

Git projects; and (iii) the software project must have been analyzed at least one related study regarding refactoring [10, 168, 172] or code smells [8, 143]. Thus, we selected projects that are known to have a considerable amount of refactorings and smell instances.

Step 2: *Single Refactoring and Composite Refactoring Detection.* – We used the RefactoringMiner 2.0 (or RefMiner 2.0) tool for detecting single refactorings applied in practice, similar to related studies [8, 10, 143, 168]. RefMiner is a tool acknowledged for reaching high precision and recall when detecting refactoring and supports several refactoring types [173]. The tool identifies 52 refactoring types. We focused on the 34 ones that are in the scope of our study. 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. Also, RefMiner 2.0 captures refactoring types that were not explored by existing approaches [143, 171]. These additional single refactorings help us to capture the composites that have other refactoring types that were not revealed by these studies. Table 4.2 presents the 34 refactoring types investigated in our work. We investigated 16 Fowler's refactoring types that were used by previous studies [143, 171], third column F-PS (Fowler and Previous Studies); ten Fowler's refactoring types that were not used by previous studies [143, 171], fourth column F-NPS (Fowler and not Previous Studies); and eight refactoring types that are not by Fowler and that were not used by previous studies [143,171], fifth column NF-NPS (not Fowler and not Previous Studies). These refactoring types not covered by Fowler were defined by [194] and are detected by RefactoringMiner. Note that, the refactorings NPS (fourth and fifth column) are fine granularity (there are transformations on variables or attributes). However, we investigated these refactorings because they can be often applied with refactorings of larger granularity (transformations on methods or classes). Therefore, these refactorings can be common in practice when developers apply refactorings on large scale as composite refactorings. However, the literature is limited about the knowledge concerning how these refactorings have helped the refactoring process in practice. For the detection of composite refactorings, we used the *range-based* heuristic [171] detailed in Section 4.2.

Step 3: *Code Smell Detection.* – Our study investigated the removal of four common code smell types: *Feature Envy*, *Long Method*, *God Class*, and *Complex Class*. These smell types have well-known recommendations of composite to remove them [25, 143]. 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.

We also investigated how the complete composites affect 19 code smell types (Table 4.3) when removing these four smell types. These 19 code smell types are frequently detected in real projects [8, 60] and they are found in methods and classes. We used the Organic tool to detect code smells [48]. Also, this tool was used by studies that proposed descriptions of complete composites. This tool uses strategies based on software metrics to collect code smells. These detection strategies were already evaluated by previous studies [8, 17, 48].

Table 4.2: Refactoring Types analyzed in this Study

Refactoring Type	**ID**	**F-PS**	**F-NPS**	**NF-NPS**
Move Attribute	MoA	✓		
Pull Up Attribute	PUP	✓		
Push Down Attribute	PDA	✓		
Rename Attribute	ReA	✓		
Replace Attribute	RpA			✓
Extract Attribute	ExA		✓	
Merge Attribute	MeA		✓	
Split Attribute	SpA		✓	
Extract Variable	ExV		✓	
Inline Variable	InV		✓	
Parameterize Variable	PaV			✓
Rename Variable	ReV		✓	
Replace Variable with Attribute	RVA			✓
Rename Parameter	ReP		✓	
Replace Variable (with Attribute)	RpV			✓
Merge Variable	MeV			✓
Change Return Type	CRT		✓	
Change Parameter Type	CPT			✓
Change Variable Type	CVT			✓
Merge Parameter	MeP			✓
Split Variable	SpV		✓	
Split Parameter	SpP		✓	
Extract Method	ExM	✓		
Inline Method	InM	✓		
Rename Method	ReM	✓		
Move Method	MoM	✓		
Pull Up Method	PUM	✓		
Push Down Method	PDM	✓		
Extract Class	ExC	✓		
Extract Subclass	ExS	✓		
Extract Superclass	EtS	✓		
Move Class	MoC	✓		
Rename Class	ReC	✓		
Extract Interface	ExI	✓		
	Total	16	10	8

Table 4.3: Code Smell Types analyzed in this Study

Code Smell Type	**ID**	**Definition**
	Method scope	
Brain Method	BrM	Method overloaded with software features
Dispersed Coupling	DsC	Method that calls too many methods
Divergent Change	DiC	Method that changes often with other ones
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 scope	
Brain Class	BrC	Class overloaded with software features
Class Data should be Private	CDP	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
Large Class	LgC	Too large 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 4: *Complete Composite Computation.* – We focused on the complete composites for removing four code smells that have previous recommendations for their removal. We then elaborated Table 4.1 that presents the recommended composites for the removal of these code smells according to studies found in the literature [25, 143, 171]. More details about the detection of complete composites in the projects can be found in [193].

Step 5: *(Side) Effect Analyses.* – We collected the code smells introduced, removed, and unaffected by the complete composites. Then, three authors manually analyzed the effect of complete composites (presented in Sections 4.4 and 4.5). We focused on these complete composite types to find possible side effects in complete composites that are commonly applied in practice. We aimed to find the relation between the introduction of code smells and the complete composites that removed the target code smell. We analyzed pieces of code that were touched by composites, other code changes, the motivations of the commits, and pull request discussions in the commits in which the complete composites were applied. It facilitates the understanding if other code changes could have introduced the code smell and if developers know these code smells.

Step 6: *Composite Validation by Experts.* – We randomly select a sample of 84 (13.5%) out of 618 composites, composed of 414 refactorings, to be manually validated by eight experts. The expert's experience has a median of 6 years. In our validated sample, before applying the composite refactorings, the source code manifested 87 code smells. After the composite refactorings, the source code manifested 115 code smells. Among the composites of this sample, 38 are complete composites in our dataset according to our scripts.

In the manual validation, each expert received a set of composite refactorings analyzing the code before and code after each single refactorings, code smells before, code smells after each composite. They answered the following questions: (i) *"Is it a complete composite for the removal of some code smell(s)?"* and (ii) *"What was the intent of the developer in this commit?"*. These questions allowed us to know about the completeness of the composites, and whether the purpose of commits was to execute refactorings.

The experts confirmed 27 (71%) out of 38 complete composites as complete (true positive), also they considered four composites that are not complete in our dataset as complete (false negative). They confirmed that the other 42 composites are not complete composites (true negative), and 11 composites are false positive for complete composites in our dataset. In that way, we have 31 complete composites according to developers (27 true positives and 4 false negatives). These results show that our script to collect complete composites has high accuracy (71%) and experts had a high agreement with our definition of complete composites. We observed that false negatives are composites that removed code smells that we are not investigating in this study, such as Duplicated Code.

Regarding the purpose of developers during commits where these 31 complete composites were identified, 19 (61%) out of 31 complete composites were applied to support refactoring. In addition, 7 (22.5%) out of 31 complete composites were applied to support refactoring and other development activities (e.g., feature addition, bug fixing). Finally, 5 (16%) out of 31 complete composites were applied to support other development activities. These results show that (83.5%) complete composites were applied to support refactoring, where developers had the intent to improve the internal code structure through the removal of code smells.

4.4 Common Complete Composites

In this section, we present the most common complete composites for four types of code smells (*Feature Envy*, *Long Method*, *Complex Class*, and *God*

Table 4.4: Most Common Complete Composites.

ID	Group	F-PS	(%)	NPS	(%)	CC	(%)
G1	[ExM]	122	19.74%	101	16.34%	**223**	**36.08%**
G2	[Refs. of Fine Granularity NPS]	0	0.00%	146	23.62%	146	23.62%
G3	Other Groups	28	4.53%	51	8.25%	79	12.78%
G4	[ExC, MoM]	17	2.75%	20	3.24%	37	5.99%
G5	[MoC]	2	0.00%	29	4.69%	31	5.02%
G6	[ExM, InM]	14	2.27%	16	2.59%	30	4.85%
G7	[MoM]	9	1.46%	16	2.59%	25	4.05%
G8	[PUM, PUA]	17	2.75%	6	0.97%	23	3.72%
G9	[ExM, MoM]	7	1.13%	7	1.13%	14	2.27%
G10	[MoC, MoM]	3	0.49%	7	1.13%	10	1.62%
	Total	219	35.44%	399	64.56%	618	100.00%

* CC: Complete Composites.
** Refs. of Fine Granularity NPS: Refactorings of fine granularity not previously studied.

Class), as justified in Step 4 (Section 4.3.2). For that, we collected types of complete composites, according to the definition of composite types presented in Section 4.2. Then, they were grouped according to the presence of at least one of Fowler's previously studied refactorings [143,171], named refactorings F-PS, along with refactorings not previously studied, named refactorings NPS. Based on that, we investigated the frequency of complete composite types that are formed by only F-PS refactorings versus those that are formed by both F-PS refactorings and other NPS refactorings. For example, a subgroup containing complete composite types formed by *Extract Methods* only and another with complete composite types formed by a mix of *Extract Methods* and refactorings NPS (e.g. Change Return Type). We collected these for all groups, and ranked the most common complete composites.

Table 4.4 presents the most common complete composites. The columns present, respectively, an identification (ID) for each group, for easier reference along with the analysis, the name for each group (defined by F-PS refactorings [143, 171]), the number of composites containing only F-PS refactorings, the number of composites that contain at least one refactoring NPS, and the last column presents the total number of composites for each group. Group G2 contains only refactorings NPS. We called this group Refactorings of Fine Granularity that are not previously studied. The existence of this group and its ranking reveals that developers often apply minor scale refactorings together. We considered G3 as a generic group of complete composites that do not belong to any other group, i.e., composite types that were observed only once.

Some of the common complete composites found are not described in existing descriptions (see Table 4.1). For example, *Extract Classes* and *Move Methods* (Group G4) are common complete composites applied by developers but existing descriptions observed in the literature do not include them. This leads us to the first finding of our study.

Finding 1: Some complete composites commonly found in practice are not documented by existing descriptions. Such existing descriptions should include these complete composites because they would better inform developers performing refactorings.

Group G1 is composed of composites with at least two or more *Extract Methods*. This group is recommended to remove *Long Methods* (see Table 4.1). We can observe that the complete composites of this group are the most common, representing 36% of the composites identified in our study. This is in consonance with previous studies, where they conclude that in fact *Extract Methods* are widely applied [143, 171]. However, in our study, we identified a different scenario. Interestingly, developers often composed these *Extract Methods* with other refactoring types that were not investigated by previous studies. Among the 223 composites of G1, 101 were not investigated by those previous studies (column NPS). A similar observation can be applied to the G7 group. Previous studies have not considered these composites in which *Move Methods* are applied with other refactoring types. However, as we could observe, a combination of them is in fact used to remove code smells. Finding 2 was derived from these observations. The implication of this finding is that existing descriptions must be revisited to further describe possible combinations of refactoring types. This would allow developers to better reason on how to apply composites when aiming to remove certain types of smells.

Finding 2: Common complete composites investigated in the literature are additionally composed with other refactoring types for better supporting smell removal.

In the previous paragraphs, we focused on the most common composites found in the literature. In addition, we also performed an overall analysis of composites not investigated in previous studies. This allows us to observe that 399 (fourth column of Table 4.4) out of 618 complete composites have refactoring types not considered as part of complete composites. This broader analysis leads to the third finding below:

Finding 3: Among all complete composites, 64% include refactoring types that were not previously investigated. Existing descriptions of complete composites may be extended to recommend composites with these refactoring types.

Regarding the group G3, we have not identified any pattern. This was expected because this group has only composites with only two or fewer instances of different types. In other words, they are not representative to conclude their effect on code smell removal. We know that descriptions should be constantly updated through empirical studies analyzing the use of composites in practice. Our study has that focus, aimed at investigating what developers commonly do in their projects. We observed that what is common in practice is not aligned with existing descriptions [25, 143, 171] (Findings 1 and 2). The descriptions assist developers on how to apply complete composites. Our study shows that some developers do not strictly follow the composites described in existing descriptions, while they are also misinformed side effects of those recommended composites. Note that informing complementary or missing, even if simple, refactoring types such as a *Change Return Type* can either assist tool developers to either detect those types or warn developers to not miss those refactorings.

4.5 (Side) Effects on Code Smells

In this study, we focused on analyzing the (side) effects of two types of complete composites that are previously [143, 171] and frequently recommended to remove the code smells analyzed in this study. In addition, we analyzed the (side) of complete composites formed by these refactoring types and refactorings not previously studied, aiming to find differences between these different combinations. We performed manual analyses (detailed in Section 4.3) to understand if the (side) effects were caused by the complete composites, find possible causes for them, and recommendations to remove or minimize these side effects. We discussed the main results of this quantitative analysis and manual analyses in this section. Additionally, more details about that are available on our website [193].

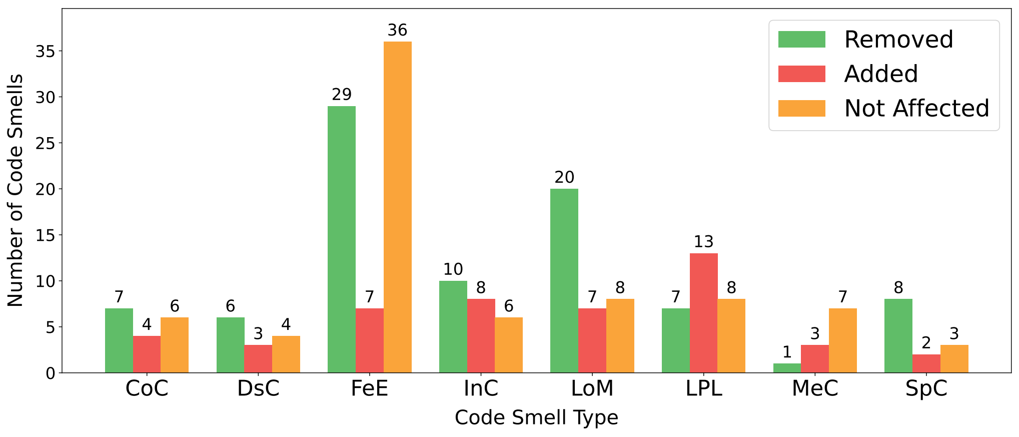
Figure 4.2(a) presents the effect of complete composites formed by only *Extract Methods*. This group of composites is recommended to remove *Long Methods*. Figure 4.2(b) presents the effect of complete composites formed by *Extract Methods* and refactorings not previously studied. In our manual analyses, we observed that they are related to the introduction of the same code smells types. The sum of side effects of Figures 4.2(a) and 4.2(b) are: 64 (7 + 57 = 10%) out of 631 (72 + 559) that introduced *Feature Envies* (FeE), 34 out of 126 (26%) introduced *Intensive Coupling* (InC), and 217 out of 328 (66%) introduced *Long Parameter Lists* (LPL). One possible cause for the cases in which the *Long Methods* (LoM) were not affected – 100 out of 256 (39%), is that the developer could not be applying the necessary number of *Extract Method* refactorings. Thus, the methods that were originally extracted continued being too long. In the case of the propagation of *Feature Envies*, a possible cause is that the code that was extracted already had that code smell. Therefore, when that code was extracted, that smell was moved alongside the method. A possible solution for this problem would be to apply more *Extract Method (s)*, *Move Method(s)*), targeting the removal of the smelly code. The extraction of many methods that implement the same concern also increased the coupling between methods, introducing the *Intensive Coupling* smell. *Extract Methods* and refactorings, such as *Merge Parameter* and *Replace variable*, can have a certain contribution to remove *Long Methods* (16 out of 222 = 7%). This can be justified because these refactorings contributed to the decrease of statements in the affected method(s), decreasing the number of lines in a long method.

We have found real examples of composites leading to the unintentional addition of side effects. One example of such side effect is in a set of commits from ElasticSearch [185] and Presto [186]. Developers applied *Extract Methods* in the methods affected by *Long Method* according to the existing recommendations [25, 143]. However, the refactorings applied led to the creation of new methods – with a high amount of parameters in their signature – thus introducing *Long Parameter List*. This smell could be prevented by the application of the *Merge Parameter* refactoring, which would combine interrelated parameters into an appropriate object. Other code smells were also affected by this group, as we can see in Figures 4.2(a) and 4.2(b). However, we were not able to find a direct relation between this group and other code smell types. We believe that those code smells may have been affected by other code changes.

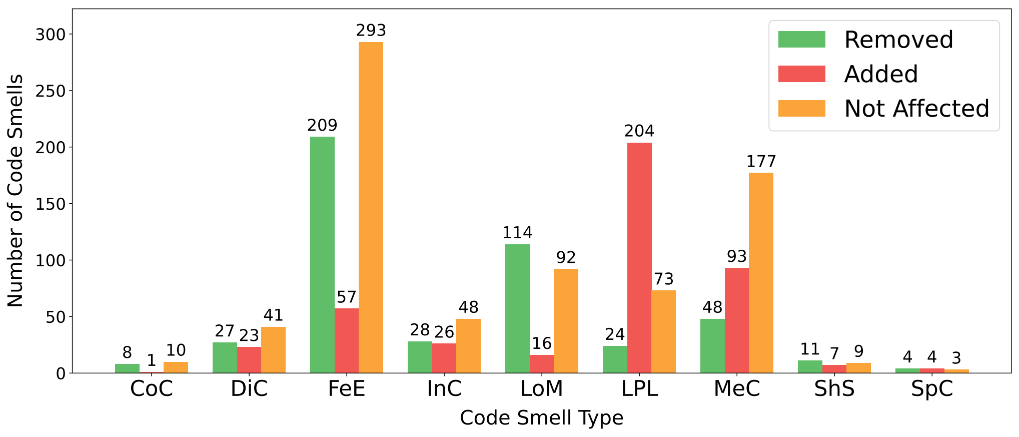
Finding 4: About 36% of complete composites formed by *Extract Methods* have introduced *Feature Envies* and *Intensive Couplings* when they remove *Long Methods*. Unfortunately, developers degrade the software internal quality by creating unnecessary high coupling.

Figure 4.2(c) presents the effect of the group of complete composites including *Move Methods* only. This group is recommended for the removal of *God Class* or *Complex Class* smells. We observed that this group is mainly associated with the removal of 23 out of 48 (48%) *Feature Envies*. We manually observed that these removals happen because those composites often moved methods that are fully envious to more appropriate classes. These types of composites also can introduce *Feature Envies* – 8 out of 48 (17%). Developers moved the envious methods to inappropriate classes, consequently creating new cases of *Feature Envy*. On other code smell types, we did not find a direct relation between the application of these composites and their effect. Those code smells could have been affected by other code changes.

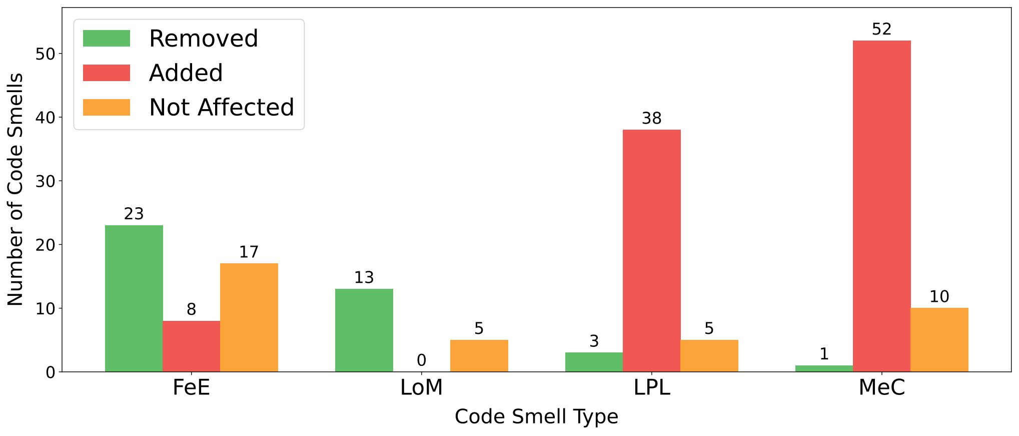
Finding 5: Almost half (48%) of *Feature Envy* smells were removed when the composite *Move Methods* were applied. This information is not documented by existing descriptions. This recommendation is not documented by existing descriptions [143,171]. These descriptions can alert developers and practitioners how to remove *Feature Envies* in methods that are fully envious.



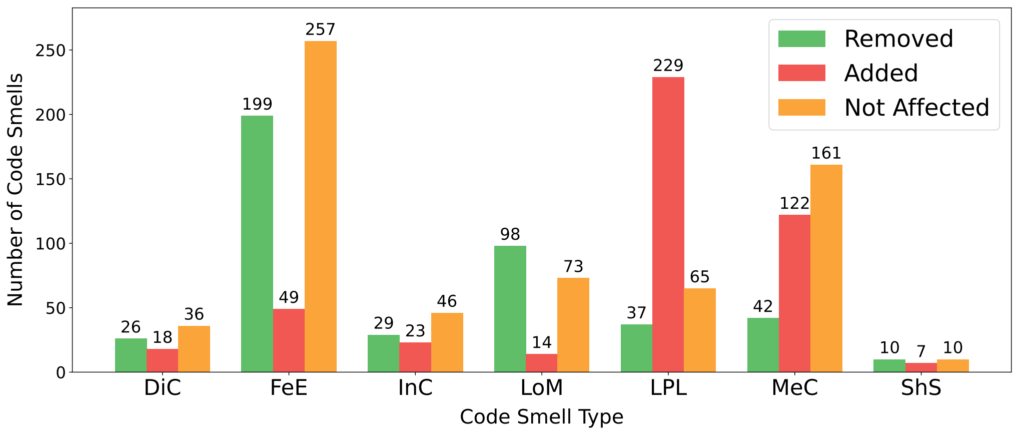
4.2(a): Extract Methods Only



4.2(b): Extract Methods and Not Previously Studied Refactoring Types



4.2(c): Move Methods Only



4.2(d): Move Methods and Not Previously Studied Refactoring Types

Figure 4.2: Side-Effect of Common Complete Composites

Table 4.5: Catalog of Recommendations with Side-Effects

Complete18	**Existing Smells**	**Target Smell**	**Reference**	**Removed Smell**	**Side-effect**	**Possible Cause of Side-effect**	**Recommendation**
ExM(s), MoM(s)	LoM, FeE	FeE	[4]. [14]	FeE, LoM	Introduction of FeE(s)	1. The envious code was extracted and moved for the extracted methods, creating new envious methods. 2. The methods were moved to inappropriate* classes, thus the method is an envious method to those classes.	Apply Extract Methods, if necessary, to separate concerns. Parameterize Variables or Merge Parameters from extracted methods if the variables are used between the concerns. Move the envious methods to the appropriate** classes
ExM	LoM, FeE	LoM	[4] [13]	LoM	Introduction of FeE(s)	1. The envious code was extracted and moved for the extracted methods, creating new envious methods.	Apply Extract Methods, if necessary, to separate concerns. Parameterize Variables or Merge Parameters from extracted methods if the variables are used between the concerns. Move the envious methods to the appropriate** classes
ExM(s)	LoM	LoM	[4] [13]	LoM	Introduction of LoM	1. The developer could not be applying the necessary number of Extract Method refactorings, the methods that were extracted, they continued being too long.	Extract more methods from long methods or apply refactorings NPS (such as Merge Parameter, Replace Variable, Change Return Type) to contribute to the decreasing of statements in the long method(s).
ExM(s)	LoM	LoM	[4] [13]	LoM	Introduction of LPL	1. The variables of the source method were transformed in an excessive number of parameters of the target methods.	Apply Merge Parameter on two or more parameters of the target method.
MoM(s)	FeE	FeE	This study	FeE	–	–	This composite may be applied when the method was fully*** envious, then the developer can move the method fully to the appropriate** class

* Inappropriate classes means classes that were not according to the responsibility of the method.

** Appropriate classes mean classes that were according to the responsibility of the method.

*** Method fully envious is a method in which most of the calls in this method are calls for other classes.

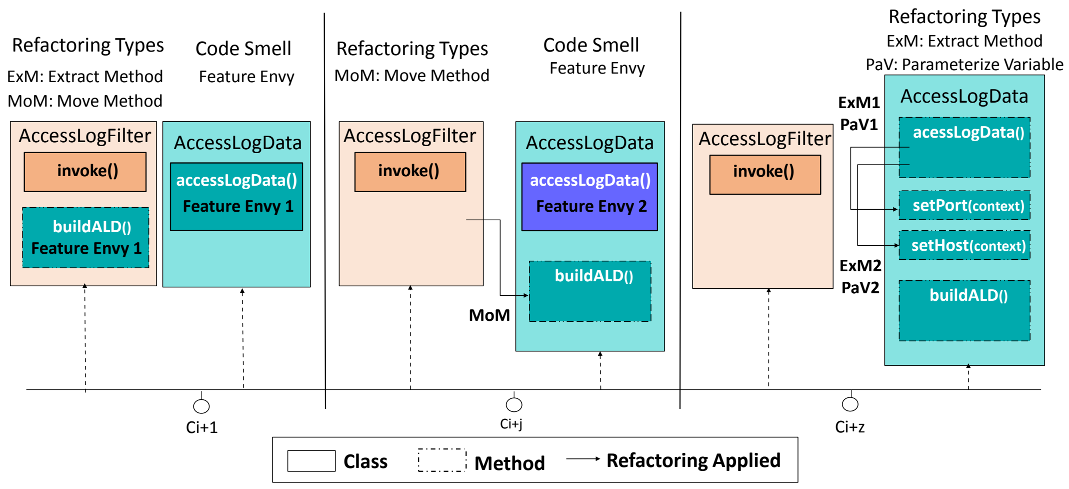


Figure 4.3: Solution of the Motivating Example

Figure 4.2(d) presents the effect of composites formed by *Move Methods* and refactorings not previously investigated [143, 171]. This type of complete composite can be related to the introduction of *Long Parameter List* (LPL) – 229 out of 331 (70%). In that case, we observed that developers applied the *Add Parameters* on methods that were moved, introducing a long list of parameters on these methods. We expected that these complete composites could have removed *Message Chains* (42 (13%) out of 325 Message Chains). However, in our manual analysis, we did not find how these refactorings could have helped to remove this code smell directly or indirectly. On other code smell types, we did not find causes to justify them as side effects of these complete composites.

Our findings reveal that developers tend to not solve the structural problem completely when they apply large-scale refactorings. They remove the target smell but introduce other code smells. This can happen because it is challenging to solve large structural problems, and developers need to decide how to combine refactorings to remove more than one code smell. This is an alert to the literature that investigates more solutions (composite refactorings) to solve large structural problems. From our results, we created a more detailed catalog of complete composites (Table 4.5). The catalog indicates the complete composite, the target smell, the existing smells before the application of composite, the side-effects, their possible causes of complete composites, and recommendations to minimize or remove these side-effects. This catalog can be used as a suggestion to improve the existing descriptions of complete composites, removing more than one code smell. It can help researchers, developers, and practitioners to identify scenarios in which each complete composite can be applied and how to minimize its side effects. Our catalog also can be useful for existing tooling support for refactorings [39, 222].

These tools can recommend complete composites. In addition, the tools can inform about the side effects and allow the developer to choose (i) the complete composite which may be applied, or (ii) the side effects to be minimized or removed, or (iii) if the complete composite may be ignored. These decisions may be according to the goal and context of developers if they are aiming to improve the code structure.

Figure 4.3 presents the solution of the motivating example (Section 4.2.4) according to our catalog. We aim to demonstrate from this real example the potential of our catalog and to have a certain acceptance of some developers. We submitted two pull requests following our catalog to remove the *Feature Envies* to verify if developers would accept the solutions of the catalog. In the first pull request (in the commit c_{i+2}) [182, 183], we applied a *Move Method*, moving the `buildALD` method to the `AccessLogData` class because we observed that this method called seven times methods of the `AccessLogData` class. Thus, this method was highly dependent on that class. This pull request was accepted. Developers did not answer why that refactoring was not applied previously, but we can hypothesize that developers did not observe that the method was envious. In the second pull request, (in the commit c_{i+3})*, we aimed to remove the *Feature Envy* of the `accessLogData` method in the `AccessLogData` class. In that case, this method calls four times the method of the `RPContext` class, and those calls involved many responsibilities related to set Port and to set Host. We then applied two *Extract Methods* to separate these responsibilities and applied two *Parameterize Variable*, adding a parameter of the `RPContext` class to avoid many calls of the methods of that class. The pull request was still open. We aim to observe if developers have an interest in the minimization of the side-effect when they applied complete composites. We submitted eight pull requests to validate our catalog, three pull requests were accepted, four are open, and one was rejected.

4.6 Threats to Validity

On detection of single refactorings, we used RefMiner 2.0 [194] because it has high accuracy, and its commonly used in the literature [8, 48]. Also, RefMiner 2.0 already ignores merging commits and treats refactoring masking [194]. On squash commits, we observed that the time between commits is short (2 weeks on average), indicating developers often refactor similar parts (composites). On code smell detection, we applied the Organic tool. This tool has detection strategies with high accuracy: 72% precision and 81% recall on average [4]. Aimed at detecting composite refactorings, we used a heuristic proposed by Sousa *et al.* [171]. This heuristic allows us to capture composites that involve multiple classes and to understand the effect of these composites on these classes. One author wrote scripts for computing complete composites, and two authors double-checked these scripts, thereby reducing the manual bias and errors.

We extended the subset of Fowler's refactoring types investigated by previous studies [143, 171]. This allows us to mitigate the threat to capture a limited number of refactorings as previous studies [143, 171]. On the RQ_2, we have a threat that code smells can be introduced or not affected due to other code changes. Aiming to mitigate this threat, we performed a manual validation with 8 developers with 84 composites of our dataset. These developers analyzed which composites removed the code smell (as described in Section 4.3). In our manual validation, we also observed how many complete composites can help to add or propagate existing smells. We observed 10 complete composites that helped to propagate existing smells, and 22 complete composites that have introduced smells. Another threat is that the side effects observed have been removed in later commits. We submitted eight pull requests for the current version of some projects (as mentioned in Section 4.5). We observed the side effects introduced by complete completes are still present in software projects. In our work, the notion of "code smell removed completely" was based on rigid thresholds of code smell detection tools [48]. Aimed to mitigate this threat, in the manual validation, we evaluated whether developers agree with the thresholds of code smell detection tools, considering that the code smells were removed completely (as described in Section 4.3).

Our focus was on performing a detailed analysis, instead of a higher smell type coverage, on selected smells that are shown to be the most related to developers' concerns [8,73]. We cannot claim generalization, but we analyzed projects with different sizes, domains, and all key findings were uniform [193]. Our understanding is not sensible or realistic at this point (when there is no study about the topic) to favor or even observe generalizability (in detriment to a first in-depth analysis per system) of a problem that is quite subjective (code smelliness) and project dependent. We claim the topic requires firstly a reliable characterization of the phenomenon in a specific set of projects, the focus of our study.

4.7 Conclusion

We investigated (i) what are the most common complete composites, and (ii) the (side) effects of complete composites on code smells. We collected 618 complete composites from 20 software projects. Our results indicated that 64% complete composites include refactoring types not covered by previous studies [143, 171]. Almost half (48%) of *Feature Envy* smells were removed when *Move Methods* were applied. These results suggest existing descriptions of complete composites should be either revisited or enhanced to explicitly include side effects. We present a catalog of complete composites with details about side effects. It can be useful to improve existing tooling support for refactorings. In future works, we aim to investigate the side effect of more composite types and interview more developers on our catalog.

5
Enhancing Recommendations of Composite Refactorings based in the Practice

In Chapter 4, we explored (i) what are the most common complete composites, and (ii) the (side) effects of complete composites on code smells. Our results indicated that 64% of complete composites include refactoring types not covered by previous studies [143, 171]. Almost half (48%) of *Feature Envy* smells were removed when *Move Methods* were applied. We then extracted five recommendations of complete composites with details about their possible side effects. However, these recommendations have main three limitations.

Firstly, the five recommendations are not well-detailed, concerning which refactoring types should be applied given certain circumstances. For instance, one of the recommendations indicate the application of *Extract Methods* together with other fine-grained refactorings, such as *Change Parameter Type* or *Change Return type*, to remove a *Long Method*. However, the recommendation does not detail which and when each fine-grained refactoring should be applied. For instance, the developer can apply *Change Parameter Type* or *Change Return type* with *Extract Method* to decrease a method size. However, when should a *Change Parameter* be applied? Should it be applied when the method has many parameters? Secondly, we did not suggest for developers complete composites that can remove code smells without inducing side effects. Finally, the third limitation is that we did not evaluate the developers' perceptions of our five recommendations.

Aiming at mitigating these limitations, we elaborated a study presented in this chapter to improve our composite recommendations addressing our RQ$_3$ (Section 1.3.1). In this chapter, we amplified our dataset in terms of the number of software projects and composite refactorings. We aimed at exploring what fine-grained refactorings are frequently applied and when they contribute to complete composites. Then, we can enhance our composite recommendations to better guide developers on applying each complete composite and alerting them on potential side effects. In this study, we also asked developers to evaluate our composite recommendations after the catalog improvements. The content of this chapter is going to be submitted to a international conference.

5.1 Introduction

Producing software projects with high design quality is the goal of every company [245, 246]. However, due to extensive maintenance and evolution in those projects, the internal software quality usually degrades [207, 247, 248]. Internal quality problems can be indicated by the presence of code smells [25]. A *code smell* is a poor code structure in a software project [25]. Developers must identify and remove code smells as soon as possible [25,249]. A well-known and widely used practice to deal with code smells is code refactoring [198]. *Code refactoring* intends to improve the code structure of the software [25]. Developers apply code refactoring aiming at fully removing code smells, even when refactorings are applied with non-refactoring changes, e.g., feature addition [60].

Despite its benefits, refactoring is a non-trivial activity, because developers need to apply at least two steps for refactoring. In the first step, smelly code have to be found [250]. In the second step, developers often have to combine refactorings [46, 143] through composite refactorings. A *composite refactoring* (composite, for short), consists of two or more interrelated single refactorings [76, 171]. 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 [171]. To make matters worse, studies have indicated that composites are generally applied manually [35, 46] and eventually combined with other code changes [46, 171]. Yet, studies indicate that only 10% of composites could remove code smells entirely [143, 171]. 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* [77].

Researchers have investigated approaches to contribute to refactoring applications for many years [251]. Recent studies indicated limitations regarding existing knowledge about refactoring, and cataloged composites that can fully remove code smells [77, 171, 227]. Catalogs are useful to guide developers (e.g., to effectively remove smells) and tool builders (e.g., to build tools that recommend effective refactorings). Also, catalogs can help developers to avoid re-refactoring the code later, decreasing the development effort. In fact, the lack of a comprehensive catalog may be one of the reasons why developers often need to re-refactor in the practice [10]. 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.

However, the existing catalogs are not aligned with the current software development practices and have certain limitations [77, 171, 227].

The first limitation of existing catalogs is that their recommendations are not well-detailed. They overlook the combination of some of the refactorings types that can be effective to remove a certain smell type. For instance, Bibiano *et al.* [77] recommend *Extract Methods* and fine-grained refactorings to remove the *Long Method* smell. However, they did not detail which fine-grained refactorings can be applied and how these refactorings can help to remove a *Long Method*. The second limitation concerns the side effects of the recommendations. Bibiano *et al.* also reported that complete composites may lead to side effects, such as the introduction or propagation of code smells. Despite indicating side effects, the authors have not performed an in-depth empirical analysis of side effects for complete composites. Another example can be found in the catalog of Brito *et al.* [227], 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 those *Pull Up Methods*.

The goal of our study is to enhance recommendations of composite refactoring based on the practice and properly support developers when applying refactorings. We aim to extend existing catalogs of complete composite refactorings that overcome the limitations mentioned above, and assess 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. These results were the basis to create our catalog of composite recommendations. Our results show that (but not limited to):

1. Developers frequently combine *Change Variable Type* or *Change Parameter Type* with other refactoring types, when extracting methods, to fully remove *Long Methods*, *Feature Envies*, and *Duplicated Code*. Interestingly, in our sample, we observed that 45% of these employed 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* that 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 (100%) agreed with our recommendations to remove the smells proposed in this study. 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 that there is a need in practice for recommendations 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. Existing state-of-the-art refactoring recommenders [39,204] neglect fine-grained refactorings, which were frequently present in complete composite recommendations.

5.2 Background and Problem Statement

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

5.2.1 Composite Refactoring (or Composite)

A single refactoring rarely removes a code smell [8]. Developers need to apply composite refactorings to eliminate the incidence of several code smell types [225]. 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 [171]. A composite c can be formed of the same refactoring type, or a combination of different refactoring types [12,143,171,172,203,226].

Due to the complexity of identifying whether a refactoring is part of a composite, recent studies proposed heuristics to detect composites [143, 171]. Several studies [76, 77, 172, 204] have used the *range-based* heuristic [171] 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 [76, 77, 171].

Different studies indicate that developers often apply composites manually [143, 171, 225]. Besides, we found evidence that composites frequently result in undesirable side effects [143, 171]. Therefore, having in-depth knowledge about composites is needed to support developers when applying refactoring. However, we observe that (i) there is a lack of 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, as described in the next subsection.

5.2.2 Completeness of Composite Refactorings

Recent studies recommend composites to remove a particular type of code smell [77]. When a composite is recommended to remove one or more code smell types, each code smell is considered a “target” of this composite [77]. For example, a recent study recommends applying *Extract Method(s)* combined with *Move Method(s)* to remove Feature Envy [171]. 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 was complete on the removal of the smell, otherwise the composite was 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 [77], 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 $TOTAL_{AFTER(s)} >= TOTAL_{BEFORE(s)}$.

An in-depth empirical investigation of the completeness of composites is necessary. Developers often spend time and effort applying composite refactorings to combat code smell incidences. However, there is no empirical knowledge evidencing the successfulness of these composites. In other words, whether they have fully eliminated code smells.

5.2.3 Fine-Grained Refactorings

A recent study mentioned that 64% of complete composites include refactorings of fine granularity [77]. A refactoring of fine granularity, or 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*, and *Extract Class*. In this study, we considered the term "coarse-grained" to better align with the term "fine-grained". Table 5.1 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, they are often ignored by studies of composite refactorings, except [77].

5.2.4 Existing Limitations about Completeness

Table 5.2 summarizes existing recommendations, presenting the complete composite that may be applied, the existing smells before the application of composite, the target smell that may be removed, the expected effect after the application of composite, and the side effects of complete composites. However, these recommendations are quite limited. The first limitation is that the existing recommendations are not well-detailed, mainly about which refactoring types may be applied and when each recommendation may be applied. Bibiano et al. [77] suggests that *Extract Methods* and FGRs may remove a *Long Method*. However, they do not detail which FGRs may be applied and how these refactorings in conjunction with an *Extract Method* help to remove a *Long Method*.

The second limitation concerns the insufficient description about the side effects of existing recommendations. For instance, Brito *et al.* [227] recommend

Table 5.1: Classification of Refactoring Types

Fine-Grained Ref. (FGR)		**Coarse-Grained Ref. (LGR)**
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**

Pull Up Methods to create a single and more general method in the superclass, achieving code reuse. However, they do not alert developers about the side effects of *Pull Up Methods* in practice. An example of a side effect is that when methods with *Feature Envy* or *Long Method* are refactored, have these smells are propagated to the superclass after the *Pull Up Methods*. Therefore, we need address these limitations to guiding developers on how and when to apply each recommendation.

Table 5.2: Existing Recommendations of Complete Composites

Complete Composites	**Target Smell**	**Effect**	**Side effect**
Extract Methods, Move Method(s) and Fine-Grained Refactorings [77]	Feature Envy	Removal of Feature Envy, Long Method	Introduction of Feature Envy(s)
Extract Method and Fine-Grained Refactorings [77]	Long Method	Removal of Long Method	Introduction of Feature Envy and Long Parameter List
Move Method(s) [77]	Feature Envy	Removal of Feature Envy	-
Move Method(s) [227]	-	Improvement of cohesion and coupling	-
Extract Methods, Move Method [227]	Duplicated Code	Promotion of reuse and removing duplication	-

5.3 Motivating Example

In this section, we present a real example of composite refactoring. This example can help on better understanding the concepts related to composite refactoring, completeness and side effects. In addition, we can show how existing limitations of the literature can affect the practice. For the motivating example, we rely on a code fragment of the Apache Ant project, in which we identified the incidence 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. This method has four code smells: *Long Method*, *FeatureEnvy*, *Duplicated Code*, and *Long Parameter List*. The developer applied a composite refactoring aiming to remove two code smells (*Long Method* and *Duplicated Code*); thus, they are the target code smells. This method was refactored through the commit `b7d1e9bde44c` [228], represented in Figure 5.1. The left side of the figure shows the method before the composite refactoring, showing the four code smells in the method (highlighted in red). The right side illustrates the method after the composite refactoring.

The smells *Long Method* and *Duplicated Code* are expressed by the several lines of duplicated code addressing the parameter `filterChains` in the method `copyWFS`. The method `copyWFS` had some code fragments duplicated, which are related to processing the parameter `filterChains`, 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 recurring calls to external methods from the class `ChainHeaderHelper`. Finally, the method signature is composed of eight parameters, indicating the incidence of a *Long Parameter List*.

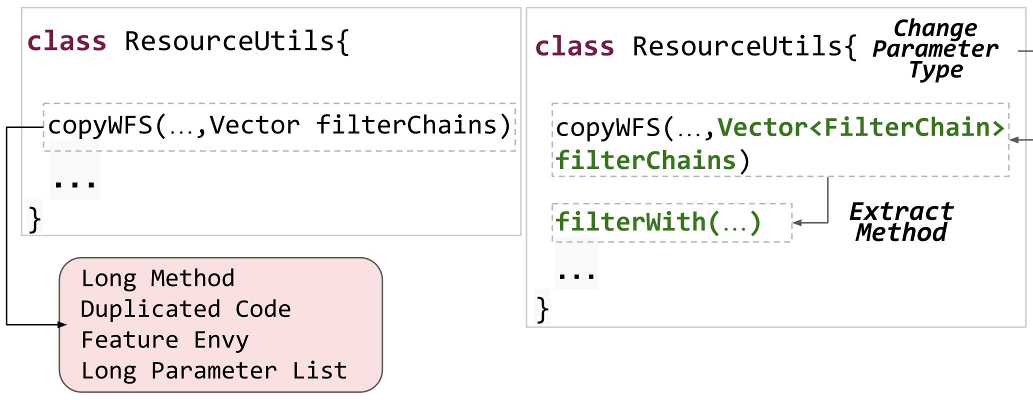

```
class ResourceUtils{
  copyWFS(…,Vector filterChains)
  ...
}
```
```
Long Method
Duplicated Code
Feature Envy
Long Parameter List
```
```
class ResourceUtils{
  copyWFS(…,Vector<FilterChain> filterChains)
  filterWith(…)
  ...
}
```

Figure 5.1: Code Smells Present in the Apache Ant Project on Commit `af74d1f6b882`

In the refered commit, the developer opted for applying a composite refactoring formed of *Extract Method* and *Change Parameter Type*. These refactoring modifications are represented in green in the right side of the Figure 5.1. 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 both *Long Method* and *Duplicated Code* in the class. The change of the parameter type led to the removal of some duplicated lines. Besides, both methods `copyWFS` and `copyWFCT` are using the extracted method now. 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 the method `copyWFS` 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. According to the definition of completeness presented in Section 5.2.2, the composite was complete because the composite removed the two target smells. However, other two code smells remained in the code (*Feature Envy* and *Long Parameter List*). Unfortunately, developers did not became aware on the propagation of *Feature Envy* and the permanence of the *Long Parameter List*. Circumstances such as this are often found in the practice. The existing literature did not provide sufficient knowledge for developers about the occurrence of multiple code smells on the same code element. The literature needs to guide developers on spotting the existence of multiple code smells and performing the complete removal of multiple code smells. Existing solutions should at least alert developers on the propagation or permanence of code smells after the application of a composite refactoring. Without that, developers 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 [77, 227]. For example, Bibiano *et al.* [77] and Chavez *et al.* [10] recommend combining *Extract Method* and *Move Method* for removing *Feature Envy* and *Long Method*. However, these studies not take into account side effects when applying these refactorings in a complex scenario that includes other smell types. For example, 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*, and the new method can inherit bad coding practices, such as the excessive number of parameters and external method calls, from the source method. This issue may especially occurs when the developer is unaware of the incidence of other smell types in the source code analyzed. We can see that the recommendations should to be aligned with the practice. This lack of alignment is because existing recommendations focus on the removal of one code smell, but in real software projects there are multiple instances of code smells on the same code element. Thus, it is necessary to investigate concrete cases in which code smells are introduced, as in the example discussed above, aiming at the improvement of recommendations for composite refactorings.

5.4 Study Settings

Our mixed-method study aims at enhancing and assessing recommendations of complete composite refactorings. To accomplish such goal, we first propose a catalog of composite recommendations (see Section 5.5.3) that fully remove code smells in light of the existing literature [77, 227] and the repository mining of open and closed-source projects. Aiming at conceiving this catalog, we merged the previous empirical-driven recommendations [77, 227] and the current knowledge that we extracted from complete composites applied in 42 real software projects. With respect to the latter, we mined the more frequent complete composites applied in those projects, and improved the existing recommendations (Section 5.2.4). Next, we evaluated our catalog of recommendations according to the developers' perceptions. We describe our research questions (RQs) and the steps of our study in what follows.

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 appears in frequent complete composites (as motivated in Section 5.2.4). Additionally, we conducted an analysis to understand the actual contribution of each refactoring type (e.g., *Extract Method* and *Move Method*) to the introduction or removal of code smells. From these observations, we can enhance existing recommendations and suggestions.

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 [77,227] 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.

5.4.1 Study Steps and Procedures

Figure 5.2 illustrates our study steps and dataset. Study steps are mainly related to the data collection and analyses. The **replication package** is available in [229].

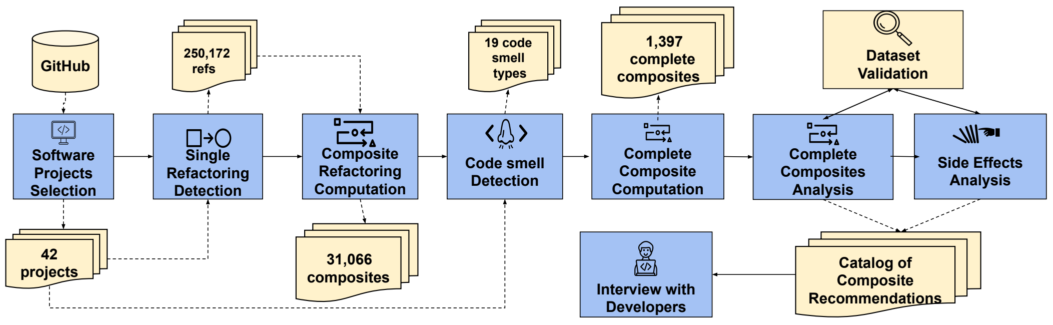

GitHub
250,172 refs
19 code smell types
1,397 complete composites
Dataset Validation
Software Projects Selection
Single Refactoring Detection
Composite Refactoring Computation
Code smell Detection
Complete Composite Computation
Complete Composites Analysis
Side Effects Analysis
42 projects
31,066 composites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

Figure 5.2: 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 [77, 171, 172, 227]. This last criterion was considered because related studies have already confirmed those software projects have occurrences of composite refactorings and code smells. Thus, our dataset can be enriched with by existing validated datasets from these studies.

Step 2: Single Refactoring Detection. For detecting single refactorings applied on software projects, we used the RefMiner 2.0 tool [256] due to its high precision and recall levels (98% and 87%, respectively). The tool also supports a total of 52 refactoring types [173]. In this study, we considered 34 refactoring types that are applied in the code scope of attributes, methods, and classes, similarly to [77]. We selected these 34 refactoring types because

Table 5.3: Classification of Refactoring Types

Fine-Grained Ref. (FGR)		**Coarse-Grained Ref. (LGR)**
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**

a recent study [77] that investigated about complete composites used these types. This study [77] showed that fine-grained refactorings are often applied in complete composites. Thus, we also explored the application of fine-grained refactorings in complete composites. Table 5.3 summarizes the classification of the 34 refactoring types in fine-grained refactorings and coarse-grained refactoring, as defined in Section 5.2.3.

Step 3: Composite Refactoring Computation. For the detection of composite refactorings, we created a Java script to implement the *range-based* heuristic [171]. As mentioned in Section 5.2.1, this heuristic captures refactorings that were applied on a common set of code elements (classes and/or methods). Thus, this heuristic fits well our study goal, as it considers multiple code elements. Besides, the reliability of this heuristic was demonstrated in [76, 77, 171]. More details about the *range-based* heuristic are available in [171]. Our script was developed in Java. Two researchers tested and validated our script.

Step 4: Code Smell Detection. Similarly to Bibiano *et al.* [77] and studies that proposed recommendations of composites [143, 171], we used the Organic tool [48] 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 [8, 17, 48]. 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. Other code

Table 5.4: 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
Divergent Change	DiC	Method that changes often with other ones
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
Large Class	LgC	Too large 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

smells can be propagated between these classes while the existing code smells are removed. Therefore, in this study, we considered code smell types that involve one or more classes. Table 5.4 lists the 19 code smell types analyzed in our study. These code smells are very common and can be removed with refactoring types investigated in this study.

Step 5: Complete Composite Computation. We focused on the complete composites for removing the 19 code smell types (Table 5.4). We then elaborated Table 5.2 which presents the recommended composites for the removal of some code smells according to prior studies [143,171,227]. However, these recommendations are limited to the full removal of four code smell types only. Thus, we aim to extend these recommendations to other code smell types investigated in our study. Additionally, we created a definition for completeness (Section 5.2.2) based on the previous work [77]. We then elaborated a script to collect complete composites according to our definition. This script was tested and validated as detailed in Section 5.6.

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 [77]. 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 what are the frequent combinations and what code smell types are often removed when they are applied. We then extracted recommendations for our catalog based on these recurring combinations of complete composites. A summary of this catalog is presented in Section 5.5.3.

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. For each composite, we analyzed the code snippets touched, other code changes, the commit messages, and pull request discussions in which the complete composites were applied. This in-depth analysis allowed us understanding whether other code changes could have introduced the code smell and if developers are aware of these code smells. The findings of this step helped us to complement our catalog with the side effects of complete composites.

Step 8: Dataset Validation. We randomly selected a sample composed of 36 complete composites from our dataset for validation. Six developers validated if the composites are complete for code smells that were detected. We provided a table for the developers with composites data: refactoring types that form each composite, the project name that was applied 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 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.

Step 9: Interviews with Developers. Firstly, we performed a pilot version with two developers, aiming to improve our interview script. After the improvements on our script, we interviewed 21 developers from different: four from Brazil, two from Austria, one from Canada, one from the USA, and one from the Netherlands. 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.* First, we asked the participants to fill out a *Characterization Form* to collect data about their development experience time, development roles, and familiarity with refactoring. Next, the participants watched a *training video* (15 minutes) about the main concepts about refactoring, code smells, composite refactoring, and their possible side effects. We decided to provide a training video to level up their knowledge about the main concepts regarding our study. Thus, we tried to reduce the bias by focusing on main concepts and presenting theoretical and practical examples.

Activity 2: *Smell identification task.* We asked participants to perform a code smell identification task. To this end, we presented a source code that contains one or more code smell types. This source code was extracted from a project of 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 emphasize that we did not say what code smell types are present in the source code. 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 from the catalog.* We presented to participants the smell definition of our catalog. For each smell, we presented an abstract example allowing then to better understand when the smell happens. We also detailed the reasons and what code elements in our example have the smell (5 minutes). We asked participants if they agree with the definition of the proposed code smells and if the code elements that they analyzed in Activity 2 contain these smells.

Activity 4: *Presentation of the refactoring recommendations from the catalog.* Before presenting our refactoring recommendations, we asked participants to talk about solutions they would apply to remove code smell(s) identified in Activity 2 (ten minutes). After that, we presented the refactoring recommendations of our catalog. For each recommendation, 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. We asked participants if their solution is more complex than our solution and if their solution has some side effects. Finally, we presented the possible side effects of our refactoring recommendations and asked the participants to explain if they thought about these side effects when they selected our refactoring recommendations.

Activity 5: *Apply the follow-up questionnaire.* We apply a questionnaire to check if our code smells are representative. For instance, we asked participants if they have ever observed smelly code similar to the code that was presented. We also asked if participants would use our recommendations in their software projects. Finally, we asked about the positive, negative, and suggestions of the interview.

5.5 Results

We present and discuss our results in this section, showing what are the most frequent combinations in complete composite refactorings, and their side effects. We then report and discuss the developers' perception of our catalog.

5.5.1 The Most Frequent Combinations in Complete Composites (RQ$_1$)

We collected the most frequent combinations applied in composite refactorings (Section 5.4.1). Table 5.5 presents these most frequent combinations of refactoring types. We found that three refactoring types of coarse granularity (LGR), i.e., *Extract Method*, *Move Method* and *Move Class*, are commonly applied with fine-grained refactorings, it also shows 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

Table 5.5: 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%)

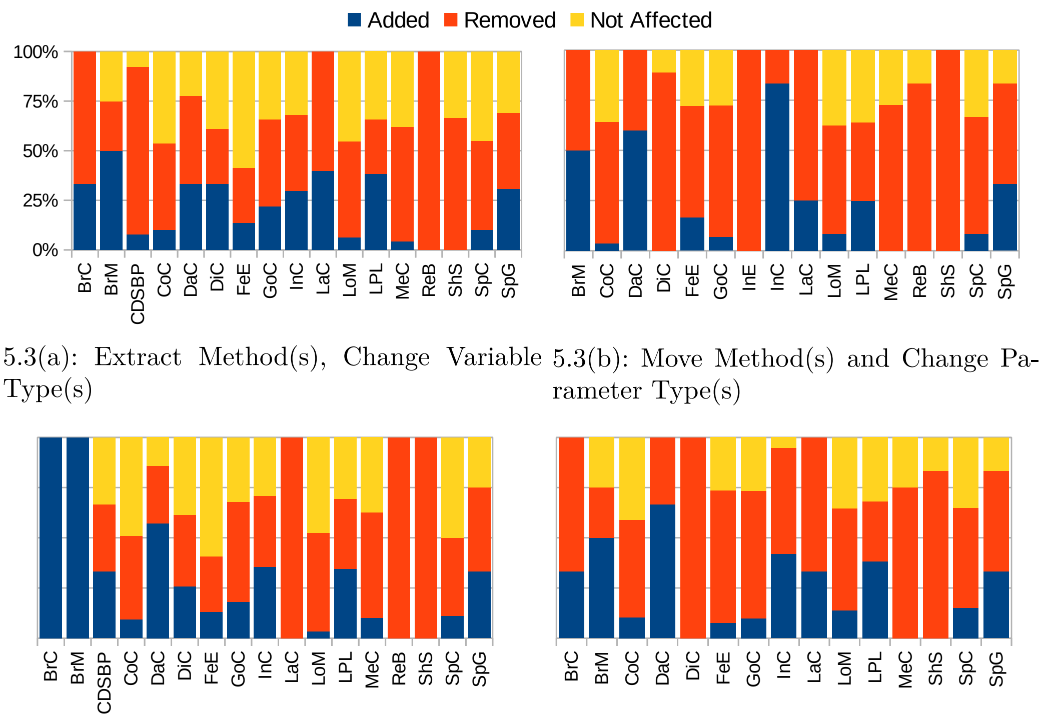
code statements, decreasing lines of code and minimizing the excessive method calls to external classes.

In summary, our results revealed that developers frequently changed the type of the method return (23%) or parameter(s) (22%) 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 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.

We noted that developers often apply a single type of FGR combined with a single type of LGR. Besides, one may see 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 5.5 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 structural modifications at the same time frequently leads to introducing new instances of code smell in the source code [143].

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.

5.5.2 Side Effects of the Frequent combinations in Complete Composites (RQ$_2$)



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

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

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

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

Figure 5.3: Side Effect of Common Complete Composites

Differently, our results reveal the combination *Extract Methods* and *Change Variable Type* introduced about 38% of *Long Parameter Lists*. This contrasting result indicates the need to reach a deep understanding of the side effects caused by each combination in complete composites. In contrast, we understand that previous work generalized the side effects for all combinations that have at least one fine-grained refactoring [77]. However, each recurring combination has a particular side effect.

As detailed in the previous section, developers apply many code modifications to support a simple combination includes large and fine-grain refactorings, generally, such a combination is applied with non-refactoring code changes. This fact, can explain the introduction (50%) of *Brain Methods*, and the prevalence (45%) of *Long Methods*), despite the developer's goal of decreasing the size and complexity of methods when he/she extracts 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 decreased as expected and the *Long Methods* are not affected. This discussion reveals another finding of our study.

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

This previous finding caught our attention because it can indicate that *Long Methods* might implement two or more features from external classes, i.e., these methods can also be *Feature Envies*, as suggested by Bibiano *et* al. [143]. However, there is no empirical evidence about the frequency of methods that are *Long Methods* and *Feature Envy* in conjunction. Thus, we randomly selected 13 software projects, and randomly selected nearly 5,000 commits of each software project to investigate the frequency of methods that have these two code smells at the same commit. Table 5.6 shows the amount of *Long Envious Methods*. We defined a *Long Envious Method* as a method that has excessive lines of code because implements one or more features from external classes. The first column of the table indicates the number of long envious methods per project. The second column of the Table indicates the number of long methods per project. Both percentages in relation to the quantity of smelly methods of each project (the third column). The percentage of smelly methods is in relation the total number of smelly methods.

As we can see in Table 5.6, *Long Envious Methods* is the most frequent code smell type in methods, confirming our initial supposition. This problem seems to occur frequently because the developers are unlikely aware about the joint occurrence of these smells on those methods. Moreover, they might focus on the removal of a single smell only because of the complexity of removing two or more smells with the same composite refactoring.

Extract Methods and *Move Methods* are frequently recommended to remove *Feature Envies* [77,171]. Figure 5.3(d) 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 motions of methods can be related to the introduction of *Long Parameter Lists* (38%) and *Intensive Coupling* (42%). Bibiano *et al.* 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 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, as suggested by the prior work [77]. 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.

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

Surprisingly, *Move Methods* and *Change Parameter Types* can be related to the introduction (83%) of *Intensive Couplings*, as showed in Figure 5.3(b). Generally, when a method is moved, the developer tends to introduce more calls of the methods from other classes. As a consequence, some parameters are also modified because the method uses attributes of other classes. It can explain why *Intensive Couplings* are frequently introduced.

Figure 5.3(c) shows the side effects of complete composites constituted of *Extract Method(s)* and *Change Parameter Type(s)*. In general, we observed that this combination can be recommended to fully remove (49%) *Long Methods*, mainly when these methods have duplicated code. Similarly to the motivating example (Section 5.3), we noted that developers changed the parameters to types that helped to remove statements code, and alongside *Extract Methods*, the *Long Methods* and *Duplicated Code* were removed.

However, our supposition about *Duplicated Code* is limited because Organic tool did not detect *Duplicated Methods*. To better investigate this possible relation between *Long Methods* and *Duplicated Methods*, we used PMD CPD [1]. We created Java scripts to mine duplicated methods through CPD output for this analysis, we consider a duplicated method if it has equals or more than 30 duplicated statements, as considered by CPD rules. We used the sample of commits that was investigated in *Long Envious Method* analysis. We then analyzed the frequency of duplicated methods that are long methods at the same commit. We called these methods of *Long Signed Clone*.

Table 5.5.2 presents the amount of *Long Signed Clones* per project, similarly to the Table 5.6. Differently from *Long Envious Methods*, the *Long Signed Clones* are not frequent in relation to smelly methods. However, in some projects the number of *Long Signed Clones* is interesting, for example, Hystrix, Geoserver and Jitwatch are projects that have many duplicated and long methods at the same commit. We consider that *Long Signed Clone* is interesting because we manually observed that the methods are long and duplicated because one or more parameter cause the repetitive and excessive

[1]<https://pmd.github.io/latest/pmd_userdocs_cpd.html>

lines of code, and frequently, developers applied *Extract Method* and *Change Parameter Types* to fully remove *Long Signed Clones*. We then can collect a recommendation from this empirical knowledge.

Table 5.6: Long Envious Method per Project

Project	LEM	LM	Smelly Methods
activiti	1665 (67.96%)	785 (32.04%)	2450 (1.99%)
bytebuddy	1512 (51.78%)	1408 (48.22%)	2920 (2.37%)
checkstyle	733 (22.57%)	2514 (77.43%)	3247 (2.64%)
geoserver	6390 (56.45%)	4930 (43,55%)	11320 (9.20%)
hystrix	40034 (78.65%)	10868 (21.35%)	50902 (41.35%)
javadriver	2589 (66.08%)	1329 (33.92%)	3918 (3.18%)
jitwatch	8974 (38.52%)	14326 (61.48%)	23300 (18.93%)
materialdialogs	834 (33.27%)	1673 (66.73%)	2507 (2.04%)
materialdrawer	1427 (50.95%)	1374 (49.05%)	2801 (2.28%)
mockito	1577 (52.31%)	1438 (47.69%)	3015 (2.45%)
quasar	5066 (64.31%)	2811 (35.69%)	7877 (6.40%)
restassured	194 (45.54%)	232 (54.46%)	426 (0.35%)
xabberandroid	4305 (51.15%)	4112 (48.85%)	8417 (6.84%)
Total	**75300 (61.17%)**	**47800 (38.83%)**	**123100 (100.00%)**

Table 5.7: Long Signed Clone per Project

Project	LSC	DuM	LeM	Smelly Methods
activiti	888 (2.23%)	37285 (93.83%)	1562 (3.93%)	39735 (5.27%)
asynchttpclient	0 (0.00%)	577 (92.77%)	45 (7.23%)	622 (0.08%)
bytebuddy	569 (1.23%)	43422 (93.70%)	2351 (5.07%)	46342 (6.15%)
checkstyle	0 (0.00%)	2932 (47.45%)	3247 (52.55%)	6179 (0.82%)
geoserver	1795 (2.71%)	54937 (82.92%)	9525 (14.38%)	66257 (8.79%)
hystrix	22365 (8.45%)	213825 (80.77%)	28537 (10.78%)	264727 (35.11%)
javadriver	709 (1.46%)	44613 (91.93%)	3209 (6.61%)	48531 (6.44%)
jitwatch	1423 (2.36%)	37040 (61.39%)	21877 (36.26%)	60340 (8.00%)
materialdialogs	52 (0.44%)	9240 (78.66%)	2455 (20.90%)	11747 (1.56%)
materialdrawer	1097 (2.96%)	34296 (92.45%)	1704 (4.59%)	37097 (4.92%)
mockito	12 (0.06%)	17277 (85.14%)	3003 (14.80%)	20292 (2.69%)
quasar	1241 (1.87%)	58349 (88.11%)	6636 (10.02%)	66226 (8.78%)
restassured	2 (0.01%)	27764 (98.49%)	424 (1.50%)	28190 (3.74%)
retrolambda	0 (0.00%)	2590 (96.46%)	95 (3.54%)	2685 (0.36%)
xabberandroid	1383 (2.51%)	46654 (84.72%)	7034 (12.77%)	55071 (7.30%)
Total	**31536 (4.18%)**	**630801 (83.66%)**	**91704 (12.16%)**	**754041 (100.00%)**

5.5.3 Evaluation of the Proposed Catalog (RQ$_3$)

Based on the common combinations in complete composite refactorings found (Section 5.5.1), we created a catalog of composite recommendations [230]. Previous composite recommendations focused on the removal of a single code smell, but our catalog is different. We provided four recommendations that remove up to three code smell types, grouped in two new smell types. Although our quantitative analyses (Section 5.5.2), *Long Envious Method* is a common smell, and *Long Signed Clone* is a smell that we observed a composite pattern for the fully removal of this smell, the application of *Change Parameter Type* and *Extract Method*. In that way, we created composite recommendations for the removal of these two smells in our catalog. These composite recommendations were obtained through our quantitative data.

From our quantitative results, we observed that *Extract Methods* and *Move Methods* are commonly applied for the removal of *Long Envious Methods*. Our catalog suggests two mechanics: First the extraction and then the moving (first mechanics), or the extraction and motion at the same time (second mechanics). Regardless of the similar refactoring result, we proposed these mechanics because developers can move two or more methods that were extracted to different classes using the first mechanics. We believe that it is more complicated using the second mechanics. 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)*. We interviewed 21 developers to characterize the developer acceptance of our catalog (RQ_3), as described in Section 5.4. Five developers analyzed methods with *Long Envious Method*, and four developers evaluated methods with *Long-signed Clone*.

Code Smell Analysis: As explained in Section 5.4.1, 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. We observed that 12 (57%) out of 21 developers detected all code smell types before we reveal them. For *Long Envious Method*, four out of five developers said that the methods had *Long Method* and *Feature Envy*. In the case of *Long-signed Clone*, two out of four developers mentioned that the method has both smells types. The other two developers only mentioned that the method has *Duplicated Code*.

We presented them the definition of the code smells. Then, we ask if 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. One developer suggested improving the definition of *Long-signed Clone*, given he/she thought the code duplication was related to a long list of parameters. The developer justified that the confusion was because the source code had *Duplicated Code* and *Long Parameter List*, and the code was duplicated because other methods had the same long list of parameters. However, when we presented the corresponding definition and example, the developer understood that the incidence of *Long-signed Clone* was not necessarily due to a long list of parameters.

Continuing with the interview, we ask them whether the source code actually had the detected code smell type. Twelve (57%) developers detected the two code smell types, while nine (43%) developer identified only one code smell type. This 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 he/she had difficulty 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*. From this answer, 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.

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*. About the *Long Envious Method*, four (36%) out of 11 developers agreed to apply the first mechanics. These developers reported that the first mechanics is more interesting for junior developers. This may be explained because junior developers have little familiarity with the source code and they have difficulty knowing what code elements may be modified by composite refactorings. According to the developers, it is more indicated that junior developers apply each code transformation at the time to analyze what code elements can be affected. This mechanics also helps to test the source code after the application of each refactoring and verifies if that refactoring did not change the software behavior. Other developers reported that applying one 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 code familiarity increases the awareness of the side effects when applying composite refactorings. Based on that, we have our next finding.

Finding 4: Applying each refactoring by time is better for junior developers because it facilitates to the analysis of side effects.

Side Effects: Seven (33%) out of 21 developers did not indicate to be aware of the side effects. Surprisingly after we mentioned the side effects, these developers reported that their own solutions could have worse side effects than our solution, 18 (85%) out of 21 developers agreed that their solutions could have worse side effects. From these answers, we can observe that some developers, that do not worry about side effects, can apply refactorings that degrade software quality. In addition, we confirm that developers need a guide to alert them about the side effects of composites because in some cases, their solutions are not the best solution to fully remove code smells.

Finding 5: Developers that do not report side effects tend to apply composite refactorings that can cause side effects.

Refactoring Complexity vs Completeness. We ask the developers whether their solutions are more complex than our recommendations. Some developers choose certain combinations because the refactoring complexity could be lowered, regardless of their side effects. An example is our motivating example (Section 5.3). For the complete removal of the four code smells in the methods `copyWFS` and `copyTFS`, the developer would need to apply *Extract Method*, *Extract Class* and *Move Method*. The *Extract Method* and *Extract Class* would remove *Duplicated Code* and *Long Parameter List*. As consequence, the size of methods would be decreased. *Move Method* would move envious code to the appropriate classes. Thus, the four code smells would be fully removed.

However, the code author and our subjects opted to apply *Extract Method* and *Change Parameter Type*, propagating *Feature Envies*. The subjects justified that these refactorings modify a few code elements and have less complexity. Developers also mentioned that the application of *Extract Class* solved the code smells, but it is necessary to modify many code elements, and these modifications could introduce other code smells. Another example is in the commit `c18fc2b` [231] of the Netty project. The developers applied *Move Attribute* and *Move Method* refactorings on the class `AbstractScheduledEventExecutor`, removing *Feature Envies*. However, the class `AbstractScheduledEventExecutor` is a *God Class*, and these refactorings made this smell become even worse. We then conclude our catalog can help developers in these situations by alerting about side effects of complete composite refactorings.

Finding 6: Some developers prefer to apply composite refactorings that are less complex, even when composites can cause side effects, which are left for later removal with other composites.

5.6 Threats to Validity

Construct Validity: Relying purely on automated detection tools may be risky for identifying code smells and refactorings [257]. 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 5.4.1). RefMiner is also beneficial because it was designed to ignore squash commits [194]. 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.* [171] for detecting composite refactorings, combined with the definition of complete composites proposed in [76, 77].

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 [48]. 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 5.4). 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.

5.7 Conclusion

Given the limitations of existing catalogs of refactorings, we presented an enhanced catalog of complete composite refactorings. We conducted a large-scale study on 42 software projects, collecting 1,397 complete composites that were the base to create our catalog. We assessed our catalog with 21 developers to have a practical view of our enhanced recommendations. 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. Regarding the catalog assessment, all interviewees agreed with the recommendations to remove smells. In addition, most of them stated that, without knowing our recommendations, their refactoring solutions could induce severe side effects.

Our main contribution is a set of the recommendations derived from the practice, which includes four complete composites to remove code smells. These recommendations can guide developers to perform composite refactorings to improve code internal quality, while alerting them about the possible side effects. As 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.

6 Exploring the Automatic Recommendation of Composite Refactorings

In our catalog derived from the practice (Chapter 5), we proposed new types of code smells, such as *Long Envious Methods*, that represent 61% of smelly methods, four composite recommendations to remove these types of smells, and their possible side effects. Twenty one developers evaluated our catalog. As the main result, most developers (85%) reported that their solutions could have worse side effects without our catalog recommendations. We then confirmed that an automated refactoring support is also necessary apply beneficial composite refactorings, as developers may struggle on tailoring our catalog recommendations to the context of their source code. Inspired by that, we explore existing automated approaches for recommendations of composite refactorings, as mentioned in our RQ_3 (Section 1.3.1). As described in Chapter 1, we considered a complete composite beneficial when it fully removes the target smell without inducing side effects.

Automated recommenders of single refactorings based on search-based algorithms have recently presented promising results [79, 146], as refactoring can be seen as an optimization task. Based on that, in this Chapter, we explore the use of search-based algorithms to recommend composite refactorings. For that, we extended an automated tool to recommend composite refactorings and alert developers about possible side effects. Then, we performed a survey with 10 developers to explore how existing search-based approaches for recommendations of composites can be improved for the beneficial removal of code smells, addressing then our RQ_3 (Section 1.3.1). The content of this chapter is going to be submitted to an international conference.

6.1 Introduction

Along the studies of our previous chapters, we perceived that developers often apply incomplete composite refactorings manually (Chapters 3, 4, and 5), and they often are not aware of the side effects of composite refactorings (Chapters 4 and 5). In these studies, we observed that a refactoring support, such as a catalog, helps developers to reason about how to fully remove multiple code smells with a composite refactoring and their possible side effects. In addition, developers reported difficulties to adapt our catalog recommendations to the context of their source code. In that way, we confirmed that many times developers need an automated refactoring support to provide beneficial removals of code smells. In light of that, we then explored how existing automated approaches for recommendations of composites can be improved for the beneficial removal of code smells.

In the literature, some studies propose automated approaches for refactoring recommendations [146, 222, 242]. These approaches use search-based algorithms to generate suggestions for composite refactorings. These studies reported promising results on the use of search-based algorithms for the recommendations of refactorings [146, 222, 242], given that refactoring can be seen as an optimization task [30, 233]. Search-based algorithms generate solutions to solve an optimization problem [30]. The solutions are optimized based on objective functions, which are also known as fitness functions. In case of recommendations for composite refactorings, the problem is the recommendation of composite for the beneficial removal of code smells. The desired solutions are composite refactorings that fully remove code smells without side effects. Fitness functions here can be defined as the full elimination of target code smells. However, there is a lack of empirical evaluations concerning whether existing search-based algorithms can leverage the completeness of composite refactorings for the beneficial removal of target code smells, mainly from the perspective of developers.

For mitigating this limitation, we performed an exploratory study investigating existing approaches of search-based algorithms for recommendations of composite refactorings. For that, we extended OrganicRef, a recent recommender of composite refactorings [244]. OrganicRef uses search-based algorithms to recommend composite refactorings. We called this extension of REComposite. Originally, OrganicRef has limitations, such as not recommending common refactoring types, e.g., *Extract Method*. In addition, the tool does not detect popular code smells, e.g., *Long Method*. Aiming to address these limitations, we extended the recommender generating a new version called REComposite. We implemented the recommendation of *Extract Method* in REComposite. It is because this refactoring type can be recommended together with the *Move Method*, previously implemented in the tool. Those refactoring types can be recommended to remove smells such as *Feature Envy*, *Long Method*, and *Long Envious Method*, as suggested in our catalog (Chapter 5). We then developed the detection of *Long Method* and *Long Envious Method* in REComposite, *Feature Envy* was originally detected. After that, we performed a survey with 10 developers to explore the recommendations of REComposite in terms of completeness and side effects, generating knowledge on how existing approaches, namely search-based algorithms, can be improved to provide complete composite refactorings.

In the next sections, we detailed the concepts related to SBSE, the fitness functions, and our results. This chapter is structured as follows. Section 6.2 describes the main concepts of this study. Our study settings are detailed in Section 6.3. We explained the survey procedures in Section 6.3.4. Section 6.4 presents the results of our survey with developers. The threats to validity and conclusion of this study are presented in Sections 6.5 and 6.6, respectively.

6.2 Background

6.2.1 Search-Based Software Engineering (SBSE)

Several problems in Software Engineering can be measured by a set of software metrics, such as measuring the coverage of tests to select which tests may be applied. Based on that, authors have used Search-based techniques to solve Software Engineering problems - i.e., Search-Based Software Engineering (SBSE) [234]. Search-based techniques generate several solutions to solve a problem and use fitness functions to select the optimal solution.

For the generation of solutions, techniques of SBSE use metaheuristic algorithms such as genetic algorithms (GAs) and simulated annealing [234]. For the evaluation of solutions, a fitness function is defined to assess the quality of the generated solutions based on the goal that may be achieved [234]. An example in the context of tests is to prioritize tests [281, 282]. In that case, a SBSE algorithm generates several tests and a fitness function verifies which tests have the high prioritization, thus they are the optimal tests. A fitness function can be mono-objective or multi-objective. A function is mono-objective when it has a single objective to assess solutions and find tests with high prioritization. A multi-objective function has two or more objectives to evaluate solutions, for example, finding tests with high prioritization and minimizing the number of tests [281].

6.2.2 Search-Based Algorithms

This section explains about the search-based algorithms that are investigated in this study. These algorithms are commonly used in recent Search-

Based Software Engineering studies [145, 222, 242, 244]. The first algorithm is the Simulated Annealing (SA) [240]. MOSA stands for Multi-Objective Simulated Annealing, which is a multi-objective version of the Simulated Annealing algorithm [240–242]. NSGA-II stands for Non-dominated Sorting Genetic Algorithm II, another multi-objecive search-based algorithm used by OrganicRef for refactoring recommendations [239].

Simulated Annealing is a general local search algorithm that seeks to minimize a single objective. The underlying idea of the method is to allow non-improving moves because they can help escape from local minima [240]. Both SA and MOSA work similarly, from an initial random population. From a initial random candidate, they calculate what is the next optimal candidate based on their objective(s); unique objective for SA e multi-objective for MOSA. This iteration continues into the best candidate is found on the local search.

Figure 6.1 illustrates a generic example of the Simulated Annealing algorithm. In this case, each blue cycle represents a solution in a search space (x, y), where $s1$ is the initial solution that was randomly chosen, and sf is the final solution according to the fitness function. In each iteration, the algorithm selects the closest solution from the current solution and checks whether this solution has a lesser or equal value than the final solution. If the solution has a lesser value, the iteration continues. Otherwise, the final solution has been found.

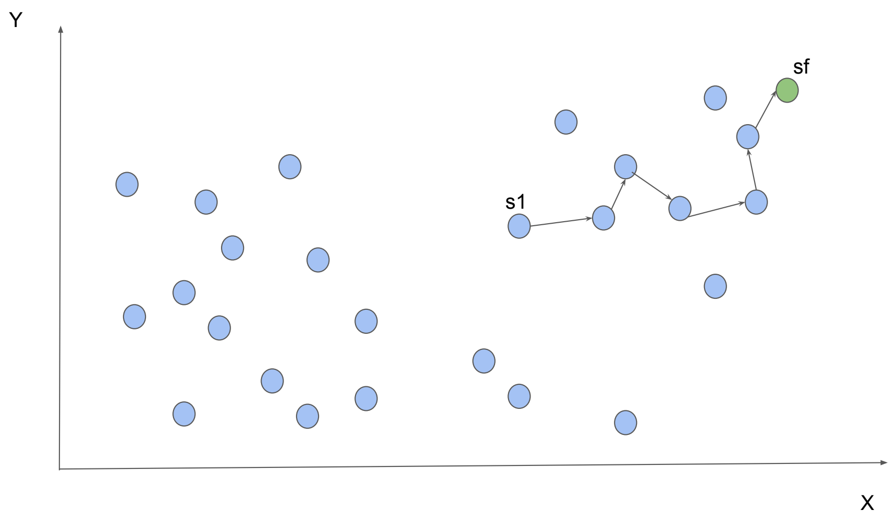


Figure 6.1: Example of Simulated Annealing

Non-dominated Sorting Genetic Algorithm II is global search algorithm. The first version of this algorithm (NSGA) finds the optimal solution verifying candidate by candidate, the worst case of optimization algorithms [30]. The NSGA-II authors modified the algorithm to consider the crowding-distance, that is a local search in which the population has their objective function values ascending in order of magnitude [239]. Each iteration, local search reduce the crowding-distance are until finding the solution. Figure 6.2 depicts a generic example of the NSGA-II algorithm. In this case, each blue cycle represents a solution in a search space (x, y), and the green cycle represents the final solution based on the fitness function. The algorithm locates the crowding-distance area by observing which solution values are increasing. In each iteration, the crowding-distance is decreased to find the final solution.

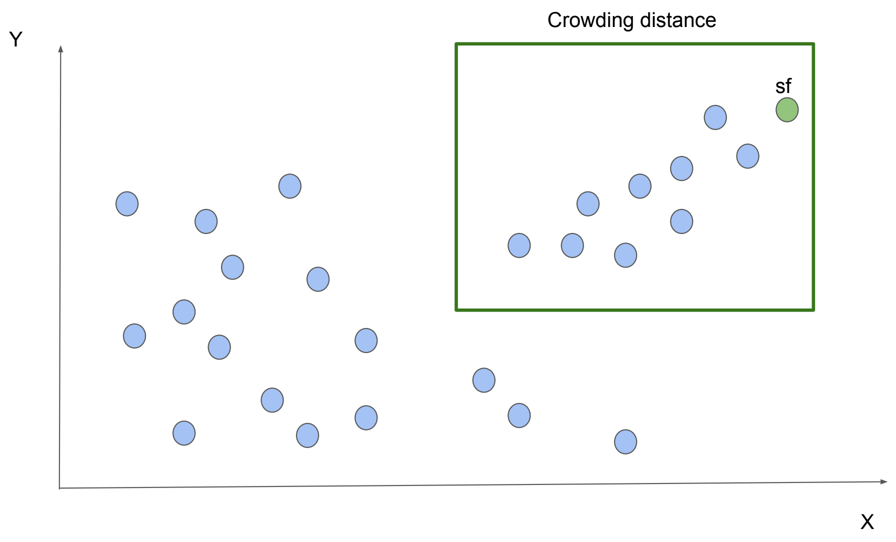


Figure 6.2: Example of NSGA-II

6.2.3 Search-Based Refactoring (SBR)

A common and challenging problem to solve in SE is selecting the optimal refactoring to improve code quality [30, 284, 285]. A previous study presented the idea of formulating the refactoring task as a search problem in the space of alternative solutions, generating a set of refactorings [233]. Harman and Tratt were the first authors to apply SBSE for refactoring, using the term Search-Based Refactoring (SBR) [30]. Harman and Tratt initially used mono-objective functions for search-based refactoring techniques [30]. However, adopting mono-objective functions is problematic when solving refactoring problems because generating optimal refactoring sequences requires multiple metrics. From then on, many researchers have investigated SBSE algorithms to solve the refactoring problem. The literature has investigated multi-objective functions to suggest optimal refactorings with objectives such as increasing cohesion [79, 145, 222] and minimizing the number of modified files [43].

Selecting a single refactoring is challenging for SBSE, finding composite refactorings is even more challenging. Therefore, the SBR problem has increased because researchers need to determine the best combination of refactorings to improve code quality. In other words, finding the optimal composite refactoring is currently challenging. Multi-objective functions for selecting optimal composite refactorings can be based on minimizing the number of refactorings or maximizing the number of code smells removed. However, composite refactorings are complex and can have side effects on source code. Thus searching for optimal composite refactoring using multi-objective functions remains challenging. In the next section, we present some existing studies that propose multi-objective functions to suggest optimal composite refactorings.

6.2.4 Related Work

Several studies proposed search-based techniques to recommend refactoring solutions. Alizadeh et al. [222] propose an intelligent refactoring bot as a GitHub app that can be easily integrated into any project repository on GitHub. This intelligent refactoring bot was called RefBot. The bot analyzes the files changed during that pull request(s) to identify refactoring opportunities using a set of quality attributes then it will find optimal composite refactorings to fix the quality issues if any. RefBot recommends the optimal composite refactorings through an automatically generated pull request. The developer can review the recommendations, and their impacts in a detailed report and select the code changes that he wants to keep or ignore [222]. After this review, the developer can close and approve the merge of the bot's pull request [222]. They used NSGA-II to generate refactoring solutions. In their fitness function, they used six QMOOD quality attributes to measure the impact of a composite refactoring on the software project [222]. These six attributes are "Reusability", "Flexibility", "Understandability", "Functionality", "Extendibility", and "Effectiveness". They compared their approach with other multi-objective approaches proposed by Ouni et al. [146].

Alizadeh et al. [222] surveyed 25 developers, asking for their opinion about the meaningfulness of the composite refactorings recommended by their technique and by the automated refactoring competitive technique. They evaluated the beta version of RefBot in one of their industrial partners during three business days (with six developers involved). During this period, they checked the ability of RefBot to select relevant refactorings for the recent pull requests introduced by the programmers during their daily activities.

On the survey with 25 developers, their results [222] indicated the percentage of meaningful recommendations is much better for RefBot compared to Ouni et al. [146] (94% for RefBot and 66% for Ouni). The percentage of refactorings that participants believe must be applied is significantly higher for Refbot as well (77% for RefBot 15% for Ouni). In the industrial experience with six developers. The six subjects confirmed that they feel more comfortable in applying composite refactorings due to the high level of control proposed by the bot to review the generated pull request, which gives them more confidence and trust in the tool [222]. This work shows that NSGA-II is an interesting algorithm to recommend composite refactorings, in terms of meaningful recommendations [222]. However, this study is limited because they do not evaluate the completeness and side effects of their recommendations of composite refactorings. This lack of evaluation can misguide developers, because the programmers can accept the pull requests recommended by RefBot, but these recommendations can not sufficient to fully remove code smells, and worse than it, the recommendations can bring side effects for the software projects.

The other related work proposes OrganicRef [244]. A recommender that uses three multi-objective approaches to generate optimal composite refactorings [244], aiming at the removal of code smells. These search-based algorithms are SA, MOSA [240–242], and NSGA-II [239], which were explained in Section 6.2.2. These algorithms have shown good results for refactoring recommendations, according to previous studies (e.g., [242] and [222]). OrganicRef builds an initial population based on the combination of aforementioned heuristics in a given context [244]. This approach differs from other approaches that usually build a random initial population. The advantage of the OrganicRef approach is that the optimization already starts with "good" solutions. Therefore, the optimization effort may be lower.

In OrganicRef, each candidate of an optimal composite refactoring is represented by a vector of refactorings [244]. Each vector position contains the refactorings applied to a context's element. Each element may contain zero to many recommended refactorings. The authors opted for this non-conventional representation because it allows OrganicRef to focus the optimization effort in the selected context [244]. This tool generates composites from three refactoring types: *Extract Class*, *Move Method*, and *Move Field*. OrganicRef detects seven types of code smells, three types from method scope and four types from class scope, as detailed in Table 6.1

Table 6.1: Code Smell Types detected by OrganicRef

Code Smell Type	**ID**	**Definition**
	Method scope	
Complex Method	CoM	Method overloaded with software features
Dispersed Coupling	DsC	Method that calls too many methods
Feature Envy	FeE	Method “envying” other classes’ features
	Class scope	
Complex Class	CoC	Too complex software features in a class
God Class	GoC	Too many software features in a class
Large Class	LgC	Too large class
Lazy Class	LaC	Too short and simple class

In OrganicRef study [244], they performed a qualitative evaluation with four developers to assess optimal composite refactorings recommended by the MOSA and NSGA-II strategies. They reported that SA was ignored in this evaluation because the algorithm did take time to generate results. Thus, they did not collect SA recommendations in time to the evaluation. In their results, developers observed that MOSA recommendations have a significant meaningfulness. Participants observed the recommendations of *Extract Class* tend to suggest the extraction of fields and methods related to the same feature. In the case of NSGA-II, they reported that composite refactorings constituted of *Extract Class* and *Move Methods* tend to remove *God Classes*. However, there is no empirical evidence whether these recommendations fully removed code smells and avoid side effects. In addition, they do not evaluate SA recommendations to explore the perception of developers on composite refactorings generated by SA algorithm. Besides, OrganicRef does not have the implementation of known refactorings such as *Extract Method*, and did not detect known code smells like *Long Method*. In summary, this previous study did not evaluate the completeness and side effects of composite recommendations according to developers’ perceptions.

6.3 Study Settings

6.3.1 Study Goal

We aimed to (i) explore search-based algorithms for the recommendation of composite refactorings through the extension of an existing recommender, (ii) collect the developers’ perceptions about the completeness and side effects of automated recommendations of composite refactorings, and (iii) provide a list of lessons learned for researchers and tool builders of automated recommenders of composite refactorings that use search-based algorithms.

6.3.2 Research Questions

RQ$_1$. To what extent are the REComposite solutions complete for developers?

Completeness is a relevant goal for a refactoring recommender because it is expected the optimal composite refactorings fully remove target code smells. However, the literature is limited to investigating to what extent existing approaches are complete to eliminate code smells, and analyze trade-offs between (in)completeness and side effects. In this research question, we aimed to address this gap. We performed a survey with developers to explore whether the REComposite solutions completely remove code smells according to developers' perceptions. We then asked it for developers that implemented the source code and for developers that not implemented the source code.

RQ$_2$. What are the side effects of REComposite solutions according to developers' perceptions?

Existing recommenders of composite refactorings did not show possible effects for developers. We then adapted REComposite. Our adaptation shows the side effects of its recommendations. However, it is possible that the tool does not show all side effects. Moreover, each search-based algorithm can induce different side effects for the recommendations of composite refactorings. Addressing these limitations, we asked to developers about possible side effects of REComposite solutions that were not showed by the recommender.

6.3.3 Study Steps

We executed the steps described below to explore the recommendations of REComposite tool. The replication package of this study is available on [283]

Step 1: Long Method Identification – We implemented the *Long Method* detection according to the detection strategies from the Organic 2.0 tool [48]. We used these strategies because it shows high precision according to prior studies [8, 77, 169].

Step 2: Long Envious Method Identification - *Long Envious Method* is a method that is long and excessively uses data from external classes. To identify this code smell, we adopted two strategies. The first strategy utilized the detection used by the Organic tool [48] for *Long Method* and *Feature Envy*. Thus, if a method that has both code smells it is considered to be a *Long Envious Method*. However, other code metrics can have been ignored to

Table 6.2: Code Smell Types detected by REComposite

Code Smell Type	**ID**	**Definition**
	Method scope	
Complex Method	CoM	Method overloaded with software features
Dispersed Coupling	DsC	Method that calls too many methods
Feature Envy	FeE	Method "envying" other classes' features
Long Method	LoM	Method with excessive lines of code
Long Envious Method	LeM	Long method that implements one or more features from external classes
	Class scope	
Complex Class	CoC	Too complex software features in a class
God Class	GoC	Too many software features in a class
Large Class	LgC	Too large class
Lazy Class	LaC	Too short and simple class

identify this new code smell due to Organic limitations. In that way, our second strategy is based on code metrics related to the code coupling of the method. A recent study [259] demonstrates that *Long Envious Methods* tend to have a coupling twice as large than isolated *Long Methods*. For example, whether a *Long Method* has a code coupling metric with value 5, a *Long Envious Method* has a value equals or greater than 10. Therefore, we used the code coupling to identify *Long Envious Methods*. Table 6.1 shows the code smells that are identified by REComposite. As we can see in the Table mentioned above, REComposite can detect *Long Method* and *Long Envious Method*, as implemented in this study.

Step 3: Extract Method Implementation - We implemented *Extract Method* using two approaches. The *Method Complexity Approach* identifies statements that can be extracted based on code metrics related to complexity, such as *Cyclomatic Complexity*. The *Envious Method Approach* identifies statements that can have *Feature Envy*, these statements are related to an external class that is used many times. These statements include parameters, local variables, and attributes of the original class. For this approach, we evaluated if the analyzed method has *Feature Envy*. If yes, we collected what class is envied, and computed what statements are related to this envied class. We then suggested an *Extract Method* to separate this envious code to another method. In that way, REComposite suggests *Extract Methods* differently to OrganicRef tool.

Step 4: Assessment of Composite Recommendations - We surveyed 10 developers to evaluate the recommendations of REComposite. Firstly, each developer completed a Characterization Web Form. Secondly, the participant watched a training video explaining the main concepts of this study. This training is necessary to align the knowledge of participants. Thirdly, we presented a webpage [1] with two optimal composite refactorings of the same project, each composite was generated by a different search-based algorithm. We asked each participant about the meaningfulness, completeness, and potential side effects of each recommended composite refactoring. They also rated the usability of the REComposite and offered suggestions for improvement. Each survey was recorded to facilitate the analysis of the answers of the subjects. Further details on the sample and procedures of our survey can be found in the next section.

6.3.4 Survey Procedures

In this section, we detailed the procedures of our survey. We then described how was (i) the selection of our sample of recommendations, (ii) the presentation of the sample of composite refactorings for developers, and (iii) analyze of the recommendations of composite refactorings.

Selection of the Sample – We selected a sample of 20 composite refactorings that were recommended for methods that have *Long Envious Methods* to assess what refactoring types are suggested to remove them. The composites are formed of, at the maximum, three refactorings to avoid the tiredness of developers during the analysis. We prioritized methods of classes that are relevant to the system, ignoring classes of tests and exceptions. In addition, we selected classes that have smells like *Large Class*, *Complex Class*, *God Class* and *Dispersed Coupling*, mainly to observe if the suggested composite help to fully or partially remove these smells of class level.

In some cases, the search-based algorithms only recommended a single refactoring to these relevant classes. In those cases, we keep them to analyze the completeness of these isolated refactorings in comparison to composite refactorings. As observed in the Chapters 4 and 5, the refactoring types recommended to fully remove *Long Envious Methods* are *Extract Method* and *Move Method*. We then aimed to verify if these refactoring types are suggested by search-based algorithms. Each developer evaluated the same class per project, independently of the search-based algorithm. For example, participant P1 evaluated the class `DefaultValue` from the software project bytebuddy, and assessed a composite generated by NSGA-II to remove code smells of this class, and another composite generated by SA to also remove code smells of the same class.

[1]REComposite Webpage Survey, <https://anacarlagb.github.io/recomposite-web/>

Table 6.3: Participants' Characterization Data

Characterization	**Years of programming experience**	**Number of developed projects**
Median	8	7
Average	7	7,8
Max	12	16
Min	2	5

Characterization of Developers – Table 6.3 presents the characterization of the survey's participants. As we can see, the developers have a considerable level of development experience in terms of programming years and the number of developed projects. Along the survey, they mentioned that have familiarity with definitions of refactoring and code smell. Table 6.3.4 shows (i) the participant identifiers, (ii) algorithms that were assessed for each developer participant, (iii) the software project that was analyzed, and (iv) the project source, i.e, the source of a project concerning the developer. We considered an external project when the participant does not implement the project that will be analyzed. An original project is when the participant implements the project that will be analyzed. It is relevant to have perceptions of developers that implement or not the analyzed software projects because we can observe if the recommendations are clear for different populations, such as developers that are new to the team and do not have familiarity with the software project. Note that ten participants analyzed composite refactorings generated by NSGA-II, five participants analyzed composites generated by SA, and other five developers assessed composites created by MOSA. As SA and MOSA are based on a similar approach (Section 6.2.2) and we have a limited number of participants, we then divided SA e MOSA between the other ten developers.

Visualization of the Recommendations of Composite Refactorings - Aiming a friendly visualization of composite recommendations for our survey, we built a web page[2] with the recommendations of REComposite. Previously, the output of OrganicRef was a JSON file only. From the web page, the developer can see the detected code smells, the recommended composite refactoring to remove the code smells, and the possible side effects of the recommended composite refactoring. As the previous output of OrganicRef was a JSON file, we then needed to manually select the fields that are shown on the web page. We selected the (i) fields related to the target code smells, (ii) the names of code elements that need to be refactored, (iii) the refactoring types that form the recommended composite refactoring, (iv) and the side effect of

[2]<https://anacarlagb.github.io/recomposite-web>

the composite. In that way, developers can easily identify the code smells in their code and understand the recommended composite refactoring. They can also assess the potential side effects of the refactoring and how it can impact the code. Currently, this web page was developed to facilitate our survey, but in the future, this web page can be integrated to REComposite source code to facilitate the visualization of its recommendations.

Analyze of the Recommendations of Composite Refactorings – The first step for the analysis of the recommendations of composite refactorings is to understand the source code that will be analyzed. In this step, the researcher that conducted the survey explain for the developer about the (i) software project domain, in case of external projects, (ii) functionality of the smelly code elements, (iii) code smells that were found in the code elements. After this explanation, the participant navigate on the source code to analyze it. In the case of original software projects, the developer accesses the code in his/her own machine. Otherwise, the subject accesses remotely the source code by one extension of the Visual Studio Code (VSC).[3] For that, the developer then needs to install on his/her machine the Visual Studio Code and an extension called Live Share[4] on the VSC tool. This extension enables participants to remotely access another Visual Studio Code with the software project to be analyzed. The installation of these tools is simple, and some developers may already be familiar with them.

The second step is the validation of the target code smells. The developer evaluates each instance of code smell and analyzes whether s/he agrees to the detection of the code smell, justifying his/her decision. Third step is the assessment of Meaningfulness. In this study, we defined refactoring as meaningful if it is appropriate for the given context and the nature of the code transformation is consistent with the code context. For example, recommending the *Extract Method* refactoring for a method too short (with few statements) would not make sense and, therefore, this refactoring would not be considered meaningful. The evaluation of meaningfulness is relevant because by taking into account the context of the code and the specific refactoring operation being performed, the approach can identify which refactorings were the most appropriate and effective at improving the quality of the code while minimizing the risk of introducing new issues or breaking existing functionality.

The fourth step, the developer analyzes whether the composite refactoring is complete to remove the target code smells. Aiming to mitigate the threat of the subjectivity on the concept of the completeness, we asked to the

[3]<https://code.visualstudio.com/>

[4]<https://marketplace.visualstudio.com/items?itemName=MS-vsliveshare.vsliveshare>

Table 6.4: Survey Information

Part. ID	Algorithm 1	Algorithm 2	Project	Project Source
P1	NSGA-II	SA	bytebuddy	external project
P2	NSGA-II	SA	bytebuddy	external project
P3	NSGA-II	SA	infproject	original project
P4	NSGA-II	MOSA	powergym	original project
P5	NSGA-II	MOSA	hystrix	external project
P6	SA	NSGA-II	jitwatch	external project
P7	SA	NSGA-II	jitwatch	external project
P8	NSGA-II	MOSA	hystrix	external project
P9	MOSA	NSGA-II	anonymous-project	original project
P10	MOSA	NSGA-II	hortaz	original project

participant what code scope was considered on the evaluation of completeness. The participant then can consider that composite refactoring is (in)complete to the method and the class scope. The final step is addressed to evaluate the side effects of each recommendation. The subject analyzes whether the side effects that were presented in REComposite can happen in the practice. In addition, the participant verifies whether other side effects can be caused by the recommended composite refactoring.

6.4 Survey Results

This section shows the results of our survey, summarizing the developers' perceptions about the code smell identification, meaningfulness, completeness, and side effects of REComposite. We extracted a list of learned lessons through these results to automated recommenders based on search-based algorithms.

6.4.1 Code Smell Agreement

In the first activities of the survey, the developer analyzes the detection of code smells. The developers verified each code smell instance of the analyzed class and argued if she/he agreed with the code smell instance. The subjects can agree totally or partially, or disagree, with the detected code smells. We considered a total agreement if the developer agrees with all smells, and partially if the subject did not agree with all smells. The case of total disagreement represents the situation where the participant did not agree with all code smells. Regarding the NSGA-II sample, seven out of ten developers partially agreed with code smell detection, while three developers totally agreed with it. In the MOSA sample, three out of five developers fully agreed with code smell detection. When it comes to the SA approach, two out of five participants answered partially agree, while three participants fully agreed with the code smell detection.

The most common reason for the partially agree responses in all the algorithms was that the developers did not fully agree with the code smell detection of some *Feature Envies*. These usually represented cases in which the "smelly" methods had only few statements. They mentioned that the methods did not have excessive calls to external classes, as answered by participant P6 "*I only disagree with one of them, the feature envy of the* `saveUnsavedEditors` *method. In this method, there is only one call to an object of another class*". On the MOSA sample, we have an interesting case of partially agree. This case was related to the detection of *Long Envious Method* in utility methods, such as `equals()` or `hash()`. The subject P2 justified that "*many of the code smells in utility methods (equals, hash) don't seem to be smells*". The participant justified that the nature of these utility methods naturally requires the use of external classes and, as a consequence, end up having many code statements. Thus, this subject does not consider there is a presence of *Long Envious Method* in those cases.

Based on those results, we observed a need for improving the detection of *Feature Envy*, mainly when a method is too short. It is because some developers do not consider that a method is *Feature Envy* when this method has few statements. For those developers, a method with few statements can not be *Feature Envy* because envious methods need to have many calls for external classes, e.g., five or more calls to external classes. In addition, we need to collect the context of methods to better classify the code smell, for example, it is important to identify when the methods are utilities, like `toString` or `equals`. In addition, on *God Classes* and *Complex Classes*, the original developers agree with code smell detection, but some classes need to be complex and with many responsibilities because they are the main classes of the software projects. However, they mentioned the improvement of these classes is possible. We learned the following lesson through those results.

Lesson 1: Recommenders of composite refactorings may take into account user preferences for detection strategies of code smells. The use of interactive search-based approaches can be motivated, in which the developer can participate in the optimization process.

6.4.2 Meaningfulness

We proposed a Likert Scale [252] to developers classify the meaningfulness of each recommended composite refactoring removing the code smells. We considered five levels in the scale, with level 1 representing low meaningfulness and level 5 representing high meaningfulness. Each developer analyzed refactoring by refactoring for each composite and classified the meaningfulness for the entire composite.

We observed that NSGA-II tends to recommend composite refactoring with high meaningfulness, as presented by Figure 6.3(c). Six out of ten developers classified composite refactorings generated by NSGA-II with high meaningfulness (levels 4 and 5). The algorithm frequently recommends *Extract Methods* on methods that have code smells like *Feature Envy* or *Long Envious Method*. NSGA-II algorithm also suggests *Extract Classes* in classes that have code smells, such as *God Class* or *Complex Class*. The approach indicated the extraction of attributes and methods that are semantically related.

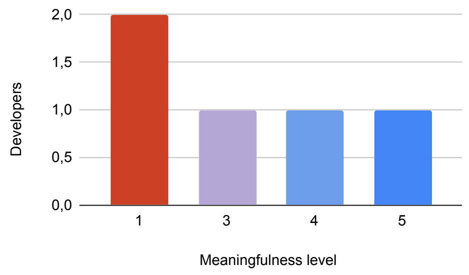
For example, on the class `HystrixCommandMetrics` from the hystrix project, the algorithm recommended the extraction of fields and methods that are related to the counting functionality. However, this algorithm had a low meaningfulness when proposed *Move Methods* on methods that are related to hierarchical and abstract classes. In the particular case of bytebuddy, the algorithm suggested *Move Method* to an abstract super class, which is not possible because it can cause syntax and semantic errors. The developer P1 mentioned other subclasses would need to implement this method that was moved; thus, this *Move Method* is not necessary because other subclasses would not use this method. In the other words, NSGA-II algorithm needs to improved for the recommendation of *Move Methods*, specially when the classes are related to hierarchy and abstractions.

Figures 6.3(a) and 6.3(b) show the results for SA and MOSA, respectively. MOSA had results better than SA: four developers classified the meaningfulness of MOSA with levels 3 and 4. SA recommends *Extract Methods* on smell-free methods and unnecessary *Move Methods* on short methods (with few statements), according to developers. MOSA indicated the motion of methods to inappropriate classes, i.e, classes that are not semantically related to the method. We then perceived that Simulated Annealing approaches have difficult to suggest composite refactorings for smelly code. Thus, we have the following lesson learned for builders of recommenders of composite refactorings, mainly for recommenders that use Simulated Annealing approaches.

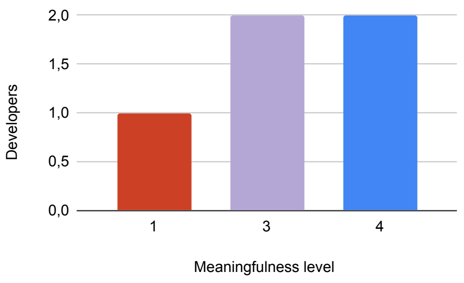
Lesson 2: Automated recommenders may allow the developer indicate the target code smells for the search-based algorithms to find composite refactorings to remove these smells.

Based on that, we confirmed that NSGA-II is the best algorithm to recommend refactoring, as indicated by [222, 244]. A possible explanation for those results is that NSGA-II better explores the search space, resulting in a higher diversity of solutions in terms of its objective functions [244]. We then observed that to use search-based algorithms to the problem related to the recommendation of composite refactorings, the problem is not basically to find the optimal composite refactoring, but also to offer new opportunities for composite refactorings.

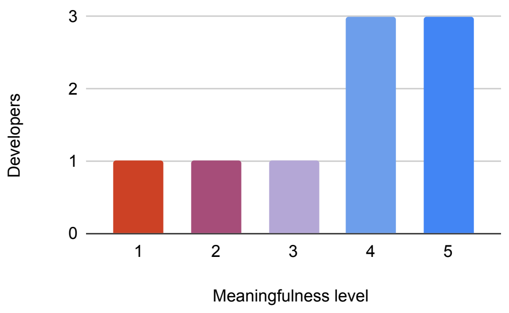
Lesson 3: Search-based algorithms that better explore the search space are motivated by the problem of recommendations for composite refactorings because they can provide new opportunities for composite refactorings.



6.3(a): Meaningfulness of SA Recommendations



6.3(b): Meaningfulness of MOSA Recommendations



6.3(c): Meaningfulness of NSGA-II Recommendations

Figure 6.3: Meaningfulness of Composite Refactorings

6.4.3 Completeness

The participants observed whether each composite refactoring can remove the code smells for at the method and class levels. Tables 6.5, 6.6, and 6.7 summarize the completeness for each recommendation from SA, MOSA, and NSGA-II algorithms, respectively. The developers could classify the composite as “Complete to this method or class and other involved classes”, “Complete composite only to this method”, “Complete composite only to this class”, “Incomplete composite to this method”, “Incomplete composite to this class”, "Incomplete composite to this method and this class", and "other situation". In the Tables, consider the following abbreviations to the refactoring types: ExM for *Extract Method*, MoM for *Move Method*, and ExC for *Extract Class*.

Regarding the SA and NSGA-II samples, developers observed that composites fully remove the smell in the method level, but the smell remains in the class. In those situations, the composite was complete to the method but incomplete to the class. Four out of five developers to the SA recommendations and four out of 10 participants to the NSGA-II recommendations mentioned that composites are complete to the method but incomplete to the class. We observed in those cases the methods are *Long Envious Methods*. In those situations, the participants justified the recommended *Extract Methods* removed the *Long Method*. However, the envious code was propagated to the new method. This problem also happens when developers manually apply refactorings, as indicated by Chapters 3 and 4. We observed that automated recommenders based on search-based algorithms currently focus to find composite refactorings for the "local" removal of target code smell, such as removals in the method. These recommenders need to be adapted for the removal of different levels of completeness, e.g., the completeness of a method and a class. We can then extract a lesson learned from this result.

Lesson 4: Search-based algorithms need to better explore the search space aiming composite refactorings that provide different levels of completeness.

On the MOSA sample, we did not discover a pattern regarding the refactoring completeness: only two composites were complete to the method, but they were incomplete to the class. In general, eight out of the 10 developers considered that NSGA-II recommendations were complete to the method or class. This is an interesting result because it can guarantee that NSGA-II recommendations can be accepted by developers.

We observed that REComposite always suggests the same refactoring type in a composite by class, e.g., one composite with only *Extract Methods* to the class `org.Main`. Aiming to understand it, we contacted the OganicRef authors (the tool that was the base of REComposite). They explained that the initial solutions are generated for each refactoring type, and from these solutions, the optimization process starts. For instance, for the *Extract Method* type, the REComposite generates a sequence of *Extract Methods* only for each class. However, it can generate incomplete composites because some composites need to have more than one refactoring type to remove some code smell types. For instance, in the case of composites only formed of *Extract Methods* applied on *Long Envious Methods*, it is also necessary to complete them with *Move*

Methods, as recommended in our catalog (Chapter 5). A possible solution for this problem is to enhance the generating of the initial solutions by increasing the diversity of refactoring types that constitute composite refactorings.

Lesson 5: Recommenders may generate initial composites constituted of all refactoring types for each class. After that, the fitness functions will analyze each combination of these refactoring types to find the optimal composite for each class.

Table 6.5: Completeness Level to SA Recommendations

SBSE	Refactorings	Completeness level
SA	ExM, ExC, MoM	Incomplete to this method and incomplete to this class
SA	ExM, ExC, MoM	Complete to this method and incomplete to this class
SA	ExM, ExM	Complete to this method and incomplete to this class
SA	ExM	Complete to this method and incomplete to this class
SA	ExM	Complete to this method and incomplete to this class

Table 6.6: Completeness Level to MOSA Recommendations

SBSE	Refactorings	Completeness level
MOSA	ExM	Complete only to this class
MOSA	MoM, MoM	Complete to this method or class and other involved classes
MOSA	MoM, MoM	Incomplete to this method
MOSA	MoM, MoM	Complete to this method and incomplete to this class
MOSA	ExC	Incomplete to this class

Table 6.7: Completeness Level to NSGA-II Recommendations

SBSE	Refactorings	Completeness level
NSGA-II	ExM, MoM, MoM	Incomplete to this method and incomplete to this class
NSGA-II	ExM, MoM, MoM	Complete to this method and incomplete to this class
NSGA-II	ExM, ExM, ExC	Complete to this method and incomplete to this class
NSGA-II	ExM, ExM	Complete only to this class
NSGA-II	ExC	Complete composite only to this class
NSGA-II	ExM, ExM	Complete to this method and incomplete to this class
NSGA-II	ExM, ExM	Complete only to this class, incomplete to this method
NSGA-II	ExC	Complete to this method or class and other involved classes
NSGA-II	ExM, ExM	Complete to this method and incomplete to this class
NSGA-II	MoM, MoM, MoM	Incomplete to this method and incomplete to this class

6.4.4 Side Effects

For each composite refactoring, we presented the possible side effects that can happen whether the composite is applied. These side effects were based on our catalog (Chapter 5). Developers evaluated what side effects can happen and if other side effects could occur. We asked them: "Do you believe that the composite refactoring you chose would have a side effect (not limited to side effects that were presented)? If so, which one?". Four developers that evaluated SA recommendations answered "No". In those cases, these developers ignored that the propagation of code smells like *Feature Envies* could be a side effect of composites that were analyzed, since the same developers previously said that composites were incomplete to the class. On the MOSA sample, two developers mentioned recommendations can have side effects as bug introductions after the *Move Methods* application, as reported by P9, *"It would break the system design, since it would be moving a get method to an exception class that do not fit the responsibility of the method."*

Surprisingly, on NSGA-II recommendations, seven developers reported that the composites can have side effects. Most of them stated that *Feature Envies* can be propagated by the source class or other involved classes. Developers also mentioned the application of several *Extract Methods* can increase the complexity of classes, increasing the *Complex Class*. In a particular case, P10 observed that *God Classes* can be introduced after the several motions of methods. Cedrim *et* al [7] alerted about the introduction of *Complex Classes* and *God Classes* after the application of several *Extract Methods* or *Move Methods*. In light of that, we need to introduce these warnings on the REComposite and enhance the recommendations based on the context of each class, avoiding the bug introduction. In addition, we observed that search-based algorithms need to be adapted to support recommendations with a minimal number of side effects.

Lesson 6: The minimization of side effects may be a parameter in the fitness functions that find composite refactoring for the beneficial removal of code smells.

6.4.5 General Evaluation of REComposite

Finally, we asked developers for suggestions to improve our recommender and our study methodology. The questions are: "Does our recommender need more information details? How can we improve it?", and "What were the positive and negative points of this study? What could be improved in the study?". Table 6.8 shows the answers to the first question, and Table 6.9 reports the answers the second question.

On the general evaluation of REComposite, developers reported the tool is innovative and easy to use. Some suggestions were made for improvement, such as (i) detailing the recommended refactorings, (ii) explaining why these refactorings were recommended, (iii) what code elements can be affected by the refactorings, (iv) considering changes in hierarchies and abstraction context, and (v) taking account that API and services classes could have a higher threshold for some smells.

Table 6.8: General Evaluation of REComposite

Part. ID	**Does our recommender need more details? What can we improve it?**
P1	"Inform more data about parameters of recommendations and properly handle changes in hierarchies and abstraction"
P2	"Yes. Perhaps a more detailed description of why a method was moved to a certain class and not to another."
P3	"Yes. In the last one, the recommended refactoring indicated to recreate the class. The code smells were right but the refactoring recommendations need improvements, in special in the target method."
P4	"The tool could have some information indicating why that code should be refactored"
P5	"Adding the description of the suggested type of recommendation would improve understanding and also judge whether the recommendation is valid or not. Would also improve the frontend and the presentation of what the recommendation wants to change, and what it wants to change in a more objective way. Presenting the full path to leave identification for the user is not always objective."
P6	"Highlight which code snippets might be causing smells like feature envy and dispersed coupling."
P7	"Improve the accuracy and context of side effects."
P8	"The tool is quite complete, informing everything that was necessary for the refactoring of code smells. The recommender in my view is great."
P9	"It would be ideal to show the code on a single page, or at least the lines where the smell is happening. "
P10	" I believe that the recommender could use the domain of the classe/software into consideration, for instance, this is an API code, the services classes could have a higher threshold for some smells, like feature envy."

On the evaluation of the methodology of our study, developers observed our methodology steps are clear and well structured. As the main positive points, they mentioned that they liked to participate in the study because they did not need to install many tools and the source code was easily shared. As the main negative points, some developers reported a lack of familiarity with the code/project being analyzed; in that case, they were participants that did not implement the analyzed code. Also, they mentioned that we need to better our explanation of the idea of linearity between the refactorings, showing why the refactorings constitute a composite. That is because some refactorings were several *Extract Methods* on the same class. The participants observed each extraction as an isolated refactoring, but these extractions form a composite on the same class. From this case, we then improved our explanation to clear the idea of linearity and the relation between the extractions.

Overall, the survey results suggest that the REComposite recommendations are useful and of good quality for software developers interested in composite refactorings. The web page offers a comprehensive and user-friendly interface for developers to improve the quality of their code using the recommendations provided by REComposite. Although, some improvements are still needed as indicated in Table 6.8.

6.5 Threats to Validity

Internal validity – On the threat of the search-based algorithms' validation, we reused the OrganicRef tool that was validated previously [244].

In that way, we mitigated the threat about the validation of implementation of SA, MOSA and NSGA-II. On REComposite implementation, we also created unit tests to validate the implementation of these algorithms and the detection of *Long Envious Method* and the suggestion of *Extract Method*. We implemented the detection of *Long Envious Method* because our previous study (Chapter 4) demonstrates that it is a common smell in the practice, and developers have difficult to detect it manually and fully remove it. We suggested *Extract Method* because it is a common refactoring type manually applied [60, 76] and this refactoring type in conjunction with other refactoring types can fully remove code smells, like *Long Envious Methods*.

Construct validity – We executed a survey with developers to explore search-based algorithms, as performed by related studies [146, 222]. The subjects watched a short training video to align the knowledge about the main concepts of the study. We did not explore other existing approaches because we had technical problems to use related recommenders [49, 222]. To mitigate this threat, we compared different search-based algorithms of OrganicRef: SA, MOSA and NSGA-II. Aiming the best understanding of source code and composite recommendations, some developers evaluated classes implemented by them, and other developers assessed classes that are well documented and we

Table 6.9: Evaluation of Methodology Study

Part. ID	What were the positive and negative points of this study? What can we improve in the study?
P1	"The interface helps; the shared code accessible by browser too."
P2	"The negative point is the lack of familiarity with the code/project being analyzed. Maybe better express the idea of linearity between the refactorings, showing that they constitute a composite."
P3	"I believe the study will help developers refactor classes faster and with justification for what code smells are on the classes. The study helps developers in refactoring methods with code smells."
P4	"I believe that the study did not have any negative points."
P5	"The 3 steps are well structured, visualizing code smells, checking suggested refactorings and then analyzing the impact of side effects was a positive distinction compared to existing tools. I believe that the formatting of the listing and sample items could be better polished. I don't see any downsides other than the above. I believe that the positive point of the study is the differential of actions that the platform brings. Recommendation along with side effects is a great idea."
P6	"Positives: Having access to the analyzed source code without having to clone any repositories. The tool is available on the web. It is not necessary much knowledge/training in smells to participate in the study. I can't see any downsides. One suggestion for the study would be to give examples of code smells before the meeting. Positive points: The study is easy to understand, does not require a lot of training or the installation of many tools."
P7	"Having a tool that optimizes the time of the refactoring process should already be a reality for any developer; this can be a good choice. The study is necessary to enable its use. No negative point."
P8	"The positive point of the study was that the details of code smells, refactoring recommendations are objective and accurate, which is very important. In this step, I did not identify negative aspects, with the exception of the complexMethod present in GetInstance(), where I did not agree with the presence of the smell."
P9	"Positives: (1) Straight forward informations presented and clear steps to follow. (2) Using the sharing tool. "
P10	"Show to the developer where there are possible points of improvement in the code. A negative aspect is that it could lead to changes that may not comply to certain common practices. Take into account the domain of the software while doing the analysis."

explained about the domain of these classes. The survey duration was about one hour per participant, avoiding the developers' tiredness. We used Likert scale in some questions of the survey to mitigate the subjectivity, each survey was recorded in video and audio to improve the understanding about each developer answer.

External validity – Aiming an acceptable representation of our results, we involved participants from different levels of development experience, and software projects from several domains and development environments with closed and open-source code. Our survey has only ten participants, but we mitigated this threat; each developer evaluated two composites, and in that way, we have 20 composite refactorings assessed.

6.6 Conclusion and Next Steps

In the study, we explored search-based algorithms for recommendations of composite refactorings through the extension REComposite. We implemented the recommendation of *Extract Method* refactoring type, and the detection of *Long Method* and *Long Envious Method* smells. REComposite uses three search-based algorithms for the recommendation of optimal composite refactorings. These algorithms are SA, MOSA, and NSGA-II. We applied a survey with 10 developers to explore the recommendations of REComposite about three points: meaningfulness, completeness, and side effects of recommended composite refactorings.

Overall, NSGA-II provides a more comprehensive approach for removing code smells from software projects, but its recommendations often can introduce side effects, according to developers. Our results revealed some suggestions to existing approaches that use search-based algorithms, such as (i) search-based algorithms need to better explore the search space aiming composite refactorings that provide different levels of completeness and (ii) the minimization of side effects may be a parameter in the fitness functions that find composite refactoring for the beneficial removal of code smells. As a general improvement for the automated refactoring recommenders, developers reported that tools need to offer more detailed explanations of the recommended refactorings.

7 Conclusion and Next Steps

This chapter describes a summary of our research results from our five empirical studies: a systematic mapping, a quantitative study on incomplete composites, an exploratory study about complete composites, the qualitative evaluation of our catalog, and the empirical exploration of an automated recommender of composite refactorings. These studies respectively resulted in a conceptual framework of composite refactoring, a list of findings about composite (in)completeness, a summary of lessons we learned with the evaluation of our catalog, and with the exploration of search-based algorithms for composite refactoring recommendations. The main contributions of our research and our next steps are described below.

1. Systematic Mapping Results: Our systematic mapping provides a conceptual framework of composite refactorings. The conceptual framework reveals seven representation models, nine characteristics, and thirty effects of composites. We found out that studies often use multidimensional representation models to (i) generate many "paths" of refactorings, and (ii) know what refactoring(s) may be suggested. Some studies also represent a composite as a sequence using vectors, or an arrays, for the recommendation of composite refactorings. These representations can be interesting to support step-wise, incremental composite refactorings because they help to know the order of each refactoring to be recommended. These results can help future studies to decide what representation model is appropriate according to the authors' approaches to supporting composites. Some studies also mentioned the characteristic of completeness of composite refactorings and indicated that this characteristic can be used to properly recommend composite refactorings. However, existing studies did not present a formation definition of completeness and not performed an in-depth empirical study about refactoring (in)completeness. On composite effects, existing studies are often limited to empirically investigate the effect on code smells, they have little evidence on how composites affect internal quality attributes in practice. In summary, our conceptual framework can guide researchers and refactoring tool builders on how to solve composite refactoring limitations and what is the appropriate characterization of composites according to each approach, like identification of composite refactorings or recommendation of composites.

2. Incomplete Composite Results: The results of the Chapter 3 help us to partially answer the RQ_2 of this proposal thesis. In our dataset with 353 incomplete composites, we confirm that composites are frequently incomplete to remove code smells. For instance, our results reveal that incomplete composite refactorings with at least one *Extract Method* are often (71%) applied without *Move Methods* on smelly classes. However, surprisingly, incomplete composites maintain the internal structural quality. We have found that most incomplete composite refactorings (58%) tended to at least maintain the internal structural quality of smelly classes, thereby not causing more harm to program comprehension. We then can observe that the incomplete nature of composites has possibly not harmed even further the program comprehensibility and other related quality attributes. This observation suggests that certain developers may be keen to maintain the structural quality of their programs through refactoring, even when they do not have the explicit intention of doing so.

3. Complete Composite Results: We performed a quantitative study to fully answer our RQ_2 (Section 1.3.1). In our dataset with 618 complete composites from 20 software projects, we have found (i) almost half (48%) of *Feature Envies* were removed when the composite *Move Methods* were applied. This information is not documented by existing composite recommendations. Since the occurrence of *Feature Envy* is a common situation [8], knowing about the usage of the *Move Methods* composite in advance can ease refactoring tasks, and (ii) about 36% of complete composites formed by *Extract Methods* to remove *Long Methods* have introduced *Feature Envies* and *Intensive Couplings* as side effects. Surprisingly, with the goal of improving code readability, by removing *Long Methods*, developers degrade the software internal quality by creating unnecessary high coupling.

4. Catalog Results: We proposed a catalog of composite recommendations to answer our RQ_3 (Section 1.3.1). Our catalog recommends four composite refactoring types to remove two new types of code smells, and the possible side effects of our recommendations. Our recommendations are based on common complete composites applied in the practice. We interviewed 21 developers to evaluate out catalog. The most (85%) of developers reported that their solutions could have the worse side effects without our catalog recommendations. Besides, we observed a significant (33%) number of developers were unaware of side effects when proposing solutions to remove code smells. These findings validate the practical necessity of recommendations to caution developers about the side effects of composite refactorings.

5. Exploration of Search-Based Algorithms Results: We performed an exploration of existing search-based algorithms to recommend composite refactorings. This empirical exploration is addressed to also answer our RQ$_3$ (Section 1.3.1). We extended an existing recommender of composite refactoring that uses search-based algorithms, OrganicRef [244]. We called this extension of REComposite. Moreover, REComposite generate composite recommendations using three search-based algorithms: SA, MOSA, and NSGA-II. The recommender identifies nine common types of code smells and recommends four refactoring types. REComposite indicates (i) the code smells that were identified, (ii) the smelly code elements, (iii) the composite that may be applied, and (iv) the side effects that may be minimized or removed. We performed a survey with ten developers to assess REComposite, exploring which search-based algorithm provides the best recommendations in terms of meaningfulness, completeness, and side effects. Our results reveal the most (80%) developers considered that NSGA-II recommendations are complete frequently; 60% of programmers mentioned that NSGA-II solutions have high meaningfulness, creating composite refactorings to be applied on smelly code elements without changing the software behavior. We then observed that NSGA-II is a search-based algorithm that better explore the search space. Thus, this algorithm generally finds new opportunities of composite refactoring. However, NSGA-II recommendations often can lead to side effects, according to 70% of developers. Based on this result, we perceived that existing search-based algorithms need to be improved to recommend complete composites without inducing side effects. We then provided a list of lessons learned for researchers and tool builders of recommenders of composite refactorings based in search, lessons such as (i) search-based algorithms need to better explore the search space aiming composite refactorings with different levels of completeness and (ii) the minimization of side effects may be a parameter to find composite refactoring for the beneficial removal of code smells.

As the main contributions, our findings reveal that developers tend not to solve structural problems completely when they apply large-scale refactorings, such as composites. These composites can remove one target code smell, but potentially introduce or do not remove other ones. This may be an alert regarding the use of existing recommendations. Our results suggest that existing recommendations of complete composites should be either revisited or enhanced to explicitly include possible side effects. In addition, we then present a catalog that can help developers in practice to fully remove code smells, minimizing side effects. Our results can inspire future research to improve existing refactoring tools, helping developers to achieve code structure improvement, based on the code context of their development activities.

Finally, we published severally key results along this doctoral research, an average of one paper published in a respectable international vehicle by year. These publications provide some indication of our research topic and findings. Table 7.1 shows a list of the papers published with the content directly related to this thesis research. Each paper was presented in a different chapter of this thesis. Table 7.2 presents the studies published that are indirectly related to this thesis. Along this doctoral research, we participated of collaborations that also investigated refactoring, o specifically composite refactoring. All of the papers were published in high quality conferences and journals (mostly Qualis A1-A4).

Table 7.1: Publications directly related to this thesis

Publication	Year	Qualis
Bibiano, A. C., Soares, V., Coutinho, D., Fernandes, E., Correia, J. L., Santos, K., Oliveira, A., Garcia, A., Gheyi, R., Fonseca, B., Ribeiro, M., Barbosa, C., Oliveira, D., (2020, July). "How does incomplete composite refactoring affect internal quality attributes?". In Proceedings of the 28th International Conference on Program Comprehension (ICPC), pp. 149-159.	2020	A2
Bibiano, A. C., Assunçao, W. K., Coutinho, D., Santos, K., Soares, V., Gheyi, R., Garcia, A., Fonseca, B., Ribeiro, M., Oliveira, D., Barbosa, C., Marques, J.L., & Oliveira, A. (2021, September). "Look ahead! revealing complete composite refactorings and their smelliness effects". In Proceedings of the 37th International Conference on Software Maintenance and Evolution (ICSME), pp. 298-308. IEEE.	2021	A2
Bibiano, A. C., "Completeness of composite refactorings for smell removal". In Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Doctoral Symposium (ICSE-DS), pp. 264-268.	2022	A1
Bibiano, A. C., Uchôa, A., Assunção, W. K., Tenório, D., Colanzi, T. E., Vergilio, S. R., & Garcia, A. (2023). "Composite refactoring: Representations, characteristics and effects on software projects". Information and Software Technology Journal (IST), 156, 107134.	2023	A1

Table 7.2: Publications indirectly related to this thesis

Publication	**Year**	**Qualis**
Bibiano, A. C., Fernandes, E., Oliveira, D., Garcia, A., Kalinowski, M., Fonseca, B., Oliveira, R., Oliveira, A.,& Cedrim, D. (2019, September). "A quantitative study on characteristics and effect of batch refactoring on code smells". In Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (MSR), pp. 1-11.	2019	A2
Paixão, M., Uchôa, A., **Bibiano, A. C.**, Oliveira, D., Garcia, A., Krinke, J., & Arvonio, E. (2020, June). "Behind the intents: An in-depth empirical study on software refactoring in modern code review". In Proceedings of the 17th Mining Software Repositories (MSR), pp. 125-136.	2020	A1
Sousa, L., Cedrim, D., Garcia, A., Oizumi, W., **Bibiano, A. C.**, Oliveira, D., Miryung, K., Oliveira, A. (2020, June). "Characterizing and identifying composite refactorings: Concepts, heuristics and patterns". In Proceedings of the 17th Mining Software Repositories (MSR), ACM/IEEE.	2020	A1
Bibiano, A. C., & Garcia, A. (2020, October). "On the characterization, detection and impact of batch refactoring in practice". In Proceedings of the 34th Brazilian Symposium on Software Engineering, on the Software Engineering Doctoral and Master Theses Competition (CTD-ES).	2020	A3
Oizumi, W., **Bibiano, A. C.**, Cedrim, D., Oliveira, A., Sousa, L., Garcia, A., & Oliveira, D. (2020, October). "Recommending composite refactorings for smell removal: Heuristics and evaluation". In Proceedings of the 34th Brazilian Symposium on Software Engineering (SBSE), pp. 72-81.	2020	A3
Oliveira, A., Neves, V., Plastino, A., **Bibiano, A.C.**, Garcia, A. Murta, L. "Do code refactorings influence the merge effort?". In Proceedings of the 45th International Conference on Software Engineering (ICSE), ACM/IEEE.	2023	A1
Oliveira, D., Assunção W. K. G., Garcia, **A., Bibiano, A.C.**, Ribeiro, M., Gheyi, R., Fonseca, B., "The untold story of code refactoring customizations in practice", In Proceedings of the 45th International Conference on Software Engineering (ICSE), ACM/IEEE.	2023	A1

Bibliography

1 BASILI, V.; ROMBACH, H.. **The TAME project: Towards improvement-oriented software environments.** IEEE Transactions on Software Engineering (TSE), 14(6):758–773, 1988.

2 BAVOTA, G.; DE CARLUCCIO, B.; DE LUCIA, A.; DI PENTA, M.; OLIVETO, R. ; STROLLO, O.. **When does a refactoring induce bugs? An empirical study.** In: PROCEEDINGS OF THE 12TH WORKING CONFERENCE ON SOURCE CODE ANALYSIS AND MANIPULATION (SCAM), p. 104–113, 2012.

4 BAVOTA, G.; LUCIA, A. D.; PENTA, M. D.; OLIVETO, R. ; PALOMBA, F.. **An experimental investigation on the innate relationship between quality and refactoring.** Journal of Systems and Software (JSS), 107:1–14, 2015.

7 CEDRIM, D.. **Understanding and Improving Batch Refactoring in Software Systems.** PhD thesis, Informatics Department (DI), Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Brazil, 2018.

8 CEDRIM, D.; GARCIA, A.; MONGIOVI, M.; GHEYI, R.; SOUSA, L.; DE MELLO, R.; FONSECA, B.; RIBEIRO, M. ; CHÁVEZ, A.. **Understanding the impact of refactoring on smells: A longitudinal study of 23 software projects.** In: PROCEEDINGS OF THE 11TH JOINT MEETING OF THE EUROPEAN SOFTWARE ENGINEERING CONFERENCE AND THE ACM SIGSOFT SYMPOSIUM ON THE FOUNDATIONS OF SOFTWARE (ESEC/FSE), p. 465–475, 2017.

10 CHÁVEZ, A.; FERREIRA, I.; FERNANDES, E.; CEDRIM, D. ; GARCIA, A.. **How does refactoring affect internal quality attributes? A multi-project study.** In: PROCEEDINGS OF THE 31ST BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING (SBES), p. 74–83, 2017.

12 Ó CINNÉIDE, M.; NIXON, P.. **Composite refactorings for java programs.** In: PROCEEDINGS OF THE WORKSHOP ON FORMAL TECHNIQUES FOR JAVA PROGRAMS, CO-LOCATED WITH THE 14TH EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING (ECOOP), p. 1–6, 2000.

13 CRESWELL, J.. **Research Design: Qualitative, Quantitative, and Mixed Methods Approaches.** SAGE Publications, 4th edition, 2014.

16 FERNANDES, E.; OLIVEIRA, J.; VALE, G.; PAIVA, T. ; FIGUEIREDO, E.. **A review-based comparative study of bad smell detection tools.** In: PROCEEDINGS OF THE 20TH INTERNATIONAL CONFERENCE ON EVALUATION AND ASSESSMENT IN SOFTWARE ENGINEERING (EASE), p. 18:1–18:12, 2016.

17 FERNANDES, E.; VALE, G.; SOUSA, L.; FIGUEIREDO, E.; GARCIA, A. ; LEE, J.. **No code anomaly is an island: Anomaly agglomeration as sign of product line instabilities**. In: PROCEEDINGS OF THE 16TH INTERNATIONAL CONFERENCE ON SOFTWARE REUSE (ICSR), p. 48–64, 2017.

19 FERNANDES, E.. **Stuck in the middle: Removing obstacles to new program features through batch refactoring**. In: PROCEEDINGS OF THE 41ST INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE): DOCTORAL SYMPOSIUM (DS), p. 1–4, 2019.

20 FERNANDES, E.; UCHOA, A.; BIBIANO, A. C. ; GARCIA, A.. **On the alternatives for composing batch refactoring**. In: PROCEEDINGS OF THE 3RD INTERNATIONAL WORKSHOP ON REFACTORING (IWOR), CO-LOCATED WITH THE 41ST INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), p. 1–4, 2019.

21 FERREIRA, I.; FERNANDES, E.; CEDRIM, D.; UCHÔA, A.; BIBIANO, A. C.; GARCIA, A.; CORREIA, J. L.; SANTOS, F.; NUNES, G.; BARBOSA, C. ; OTHERS. **The buggy side of code refactoring: Understanding the relationship between refactorings and bugs**. In: PROCEEDINGS OF THE 40TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE): POSTER TRACK, p. 406–407, 2018.

25 FOWLER, M.. **Refactoring: Improving the Design of Existing Code**. Addison-Wesley Professional, 1st edition, 1999.

28 GRIFFITH, I.; WAHL, S. ; IZURIETA, C.. **Truerefactor: An automated refactoring tool to improve legacy system and application comprehensibility**. In: PROCEEDINGS OF THE 24TH INTERNATIONAL CONFERENCE ON COMPUTER APPLICATIONS IN INDUSTRY AND ENGINEERING (CAINE), p. 1–6, 2011.

30 HARMAN, M.; TRATT, L.. **Pareto optimal search based refactoring at the design level**. In: PROCEEDINGS OF THE 9TH GENETIC AND EVOLUTIONARY COMPUTATION CONFERENCE (GECCO), p. 1106–1113, 2007.

31 JALALI, S.; WOHLIN, C.. **Systematic literature studies: Database searches vs. backward snowballing**. In: PROCEEDINGS OF THE 6TH INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING AND MEASUREMENT (ESEM), p. 29–38, 2012.

32 KANG, K.; COHEN, S.; HESS, J.; NOVAK, W. ; PETERSON, A.. **Feature-oriented domain analysis (FODA) feasibility study**. Technical report, CMU-SEI-90-TR-21 and ESD-90-TR-222, Software Engineering Insttitute (SEI), Carnegie Mellon University (CMU), 1990.

35 KIM, M.; ZIMMERMANN, T. ; NAGAPPAN, N.. **An empirical study of refactoring: Challenges and benefits at Microsoft**. IEEE Transactions on Software Engineering (TSE), 40(7):633–649, 2014.

36 KITCHENHAM, B.; CHARTERS, S.. **Guidelines for performing systematic literature reviews in software engineering**. Technical report, EBSE 2007-001, Version 2.3, Keele University and University of Durham, 2007.

37 KUHLEMANN, M.; LIANG, L. ; SAAKE, G.. **Algebraic and cost-based optimization of refactoring sequences**. In: PROCEEDINGS OF THE 2ND INTERNATIONAL WORKSHOP ON MODEL-DRIVEN PRODUCT LINE ENGINEERING (MDPLE), CO-LOCATED WITH THE 6TH EUROPEAN CONFERENCE ON MODELLING FOUNDATIONS AND APPLICATIONS (ECMFA), p. 37–48, 2010.

39 LIN, Y.; PENG, X.; CAI, Y.; DIG, D.; ZHENG, D. ; ZHAO, W.. **Interactive and guided architectural refactoring with search-based recommendation**. In: PROCEEDINGS OF THE 24TH INTERNATIONAL SYMPOSIUM ON FOUNDATIONS OF SOFTWARE ENGINEERING (FSE), p. 535–546, 2016.

43 MEANANEATRA, P.. **Identifying refactoring sequences for improving software maintainability**. In: PROCEEDINGS OF THE 27TH INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING (ASE), p. 406–409, 2012.

44 MKAOUER, M. W.; KESSENTINI, M.; BECHIKH, S.; DEB, K. ; Ó CINNÉIDE, M.. **Recommendation system for software refactoring using innovization and interactive dynamic optimization**. In: PROCEEDINGS OF THE 29TH INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING (ASE), p. 331–336, 2014.

46 MURPHY-HILL, E.; PARNIN, C. ; BLACK, A.. **How we refactor, and how we know it**. IEEE Transactions on Software Engineering (TSE), 38(1):5–18, 2012.

47 Ó CINNÉIDE, M.; TRATT, L.; HARMAN, M.; COUNSELL, S. ; HEMATI MOGHADAM, I.. **Experimental assessment of software metrics using automated refactoring**. In: PROCEEDINGS OF THE 5TH INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING AND MEASUREMENT (ESEM), p. 49–58, 2012.

48 OIZUMI, W.; GARCIA, A.; SOUSA, L.; CAFEO, B. ; ZHAO, Y.. **Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems**. In: PROCEEDINGS OF THE 38TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), p. 440–451, 2016.

49 OUNI, A.; KESSENTINI, M.; SAHRAOUI, H. ; HAMDI, M. S.. **Search-based refactoring: Towards semantics preservation**. In: PROCEEDINGS OF THE 28TH IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE (ICSM), p. 347–356, 2012.

50 OUNI, A.; KESSENTINI, M. ; SAHRAOUI, H.. **Search-based refactoring using recorded code changes**. In: PROCEEDINGS OF THE 17TH EUROPEAN CONFERENCE ON SOFTWARE MAINTENANCE AND REENGINEERING (CSMR), p. 221–230, 2013.

55 PIVETA, E.; ARAÚJO, J.; PIMENTA, M.; MOREIRA, A.; GUERREIRO, P. ; PRICE, R. T.. **Searching for opportunities of refactoring sequences: Reducing the search space**. In: PROCEEDINGS OF THE 32ND INTERNATIONAL CONFERENCE ON COMPUTER SOFTWARE AND APPLICATIONS (COMPSAC), p. 319–326, 2008.

60 SILVA, D.; TSANTALIS, N. ; VALENTE, M. T.. **Why we refactor? Confessions of GitHub contributors**. In: PROCEEDINGS OF THE 24TH INTERNATIONAL SYMPOSIUM ON FOUNDATIONS OF SOFTWARE ENGINEERING (FSE), p. 858–870, 2016.

62 SOUSA, L.; OLIVEIRA, A.; OIZUMI, W.; BARBOSA, S.; GARCIA, A.; LEE, J.; KALINOWSKI, M.; DE MELLO, R.; FONSECA, B.; OLIVEIRA, R.; LUCENA, C. ; PAES, R.. **Identifying design problems in the source code: A grounded theory**. In: PROCEEDINGS OF THE 40TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), p. 921–931, 2018.

63 STOL, K.-J.; RALPH, P. ; FITZGERALD, B.. **Grounded theory in software engineering research: A critical review and guidelines**. In: PROCEEDINGS OF THE 38TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), p. 120–131, 2016.

64 SZŐKE, G.; NAGY, C.; FÜLÖP, L.; FERENC, R. ; GYIMÓTHY, T.. **FaultBuster: An automatic code smell refactoring toolset**. In: PROCEEDINGS OF THE 15TH WORKING CONFERENCE ON SOURCE CODE ANALYSIS AND MANIPULATION (SCAM), p. 253–258, 2015.

65 SZOKE, G.; NAGY, C.; FERENC, R. ; GYIMÓTHY, T.. **Designing and developing automated refactoring transformations: An experience report**. In: PROCEEDINGS OF THE 23RD INTERNATIONAL CONFERENCE ON SOFTWARE ANALYSIS, EVOLUTION AND REENGINEERING (SANER), p. 693–697, 2016.

67 TSANTALIS, N.; CHATZIGEORGIOU, A.. **Identification of extract method refactoring opportunities for the decomposition of methods**. Journal of Systems and Software (JSS), 84(10):1757–1782, 2011.

70 WOHLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M.; REGNELL, B. ; WESSLÉN, A.. **Experimentation in Software Engineering**. Springer Science & Business Media, 1st edition, 2012.

73 YAMASHITA, A.; MOONEN, L.. **Do developers care about code smells? An exploratory survey**. In: PROCEEDINGS OF THE 20TH WORKING CONFERENCE ON REVERSE ENGINEERING (WCRE), p. 242–251, 2013.

76 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. ; OLIVEIRA, F.. **How does incomplete composite refactoring affect internal quality attributes?** In: 28TH INTERNATIONAL CONFERENCE ON PROGRAM COMPREHENSION (ICPC), 2020.

77 BIBIANO, A. C.; ASSUNÇAO, W.; COUTINHO, D.; SANTOS, K.; SOARES, V.; GHEYI, R.; GARCIA, A.; FONSECA, B.; RIBEIRO, M.; OLIVEIRA, D. ; OTHERS. **Look ahead! revealing complete composite refactorings and their smelliness effects.** In: 37TH INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE AND EVOLUTION (ICSME), 2021.

79 MEANANEATRA, P.; RONGVIRIYAPANISH, S. ; APIWATTANAPONG, T.. **Refactoring opportunity identification methodology for removing long method smells and improving code analyzability.** IEICE Transactions on Information and Systems, 101(7):1766–1779, 2018.

80 RANI, A.; CHHABRA, J. K.. **Prioritization of smelly classes: A two phase approach (reducing refactoring efforts).** In: 2017 3RD INTERNATIONAL CONFERENCE ON COMPUTATIONAL INTELLIGENCE & COMMUNICATION TECHNOLOGY (CICT), p. 1–6. IEEE, 2017.

94 KESSENTINI, M.; MAHAOUACHI, R. ; GHEDIRA, K.. **What you like in design use to correct bad-smells.** Software Quality Journal, 21(4):551–571, 2013.

95 JENSEN, A. C.; CHENG, B. H.. **On the use of genetic programming for automated refactoring and the introduction of design patterns.** In: PROCEEDINGS OF THE 12TH ANNUAL CONFERENCE ON GENETIC AND EVOLUTIONARY COMPUTATION, p. 1341–1348, 2010.

98 KESSENTINI, M.; WANG, H.. **Detecting refactorings among multiple web service releases: A heuristic-based approach.** In: 2017 IEEE INTERNATIONAL CONFERENCE ON WEB SERVICES (ICWS), p. 365–372. IEEE, 2017.

S8 RAZANI, Z.; KEYVANPOUR, M.. **Sbsr solution evaluation: Methods and challenges classification.** In: 2019 5TH CONFERENCE ON KNOWLEDGE BASED ENGINEERING AND INNOVATION (KBEI), p. 181–188. IEEE.

S9 FERNANDES, E.. **Stuck in the middle: Removing obstacles to new program features through batch refactoring.** In: 2019 IEEE/ACM 41ST INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING: COMPANION PROCEEDINGS (ICSE-COMPANION), p. 206–209. IEEE, 2019.

104 COUNSELL, S.; LIU, X.; SWIFT, S.; BUCKLEY, J.; ENGLISH, M.; HEROLD, S.; ELDH, S. ; ERMEDAHL, A.. **An exploration of the'introduce explaining variable'refactoring.** In: SCIENTIFIC WORKSHOP PROCEEDINGS OF THE XP2015, p. 1–5, 2015.

105 ZARRAS, A. V.; VARTZIOTIS, T. ; VASSILIADIS, P.. **Navigating through the archipelago of refactorings.** In: PROCEEDINGS OF THE 2015 10TH JOINT MEETING ON FOUNDATIONS OF SOFTWARE ENGINEERING, p. 922–925, 2015.

109 MARTICORENA, R.; LÓPEZ, C.; PÉREZ, J. ; CRESPO, Y.. **Assisting refactoring tool development through refactoring characterization.** In: PROCEEDINGS OF THE 6TH INTERNATIONAL CONFERENCE ON SOFTWARE AND DATA TECHNOLOGIES, volumen 2, 2011.

111 MOESUS, N.; SCHOLZE, M.; SCHLESINGER, S. ; HERBER, P.. **Automated selection of software refactorings that improve performance.** In: ICSOFT, p. 67–78, 2018.

116 GAITANI, M. A. G.; ZAFEIRIS, V. E.; DIAMANTIDIS, N. ; GIAKOUMAKIS, E. A.. **Automated refactoring to the null object design pattern.** Information and Software Technology, 59:33–52, 2015.

132 TARWANI, S.; CHUG, A.. **Sequencing of refactoring techniques by greedy algorithm for maximizing maintainability.** In: 2016 INTERNATIONAL CONFERENCE ON ADVANCES IN COMPUTING, COMMUNICATIONS AND INFORMATICS (ICACCI), p. 1397–1403. IEEE, 2016.

143 BIBIANO, A. C.; FERNANDES, E.; OLIVEIRA, D.; GARCIA, A.; KALINOWSKI, M.; FONSECA, B.; OLIVEIRA, R.; OLIVEIRA, A. ; CEDRIM, D.. **A quantitative study on characteristics and effect of batch refactoring on code smells.** In: PROCEEDINGS OF THE 13TH INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING AND MEASUREMENT (ESEM), p. 1–11. IEEE, 2019.

145 OUNI, A.; KESSENTINI, M.; Ó CINNÉIDE, M.; SAHRAOUI, H.; DEB, K. ; INOUE, K.. **More: A multi-objective refactoring recommendation approach to introducing design patterns and fixing code smells.** Journal of Software: Evolution and Process, 29(5):e1843, 2017.

146 OUNI, A.; KESSENTINI, M.; SAHRAOUI, H.; INOUE, K. ; DEB, K.. **Multi-criteria code refactoring using search-based software engineering: An industrial case study.** ACM Transactions on Software Engineering and Methodology (TOSEM), 25(3):1–53, 2016.

156 MOHAN, M.; GREER, D.. **Using a many-objective approach to investigate automated refactoring.** Information and Software Technology, 112:83–101, 2019.

157 BOTELHO, G.; BEZERRA, L.; BRITTO, A. ; SILVA, L.. **A many-objective estimation distributed algorithm applied to search based software refactoring.** In: 2018 IEEE CONGRESS ON EVOLUTIONARY COMPUTATION (CEC), p. 1–8. IEEE, 2018.

167 PETERSEN, K.; VAKKALANKA, S. ; KUZNIARZ, L.. **Guidelines for conducting systematic mapping studies in software engineering: An update.** Information and Software Technology, 64:1 – 18, 2015.

167 PAIXÃO, M.; UCHÔA, A.; BIBIANO, A. C.; OLIVEIRA, D.; GARCIA, A.; KRINKE, J. ; ARVONIO, E.. **Behind the intents: An in-depth empirical study on software refactoring in modern code review.** In: 17TH MINING SOFTWARE REPOSITORIES (MSR), 2020.

168 ABID, C.; ALIZADEH, V.; KESSENTINI, M.; FERREIRA, T. D. N. ; DIG, D.. **30 years of software refactoring research: A systematic literature review.** Transaction of Software Engineering (TSE), 2020.

168 ALOMAR, E.; MKAOUER, M.; OUNI, A. ; KESSENTINI, M.. **On the impact of refactoring on the relationship between quality attributes and design metrics**. In: 13TH INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING AND MEASUREMENT (ESEM), p. 1–11, 2019.

169 BIBIANO, A. C.. **Understanding characteristics and structural effects of batch refactoring in practice**. 2019.

170 VILLAMIZAR, H.; KALINOWSKI, M.; VIANA, M. ; FERNÁNDEZ, D. M.. **A systematic mapping study on security in agile requirements engineering**. In: 2018 44TH EUROMICRO CONFERENCE ON SOFTWARE ENGINEERING AND ADVANCED APPLICATIONS (SEAA), p. 454–461, 2018.

171 SOUSA, L.; CEDRIM, D.; GARCIA, A.; OIZUMI, W.; BIBIANO, A. C.; TENORIO, D.; KIM, M. ; OLIVEIRA, A.. **Characterizing and identifying composite refactorings: Concepts, heuristics and patterns**. In: 17TH MSR (2020), 2020.

172 BRITO, A.; HORA, A. ; VALENTE, M. T.. **Refactoring graphs: Assessing refactoring over time**. In: PROCEEDINGS OF THE 26TH INTERNATIONAL CONFERENCE ON SOFTWARE ANALYSIS, EVOLUTION AND REENGINEERING (SANER), p. 504–507, 2019.

173 TSANTALIS, N.; MANSOURI, M.; ESHKEVARI, L.; MAZINANIAN, D. ; DIG, D.. **Accurate and efficient refactoring detection in commit history**. In: PROCEEDINGS OF THE 40TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), p. 483–494, 2018.

174 VIDAL, S. A.; OIZUMI, W. N.; GARCIA, A.; DIAZ-PACE, J. A. ; MARCOS, C.. **Ranking architecturally critical agglomerations of code smells**. Sci. Comput. Program. (2019), 182:64–85, 2019.

175 OIZUMI, W. N.; DA SILVA SOUSA, L.; OLIVEIRA, A.; GARCIA, A.; AGBACHI, O. I. A. B.; OLIVEIRA, R. F. ; LUCENA, C.. **On the identification of design problems in stinky code: experiences and tool support**. J. Braz. Comp. Soc. (2018), 24(1):13:1–13:30, 2018.

176 OIZUMI, W. N.; GARCIA, A. F.; DA SILVA SOUSA, L.; CAFEO, B. B. P. ; ZHAO, Y.. **Code anomalies flock together: exploring code anomaly agglomerations for locating design problems**. In: Dillon, L. K.; Visser, W. ; Williams, L., editors, 38TH ICSE (2016), p. 440–451. ACM, 2016.

177 PALOMBA, F.; ZAIDMAN, A.; OLIVETO, R. ; DE LUCIA, A.. **An exploratory study on the relationship between changes and refactoring**. In: 25TH ICPC (2017), p. 176–185. IEEE, 2017.

178 GUIMARÃES, E. T.; GARCIA, A. F. ; CAI, Y.. **Architecture-sensitive heuristics for prioritizing critical code anomalies**. In: France, R. B.; Ghosh, S. ; Leavens, G. T., editors, 14TH INTERNATIONAL CONFERENCE ON MODULARITY (2015), p. 68–80. ACM, 2015.

179 SANTOS, J. A. M.; ROCHA-JUNIOR, J. B.; PRATES, L. C. L.; DO NASCIMENTO, R. S.; FREITAS, M. F. ; DE MENDONÇA, M. G.. **A systematic review on the code smell effect**. JSS (2018), 144:450–477, 2018.

180 DUBBO. **Acesslog dateformat enhancemnet**, 2019. Available at: <https://github.com/apache/dubbo/commit/5146f6d6>.

181 DUBBO. **Dubbo github project**, 2019. Available at: <https://github.com/apache/dubbo>.

182 SANTOS, K.. **Refactoring (move method) to accesslogdata class**, 2020. Available at: <https://github.com/apache/dubbo/pull/6151>.

183 SANTOS, K.. **Refactoring to remove envious methods (extract methods)**, 2021. Available at: <https://github.com/apache/dubbo/pull/7647>.

184 DUBBO. **Acesslog dateformat enhancemnet**, 2019. Available at: <https://github.com/apache/dubbo/pull/3274>.

185 ELASTICSEARCH. **Support date math for origin decay function parsing**, 2013. Available at: <https://github.com/elastic/elasticsearch/commit/2d523ac>.

186 PRESTODB. **Parse hive column values as needed instead of all up front**, 2013. Available at: <https://github.com/prestodb/presto/commit/b4bbb4b>.

187 DUBBO. **Rewrite uts**, 2019. Available at: <https://github.com/apache/dubbo/commit/66fbd320>.

188 DUBBO. **Refactoring to remove duplicate methods and feature envy.**, 2019. Available at: <https://github.com/apache/dubbo/pull/5506>.

189 DUBBO. **Refactoring to remove feature envy**, 2019. Available at: <https://github.com/apache/dubbo/pull/5559>.

190 DUBBO. **refactoring to remove feature envy**, 2019. Available at: <https://github.com/apache/dubbo/pull/5529>.

191 ALSHAYEB, M.. **Empirical investigation of refactoring effect on software quality**. 51(9):1319–1326, 2009.

192 DU BOIS, B.; DEMEYER, S. ; VERELST, J.. **Refactoring: Improving coupling and cohesion of existing code**. In: 11TH WCRE (2004), p. 144–151, 2004.

193 BIBIANO, A. C.. **Complete composite website**, 2021. Available at: <https://anacarlagb.github.io/icsme2021-complete-composite/>.

194 TSANTALIS, N.; KETKAR, A. ; DIG, D.. **Refactoringminer 2.0**. IEEE Transactions on Software Engineering (TSE), 2020.

195 OLIVEIRA, R. F.; DE MELLO, R. M.; FERNANDES, E.; GARCIA, A. ; LUCENA, C.. **Collaborative or individual identification of code smells? on the effectiveness of novice and professional developers**. Information and Software Technology (IST), 120, 2020.

196 DE MELLO, R. M.; UCHÔA, A. G.; OLIVEIRA, R. F.; OIZUMI, W. N.; SOUZA, J.; MENDES, K.; OLIVEIRA, D.; FONSECA, B. ; GARCIA, A.. **Do research and practice of code smell identification walk together? A social representations analysis**. In: 13TH INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING AND MEASUREMENT (ESEM), p. 1–6. IEEE, 2019.

197 FONTANA, F.; MANGIACAVALLI, M.; POCHIERO, D. ; ZANONI, M.. **On experimenting refactoring tools to remove code smells**. In: PROCEEDINGS OF THE 16TH INTERNATIONAL CONFERENCE ON AGILE SOFTWARE DEVELOPMENT (XP), SCIENTIFIC WORKSHOPS, p. 1–7, 2015.

198 YOSHIDA, N.; SAIKA, T.; CHOI, E.; OUNI, A. ; INOUE, K.. **Revisiting the relationship between code smells and refactoring**. In: PROCEEDINGS OF THE 24TH INTERNATIONAL CONFERENCE ON PROGRAM COMPREHENSION (ICPC), p. 1–4, 2016.

199 MENS, T.; TOURWÉ, T.. **A survey of software refactoring**. IEEE Transactions on Software Engineering (TSE), 30(2):126–139, 2004.

200 MARTINS, J.; BEZERRA, C.; UCHÔA, A. ; GARCIA, A.. **How do code smell co-occurrences removal impact internal quality attributes? a developers' perspective**. In: 35TH SBES, p. 54–63, 2021.

201 TSANTALIS, N.; CHAIKALIS, T. ; CHATZIGEORGIOU, A.. **Ten years of JDeodorant: Lessons learned from the hunt for smells**. In: P25TH SANER (2018), p. 4–14, 2018.

202 OLIVEIRA, R. F.; DA SILVA SOUSA, L.; DE MELLO, R. M.; VALENTIM, N. M. C.; LOPES, A.; CONTE, T.; GARCIA, A. F.; DE OLIVEIRA, E. C. C. ; DE LUCENA, C. J. P.. **Collaborative identification of code smells: A multi-case study**. In: 39TH ICSE-SEIP (2017), p. 33–42. IEEE Computer Society, 2017.

203 BIBIANO, A. C.; GARCIA, A.. **On the characterization, detection and impact of batch refactoring in practice**. In: 34TH BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERINGSOFTWARE ENGINEERING - DOCTORAL AND MASTER THESES COMPETITION (SBES-CTD), p. 165–179, Porto Alegre, RS, Brasil, 2020. SBC.

204 OIZUMI, W.; BIBIANO, A. C.; CEDRIM, D.; OLIVEIRA, A.; SOUSA, L.; GARCIA, A. ; OLIVEIRA, D.. **Recommending composite refactorings for smell removal: Heuristics and evaluation**. In: 34TH BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING (SBES), p. 72–81, 2020.

205 SOARES, V.; OLIVEIRA, A.; PEREIRA, J. A.; BIBANO, A. C.; GARCIA, A.; FARAH, P. R.; VERGILIO, S. R.; SCHOTS, M.; SILVA, C.; COUTINHO,

D. ; OTHERS. **On the relation between complexity, explicitness, effectiveness of refactorings and non-functional concerns**. In: 34TH BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING (SBES), p. 788–797, 2020.

206 TAHIR, A.; DIETRICH, J.; COUNSELL, S.; LICORISH, S. ; YAMASHITA, A.. **A large scale study on how developers discuss code smells and anti-pattern in stack exchange sites**. Information and Software Technology, 125:106333, sep 2020.

207 UCHÔA, A.; BARBOSA, C.; OIZUMI, W.; BLENILIO, P.; LIMA, R.; GARCIA, A. ; BEZERRA, C.. **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), p. 511–522. IEEE, 2020.

208 BARBOSA, C.; UCHÔA, A.; COUTINHO, D.; FALCÃO, F.; BRITO, H.; AMARAL, G.; SOARES, V.; GARCIA, A.; FONSECA, B.; RIBEIRO, M. ; OTHERS. **Revealing the social aspects of design decay: A retrospective study of pull requests**. In: 34TH BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING (SBES), p. 364–373, 2020.

209 UCHÔA, A.; BARBOSA, C.; COUTINHO, D.; OIZUMI, W.; ASSUNÇAO, W. K.; VERGILIO, S. R.; PEREIRA, J. A.; OLIVEIRA, A. ; GARCIA, A.. **Predicting design impactful changes in modern code review: A large-scale empirical study**. 2021.

211 OLIVEIRA, J.; GHEYI, R.; MONGIOVI, M.; SOARES, G.; RIBEIRO, M. ; GARCIA, A.. **Revisiting the refactoring mechanics**. Information and Software Technology, 110:136–138, 2019.

213 STÖRRLE, H.. **How are conceptual models used in industrial software development? a descriptive survey**. In: PROCEEDINGS OF THE 21ST INTERNATIONAL CONFERENCE ON EVALUATION AND ASSESSMENT IN SOFTWARE ENGINEERING, p. 160–169, 2017.

214 WHITE, J.; DOUGHERTY, B.; SCHMIDT, D. C. ; BENAVIDES CUEVAS, D. F.. **Automated reasoning for multi-step feature model configuration problems**. In: SPLC 2009: 13TH INTERNATIONAL SOFTWARE PRODUCT LINE CONFERENCE (2009), P 11-20. ACM, 2009.

216 OLIVEIRA, R.; ESTÁCIO, B.; GARCIA, A.; MARCZAK, S.; PRIKLADNICKI, R.; KALINOWSKI, M. ; LUCENA, C.. **Identifying code smells with collaborative practices: A controlled experiment**. In: PROCEEDINGS OF THE BRAZILIAN SYMPOSIUM ON SOFTWARE COMPONENTS, ARCHITECTURES AND REUSE (SBCARS), p. 61–70. IEEE, 2016.

217 NETO, A. A.; KALINOWSKI, M.; GARCIA, A.; WINKLER, D. ; BIFFL, S.. **A preliminary comparison of using variability modeling approaches to represent experiment families**. In: PROCEEDINGS OF THE EVALUATION AND ASSESSMENT ON SOFTWARE ENGINEERING, p. 333–338, 2019.

218 TRUJILLO, S.; BATORY, D. ; DIAZ, O.. **Feature refactoring a multi-representation program into a product line**. In: PROCEEDINGS OF THE 5TH INTERNATIONAL CONFERENCE ON GENERATIVE PROGRAMMING AND COMPONENT ENGINEERING, p. 191–200, 2006.

219 BIBIANO, A. C.; UCHÔA, A.; ASSUNÇAO, W. K.; OLIVEIRA, D.; E. COLANZI, T.; VERGILIO, S. R. ; GARCIA, A.. **Composite refactoring: Representation models, characteristics and effect on software projects, url: https://shorturl.at/gstzd**. 2022.

220 BIBIANO, A. C. G.. **Understanding Characteristics and Structural Effects of Batch Refactorings in Practice**. PhD thesis, Master's dissertation, PUC-Rio, 2019.

221 BENAVIDES, D.; SEGURA, S. ; RUIZ-CORTÉS, A.. **Automated analysis of feature models 20 years later: A literature review**. Information systems, 35(6):615–636, 2010.

222 ALIZADEH, V.; OUALI, M. A.; KESSENTINI, M. ; CHATER, M.. **Refbot: Intelligent software refactoring bot**. In: INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING (ASE), p. 823–834, 2019.

223 VIDAL, S. A.; OIZUMI, W. N.; GARCIA, A.; DIAZ-PACE, J. A. ; MARCOS, C.. **Ranking architecturally critical agglomerations of code smells**. Sci. Comput. Program. (2019), 182:64–85, 2019.

224 BIBIANO, A. C.. **Incomplete composite website**, 2020. Available at: <https://researcher-icpc-104.github.io/icpc2020_incomplete_composite>.

225 SZŐKE, G.; ANTAL, G.; NAGY, C.; FERENC, R. ; GYIMÓTHY, T.. **Empirical study on refactoring large-scale industrial systems and its effects on maintainability**. Journal of Systems and Software (JSS), 129:107–126, 2017.

226 TENORIO, D.; BIBIANO, A. C. ; GARCIA, A.. **On the customization of batch refactoring**. In: 3RD INTERNATIONAL WORKSHOP ON REFACTORING, CO-ALOCATED INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (IWOR-ICSE), p. 13–16. IEEE Press, 2019.

227 BRITO, A.; HORA, A. ; VALENTE, M. T.. **Towards a catalog of composite refactorings**. Journal of Software: Evolution and Process, 2022.

228 ANT. **Apache ant**, 2017. Available at: <https://github.com/apache/ant/commit/b7d1e9bde44c>.

229 BIBIANO, A. C.. **Complete composite website**, 2022. Available at: <https://compositerefactoring.github.io/site/>.

230 BIBIANO, A. C.. **Catalog of complete composites**, 2022. Available at: <https://compositerefactoring.github.io/catalog>.

231 NETTY. **Allow controlling time flow for embeddedeventloop**, 2022. Available at: <https://github.com/netty/netty/commit/c18fc2b>.

233 O'KEEFFE, M.; CINNÉIDE, M. O.. **A stochastic approach to automated design improvement**. In: ACM INTERNATIONAL CONFERENCE PROCEEDING SERIES, volumen 42, p. 59–62. Citeseer, 2003.

234 HARMAN, M.; JONNES, B.. **Search-based software engineering**. In: INFORMATION AND SOFTWARE TECHNOLOGY, volumen 43, p. 833–839, 2001.

239 DEB, K.; PRATAP, A.; AGARWAL, S. ; MEYARIVAN, T.. **A fast and elitist multiobjective genetic algorithm: Nsga-ii**. IEEE transactions on evolutionary computation, 6(2):182–197, 2002.

240 ULUNGU, E. L.; TEGHEM, J.; FORTEMPS, P. ; TUYTTENS, D.. **Mosa method: a tool for solving multiobjective combinatorial optimization problems**. Journal of multicriteria decision analysis, 8(4):221, 1999.

241 FRAIRE HUACUJA, H. J.; SOTO, C.; DORRONSORO, B.; SANTILLÁN, C. G.; VALDEZ, N. R. ; BALDERAS-JARAMILLO, F.. **Amosa with analytical tuning parameters and fuzzy logic controller for heterogeneous computing scheduling problem**. Intuitionistic and Type-2 Fuzzy Logic Enhancements in Neural and Optimization Algorithms: Theory and Applications, p. 195–208, 2020.

242 KESSENTINI, M.; DEA, T. J. ; OUNI, A.. **A context-based refactoring recommendation approach using simulated annealing: two industrial case studies**. In: PROCEEDINGS OF THE GENETIC AND EVOLUTIONARY COMPUTATION CONFERENCE, p. 1303–1310, 2017.

244 OIZUMI, W.. **Identification and Refactoring of Design Problems in Software Systems**. PhD thesis, Informatics Department (DI), Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Brazil, 2022.

245 GALIN, D.. **Software quality: concepts and practice**. John Wiley & Sons, 2018.

246 LAPORTE, C. Y.; APRIL, A.. **Software quality assurance**. John Wiley & Sons, 2018.

247 BAABAD, A.; ZULZALIL, H. B.; HASSAN, S. ; BAHAROM, S. B.. **Software architecture degradation in open source software: A systematic literature review**. IEEE Access, 8:173681–173709, 2020.

248 OIZUMI, W.; SOUSA, L.; OLIVEIRA, A.; CARVALHO, L.; GARCIA, A.; COLANZI, T. ; OLIVEIRA, R.. **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), p. 346–357, 2019.

249 OLIVEIRA, D.; ASSUNÇÃO, W. K. G.; GARCIA, A.; FONSECA, B. ; RIBEIRO, M.. **Developers' perception matters: machine learning to detect developer-sensitive smells**. Empirical Software Engineering, 27(7).

250 FERNANDES, E.; CHÁVEZ, A.; GARCIA, A.; FERREIRA, I.; CEDRIM, D.; SOUSA, L. ; OIZUMI, W.. **Refactoring effect on internal quality attributes: What haven't they told you yet?** Information and Software Technology, 126:106347, 2020.

251 LACERDA, G.; PETRILLO, F.; PIMENTA, M. ; GUÉHÉNEUC, Y. G.. **Code smells and refactoring: A tertiary systematic review of challenges and observations**. Journal of Systems and Software (JSS), 167:110610, 2020.

252 YAMASHITA, T.; MILLAR, R. J.. **Likert Scale**, p. 2938–2941. Springer International Publishing, Cham, 2021.

253 OPDYKE, W. F.. **Refactoring: An aid in designing application frameworks and evolving object-oriented systems**. In: PROC. OF 1990 SYMPOSIUM ON OBJECT-ORIENTED PROGRAMMING EMPHASIZING PRACTICAL APPLICATIONS (SOOPPA), 1990.

254 QAYUM, F.; HECKEL, R.. **Search-based refactoring using unfolding of graph transformation systems**. Electronic Communications of the EASST, 38, 2011.

255 GALSTER, M.; WEYNS, D.; TOFAN, D.; MICHALIK, B. ; AVGERIOU, P.. **Variability in software systems—a systematic literature review**. IEEE Transactions on Software Engineering, 40(3):282–306, 2013.

256 TSANTALIS, N.; KETKAR, A. ; DIG, D.. **Refactoringminer 2.0**. IEEE Transactions on Software Engineering, 2020.

257 DE MELLO, R.; OLIVEIRA, R.; UCHÔA, A.; OIZUMI, W.; GARCIA, A.; FONSECA, B. ; DE MELLO, F.. **Recommendations for developers identifying code smells**. IEEE Software, 2022.

259 VASCONCELOS, A.. **Explorando métricas de código para a detecção de long envious method**. Trabalho de Conclusão de Curso, Computing Institute (IC), Federal University of Alagoas, Brazil, 2023.

260 BIBIANO, A. C.; UCHÔA, A.; ASSUNÇÃO, W. K.; TENÓRIO, D.; COLANZI, T. E.; VERGILIO, S. R. ; GARCIA, A.. **Composite refactoring: Representations, characteristics and effects on software projects**. Information and Software Technology, 156:107134, 2023.

261 TUFANO, M.; PALOMBA, F.; BAVOTA, G.; OLIVETO, R.; DI PENTA, M.; DE LUCIA, A. ; POSHYVANYK, D.. **When and why your code starts to smell bad (and whether the smells go away)**. TSE (2017), 43(11):1063–1088, 2017.

262 PALOMBA, F.; BAVOTA, G.; DI PENTA, M.; FASANO, F.; OLIVETO, R. ; DE LUCIA, A.. **A large-scale empirical study on the lifecycle of code smell co-occurrences**. IST (2018), 99:1–10, 2018.

263 OPDYKE, W. F.. **Refactoring Object-Oriented Frameworks**. PhD thesis, USA, 1992.

264 ZHAO, S.; BIAN, Y. ; ZHANG, S.. **A review on refactoring sequential program to parallel code in multicore era**. In: INTERNATIONAL CONFERENCE ON INTELLIGENT COMPUTING AND INTERNET OF THINGS. IEEE, jan 2015.

265 LAGUNA, M. A.; CRESPO, Y.. **A systematic mapping study on software product line evolution: From legacy system reengineering to product line refactoring**. Science of Computer Programming, 78(8):1010–1034, aug 2013.

266 MISBHAUDDIN, M.; ALSHAYEB, M.. **UML model refactoring: a systematic literature review**. Empirical Software Engineering, 20(1):206–251, oct 2015.

267 SIDHU, B. K.; SINGH, K. ; SHARMA, N.. **Refactoring UML models of object-oriented software: A systematic review**. International Journal of Software Engineering and Knowledge Engineering, 28(09):1287–1319, sep 2018.

268 TAVARES, C. S.; FERREIRA, F. ; FIGUEIREDO, E.. **A systematic mapping of literature on software refactoring tools**. In: XIV BRAZILIAN SYMPOSIUM ON INFORMATION SYSTEMS (SBSI). ACM, 2018.

269 COELHO, F.; MASSONI, T. ; ALVES, E. L.. **Refactoring-aware code review: A systematic mapping study**. In: IEEE/ACM 3RD INTERNATIONAL WORKSHOP ON REFACTORING (IWoR). IEEE, may 2019.

270 VASSALLO, C.; PALOMBA, F. ; GALL, H. C.. **Continuous refactoring in CI: A preliminary study on the perceived advantages and barriers**. In: IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE AND EVOLUTION (ICSME). IEEE, sep 2018.

271 BAQAIS, A. A. B.; ALSHAYEB, M.. **Automatic software refactoring: a systematic literature review**. Software Quality Journal, 28(2):459–502, dec 2019.

272 MARIANI, T.; VERGILIO, S. R.. **A systematic review on search-based refactoring**. Information and Software Technology, 83:14–34, mar 2017.

273 MOHAN, M.; GREER, D.. **A survey of search-based refactoring for software maintenance**. Journal of Software Engineering Research and Development, 6(1), feb 2018.

274 MENS, T.; TOURWE, T.. **A survey of software refactoring**. IEEE Transactions on Software Engineering, 30(2):126–139, feb 2004.

275 ABEBE, M.; ; AND, C.-J. Y.. **Classification and summarization of software refactoring researches: A literature review approach**. In: ADVANCED SCIENCE AND TECHNOLOGY LETTERS. Science & Engineering Research Support soCiety, apr 2014.

276 DALLAL, J. A.. **Identifying refactoring opportunities in object-oriented code: A systematic literature review**. Information and Software Technology, 58:231–249, feb 2015.

277 SINGH, S.; KAUR, S.. **A systematic literature review: Refactoring for disclosing code smells in object oriented software.** Ain Shams Engineering Journal, 9(4):2129–2151, dec 2018.

278 DALLAL, J. A.; ABDIN, A.. **Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: A systematic literature review.** IEEE Transactions on Software Engineering, 44(1):44–69, jan 2018.

279 KAUR, S.; SINGH, P.. **How does object-oriented code refactoring influence software quality? research landscape and challenges.** Journal of Systems and Software, 157:110394, nov 2019.

280 SHARMA, T.; SURYANARAYANA, G. ; SAMARTHYAM, G.. **Challenges to and solutions for refactoring adoption: An industrial perspective.** IEEE Software, 32(6):44–51, nov 2015.

281 LI, Z.; HARMAN, M. ; HIERONS, R. M.. **Search algorithms for regression test case prioritization.** IEEE Transactions on software engineering, 33(4):225–237, 2007.

282 MANSOUR, N.; BAHSOON, R. ; BARADHI, G.. **Empirical comparison of regression test selection algorithms.** Journal of Systems and Software, 57(1):79–90, 2001.

283 BIBIANO, A. C.. **Replication package of survey on recomposite**, 2023. Available at: <https://doi.org/10.5281/zenodo.7883634>.

284 BORBA, P.; SAMPAIO, A. ; CORNÉLIO, M.. **A refinement algebra for object-oriented programming.** In: ECOOP 2003–OBJECT-ORIENTED PROGRAMMING: 17TH EUROPEAN CONFERENCE, DARMSTADT, GERMANY, JULY 21-25, 2003. PROCEEDINGS 17, p. 457–482. Springer, 2003.

285 GHEYI, R.; BORBA, P.. **Refactoring alloy specifications.** Electronic Notes in Theoretical Computer Science, 95:227–243, 2004.

PS1 Ó CINNÉIDE, M.; NIXON, P.. **A methodology for the automated introduction of design patterns.** In: PROCEEDINGS OF THE 7TH INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE (ICSM), p. 463–472, 1999.

PS10 SZŐKE, G.; NAGY, C.; FÜLÖP, L.; FERENC, R. ; GYIMÓTHY, T.. **Faultbuster: An automatic code smell refactoring toolset.** In: 15TH SCAM, p. 253–258, 2015.

PS11 MURPHY-HILL, E.; PARNIN, C. ; BLACK, A.. **How we refactor, and how we know it.** IEEE Trans. Softw. Eng., 38(1):5–18, 2012.

PS12 MEANANEATRA, P.. **Identifying refactoring sequences for improving software maintainability.** In: 27TH ASE, p. 406–409, 2012.

PS13 KIM, J.; BATORY, D.; DIG, D. ; AZANZA, M.. **Improving refactoring speed by 10x.** In: PROCEEDINGS OF THE 38TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), p. 1145–1156, 2016.

PS14 LIN, Y.; PENG, X.; CAI, Y.; DIG, D.; ZHENG, D. ; ZHAO, W.. **Interactive and guided architectural refactoring with search-based recommendation.** In: PROCEEDINGS OF THE 24TH INTERNATIONAL SYMPOSIUM ON FOUNDATIONS OF SOFTWARE ENGINEERING (FSE), p. 535–546, 2016.

PS15 HARMAN, M.; TRATT, L.. **Pareto optimal search based refactoring at the design level.** In: PROCEEDINGS OF THE 9TH GENETIC AND EVOLUTIONARY COMPUTATION CONFERENCE (GECCO), p. 1106–1113, 2007.

PS152 QAYUM, F.; HECKEL, R.. **Search-based refactoring using unfolding of graph transformation systems.** Electronic Communications of the EASST, 38, 2011.

PS16 MKAOUER, M. W.; KESSENTINI, M.; BECHIKH, S.; DEB, K. ; Ó CINNÉIDE, M.. **Recommendation system for software refactoring using innovization and interactive dynamic optimization.** In: PROCEEDINGS OF THE 29TH INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING (ASE), p. 331–336, 2014.

PS17 RAYCHEV, V.; SCHÄFER, M.; SRIDHARAN, M. ; VECHEV, M.. **Refactoring with synthesis.** ACM SIGPLAN Notices, 48(10):339–354, 2013.

PS18 KIM, J.; BATORY, D. ; DIG, D.. **Scripting parametric refactorings in Java to retrofit design patterns.** In: PROCEEDINGS OF THE 31ST INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE AND EVOLUTION (ICSME), p. 211–220, 2015.

PS19 BEN FADHEL, A.; KESSENTINI, M.; LANGER, P. ; WIMMER, M.. **Search-based detection of high-level model changes.** In: PROCEEDINGS OF THE 28TH IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE (ICSM), p. 212–221, 2012.

PS2 VILLAVICENCIO, G.. **A new software maintenance scenario based on refactoring techniques.** In: PROCEEDINGS OF THE 16TH EUROPEAN CONFERENCE ON SOFTWARE MAINTENANCE AND REENGINEERING (CSMR), p. 341–346, 2012.

PS20 QAYUM, F.; HECKEL, R.; CORRADINI, A.; MARGARIA, T.; PADBERG, J. ; TAENTZER, G.. **Search-based refactoring based on unfolding of graph transformation systems.** In: PROCEEDINGS OF THE 5TH INTERNATIONAL CONFERENCE ON GRAPH TRANSFORMATION (ICGT): DOCTORAL SYMPOSIUM (DS), p. 1–14, 2010.

PS21 MAHOUACHI, R.; KESSENTINI, M. ; Ó CINNÉIDE, M.. **Search-based refactoring detection.** In: PROCEEDINGS OF THE 15TH GENETIC AND EVOLUTIONARY COMPUTATION CONFERENCE (GECCO), p. 205–206, 2013.

PS22 OUNI, A.; KESSENTINI, M. ; SAHRAOUI, H.. **Search-based refactoring using recorded code changes.** In: PROCEEDINGS OF THE 17TH EUROPEAN CONFERENCE ON SOFTWARE MAINTENANCE AND REENGINEERING (CSMR), p. 221–230, 2013.

PS23 OUNI, A.; KESSENTINI, M.; SAHRAOUI, H. ; HAMDI, M. S.. **Search-based refactoring: Towards semantics preservation.** In: PROCEEDINGS OF THE 28TH IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE (ICSM), p. 347–356, 2012.

PS24 PIVETA, E.; ARAÚJO, J.; PIMENTA, M.; MOREIRA, A.; GUERREIRO, P. ; PRICE, R. T.. **Searching for opportunities of refactoring sequences: Reducing the search space.** In: PROCEEDINGS OF THE 32ND INTERNATIONAL CONFERENCE ON COMPUTER SOFTWARE AND APPLICATIONS (COMPSAC), p. 319–326, 2008.

PS25 PRETE, K.; RACHATASUMRIT, N.; SUDAN, N. ; KIM, M.. **Template-based reconstruction of complex refactorings.** In: PROCEEDINGS OF THE 26TH INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE (ICSM), p. 1–10, 2010.

PS26 OUNI, A.; KESSENTINI, M.; SAHRAOUI, H. ; HAMDI, M. S.. **The use of development history in software refactoring using a multi-objective evolutionary algorithm.** In: PROCEEDINGS OF THE 15TH GENETIC AND EVOLUTIONARY COMPUTATION CONFERENCE (GECCO), p. 1461–1468, 2013.

PS27 GRIFFITH, I.; WAHL, S. ; IZURIETA, C.. **Truerefactor: An automated refactoring tool to improve legacy system and application comprehensibility.** In: PROCEEDINGS OF THE 24TH INTERNATIONAL CONFERENCE ON COMPUTER APPLICATIONS IN INDUSTRY AND ENGINEERING (CAINE), p. 1–6, 2011.

PS28 FOSTER, S.; GRISWOLD, W. ; LERNER, S.. **WitchDoctor: IDE support for real-time auto-completion of refactorings.** In: PROCEEDINGS OF THE 34TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), p. 222–232, 2012.

PS3 KUHLEMANN, M.; LIANG, L. ; SAAKE, G.. **Algebraic and cost-based optimization of refactoring sequences.** In: 2ND MDPLE CO-LOCATED WITH 6TH ECMFA, 2010.

PS5 Ó CINNÉIDE, M.; NIXON, P.. **Composite refactorings for java programs.** In: PROCEEDINGS OF THE WORKSHOP ON FORMAL TECHNIQUES FOR JAVA PROGRAMS, CO-LOCATED WITH THE 14TH EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING (ECOOP), p. 1–6, 2000.

PS6 SZOKE, G.; NAGY, C.; FERENC, R. ; GYIMÓTHY, T.. **Designing and developing automated refactoring transformations: An experience report.** In: PROCEEDINGS OF THE 23RD INTERNATIONAL CONFERENCE ON SOFTWARE ANALYSIS, EVOLUTION AND REENGINEERING (SANER), p. 693–697, 2016.

PS7 DE OLIVEIRA, M. C.. **DRACO: Discovering refactorings that improve architecture using fine-grained co-change dependencies.** In: PROCEEDINGS OF THE 11TH JOINT MEETING ON FOUNDATIONS OF SOFTWARE ENGINEERING (FSE), p. 1018–1021, 2017.

PS8 FATIREGUN, D.; HARMAN, M. ; HIERONS, R.. **Evolving transformation sequences using genetic algorithms.** In: PROCEEDINGS OF THE 4TH INTERNATIONAL WORKSHOP ON SOURCE CODE ANALYSIS AND MANIPULATION (SCAM), p. 65–74, 2004.

PS9 Ó CINNÉIDE, M.; TRATT, L.; HARMAN, M.; COUNSELL, S. ; HEMATI MOGHADAM, I.. **Experimental assessment of software metrics using automated refactoring.** In: PROCEEDINGS OF THE 5TH INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING AND MEASUREMENT (ESEM), p. 49–58, 2012.

S02 BIBIANO, A. C.; FERNANDES, E.; OLIVEIRA, D.; GARCIA, A.; KALINOWSKI, M.; FONSECA, B.; OLIVEIRA, R.; OLIVEIRA, A. ; CEDRIM, D.. **A quantitative study on characteristics and effect of batch refactoring on code smells.** In: PROCEEDINGS OF THE 13TH INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING AND MEASUREMENT (ESEM), p. 1–11. IEEE, 2019.

S04 KIM, M.; ZIMMERMANN, T. ; NAGAPPAN, N.. **An empirical study of refactoring challenges and benefits at Microsoft.** IEEE Trans. Softw. Eng., 40(7):633–649, 2014.

S04 MOESUS, N.; SCHOLZE, M.; SCHLESINGER, S. ; HERBER, P.. **Automated selection of software refactorings that improve performance.** In: ICSOFT, p. 67–78, 2018.

S10 MOHAN, M.; GREER, D.. **Using a many-objective approach to investigate automated refactoring.** Information and Software Technology, 112:83–101, 2019.

S101 HEROLD, S.; MAIR, M.. **Recommending refactorings to re-establish architectural consistency.** In: EUROPEAN CONFERENCE ON SOFTWARE ARCHITECTURE, p. 390–397. Springer, 2014.

S103 YANG, L.; KAMIYA, T.; SAKAMOTO, K.; WASHIZAKI, H. ; FUKAZAWA, Y.. **Refactoringscript: A script and its processor for composite refactoring.** In: SEKE, p. 711–716, 2014.

S11 BOTELHO, G.; BEZERRA, L.; BRITTO, A. ; SILVA, L.. **A many-objective estimation distributed algorithm applied to search based software refactoring.** In: 2018 IEEE CONGRESS ON EVOLUTIONARY COMPUTATION (CEC), p. 1–8. IEEE, 2018.

S110 NEGARA, S.; CHEN, N.; VAKILIAN, M.; JOHNSON, R. E. ; DIG, D.. **A comparative study of manual and automated refactorings.** In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, p. 552–576. Springer, 2013.

S111 VAKILIAN, M.; CHEN, N.; MOGHADDAM, R. Z.; NEGARA, S. ; JOHNSON, R. E.. **A compositional paradigm of automating refactorings.** In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, p. 527–551. Springer, 2013.

S112 SOETENS, Q. D.; PEREZ, J. ; DEMEYER, S.. **An initial investigation into change-based reconstruction of floss-refactorings.** In: PROCEEDINGS OF THE 29TH INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE, p. 384–387. IEEE, 2013.

S115 ZIBRAN, M. F.; ROY, C. K.. **Conflict-aware optimal scheduling of prioritised code clone refactoring.** IET software, 7(3):167–186, 2013.

S117 DORA, S. K.; KANHAR, D.. **Identifying refactoring opportunity in an application: A metric based approach.** In: PROCEEDINGS OF INTERNATIONAL CONFERENCE ON ADVANCES IN COMPUTING, p. 107–112. Springer, 2013.

S118 OUNI, A.; KESSENTINI, M.; SAHRAOUI, H. ; BOUKADOUM, M.. **Maintainability defects detection and correction: a multi-objective approach.** Automated Software Engineering, 20(1):47–79, 2013.

S119 GHANNEM, A.; EL BOUSSAIDI, G. ; KESSENTINI, M.. **Model refactoring using interactive genetic algorithm.** In: PROCEEDINGS OF THE 5TH INTERNATIONAL SYMPOSIUM ON SEARCH BASED SOFTWARE ENGINEERING, p. 96–110. Springer, 2013.

S120 COHEN, J.; AJOULI, A.. **Practical use of static composition of refactoring operations.** In: PROCEEDINGS OF THE 28TH ANNUAL ACM SYMPOSIUM ON APPLIED COMPUTING, p. 1700–1705, 2013.

S122 PÉREZ, J.. **Refactoring planning for design smell correction: Summary, opportunities and lessons learned.** In: 2013 IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE, p. 572–577. IEEE, 2013.

S123 SAADEH, E.; KOURIE, D. G.. **Refactoring with ordered collections of fine-grain transformations.** International Journal of Software Engineering and Knowledge Engineering, 23(03):309–339, 2013.

S124 MAHOUACHI, R.; KESSENTINI, M. ; CINNÉIDE, M. Ó.. **Search-based refactoring detection using software metrics variation.** In: PROCEEDINGS OF THE 5TH INTERNATIONAL SYMPOSIUM ON SEARCH BASED SOFTWARE ENGINEERING, p. 126–140. Springer, 2013.

S125 WONGPIANG, R.; MUENCHAISRI, P.. **Selecting sequence of refactoring techniques usage for code changing using greedy algorithm.** In: PROCEEDINGS OF THE 4TH INTERNATIONAL CONFERENCE ON ELECTRONICS INFORMATION AND EMERGENCY COMMUNICATION, p. 160–164. IEEE, 2013.

S126 AJOULI, A.; COHEN, J. ; ROYER, J.-C.. **Transformations between composite and visitor implementations in java.** In: PROCEEDINGS OF THE 39TH EUROMICRO CONFERENCE ON SOFTWARE ENGINEERING AND ADVANCED APPLICATIONS, p. 25–32. IEEE, 2013.

S127 KESSENTINI, M.; MAHAOUACHI, R. ; GHEDIRA, K.. **What you like in design use to correct bad-smells.** Software Quality Journal, 21(4):551–571, 2013.

S129 KIM, M.; ZIMMERMANN, T. ; NAGAPPAN, N.. **A field study of refactoring challenges and benefits.** In: PROCEEDINGS OF THE ACM SIGSOFT 20TH INTERNATIONAL SYMPOSIUM ON THE FOUNDATIONS OF SOFTWARE ENGINEERING, p. 1–11, 2012.

S131 CHRISTOPOULOU, A.; GIAKOUMAKIS, E. A.; ZAFEIRIS, V. E. ; SOUKARA, V.. **Automated refactoring to the strategy design pattern.** Information and Software Technology, 54(11):1202–1214, 2012.

S133 GE, X.; DUBOSE, Q. L. ; MURPHY-HILL, E.. **Reconciling manual and automatic refactoring.** In: 2012 34TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), p. 211–221. IEEE, 2012.

S135 HILLS, M.; KLINT, P. ; VINJU, J. J.. **Scripting a refactoring with rascal and eclipse.** In: PROCEEDINGS OF THE FIFTH WORKSHOP ON REFACTORING TOOLS, p. 40–49, 2012.

S136 CZIBULA, G.; CZIBULA, I. G.. **Unsupervised restructuring of object-oriented software systems using self-organizing feature maps.** International Journal of Innovative Computing, Information and Control, 8(3 (A)):1689–1705, 2012.

S137 BAVOTA, G.; DE CARLUCCIO, B.; DE LUCIA, A.; DI PENTA, M.; OLIVETO, R. ; STROLLO, O.. **When does a refactoring induce bugs? an empirical study.** In: PROCEEDINGS OF THE 12TH INTERNATIONAL WORKING CONFERENCE ON SOURCE CODE ANALYSIS AND MANIPULATION, p. 104–113. IEEE, 2012.

S138 ZIBRAN, M. F.; ROY, C. K.. **A constraint programming approach to conflict-aware optimal scheduling of prioritized code clone refactoring.** In: 2011 IEEE 11TH INTERNATIONAL WORKING CONFERENCE ON SOURCE CODE ANALYSIS AND MANIPULATION, p. 105–114. IEEE, 2011.

S140 MURGIA, A.; TONELLI, R.; COUNSELL, S.; CONCAS, G. ; MARCHESI, M.. **An empirical study of refactoring in the context of fanin and fanout coupling.** In: 2011 18TH WORKING CONFERENCE ON REVERSE ENGINEERING, p. 372–376. IEEE, 2011.

S141 MARTICORENA, R.; LÓPEZ, C.; PÉREZ, J. ; CRESPO, Y.. **Assisting refactoring tool development through refactoring characterization.** In: PROCEEDINGS OF THE 6TH INTERNATIONAL CONFERENCE ON SOFTWARE AND DATA TECHNOLOGIES, volumen 2, 2011.

S143 LEE, S.; BAE, G.; CHAE, H. S.; BAE, D.-H. ; KWON, Y. R.. **Automated scheduling for clone-based refactoring using a competent ga.** Software: Practice and Experience, 41(5):521–550, 2011.

S145 BIEGEL, B.; SOETENS, Q. D.; HORNIG, W.; DIEHL, S. ; DEMEYER, S.. **Comparison of similarity metrics for refactoring detection.** In: PROCEEDINGS OF THE 8TH WORKING CONFERENCE ON MINING SOFTWARE REPOSITORIES, p. 53–62, 2011.

S146 ZIBRAN, M. F.; ROY, C. K.. **Conflict-aware optimal scheduling of code clone refactoring: A constraint programming approach.** In: 2011 IEEE 19TH INTERNATIONAL CONFERENCE ON PROGRAM COMPREHENSION, p. 266–269. IEEE, 2011.

S147 SANDALSKI, M.; STOYANOVA-DOYCHEVA, A.; POPCHEV, I. ; STOYANOV, S.. **Development of a refactoring learning environment.** Cybernetics and Information Technologies (CIT), 11(2), 2011.

S148 HUANG, J.; CARMINATI, F.; BETEV, L.; ZHU, J. ; LUZZI, C.. **Extractor: An extensible framework for identifying aspect-oriented refactoring opportunities.** In: INTERNATIONAL CONFERENCE ON SYSTEM SCIENCE, ENGINEERING DESIGN AND MANUFACTURING INFORMATIZATION, volumen 2, p. 222–226. IEEE, 2011.

S149 HUANG, J.; CARMINATI, F.; BETEV, L.; LUZZI, C.; LU, Y. ; ZHOU, D.. **Identifying composite refactorings with a scripting language.** In: PROCEEDINGS OF THE 3RD INTERNATIONAL CONFERENCE ON COMMUNICATION SOFTWARE AND NETWORKS, p. 267–271. IEEE, 2011.

S15 NADER-PALACIO, D.; RODRÍGUEZ-CÁRDENAS, D. ; GOMEZ, J.. **Assessing single-objective performance convergence and time complexity for refactoring detection.** In: PROCEEDINGS OF THE GENETIC AND EVOLUTIONARY COMPUTATION CONFERENCE COMPANION, p. 1606–1613, 2018.

S150 MEANANEATRA, P.; RONGVIRIYAPANISH, S. ; APIWATTANAPONG, T.. **Identifying refactoring through formal model based on data flow graph.** In: PROCEEDINGS OF THE 5TH MALAYSIAN CONFERENCE IN SOFTWARE ENGINEERING, p. 113–118. IEEE, 2011.

S151 TSANTALIS, N.; CHATZIGEORGIOU, A.. **Ranking refactoring suggestions based on historical volatility.** In: PROCEEDINGS OF THE 15TH EUROPEAN CONFERENCE ON SOFTWARE MAINTENANCE AND REENGINEERING, p. 25–34. IEEE, 2011.

S154 QAYUM, F.. **Automated assistance for search-based refactoring using unfolding of graph transformation systems.** In: PROCEEDINGS OF THE 5TH INTERNATIONAL CONFERENCE ON GRAPH TRANSFORMATION, p. 407–409. Springer, 2010.

S155 CZIBULA, G.; CZIBULA, I. G.. **Clustering based adaptive refactoring.** Wseas Transactions on Computers, 2010.

S156 COUNSELL, S.; LOIZOU, G. ; NAJJAR, R.. **Evaluation of the 'replace constructors with creation methods' refactoring in java systems.** IET software, 4(5):318–333, 2010.

S158 JENSEN, A. C.; CHENG, B. H.. **On the use of genetic programming for automated refactoring and the introduction of design patterns.** In: PROCEEDINGS OF THE 12TH ANNUAL CONFERENCE ON GENETIC AND EVOLUTIONARY COMPUTATION, p. 1341–1348, 2010.

S16 CHEN, Z.; KWON, Y.-W. ; SONG, M.. **Clone refactoring inspection by summarizing clone refactorings and detecting inconsistent changes during software evolution.** Journal of Software: Evolution and Process, 30(10):e1951, 2018.

S17 OMORI, T.; MARUYAMA, K.. **Comparative study between two approaches using edit operations and code differences to detect past refactorings.** IEICE Transactions on Information and Systems, 101(3):644–658, 2018.

S19 OO, T.; LIU, H. ; NYIRONGO, B.. **Dynamic ranking of refactoring menu items for integrated development environment.** IEEE Access, 6:76025–76035, 2018.

S22 KOZSIK, T.; TÓTH, M. ; BOZÓ, I.. **Free the conqueror! refactoring divide-and-conquer functions.** Future Generation Computer Systems, 79:687–699, 2018.

S23 PANTIUCHINA, J.; LANZA, M. ; BAVOTA, G.. **Improving code: The (mis) perception of quality metrics.** In: 2018 IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE AND EVOLUTION (ICSME), p. 80–91. IEEE, 2018.

S247 SOUSA, L.; CEDRIM, D.; GARCIA, A.; OIZUMI, W.; BIBIANO, A. C.; OLIVEIRA, D.; KIM, M. ; OLIVEIRA, A.. **Characterizing and identifying composite refactorings: Concepts, heuristics and patterns.** In: PROCEEDINGS OF THE 17TH INTERNATIONAL CONFERENCE ON MINING SOFTWARE REPOSITORIES, p. 186–197, 2020.

S248 ALIZADEH, V.; KESSENTINI, M.; MKAOUER, M. W.; OCINNEIDE, M.; OUNI, A. ; CAI, Y.. **An interactive and dynamic search-based approach to software refactoring recommendations.** IEEE Transactions on Software Engineering, 46(9):932–961, 2018.

S25 NYAMAWE, A. S.; LIU, H.; NIU, Z.; WANG, W. ; NIU, N.. **Recommending refactoring solutions based on traceability and code metrics.** IEEE Access, 6:49460–49475, 2018.

S250 BRITO, A.; HORA, A. ; VALENTE, M. T.. **Refactoring graphs: Assessing refactoring over time.** In: 2020 IEEE 27TH INTERNATIONAL CONFERENCE ON SOFTWARE ANALYSIS, EVOLUTION AND REENGINEERING (SANER), p. 367–377. IEEE, 2020.

S258 TARWANI, S.; CHUG, A.. **Application of ao* algorithm in recognizing the optimum refactoring sequence for examining the effect on maintainability: An empirical study.** In: 2021 11TH INTERNATIONAL CONFERENCE ON CLOUD COMPUTING, DATA SCIENCE & ENGINEERING (CONFLUENCE), p. 188–195. IEEE, 2021.

S259 BIBIANO, A. C.; ASSUNÇAO, W. K.; COUTINHO, D.; SANTOS, K.; SOARES, V.; GHEYI, R.; GARCIA, A.; FONSECA, B.; RIBEIRO, M.; OLIVEIRA, D. ; OTHERS. **Look ahead! revealing complete composite refactorings and their smelliness effects.** In: 2021 IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE AND EVOLUTION (ICSME), p. 298–308. IEEE, 2021.

S261 BRITO, A.; HORA, A. ; VALENTE, M. T.. **Characterizing refactoring graphs in java and javascript projects.** Empirical Software Engineering, 26(6):1–43, 2021.

S265 FREIRE, S.; PASSOS, A.; MENDONÇA, M.; SANT'ANNA, C. ; SPÍNOLA, R. O.. **On the influence of uml class diagrams refactoring on code debt: A family of replicated empirical studies.** In: 2020 46TH EUROMICRO CONFERENCE ON SOFTWARE ENGINEERING AND ADVANCED APPLICATIONS (SEAA), p. 346–353. IEEE, 2020.

S266 HOU, D.; YIN, Y.; SU, Q. ; LIU, L.. **Software refactoring scheme based on nsga-ii algorithm.** In: 2020 7TH INTERNATIONAL CONFERENCE ON DEPENDABLE SYSTEMS AND THEIR APPLICATIONS (DSA), p. 447–452. IEEE, 2020.

S27 MEANANEATRA, P.; RONGVIRIYAPANISH, S. ; APIWATTANAPONG, T.. **Refactoring opportunity identification methodology for removing long method smells and improving code analyzability.** IEICE Transactions on Information and Systems, 101(7):1766–1779, 2018.

S273 ADLER, F.; FRASER, G.; GRÜNDINGER, E.; KÖRBER, N.; LABRENZ, S.; LERCHENBERGER, J.; LUKASCZYK, S. ; SCHWEIKL, S.. **Improving readability of scratch programs with search-based refactoring.** In: 2021 IEEE 21ST INTERNATIONAL WORKING CONFERENCE ON SOURCE CODE ANALYSIS AND MANIPULATION (SCAM), p. 120–130. IEEE, 2021.

S274 SGHAIER, O. B.; SAHRAOUI, H. ; FAMELIS, M.. **Metamodel refactoring using constraint solving: a quality-based perspective.** In: 2021 ACM/IEEE INTERNATIONAL CONFERENCE ON MODEL DRIVEN ENGINEERING LANGUAGES AND SYSTEMS COMPANION (MODELS-C), p. 797–806. IEEE, 2021.

S275 CORTELLESSA, V.; DI POMPEO, D.; STOICO, V. ; TUCCI, M.. **On the impact of performance antipatterns in multi-objective software model refactoring optimization.** In: 2021 47TH EUROMICRO CONFERENCE ON SOFTWARE ENGINEERING AND ADVANCED APPLICATIONS (SEAA), p. 224–233. IEEE, 2021.

S276 COUNSELL, S.; DESTEFANIS, G.; SWIFT, S.; ARZOKY, M. ; TAIBI, D.. **On the link between refactoring activity and class cohesion through the prism of two cohesion-based metrics.** In: 2020 IEEE 20TH INTERNATIONAL CONFERENCE ON SOFTWARE QUALITY, RELIABILITY AND SECURITY (QRS), p. 91–98. IEEE, 2020.

S280 FARAH, P. R.; MARIANI, T.; DA ROZA, E. A.; SILVA, R. C. ; VERGILIO, S. R.. **Unsupervised learning for refactoring pattern detection**. In: 2021 IEEE CONGRESS ON EVOLUTIONARY COMPUTATION (CEC), p. 2377–2384. IEEE, 2021.

S29 MAHOUACHI, R.. **Search-based cost-effective software remodularization**. Journal of Computer Science and Technology, 33(6):1320–1336, 2018.

S30 HAN, A.-R.; CHA, S.. **Two-phase assessment approach to improve the efficiency of refactoring identification**. IEEE Transactions on Software Engineering, 44(10):1001–1023, 2017.

S31 KESSENTINI, M.; DEA, T. J. ; OUNI, A.. **A context-based refactoring recommendation approach using simulated annealing: two industrial case studies**. In: PROCEEDINGS OF THE GENETIC AND EVOLUTIONARY COMPUTATION CONFERENCE, p. 1303–1310, 2017.

S32 KEBIR, S.; BORNE, I. ; MESLATI, D.. **A genetic algorithm-based approach for automated refactoring of component-based software**. Information and Software Technology, 88:17–36, 2017.

S33 MKAOUER, M. W.; KESSENTINI, M.; CINNÉIDE, M. Ó.; HAYASHI, S. ; DEB, K.. **A robust multi-objective approach to balance severity and importance of refactoring opportunities**. Empirical Software Engineering, 22(2):894–927, 2017.

S34 CINNÉIDE, M. O.; MOGHADAM, I. H.; HARMAN, M.; COUNSELL, S. ; TRATT, L.. **An experimental search-based approach to cohesion metric evaluation**. Empirical Software Engineering, 22(1):292–329, 2017.

S37 ZAFEIRIS, V. E.; POULIAS, S. H.; DIAMANTIDIS, N. ; GIAKOUMAKIS, E. A.. **Automated refactoring of super-class method invocations to the template method design pattern**. Information and Software Technology, 82:19–35, 2017.

S38 KESSENTINI, M.; WANG, H.. **Detecting refactorings among multiple web service releases: A heuristic-based approach**. In: 2017 IEEE INTERNATIONAL CONFERENCE ON WEB SERVICES (ICWS), p. 365–372. IEEE, 2017.

S39 CHUG, A.; TARWANI, S.. **Determination of optimum refactoring sequence using a* algorithm after prioritization of classes**. In: 2017 INTERNATIONAL CONFERENCE ON ADVANCES IN COMPUTING, COMMUNICATIONS AND INFORMATICS (ICACCI), p. 1624–1630. IEEE, 2017.

S42 CHARALAMPIDOU, S.; AMPATZOGLOU, A.; CHATZIGEORGIOU, A.; GKORTZIS, A. ; AVGERIOU, P.. **Identifying extract method refactoring opportunities based on functional relevance**. IEEE Transactions on Software Engineering, 43(10):954–974, 2016.

S43 OUNI, A.; KESSENTINI, M.; Ó CINNÉIDE, M.; SAHRAOUI, H.; DEB, K. ; INOUE, K.. **More: A multi-objective refactoring recommendation**

approach to introducing design patterns and fixing code smells. Journal of Software: Evolution and Process, 29(5):e1843, 2017.

S45 MORALES, R.; SOH, Z.; KHOMH, F.; ANTONIOL, G. ; CHICANO, F.. **On the use of developers' context for automatic refactoring of software anti-patterns.** Journal of systems and software, 128:236–251, 2017.

S46 RANI, A.; CHHABRA, J. K.. **Prioritization of smelly classes: A two phase approach (reducing refactoring efforts).** In: 2017 3RD INTERNATIONAL CONFERENCE ON COMPUTATIONAL INTELLIGENCE & COMMUNICATION TECHNOLOGY (CICT), p. 1–6. IEEE, 2017.

S47 GE, X.; SARKAR, S.; WITSCHEY, J. ; MURPHY-HILL, E.. **Refactoring-aware code review.** In: 2017 IEEE SYMPOSIUM ON VISUAL LANGUAGES AND HUMAN-CENTRIC COMPUTING (VL/HCC), p. 71–79. IEEE, 2017.

S49 KESSENTINI, M.; MANSOOR, U.; WIMMER, M.; OUNI, A. ; DEB, K.. **Search-based detection of model level changes.** Empirical Software Engineering, 22(2):670–715, 2017.

S50 CHEN, Z.; MOHANAVILASAM, M.; KWON, Y.-W. ; SONG, M.. **Tool support for managing clone refactorings to facilitate code review in evolving software.** In: 2017 IEEE 41ST ANNUAL COMPUTER SOFTWARE AND APPLICATIONS CONFERENCE (COMPSAC), volumen 1, p. 288–297. IEEE, 2017.

S52 RISSETTI, G.; CHARÃO, A. S. ; PIVETA, E. K.. **A set of refactorings for the evolution of fortran programs.** International Journal of High Performance Systems Architecture, 6(2):98–109, 2016.

S53 KHRISHE, Y.; ALSHAYEB, M.. **An empirical study on the effect of the order of applying software refactoring.** In: 2016 7TH INTERNATIONAL CONFERENCE ON COMPUTER SCIENCE AND INFORMATION TECHNOLOGY (CSIT), p. 1–4. IEEE, 2016.

S54 KÁDÁR, I.; HEGEDŰS, P.; FERENC, R. ; GYIMÓTHY, T.. **Assessment of the code refactoring dataset regarding the maintainability of methods.** In: INTERNATIONAL CONFERENCE ON COMPUTATIONAL SCIENCE AND ITS APPLICATIONS, p. 610–624. Springer, 2016.

S55 TAKAHASHI, Y.; NITTA, N.. **Composite refactoring for decoupling multiple classes.** In: 2016 IEEE 23RD INTERNATIONAL CONFERENCE ON SOFTWARE ANALYSIS, EVOLUTION, AND REENGINEERING (SANER), volumen 1, p. 594–598. IEEE, 2016.

S57 SCHUSTER, C.; DISNEY, T. ; FLANAGAN, C.. **Macrofication: Refactoring by reverse macro expansion.** In: EUROPEAN SYMPOSIUM ON PROGRAMMING, p. 644–671. Springer, 2016.

S59 CARACCIOLO, A.; AGA, B.; LUNGU, M. ; NIERSTRASZ, O.. **Marea: A semi-automatic decision support system for breaking dependency cycles.** In: 2016 IEEE 23RD INTERNATIONAL CONFERENCE ON SOFTWARE

ANALYSIS, EVOLUTION, AND REENGINEERING (SANER), volumen 1, p. 482–492. IEEE, 2016.

S60 OUNI, A.; KESSENTINI, M.; SAHRAOUI, H.; INOUE, K. ; DEB, K.. **Multi-criteria code refactoring using search-based software engineering: An industrial case study**. ACM Transactions on Software Engineering and Methodology (TOSEM), 25(3):1–53, 2016.

S62 TARWANI, S.; CHUG, A.. **Sequencing of refactoring techniques by greedy algorithm for maximizing maintainability**. In: 2016 INTERNATIONAL CONFERENCE ON ADVANCES IN COMPUTING, COMMUNICATIONS AND INFORMATICS (ICACCI), p. 1397–1403. IEEE, 2016.

S63 MOHAN, M.; GREER, D. ; MCMULLAN, P.. **Technical debt reduction using search based automated refactoring**. Journal of Systems and Software, 120:183–194, 2016.

S66 FONTANA, F. A.; ZANONI, M. ; ZANONI, F.. **A duplicated code refactoring advisor**. In: INTERNATIONAL CONFERENCE ON AGILE SOFTWARE DEVELOPMENT, p. 3–14. Springer, 2015.

S67 KEBIR, S.; BORNE, I. ; MESLATI, D.. **A genetic algorithm for automated refactoring of component-based software**. In: PROCEEDINGS OF THE 9TH EAI INTERNATIONAL CONFERENCE ON BIO-INSPIRED INFORMATION AND COMMUNICATIONS TECHNOLOGIES (FORMERLY BIONETICS), p. 221–228, 2016.

S68 VEREBI, I.. **A model-based approach to software refactoring**. In: 31ST INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE AND EVOLUTION (ICSME), p. 606–609. IEEE, 2015.

S69 HAN, A.-R.; BAE, D.-H. ; CHA, S.. **An efficient approach to identify multiple and independent move method refactoring candidates**. Information and Software Technology, 59:53–66, 2015.

S71 COUNSELL, S.; LIU, X.; SWIFT, S.; BUCKLEY, J.; ENGLISH, M.; HEROLD, S.; ELDH, S. ; ERMEDAHL, A.. **An exploration of the'introduce explaining variable'refactoring**. In: SCIENTIFIC WORKSHOP PROCEEDINGS OF THE XP2015, p. 1–5, 2015.

S72 GAITANI, M. A. G.; ZAFEIRIS, V. E.; DIAMANTIDIS, N. ; GIAKOUMAKIS, E. A.. **Automated refactoring to the null object design pattern**. Information and Software Technology, 59:33–52, 2015.

S74 LIU, H.; LIU, Q.; LIU, Y. ; WANG, Z.. **Identifying renaming opportunities by expanding conducted rename refactorings**. IEEE Transactions on Software Engineering, 41(9):887–900, 2015.

S75 OUNI, A.; KESSENTINI, M.; SAHRAOUI, H.; INOUE, K. ; HAMDI, M. S.. **Improving multi-objective code-smells correction using development history**. Journal of Systems and Software, 105:18–39, 2015.

S76 MKAOUER, W.; KESSENTINI, M.; SHAOUT, A.; KOLIGHEU, P.; BECHIKH, S.; DEB, K. ; OUNI, A.. **Many-objective software remodularization using nsga-iii**. ACM Transactions on Software Engineering and Methodology (TOSEM), 24(3):1–45, 2015.

S77 ZARRAS, A. V.; VARTZIOTIS, T. ; VASSILIADIS, P.. **Navigating through the archipelago of refactorings**. In: PROCEEDINGS OF THE 2015 10TH JOINT MEETING ON FOUNDATIONS OF SOFTWARE ENGINEERING, p. 922–925, 2015.

S79 OUNI, A.; KESSENTINI, M.; BECHIKH, S. ; SAHRAOUI, H.. **Prioritizing code-smells correction tasks using chemical reaction optimization**. Software Quality Journal, 23(2):323–361, 2015.

S81 SANTOS, H.; PIMENTEL, J. F.; DA SILVA, V. T. ; MURTA, L.. **Software rejuvenation via a multi-agent approach**. Journal of Systems and Software, 104:41–59, 2015.

S85 NAIYA, N.; COUNSELL, S. ; HALL, T.. **The relationship between depth of inheritance and refactoring: An empirical study of eclipse releases**. In: 2015 41ST EUROMICRO CONFERENCE ON SOFTWARE ENGINEERING AND ADVANCED APPLICATIONS, p. 88–91. IEEE, 2015.

S86 MKAOUER, M. W.; KESSENTINI, M.; BECHIKH, S. ; CINNÉIDE, M. Ó.. **A robust multi-objective approach for software refactoring under uncertainty**. In: PROCEEDINGS OF THE 6TH INTERNATIONAL SYMPOSIUM ON SEARCH BASED SOFTWARE ENGINEERING, p. 168–183. Springer, 2014.

S89 COUNSELL, S.; SWIFT, S.; MURGIA, A.; TONELLI, R.; MARCHESI, M. ; CONCAS, G.. **Are some refactorings attached to fault-prone classes and others to fault-free classes?** In: INTERNATIONAL CONFERENCE ON AGILE SOFTWARE DEVELOPMENT, p. 136–147. Springer, 2014.

S90 GLIGORIC, M.; SCHULTE, W.; PRASAD, C.; VAN VELZEN, D.; NARASAMDYA, I. ; LIVSHITS, B.. **Automated migration of build scripts using dynamic analysis and search-based refactoring**. ACM SIGPLAN Notices, 49(10):599–616, 2014.

S91 BAVOTA, G.; DE LUCIA, A.; MARCUS, A. ; OLIVETO, R.. **Automating extract class refactoring: an improved method and its evaluation**. Empirical Software Engineering, 19(6):1617–1664, 2014.

S93 WONGPIANG, R.; MUENCHAISRI, P.. **Comparing heuristic search methods for selecting sequence of refactoring techniques usage for code changing**. In: INTERNATIONAL MULTICONFERENCE OF ENGINEERS AND COMPUTER SCIENTISTIS (IMECS2014), volumen 1, 2014.

S94 CHISALITA-CRETU, C.. **Evolutionary approach for the strategy-based refactoring selection**. In: PROCEEDINGS OF THE WORLD CONGRESS ON ENGINEERING AND COMPUTER SCIENCE, volumen 1, 2014.

S95 MKAOUER, M. W.; KESSENTINI, M.; BECHIKH, S.; DEB, K. ; Ó CINNÉIDE, M.. **High dimensional search-based software engineering: finding tradeoffs among 15 objectives for automating software refactoring using nsga-iii.** In: PROCEEDINGS OF THE 2014 ANNUAL CONFERENCE ON GENETIC AND EVOLUTIONARY COMPUTATION, p. 1263–1270, 2014.

S96 UNTERHOLZNER, M.. **Improving refactoring tools in smalltalk using static type inference.** Science of Computer Programming, 96:70–83, 2014.

S99 AMAL, B.; KESSENTINI, M.; BECHIKH, S.; DEA, J. ; SAID, L. B.. **On the use of machine learning and search-based software engineering for ill-defined fitness function: a case study on software refactoring.** In: PROCEEDINGS OF THE 6TH INTERNATIONAL SYMPOSIUM ON SEARCH BASED SOFTWARE ENGINEERING, p. 31–45. Springer, 2014.

aaaa

aaaa