Characterizing and Identifying Composite Refactorings: Concepts, Heuristics and Patterns

Leonardo Sousa
Electrical & Computer Engineering
Carnegie Mellon University, USA
leo.sousa@sv.cmu.edu

Diego Cedrim
Amazon
Brazil
dccedrim@amazon.com

Alessandro Garcia, Willian Oizumi
PUC-Rio, Brazil
{afgarcia,woizumi}@inf.puc-rio.br

Ana C. Bibiano, Daniel Oliveira
PUC-Rio, Brazil
{abibiano,doliveira}@inf.puc-rio.br

Miryung Kim
UCLA, USA
miryung@cs.ucla.edu

Anderson Oliveira
PUC-Rio, Brazil
aoliveira@inf.puc-rio.br

ABSTRACT

Refactoring consists of a program transformation applied to improve the internal structure of a program, for instance, by contributing to remove code smells. Developers often apply multiple interrelated refactorings called *composite refactoring*. Even though composite refactoring is a common practice, an investigation from different points of view on how composite refactoring manifests in practice is missing. Previous empirical studies also neglect how different kinds of composite refactorings affect the removal, prevalence or introduction of smells. To address these matters, we provide a conceptual framework and two heuristics to respectively characterize and identify composite refactorings within and across commits. Then, we mined the commit history of 48 GitHub software projects, in which we identified and analyzed 24,911 composite refactorings involving 104,505 single refactorings. Amongst several findings, we observed that most composite refactorings occur in the same commit and have the same refactoring type. We also found that several refactorings are semantically related to each other, which occur in different parts of the system but are still related to the same task. Moreover our study is the first to reveal that many smells are introduced in a program due to "incomplete" composite refactorings. Additionally, our study is also the first to reveal 111 patterns of composite refactorings that frequently introduce or remove certain smell types. These patterns can be used as guidelines for developers to improve their refactoring practices as well as for designers of recommender systems.

ACM Reference Format:
Leonardo Sousa, Diego Cedrim, Alessandro Garcia, Willian Oizumi, Ana C. Bibiano, Daniel Oliveira, Miryung Kim, and Anderson Oliveira. 2020. Characterizing and Identifying Composite Refactorings: Concepts, Heuristics and Patterns. In *Proceedings of (MSR'20)*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Software refactoring is a widely used technique in practice [9, 11, 13, 18, 19, 32, 47]. Refactoring consists of a program transformation used to improve software structure, such as removing code smells [14]. Well-known refactoring types include *Extract Method*, *Rename Method*, and *Move Method*. Since the term refactoring first appeared in the literature [14, 35], studies have been actively investigating it [2, 3, 8, 11, 13, 18, 19, 24, 31, 32, 42, 47]. Most of these studies analyze the characteristics and the impact of each single refactoring on the software structure.

However, from 40% to 60% of the times, developers apply more than one refactoring in conjunction [7, 32], even for removing simple code smells, such as Long Methods [14]. In other words, developers often apply which we call here as *composite refactoring*. A composite refactoring – from now on also called composites – comprises two or more interrelated refactorings that affect one or more elements [7, 9, 33, 41]. There are two broad categories of composites: (i) temporally-related composite, *i.e.*, those refactorings applied in the same commit and are likely to be related to the same developer's task, and (ii) spatial composite, *i.e.*, a set of refactorings applied in structurally related code elements, regardless whether they are performed at the same change (commit) or not.

However, recent studies (e.g., [7, 9, 39, 48]) have strictly focused their analysis on a single category of composite (Section 2). For example, Palomba *et al.* [39] and Tufano *et al.* [48] only analyze temporally-related composites, while Bibiano *et al.* [7] and Brito *et al.* [9] explore spatial composites. As there is no study that analyzes these different categories all together, a more comprehensive understanding of composites is missing. There is not even a unified conceptual framework that supports such a holistic characterization and study of composites.

Moreover, when composite categories are studied only under a single perspective, the actual impact of refactoring on the program structure – *e.g.*, removal or introduction of smells – is not properly understood (Section 2). For example, while certain complex smells are likely to be fully removed over time (*e.g.*, a God Class) through a spatial composite refactoring, other smells (*e.g.*, Shotgun Surgery) may be removed in a single commit, but require changes in non-structurally related parts of the program. Unfortunately, existing studies that assess the impact of refactoring on code smells [5, 7, 11, 48] do not consider both categories of composites.

To address the aforementioned issues, we mined the commit history of 48 GitHub software projects (i) to identify the characteristics of different categories of composite refactorings, and (ii) their effect on either removing or introducing smells. To support our study, we provide a conceptual framework and two heuristics for detecting composites. The heuristics are named *commit-based* and *range-based* heuristics, and they serve to automatically identify composites in software projects. The first supports the analysis of

MSR'20, May 2020, Seoul, Korea
2020. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

refactorings which have a temporal relation. The second intends to capture refactorings that have a spatial relation. These heuristics enabled us to investigate composites and their impact on smells from different perspectives. We expect that our contributions and study findings can help tool builders by uncovering the blind spots on the relation between composite refactoring and smells that have not been properly addressed by the community. Our contributions and study findings can be summarized as follows.

First, we provide a formal and unambiguous definition for composites, which also serves to guide researchers who aim to further investigate composites. Our two proposed heuristics enabled us to reveal characteristics of composites in practice, which are overlooked by previous studies [7, 9, 32]. We present some of these characteristics below.

Second, we observe that nearly 41% of composites are complex, *i.e.*, are comprised by 3 to 20 interrelated refactorings, which contradicts a recent finding [7]. The majority of the composites are confined to the same commit and homogeneously formed by refactorings of the same type, *e.g.*, various syntactically related method extractions. There is also a non-negligible frequency of: (i) heterogeneous and cross-commit composites, and (ii) semantically related composites within the same commit, *i.e.*, sequences of refactorings located in different parts of the code, but still related to the same task (*e.g.*, removing non-trivial, scattered smells).

Third, contradicting previous findings [6, 7, 11, 44], we observe that refactoring do have a considerable effect on smells. We found that nearly 50% of composites either remove or introduce smells. Previous studies often suggest otherwise. For instance, Bavota *et al* [6] stated that refactorings are not related to smell removal. Cedrim *et al.* [11] and Bibiano *et al* [7] reported that refactorings are most often neutral, *i.e.*, neither introduce nor remove smells. These studies either analyze each single refactoring individuallyy or multiple refactorings affecting only a single element.

Fourth, our heuristics enabled us to identify patterns of composites that recurrently introduce or remove specific smell types. No existing study in the literature, including recent studies (e.g., [7, 11]), systematically derived and documented such a comprehensive set of smell-affecting composite patterns. A manual analysis confirmed a total of 111 composite-smell patterns: 84 smell-removing patterns and 27 smell-introducing patterns. As refactoring tools tend to be underused [32], these patterns can be used to improve recommendation systems [17, 23, 30, 34, 36] by recommending removal patterns that developers do in practice; thus, increasing the chance of them adopt automated refactoring tools.

Fifth, our study also contributes with a comprehensive replication package [38]. Our dataset is available for other researchers who are interested in studying composites and their effects on smells. We also provide the scripts that we used to implement the proposed heuristics as well as the catalog of composite-smell patterns for eleven smell types.

2 RELATED WORK AND EXAMPLE

Diverse views on composite refactoring. Many researchers have investigated composites [7, 9, 27, 32, 45, 48, 49]. However, they use different terms (*e.g.*, *batch refactoring* [7]) or definitions to refer to composite refactoring. Some studies consider a composite as a set of two or more interrelated refactorings applied by the same developer [7, 23, 30, 32, 46]. Other studies define a composite as a set of refactorings applied by *multiple developers* [19, 27, 45]. Bibiano *et al.* [7] consider the scope of a composite refactoring as an *individual code element*. Other studies consider that a composite refactoring may be applied in the scope of *multiple elements* [19, 27, 30, 32, 45, 46]. There is even a study that assumes time constraints to define a composite [32]. There are also studies that have proposed approaches to recommend composite refactorings [23, 30, 46].

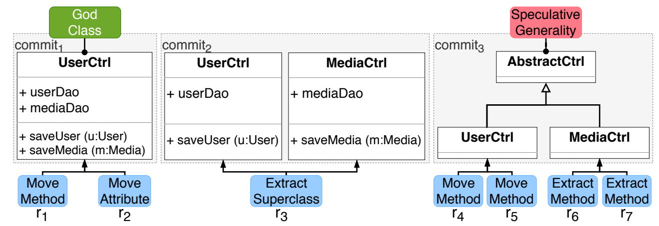

Figure 1: Refactorings applied to the Mobile Media

To the extent of our knowledge, Bibiano *et al.* [7], Vassalo *et al.* [49], and Brito *et al.* [9] are the most recent studies that investigate composites. Unfortunately, these studies tend to only investigate composite through a single perspective. Additionally, neither of them provided both a clear definition of composite refactoring and also a systematic investigation about its effects on smells. For example, Bibiano *et al.* [7] only provided a partial view on composite refactoring since they analyze only composites in the scope of individual code elements. Hence, composite refactorings that crosscut two or more elements were not completely investigated. Moreover, their overly restrictive definition of composite can lead to some findings that may not hold in practice. Next, we present an example that illustrates how their restrictive analysis of composites can lead to misleading results.

Effect of composites on smells. For this discussion, we will rely on the example of Figure 1. This figure shows three commits of Mobile Media (MM), a software product line to derive mobile applications [53]. A developer performed seven refactorings: $r_1, r_2, .., r_7$ along these commits. We may have different instances of composites according to the chosen composite definition. Bibiano *et al.* [7] define composite as two or more refactorings within the scope of a single element. Thus, they would consider only $cr_1 = [r_1, r_2, r_3, r_4, r_5]$ and $cr_2 = [r_3, r_6, r_7]$ as composites. But only restricting composites to those occurring in the context of an element may be inappropriate to investigate the effects of composites on smells. For example, in Figure 1, the refactorings r_1 and r_2 removed the God Class. As these refactorings belong to the composite cr_1, Bibiano *et al.* would conclude that composites have a positive effect on the program structure since cr_1 reduced the incidence of smells. However, this conclusion is misleading due to their narrow composite definition.

Let us consider the r_3 refactoring (*Extract Superclass*), which crosscuts multiple elements. This refactoring creates a superclass (`AbstractCtrl`) shared by `UserCtrl` and `MediaCtrl`, which led to the introduction of the Speculative Generality [14]. Since the smell is introduced in the scope of another element, Bibiano *et al.* would not consider it when assessing the effect of a composite. In this

scenario, the composite removed a smell (God Class) but introduced another (Speculative Generality). Therefore, Bibiano *et al.* should have concluded that composites have no effect on the introduction or removal of smells. As Bibiano *et al.* do not consider the scope of all elements affected by the refactorings, they only provide a partial view of the effects of composite on smells. To have a better understanding on composite refactorings and their effect on smells, we propose two heuristics (Section 3.3) to identify composites that affect the scope of one to multiple elements.

3 CHARACTERIZING AND IDENTIFYING COMPOSITE REFACTORING

In this section, we define basic concepts for supporting the understanding of composite refactoring (Section 3.1). We use them to identify the limitations of an existing heuristic (Section 3.2) and to propose two new heuristics (Section 3.3).

3.1 A Conceptual Framework

This section presents a conceptual framework for composite refactoring. We used this framework to provide a foundation for our heuristics (Section 3.3) and our empirical study. Other researchers can also use it to conduct studies based on unambiguous concepts.

3.1.1 Composite Refactoring. Composite refactoring occurs when two or more interrelated refactorings are applied to a set of code elements. Thus, $cr = [r_1, r_2, \cdots, r_n]$ is a composite of size n if $n \geq 2$. Additionally, the refactorings within the composite should be interrelated. The notion of interrelation depends on the composite scope (Section 3.1.4). Most studies restrict the composite to refactorings applied by the same developer [7, 32, 37, 43]. However, developers can work together to apply a composite [19]. This scenario can happen, for example, when they have to team up to plan and perform a major restructuring in the system, or when they create branches to apply refactoring exclusively [19].

3.1.2 Composite Uniformity. All the refactorings in the composite can have the same type or not, which we define as composite *uniformity*. In this context, $type(r_i)$ is a function that returns the type of the refactoring r_i. In our example of Figure 1, $type(r_1) =$ *Move Method*. Therefore, the composite $cr = [r_1, r_2, \cdots, r_n]$ is *heterogeneous* if and only if $|type(r_1) \cup type(r_2) \cdots \cup type(r_n)| > 1$. If $|type(r_1) \cup type(r_2) \cdots \cup type(r_n)| = 1$, then the composite is *homogeneous*. Most studies do not consider that a composite only exists if all refactorings have the same type [32, 37, 40, 43].

3.1.3 Composite Timespan. A developer can start a composite in a commit and finish it in the same commit or in the subsequent commits. In this sense, *composite timespan* indicates if the composite is either *single-commit* or *cross-commit*. To identify the timespan, let us define the function $commit(r)$ to find the commit where the refactoring r was performed. Thus, a composite $cr = [r_1, r_2, \cdots, r_n]$ is *cross-commit* if and only if $|commit(r_1) \cup \cdots \cup commit(r_n)| > 1$. Similarly, if $|commit(r_1) \cup \cdots \cup commit(r_n)| = 1$, then cr is *single-commit*. Several studies of refactoring only consider major version [6] or a single commit [11], or the entire project history [7].

3.1.4 Refactoring and Composite Scope. Elements directly affected by the refactoring constitute the *refactoring scope*. Given a refactoring r, $scope(r)$ is a function that returns the set of elements belonging to the scope of r. For instance, the refactoring r_1 in Figure 1 (*Move Method*) moved the method `mediaDao` from class `UserCtrl` to `MediaCtrl`. Hence, the refactoring scope is $\{mediaDao, UserCtrl, MediaCtrl\}$. Similar to a single refactoring, composites also have a scope. The *composite scope* is the set of code elements affected by the refactorings within a composite. The composite scope also indicates how the refactorings within the composite are interrelated.

One might naturally say the union of all refactoring scopes from a composite determines the composite scope, but this is not necessarily true in all scenarios. Related studies have different ways to define the composite scope. In general, these studies can be divided into two groups: composite refactoring affects only the scope of a single element [21, 29, 40] or the scope of multiple elements [19, 37]. In the first group, all refactorings within the composite are related to each other because they affect the same element. In the second group, if a refactoring crosscuts two elements, then all refactorings in one element will be related to the refactorings in the other element. For example, a developer applied refactoring r_1 to class `A` and r_2 to class `B`. These two refactorings are not related to each; thus they do not compose a composite. However, the developer applied a refactoring r_3, which moves a method from `A` to `B`. Thus, the three refactorings became related to each other, creating a composite. In this case, the composite scope includes both classes.

3.1.5 Composite Synthesis. The process of grouping interrelated refactorings to find composites is defined as *composite synthesis*. To synthesize a composite, we need first to detect the refactorings that occurred in the system. Related studies have different strategies to identify refactorings applied by developers. A strategy is to analyze the commit message to identify the refactorings [42]. Other strategy is to use a tool that compares two subsequent commits to identify them [47]. For the sake of explanation, let assumes that a refactoring detection tool implements a function R. This function expresses all refactorings in the history H of a system s, which is composed of all refactorings detected between subsequent pairs of commits: $H(s) = \bigcup_{i=1}^{|Commits(s)|-1} R(c_i, c_{i+1})$. To illustrate the output of function $H(s)$, let us visit the MM system presented in Figure 1. This system has four commits, where three of them are represented in the figure. The fourth one is produced as the result of applying the refactorings $\{r_4, r_5, r_6, r_7\}$. Hence, $H(s_1) = R(c_1, c_2) \cup R(c_2, c_3) \cup R(c_3, c_4)$. In other words, $H(s_1)$ contains all refactorings presented in Figure 1, which are $\{r_1, r_2, r_3, r_4, r_5, r_6, r_7\}$.

3.2 Element-Based Heuristic

This section presents a formal definition of the *element-based* heuristic [7], which we will use in our study.

Formal Definition. A heuristic that synthesizes composites using as scope an individual code element, *i.e.*, either a method or a class. The goal of this heuristic is to investigate how composites affect an specific element. Formally, a given composite $cr = [r_1, r_2, \cdots, r_n]$ is synthesized by the element-based heuristic if and only if there is an element e such as $e \in scope(r_i)\ \forall r_i \in cr$. For instance, let $CR_e(h)$ be the function that implements the element-based heuristic over a particular refactoring history h (Figure 1).

So, $CR_e(H(s_1)) = \{cr_a[r_1, r_2, r_3, r_4, r_5], cr_b[r_3, r_6, r_7]\}$. Thus, this heuristic synthesizes two composites. The first one, cr_a, is a composite because $[r_1, r_2, r_3, r_4, r_5]$ affected the same element: *UserCtrl*. The second composite, cr_b, affects the *MediaCtrl* class.

Scope. In this heuristic, the composite scope is determined by the element used to synthesize the composites. In this way, $scope(c_a) = \{UserCtrl\}$, and $scope(c_b) = \{MediaCtrl\}$.

The element-based heuristic focuses on the element to find composites. Focusing on the element is a strength as it allows us to investigate what occurs with the element during its evolution. At the same time, focusing on the element is also a weakness. The scope of some refactoring types goes beyond a single element. Suppose a developer applies an *Extracted Method* in class A, and then a *Move Method* from class A to B. The heuristic will only synthesize a composite in class A. Since class B is out of scope, the effects of the composite in B will not be considered. As the effect in each element will be treated independently, this heuristic may not be entirely appropriate to investigate the effect of composites on smells.

3.3 Composite Synthesis Heuristics

We propose here two heuristics to synthesize composites.

3.3.1 Commit-Based Heuristic. The composite scope also indicates how the refactorings are interrelated (Section 3.1.4). Sometimes the refactorings are not structurally related to each other but they occur in the same context. For example, a developer may apply several refactorings to address a task associated with a commit. Hence, it makes sense to group these refactorings. For this purpose, this heuristic considers a single commit as the timespan (Section 3.1.3). In fact, there is a commit policy, widely accepted in the community, that recommends developers not to perform code changes for multiple tasks in the same commit [20]. Thus, each commit should have refactorings somehow related to the same task.

Formal Definition. The *commit-based* composite heuristic synthesizes as a composite all refactorings performed within a commit. The goal of this heuristic is to capture a temporal relation among the refactorings made at the time frame of a single commit. Formally, a composite $cr = [r_1, r_2, \cdots, r_n]$ is synthesized if and only if $|commit(r_1) \cup commit(r_2) \cdots \cup commit(r_n)| = 1$. For instance, consider $H(s_1) = [r_1, \cdots, r_7]$ (Figure 1). Now, let $CR_c(h)$ be the function that implements the commit-based heuristic over a refactoring history h. Thus, the commit-based heuristic produces two composites: $CR_c(H(s_1)) = \{cr_c[r_1, r_2], cr_d[r_4, r_5, r_6, r_7]\}$.

Scope. The composite scope includes the elements affected by the refactorings within the commit. Thus, $scope(cr_c) = \{UserCtrl, MediaCtrl\}$, and $scope(cr_d) = \{UserCtrl, MediaCtrl, AbstractCtrl\}$.

The commit-based heuristic is useful to observe the effect of all refactorings that occur in a commit. Assuming that all the changes within a commit are related to the same task [20], researchers can use this heuristic to understand how refactorings affect elements related to a task. This heuristic solves (partially) the limitation of the element-based heuristic. Instead of considering only the scope of a single element, it considers all elements affected by the refactorings made along the commit's task.

As this heuristic considers all elements, it does not discard refactorings that crosscut elements. However, there are cases that the commit-based heuristic discards refactorings to which it should not. A developer can start a composite in a commit and finish it in the subsequent commits. For example, a developer can start a composite, then, s/he can commit the changes and continue on refactoring the same elements. In this case, the commit-based heuristic would synthesize two composites rather than one.

3.3.2 Range-Based Heuristic. Some refactorings are structurally related to each other because they affect elements that are located in the same part of the source code. Thus, if we want to understand the effect of composites on the program structure, we need to analyze how these structurally related refactorings affect the elements. For example, if a refactoring crosscuts two elements, both elements should be analyzed to understand the effect of the refactoring. We propose the range-based heuristic to identify composites in which their refactorings affect the same location in the code.

Formal Definition. The *range-based* composite heuristic considers the notion of refactoring scope to synthesize composites. In this heuristic, the scope of all refactorings form the composite scope. A composite starts with an arbitrary refactoring r_a. A second refactoring r_b is part of the same composite if and only if r_a and $\exists e \in scope(r_b)$ such as $e \in scope(r_a)$. A possible third refactoring r_c will be added to the composite if $\exists e \in scope(r_c)$ such as $e \in scope(r_a)$ or $e \in scope(r_b)$. This process continues until all refactorings in a particular history are explored.

Scope. In this heuristic, the composite scope is determined by the union of the scopes of all refactorings. In this way, the scope is defined as $\cup_{i=1}^{n} scope(r_i)$. The r_1 and r_2 refactorings in Figure 1 moved elements from *UserCtrl* to *MediaCtrl* classes. Hence, $scope(r_1) = scope(r_2) = \{UserCtrl, MediaCtrl\}$. The composite synthesis in this example starts with r_1. As r_2 was applied in one element of $scope(r_1)$, then the composite grows bigger and turns into $[r_1, r_2]$. The r_3 refactoring affects elements of $scope(r_1)$, then the composite is now $[r_1, r_2, r_3]$. The same reasoning can be used for the remaining refactorings, so the composite synthesis produce the composite $c_e = [r_1, r_2, r_3, r_4, r_5, r_6, r_7]$.

4 STUDY PLANNING

4.1 Research Questions

In the previous section, we proposed heuristics to identify composites. These heuristics allow one to analyze composites from different, albeit complementary, perspectives. To propose them, we formally defined concepts that characterize a composite. Our goal is to use these concepts to understand (i) how composites manifest in software systems and (ii) their effect on smells. To achieve this goal, we aim to answer the following research question:

> **RQ_1.** What are the characteristics of composites in software systems?

We address $\mathbf{RQ_1}$ by applying the heuristics to identify three categories of composites: *element-based*, *commit-based*, and *range-based composites*. We rely on the concepts defined in our conceptual framework to compare these categories of composites. The analysis of these categories also allows us to have a better understanding of the effect of composites on the program structure. For this purpose, we investigate if composites affect the incidence of code smells. Thus, our following research question addresses this investigation:

465 466 467 468 469 470 471 472 473 474 475 476 477 478 479 480 481 482 483 484 485 486 487 488 489 490 491 492 493 494 495 496 497 498 499 500 501 502 503 504 505 506 507 508 509 510 511 512 513 514 515 516 517 518 519 520 521 522

$\mathbf{RQ_2}$. How does composite affect the incidence of smells?

We address $\mathbf{RQ_2}$ by investigating the influence of the composites on the incidence of code smells. Notice that such investigation is not trivial. First, we need to identify the elements affected by each category of composite, but taking into consideration their composite scope. Then, we analyze what happened with the smells before and after developers apply the composites. To support this analysis, we rely on the classification of each composite according to their effect on the incidence of smells. Thus, we classify a composite as a **positive** one if it reduces the number of code smells. Conversely, we classify it as **negative** composite if it increases the number of smells. Otherwise, we classify it as **neutral** composite, i.e., if it neither increases nor decreases the number of smells. This type of analysis has been applied in other empirical studies [7, 10–12]. Consequently, we can directly contrast our findings with theirs.

As a complement to $\mathrm{RQ_2}$, understanding and distinguishing the effect of specific types of composites on smells is an essential investigation. First, our investigation may help tool builders by uncovering the blind spots on the relation between refactoring and smells. Second, this investigation aims (i) to identify topics that require further investigation and (ii) to contrast the results with findings established in the literature. For example, Fowler [14] presented a catalog of composite types that can be used to remove code smells, which we named as a composite-smell pattern. A *composite-smell pattern* establishes a frequently observed relationship between a composite type and the introduction or removal of a smell type. For instance, suppose that there is a method affected by the Feature Envy code smell. In this case, Fowler recommends to apply a composite pattern composed of *Extract Method* followed by a *Move Method*. Unfortunately, we do not know if developers apply this composite pattern in practice. More specifically, we do not know which patterns govern the relation between refactorings and smells. These patterns are the focus of our next research question:

$\mathbf{RQ_3}$. What are the patterns governing composites and smells?

We address $\mathbf{RQ_3}$ by investigating creational and removal patterns. A **creational pattern** represents a recurring case where the composite tends to introduce a code smell. A **removal pattern** represents a recurring case where the composite tends to remove a smell. We detect these patterns by analyzing the impact of composites on smells located in the elements forming the composite scope. There is no empirical study in the literature that reports composites that typically remove or introduce smells. By answering $\mathbf{RQ_3}$, we are able to reveal composites used by developers not only to remove, but also to inadvertently introduce smells. The knowledge about creational patterns make developers informed about the risks of introducing certain smells along composite refactoring. The removal patterns can be useful to implement recommendation systems to support developers when removing smells.

4.2 Study Phases

This section presents the five phases of the study design.

Phase 1: Dataset Acquisition. In this phase, we choose a set S of software projects to analyze. We established GitHub as the source of projects. To select them, we followed criteria based on closely related studies [7, 11]. We selected projects with (1) different levels of popularity – based on the number of Github stars, (2) an active issue tracking system, and (3) at least 90% of code written in Java. These criteria allowed us to select 48 projects with a diversity of structure, domain, size and popularity. The replication package contains information about them [38], including name, domain, number of lines of code, commits, and Github stars.

Phase 2: Smell and Refactoring Detection. In this phase, we detected (i) the refactorings in all subsequent pairs of commits c_i and c_{i+1}, and (ii) all smells in each commit $c_i \in commit(s)$. We chose Refactoring Miner [47] to detect refactorings for two reasons. First, the tool has precision of 98% and recall of 87% as reported by Tsantalis *et al.* [47], which leads to a very low rate of false positives and false negatives. Second, the tool identifies the most common refactoring types applied by developers [32]. We considered all 14 refactoring types identified by the tool. Refactoring Miner gives us as output a list of refactorings $R(c_i, c_{i+1}) = \{r_1, \cdots, r_k\}$ as defined before, where k is the number of identified refactorings.

Code smells are often detected with metric-based strategies [4]. Each strategy is defined based on a set of metrics and thresholds. After collecting metrics for all projects, we applied the rules to detect smells [6, 22, 26]. These rules were used because: (i) they represent refinements of well-known rules proposed by Lanza *et al.* [22], which are used in related studies [7, 11, 28, 51]; and (ii) they have, on average, precision of 72% and recall of 81% [25]. We collected 19 smells: *Brain Class*, *Brain Method*, *Class Data Should Be Private*, *Complex Class*, *Data Class*, *Dispersed Coupling*, *Divergent Change*, *Feature Envy*, *God Class*, *Intensive Coupling*, *Large Class*, *Lazy Class*, *Long Method*, *Long Parameter List*, *Message Chain*, *Refused Bequest*, *Shotgun Surgery*, *Spaghetti Code*, *Speculative Generality*.

Phase 3: Manual Validation. We randomly sampled refactorings from each type to manually validated them. To ensure an acceptable confidence level in the results, we calculated the sample size of each refactoring type based on a confidence level of 95% and a confidence interval of 5 points. We recruited ten undergraduate students from another research group to also analyze the samples. The samples were divided into ten disjointed sets, and each student validated one. For each pair of elements, they had to mark it as a valid refactoring or not. Thus, we estimated the number of false positives generated by the Refactoring Miner [47]. We highlight that our goal was to ensure the trustability of the tool for our set of systems. For that matter, we relied on students, familiar with refactoring, to validate the tool. After the manual validation, we observed that the tool achieve high precision for all refactoring types, in which the median was 88.36%. The precision for all refactoring types is within one standard deviation (7.73). Applying the Grubb outlier test (alpha=0.05), we did not find any outlier. This result indicates that no refactoring type is strongly influencing the median precision. Thus, the precision for all the refactorings in the validated sample provides trustability to our results.

Some smells can be introduced by functional changes, such as the implementation of a new feature. Thus, we also validated if the smells were introduce or removed by the refactorings. First, we ran the eGit plugin and the Linux diff tool to find changes between commits. Then, we manually analyzed each change. When we identified a functional change, we classified it as non-pure refactoring [32]; otherwise, we classified it as pure refactoring. We validated

523 524 525 526 527 528 529 530 531 532 533 534 535 536 537 538 539 540 541 542 543 544 545 546 547 548 549 550 551 552 553 554 555 556 557 558 559 560 561 562 563 564 565 566 567 568 569 570 571 572 573 574 575 576 577 578 579 580

1,168 pure refactorings and 3,817 non-pure refactorings. We used the pure refactorings to confirm some results in Sections 5 and 6.

Phase 4: Synthesis and Classification of Composites. The heuristics to synthesize composites require a refactoring history as input (Section 3.3). We collected this history for each project in Phase 2. Each refactoring history was submitted to the algorithms that implement the heuristics, allowing us to collect: (i) element-based, (ii) range-based, and (iii) commit-based composites. After collecting them, they were classified according to their effect on smells. Thus, composites were classified as positive, negative, and neutral. Finally, we identified composite patterns related to the introduction and removal of specific types of smell. More details about the composite patterns are provided in Section 6. The algorithms (scripts) that implement the heuristics and classify the composites are available in the replication package [38].

Phase 5: Systematic Validation of Composite Patterns. To increase the reliability of our results, we conducted a systematic manual validation of a random sample of composites. First, we selected 130 composites associated with the introduction and removal of Feature Envy and God Class. We focused on these smells since they are the ones with the most complex composites (Section 6). Then, we randomly divided the composites among 4 researchers. For each composite, the researcher conducted the following steps.

1. Select the GitHub project where the composite happened;
2. Identify the commits where the composite occur;
3. Validate the refactorings and the smells in the elements;
4. Confirm if the composite is a creational or removal pattern;
   - (a) If yes: confirm if the composite explicitly introduced/removed the smell or if it is at least associated with the smell introduction/removal.
   - (b) If no: verify if the composite is an incomplete one, *i.e.*, if one or more refactorings in the removal pattern would have removed the smell.
5. Analyze the commit messages to find the developer's intention when performing the composite.

We validated 40 creational patterns, 43 removal patterns and 47 incomplete composites. We will use the validated composites to exemplify our discussions. In these cases, we will identify the composite by the "#" symbol followed by its id, *e.g.*, composite #21517). Our replication package contains all the validated instances and the detailed steps and information to validate them.

5 COMPOSITES: OCCURRENCE AND EFFECT

We identified 27,911 composites in our dataset. We present their characteristics (Section 5.1) and smell effects (Section 5.2).

5.1 Synthesized Composites

5.1.1 Quantity and Size. This section addresses our **RQ$_1$**. Table 1 shows, for each heuristic (1st*column*), the quantity (2nd*column*) and size of composites.

Providing a broader view on the composites. In Section 3.2, we discussed the element-based heuristic proposed by Bibiano *et al.* [7]. We mentioned that there were several elements affected by the refactorings that they were probably ignoring. Indeed, the number of refactored elements in the element-based composites is lower when compared to the other categories of composites (last column in the Table 1). When we compare the average size of element-based composites with the commit-based and range-based composites (7th*column*), we notice a huge difference in the number of refactorings in each category of composite. Comparing the number of elements with the average size, we notice that the commit-based and range-based composites are fragmented in the element-based composites. This result shows how the element-based heuristic only provides a partial view of the composites. The analysis of refactored elements leads to our first finding:

Table 1: Quantity and size of composites by heuristic

Heur.	№ Comp.	Ref. in Comp.	Size				Std. Dev.	Grubbs Test	№ Elem.
			Min	Med.	Max	Avg			
Element	12,636	28,394 (54%)	2	2	333	3.9	6.6	49.89538	4,579
Commit	11,545	47,218 (91%)	2	3	2,562	8.0	44.4	57.76980	51,472
Range	3,730	28,883 (55%)	2	2	2,556	7.7	62.2	41.09278	18,132

> **Finding 1**: Commit-based and range-based heuristics allow a broader assessment on the interrelation among refactored code elements.

Capturing complex composites. We also observed that our heuristics are helpful to find complex composites. A composite is considered complex when it is composed of a high number of refactorings, usually affecting multiple code elements. When we consider the average of refactorings in a composite (7th*column*), it becomes clear that the size of commit-based (8.0) and range-based (7.7) composites is near twice the size of element-based composites (3.9). This comparison shows that the number of interrelated refactorings (in commit-based or range-based composites) is much larger than any occurrence in the context of a single element. We also have found that 1,545 (41%) out of 3,761 composites of range-based heuristic, and 5,793 (50%) out of 11,659 composites of commit-based heuristic have 3 to 20 interrelated refactorings in conjunction. Therefore, studies that investigated only single refactorings or only refactorings affecting an element [6, 8, 11–13, 15, 16, 42, 52] are not able to identify complex composites. Thus, they are oversimplifying the study on refactoring. This result leads us to our next finding:

> **Finding 2**: There is a non-ignorable frequency of complex composites that most empirical studies missed.

Most refactorings are interrelated. After applying the heuristics, a given refactoring will be either classified as a single refactoring or interrelated with others in a composite. In this vein, the 3rd*column* of Table 1 presents the quantity of interrelated refactorings. As expected, the commit-based heuristic was the one that grouped the highest number of interrelated refactorings. The heuristic synthesized 11,545 composites, totaling 47,218 interrelated refactorings, which represents 91% of the total of refactorings in our dataset. This result indicates that refactoring composites are much more complex. Previous empirical studies [11, 32] reported that *Extract Method* and *Rename Method* are the commonest refactoring types applied by developers. These studies may give the simplistic impression that developers tend to most commonly apply single

refactorings with a very strict scope, *i.e.*, refactorings that affect one or two methods of a single class. However, this is not the case.

Even though *Extract* and *Rename Method* are the most common refactoring types, they are most often interrelated with other refactorings and they tend to be complex. For example, when we manually validated the 130 composite instances, we found that when these two refactoring types are applied, they are frequently part of a much more complex transformation that goes beyond the scope of a single method or class. For instance, when developers had the intention to improve the source code, all the refactorings were associated to the same task: code improvement (*e.g.*, composites #22691 and #22703[1]). This is even clearer for the commit-based composites. Since most of the refactorings occur within a commit (91%), the refactorings are associated with the task's commit.

> **Finding 3**: Refactoring composites are much more complex than what existing empirical studies suggest.

Semantic relation among refactorings. When we analyze the commit-based composites, only 9% of the refactorings do not belong to a composite. This result indicates that 91% of the refactorings are interrelated. Thus, either these refactorings are part of range-based composites (55%) or they occur in elements that are not structurally related to each other. This result indicates that when developers are working on a task, there are several refactorings that are not syntactically related to each other. As the refactorings in the commit-based composites are not syntactically related, we investigated if they had any relation. We found that several of these refactorings are semantically associated with the task that the developer is addressing in the commit. For example, several of the refactorings were applied to remove smells in different elements. These refactorings were not structurally related to each other, but they were semantically related to each since they aimed to remove smells (Section 5.2). Notice that if one analyzes only the range-based composite, s/he would not be able to identify the semantic relation between the refactorings. This result leads us to our next finding:

> **Finding 4**: Several commit-based composites contain refactorings that are semantically related to each other.

This finding may jeopardize most refactoring recommendation systems [17, 23, 30, 34, 36, 37]. These systems tend to consider only the structurally related refactorings to learn how to recommend refactorings. However, they do not explore the semantic relation among refactorings. Only considering structurally related refactorings may not suffice to provide recommendations for developers.

In our dataset, we also found extremely large composites, as presented in Table 1. However, we consider these composites as being outliers, since they are extremely rare. For the commit-based heuristic, for example, 87% of the composites are composed by 10 or less refactorings. On the other hand, only 0.004% of the commit-based composites have more than 100 refactorings. Thus, to confirm that large composites are outliers, we applied the Grubbs test for one outlier. Table 1 shows the Grubbs score in the penultimate column. The test is calculated as the highest size minus mean, divided by standard deviation. We observed p-values smaller than 0.00001 for all heuristics. This means that we can accept the hypothesis that the highest sizes of all heuristics are outliers. In our replication package [38], we have a manual analysis about these outliers.

[1] These composites are available in our replication package [38]

5.1.2 Heterogeneity and Timespan of Composites. Table 2 presents the results about the timespan and uniformity of composites.

Table 2: Timespan and uniformity characteristics

	Timespan		Uniformity	
Heur.	**Single-Commit**	**Cross-Commit**	**Homoge.**	**Heteroge.**
Element	9,094 (72.0%)	3,542 (28.0%)	11,107 (87.9%)	1,529 (12.1%)
Commit	11,545 (100.0%)	0 (0.0%)	6,484 (56.0%)	5,061 (44.0%)
Range	3,486 (93.5%)	244 (6.5%)	2,875 (77.0%)	855 (23.0%)

Most composites are single-commit. Different from our expectation, Table 2 shows that most composites are single-commit. This occurs even in the case of the range-based composites, where there is the possibility of having a larger composite scope. We were expecting that developers could start a composite in a commit and finish it in the following commits. However, out results show that developers tend to limit the composites to a single commit. This suggests that they intend to perform all refactorings at once, without splitting the task into multiple commits.

Most composites are homogeneous. Regarding uniformity, Table 2 shows that most composites are homogeneous, *i.e.*, they have the same refactoring type. We were not expecting this result. Fowler [14] in his book presents a catalog of multiple refactorings that can be applied to remove some smells. Hence, we assumed that developers would apply heterogeneous composites in practice. However, our assumption does not hold in practice. Regardless the heuristics, most composites are homogeneous. The highest incidence of heterogeneous composites are from the commit-based composites, which can be explained due to the semantic relation among refactorings. As discussed, any refactoring performed in a given commit can be semantically related to the same task, even if these refactorings are applied in structurally unrelated elements. The result about uniformity indicates that developers frequently apply the same refactoring type when restructuring related elements. After analyzing the uniformity and timespan characteristics, our results lead us to our next finding:

> **Finding 5**: Even though homogeneous and single-commit composites are more frequent than their counterparts, heterogeneous and cross-commits composites occur with a non-ignorable frequency, which should not be overlooked.

5.2 Effect of Composites on Code Smells

To answer $\mathbf{RQ_2}$, we classified the composites as positive, negative or neutral according to their effect on the incidence of smells. Table 3 shows the classification for each heuristic.

Table 3: Composite classification by heuristic

Heuristic	Positive	Neutral	Negative
Element-based	751 (6.0%)	11,264 (89.1%)	621 (4.9%)
Commit-based	1,653 (14.3%)	6,019 (52.1%)	3,873 (33.6%)
Range-based	542 (14.5%)	2,020 (54.2%)	1,168 (31.3%)

Several positive and negative composites. We can notice in Table 3 that the frequency of positive, negative and neutral composites differs between the element-based heuristic and the commit-based and range-based heuristics. First, Bibiano *et al.* found similar values for the element-based heuristic. However, if we analyze only from the perspective of element-based heuristic, we will conclude that the frequency of positive and negative composites is almost negligible. However, this conclusion is not correct. The other heuristics show that the positive and negative composites are almost as frequent as neutral composites. In fact, the frequency of positive, negative and neutral composites is higher than the results reported in the literature [6, 7, 11]. As discussed, the scope of some refactoring types goes beyond a single element. However, the element-based heuristic only consider the scope of a single element. Thus, this heuristic is not entirely appropriate to investigate refactorings that crosscut elements. This limitation compromises the study of Bibiano *et al.* [7]. In their study, the effect of several refactorings out of the composite scope is mistakenly ignored. Thus, they provide a partial view of composites, which, in the worst scenario, can be an erroneous view. This result leads to our next finding:

> **Finding 6**: Effects of composites often can only be observed through the reasoning of refactoring's relations in the scope of a range or a commit.

Negative composites are most likely than positive ones. We had an increase in the number of positive composites when we compare the element-based composites with the other categories. As discussed in *Finding 4* (Section 5.1.1), several refactorings are not syntactically related to each other but are semantically related. This scenario occurred, for instance, when developers had the task of removing Duplicate Code smell scattered over different parts of the system. We found several instances of the following commit-based composite cr_1 = {*Extract Superclass*, *Rename Method*} to remove this smell. The developer applied the *Extract Superclass* to create a superclass for the classes with the smell. Then, s/he renamed the method in the superclass to be consistent with the functionality provided. We found a case that a system had three different unrelated instances of Duplicate Code in the same commit. For each instance, the developer applied the composite cr_1. Despite the increase in positive composites, developers are most likely to introduce smells, as shown in Table 3. This result leads to the next finding:

> **Finding 7**: Even though most composites are neutral, a non-ignorable frequency of composites introduce smells.

Effect of the composite on the smell type. We relied on the classification of each composite to investigate its influence on the incidence of smells (Section 4.1). We found a case in which the developer applied a composite to a class that had two smells: Feature Envy and Message Chain. After the composite has been applied, we noticed that the developer removed the Message Chain, but s/he introduced a God Class. In this case, our classification scheme would classify the composite as neutral. However, a God Class would be often considered worse than a Message Chain. Hence, it would not be fair to label the composite as neutral. Considering the "criticality" of the smell, this composite is more likely to be considered negative because the structure is worse than before. To mitigate the risk of misclassifying neutral composites, we verified in our datset the smells presented before and after each neutral composite. We observed only 30 cases, in a set that contains 27,911 composites, in which a smell was replaced by other from a different type. This investigation leads to our next finding:

> **Finding 8**: The refactorings in neutral composites very often do not replace a smell type for another type.

6 COMPOSITE-SMELL PATTERNS

To address **RQ**$_3$, we analyzed removal and creational patterns emerging from the relationship between range-based composites and smells (Section 4.1). We focus on discussing here the patterns of range-based composites that affect Feature Envy and God Class. We discuss these smells because they are usually associated with the system structural degradation [1, 26, 50]. Patterns for the other nine smells are available in our replication package as well as patterns for the other categories of composites [38]. We manually inspected several instances of the patterns to understand what happened. In particular, we also confirmed whether the composites were directly related to the removal or introduction of the smell. After this analysis, we ended up identifying a total of 111 composite-smell patterns: 84 removal patterns and 27 creational patterns.

6.1 Feature Envy

Feature Envy is a code smell that represents a method much more interested in the data of a class other than the one it is actually declared [14]. This smell is the most frequent one in our dataset. Figure 2 presents all 13 composite types related to Feature Envy. Green boxes represent the removal patterns; they appear in the right side of Figure 2. The red ones, in the left side, represent the creational patterns. The content of each box represents the type of composite involved in the pattern. There is a caveat regarding the repetition structure: the $\{n\}$ symbol indicates the refactoring type was observed more than once in the composite structure.

The arrow weight indicates the frequency of a pattern with: (i) a removal behavior if the arrow is pointing to a green box, and (ii) creational behavior if the arrow is departing from a red box. For instance, the top-right green box indicates that in 77% of the times a composite with more than one *Inline Method* followed by more than one *Extract Method* removes one instance of Feature Envy. The same rationale is used to interpret the creational patterns.

We discussed in Section 5.1.1 that *Extract Method* is one of the most common refactorings and it is most often interrelated with other refactorings. Indeed, Figure 2 shows that all patterns have by, at least, one *Extract Method (EM)*. Neither the discussion about Extract Method in Section 5.1.1, nor the identification of composite patterns would be possible if (i) we had only analyzed single refactorings or (ii) used the element-based heuristic.

Incomplete composites. We noticed cases of composites consistently introducing Feature Envies in 31 projects. Composites with *Move Attribute, Extract Method* introduced Feature Envies in more than 60% of the cases as shown in Figure 2. These creational patterns indicate that the composites are "incomplete", which contributed to the introduction (rather than the removal) of the Feature Envy. An *incomplete composite* occurs when a set of refactorings

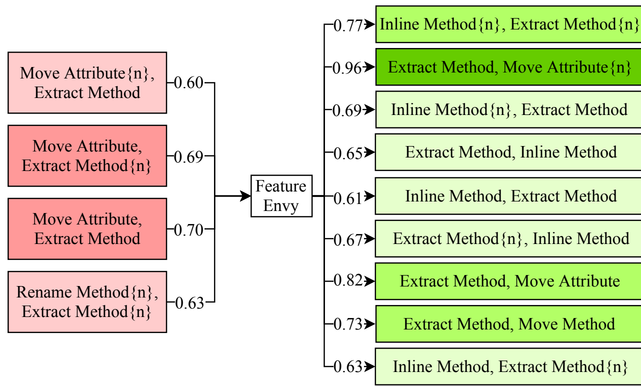

Figure 2: Feature Envy patterns

affect the smelly structure, but are not sufficient to fully remove a smell. In certain cases, it may even worsen the smelly structure. For instance, the developers moved attributes in the three first creational patterns in Figure 2; however, they did not move the corresponding extracted methods to fully remove the envy structures. Consequently, the "unmoved methods" became more interested in the classes to which the attributes were moved. Thus, these composites led to the introduction of the Feature Envy because they were incomplete; *i.e.*, a *Move Method* should also be part of such composites. Examples falling into this scenario include composites #22092, #22156 and #22419.

This type of scenario reinforces our discussion about the high number of negative composites (*Finding 7*). As we discussed in Section 5.2, our heuristics show that several composites are negative. This increase in the number of negative impacts is related to the incomplete composites. We found that developers are trying to improve the program structure during the refactoring process but, for different reasons, they are not necessarily completing the restructuring process to fully remove the smelly structure. As a consequence, incomplete composites lead to the introduction of smells, such as the Feature Envy. These incomplete composites were also observed on patterns for the other smell types.

> **Finding 9**: Developers tend to introduce smells, such as Feature Envies, due to incomplete composites.

Avoiding misleading results. As discussed, Bibiano et al. [7] do not provide a broader understanding of the effect of composites on smells, which can lead to misleading results. The same occurs with studies that only focus on single refactorings [6, 11]. For example, Bavota *et al.* [6] did not find any relation between specific smells (*e.g.*, Feature Envy) and specific refactorings (*e.g.*, EM). To illustrate how these studies are not able to either provide a broader view or find relation between refactorings and smells, let us consider the EM refactoring since it occurs in all the patterns associated with the Feature Envy (Figure 2).We applied the Fisher's Exact Test to investigate the relation between *EM* and Feature Envy (Table 4). For each heuristic (1st *column*), we present the number of composites containing *EM* that removed and introduced Feature Envies, 2nd and 3rd *columns* respectively. The 4th and 5th columns show the same information for composites without *EM*. The last two columns show the p-value and odds ratio (OR) for the Fisher's Exact Test.

Table 4: Fisher's test results for Feature Envy patterns

Heuristic	Positive With EM	Negative With EM	Positive Without EM	Negative Without EM	p-value	OR
Element	496	86	0	0	1	0
Commit	15,632	2,013	31,398	39,000	<0.000001	9.64
Range	360	110	25	0	0.002338	0

We ran the test with 95% of confidence, which means that we can reject the null hypothesis (H0) when the p-value is smaller than 0.05. In our case, the H0 is that the introduction or removal of Feature Envies by composites is independent of the presence of *EM*. Given the p-values, only in the case of the element-based heuristic that we cannot reject H0. Therefore, the element-based composites mislead us to believe that composites without *EM* will never remove or introduce Feature Envies. However, the results of the other heuristisc show the opposite, especially in the case of commit-based composites. Thus, our heuristics were able to reveal that *EM* often "partially" contributes to the removal (and introduction) of Feature Envy, when performed with other refactorings (composites). In summary, only analyzing element-based composites [7] or single composites [6, 11] does not provide a broader understanding of composite, or, in the worst-case scenario, it can lead to an erroneous result. This discussion reinforces *Finding 1* (Section 5.1.1).

6.2 God Class

Our second set of composite-smell patterns concerns the God Class. This smell exists when a class accumulates several responsibilities [14]. We found out that this smell is more frequent than one might expect. We found 425 distinct instances of God Class distributed into 26 projects. Figure 3 presents all the 12 patterns.

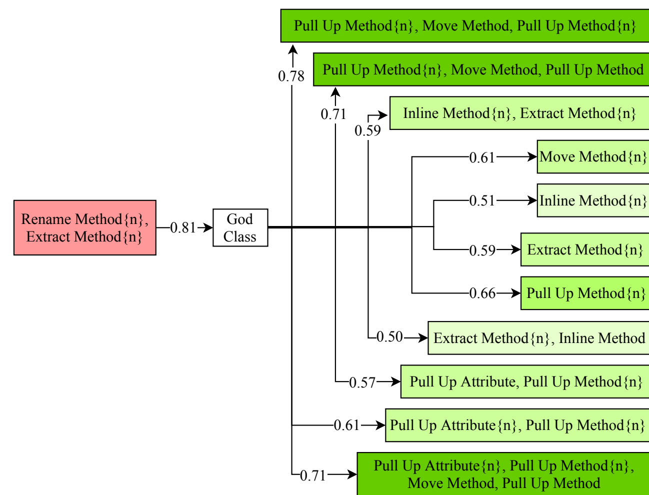

Figure 3: God Class patterns

Palomba et al. showed that when developers implement new features, they often apply complex refactorings to improve the code cohesion [39]. Our results provide a new perspective regarding

this scenario. We found that developers tend to decrease the code cohesion when interleaving refactorings with additional changes. For example, when developers apply composites of *Rename Methods* and *Extract Methods*, they tend to introduce God Class, as show in Figure 3. At first sight, this pattern is not intuitive. Developers are not expected to increase the size of classes while performing *Rename* and *Extract Methods*. We analyzed these composites to understand why they led to the God Class.

Inappropriate additional changes. We found that this creational pattern exists when developers interleave refactoring with additional changes and if they are not performed in conjunction with other refactorings (*e.g.*, composites #21517 and #20932). The additional changes comprise the creation of new methods (*Extract Methods*), which are, unfortunately, implementing unrelated functionalities. As a consequence of these additions in the extracted methods, developers have to change the methods' name to express the new functionalities (*Rename Methods*). As new functionalities are introduced, the class cohesion decreases, which leads to the appearance of a God Class. The composites with *Rename Methods* and *Extract Methods* were not the main reason for the introduction of the God Class. Still, a recommender system can use this pattern to improve their refactoring recommendation. For example, if a developer is introducing non-structural changes along with *Rename Methods* and *Extract Methods*, the system can alert the developer that s/he may introduce a structural problem.

Moving data to remove the God Class. We identified 11 removal patterns associated with the God Class. This result shows that developers often apply a wide range of non-trivial composites to remove the smell across software project. For example, as discussed in the previous paragraphs, the God Class was introduced when the composites of *Rename Methods* and *Extract Methods* occurred with additional changes. We found that these changes introduced pieces of code that should not be in the classes, contributing to the God Class. Later on, developers had to apply several refactorings to move these pieces of code to the classes that suit them better, removing the God Class. This behavior of applying refactorings that move data is reflected in the removal patterns. All the removal patterns had refactorings that moved data between classes, except for *Inline Method* and *Extract Method*. This scenario is another example of why an element-based heuristic fails to show the effect of composites on smells. To remove God Class, developers apply refactorings that affect multiple elements, such as the classes to which the data is moved. However, if we analyze only the scope of a single element, we would not be able to notice that composites moving data play a central role in the addition and removal of God Classes. This behavior leads us to our next finding:

> **Finding 10**: The range-based heuristic detects how data is moved among classes to either introduce or remove God Class.

Providing knowledge based on practice. Although some patterns emerge in the element-based heuristic, they only provide a partial view of composite effects. Several of the composite patterns reported here and in the replication package can only be identified with range-based and commit-based heuristics. Even Fowler's catalog [14], which lists common composites to remove smells, does not report our patterns. For example, Fowler's catalog indicates that developers should apply *Extract Class* or *Extract Subclass* to remove a God Class. However, we noticed that developers much more often follow other strategies regarding the refactoring types: *Inline Method*, *Extract Method*, *Pull Up Method* and *Attribute*, and *Move Method*. Thus, our results suggest that existing refactoring catalogs [14] may not reflect the practice. We also observed that existing recommenders for code smell removal do not recommend these patterns [30, 36, 46]. They should refine their recommendations with our smell-removal composite patterns.

7 THREATS TO VALIDITY

To apply the heuristics, we had to identify the refactorings that occurred in each system. For this identification, we relied on the Refactoring Miner [47]. Thus, there is a threat associated with the false positives generated by the tool. To minimize this threat, we conducted a manual validation for each refactoring type (Section 4.2). We observed a high precision for each refactoring type.

Some findings are centered around the difference among positive, negative and neutral composites. However, if our classification procedure is somewhat inaccurate in identifying them, then we have a major threat to the validity of our data. In order to mitigate that, we studied all the cases where the classification procedure could be inaccurate (Section 5.2). We found a risk of the classification scheme being wrong on 0.01% of the cases. In this way, this risk was mitigated by the data disposition.

We also presented several patterns where range-based composites removed or introduced smells. We computed them by verifying how often they happen in the analyzed projects, so they might suffer from lack of generality. To avoid this threat, we only reported those patterns that happened in more than 50% of the instances in our dataset. Additionally, to make sure these patterns happened in all three types of composites, we verified the intersection of the element-based, commit-based and range-based heuristics. We found that 16 (out of 27) creational pattern and 80 (out of 84) removal patterns were found by all heuristics.

8 CONCLUSION

Composite refactoring is common in practice, but a wide empirical knowledge about it is scarce. To tackle this issue, we conducted a study with two purposes. First, we provided a conceptual characterization of composites and defined two heuristics to identify composites in different categories. Second, our study aimed to understand how composites manifest in practice, and how they affect the program structure. Our results show that to study composite refactoring we need indeed to rely on different heuristics: they are complementary to each other, but most empirical studies tend to use only a single heuristic (Section 2). For example, the identification of the semantically-related refactorings was only possible using the commit-based and range-based heuristics together. Similarly, the identification of several composite-smell patterns were only possible with the range-based heuristic. Thus, studies that investigate only a single composite perspective fall short in providing a full understanding of temporal and spatial refactoring effects.

Our results can be useful both for researchers and practitioners. In particular, our study helped to explain conflicting results in the literature. For instance, different studies (*e.g.*, [6] and [7]) have come

to different conclusions regarding the relation of refactoring types with specific code smells. Thus, we provided new evidence that there are composite patterns strongly related to the introduction or removal of specific code smells (which explain the divergence in their results). On the practical side, we contributed with insights and a set of composite-smell patterns that are useful for improving existing refactoring detection tools or recommender systems.

REFERENCES

[1] M Abbes, F Khomh, Y Gueheneuc, and G Antoniol. 2011. An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, on Program Comprehension. In *Proceedings of the 15th European Software Engineering Conference; Oldenburg, Germany*. 181–190.

[2] Vahid Alizadeh and Marouane Kessentini. 2018. Reducing Interactive Refactoring Effort via Clustering-based Multi-objective Search. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)*. ACM, New York, NY, USA, 464–474. https://doi.org/10.1145/3238147.3238217

[3] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, Ali Ouni, and Marouane Kessentini. 2019. Do Design Metrics Capture Developers Perception of Quality? An Empirical Study on Self-Affirmed Refactoring Activities. In *13th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM 2019)*.

[4] Roberta Arcoverde, Isela Macia, Alessandro Garcia, and Arndt von Staa. 2012. Automatically Detecting Architecturally-Relevant Code Anomalies. *Proceedings of the International Workshop on Recommendation Systems for Software Engineering* (2012), 90–91. https://doi.org/10.1109/RSSE.2012.6233419

[5] Gabriele Bavota, Bernardino De Carluccio, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Orazio Strollo. 2012. When Does a Refactoring Induce Bugs? An Empirical Study. *Proceedings of the IEEE 12th International Working Conference on Source Code Analysis and Manipulation* (2012), 104–113. https://doi.org/10.1109/SCAM.2012.20

[6] Gabriele Bavota, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Fabio Palomba. 2015. An Experimental Investigation On The Innate Relationship Between Quality And Refactoring. *Journal of Systems and Software* 107 (2015), 1–14. https://doi.org/10.1016/j.jss.2015.05.024

[7] Ana Carla Bibiano, Eduardo Fernandes, Daniel Oliveira, Alessandro Garcia, Marcos Kalinowski, Baldoino Fonseca, Roberto Oliveira, Anderson Oliveira, and Diego Cedrim. 2019. A Quantitative Study on Characteristics and Effect of Batch Refactoring on Code Smells. In *13th International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 1–11.

[8] Arnaud Blouin, Valéria Lelli, Benoit Baudry, and Fabien Coulon. 2018. User interface design smell: Automatic detection and refactoring of Blob listeners. *Information and Software Technology* 102 (2018), 49 – 64. https://doi.org/10.1016/j.infsof.2018.05.005

[9] Aline Brito, Andre Hora, and Marco Tulio Valente. 2020. Refactoring Graphs: Assessing Refactoring over Time. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE.

[10] Diego Cedrim, Leonardo da Silva Sousa, Alessandro F. Garcia, and Rohit Gheyi. 2016. Does Refactoring Improve Software Structural Quality? A Longitudinal Study of 25 Projects. In *Proceedings of the 30th Brazilian Symposium on Software Engineering*. ACM, New York, NY, USA, 73–82. https://doi.org/10.1145/2973839.2973848

[11] Diego Cedrim, Alessandro Garcia, Melina Mongiovi, Rohit Gheyi, Leonardo Sousa, Rafael de Mello, Baldoino Fonseca, Márcio Ribeiro, and Alexander Chávez. 2017. Understanding the Impact of Refactoring on Smells: A Longitudinal Study of 23 Software Projects. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 465–475. https://doi.org/10.1145/3106237.3106259

[12] Alexander Chávez, Isabella Ferreira, Eduardo Fernandes, Diego Cedrim, and Alessandro Garcia. 2017. How Does Refactoring Affect Internal Quality Attributes? A Multi-Project Study. In *Proceedings of the 31st Brazilian Symposium on Software Engineering (SBES'17)*. ACM, New York, NY, USA, 74–83. https://doi.org/10.1145/3131151.3131171

[13] Danny Dig, Kashif Manzoor, Ralph Johnson, and Tien N. Nguyen. 2007. Refactoring-Aware Configuration Management for Object-Oriented Programs. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*. IEEE Computer Society, Washington, DC, USA, 427–436. https://doi.org/10.1109/ICSE.2007.71

[14] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. 1999. *Refactoring: Improving The Design Of Existing Code* (1st ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. 464 pages.

[15] Kenji Fujiwara, Kyohei Fushida, Norihiro Yoshida, and Hajimu Iida. 2013. *Assessing Refactoring Instances and the Maintainability Benefits of Them from Version Archives*. Springer Berlin Heidelberg, Berlin, Heidelberg, 313–323. https://doi.org/10.1007/978-3-642-39259-7_25

[16] Birgit Geppert, Audris Mockus, and Frank Rossler. 2005. Refactoring for Changeability: A Way to Go?. In *Proceedings of the 11th IEEE International Software Metrics Symposium (METRICS '05)*. IEEE Computer Society, Washington, DC, USA, 13–. https://doi.org/10.1109/METRICS.2005.40

[17] Mark Harman and Laurence Tratt. 2007. Pareto optimal search based refactoring at the design level. In *9th Genetic and Evolutionary Computation Conference (GECCO)*. 1106–1113.

[18] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2012. A Field Study of Refactoring Challenges and Benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*. ACM, New York, NY, USA, Article 50, 11 pages. https://doi.org/10.1145/2393596.2393655

[19] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2014. An Empirical Study of Refactoring Challenges and Benefits at Microsoft. *IEEE Transactions on Software Engineering* 40, 7 (2014), 633–649. https://doi.org/10.1109/TSE.2014.2318734

[20] H. Kirinuki, Y. Higo, K. Hotta, and S. Kusumoto. 2016. Splitting Commits via Past Code Changes. In *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*. 129–136. https://doi.org/10.1109/APSEC.2016.028

[21] Martin Kuhlemann, Liang Liang, and Gunter Saake. 2010. Algebraic and cost-based optimization of refactoring sequences. In *2nd International Workshop on Model-driven Product Line Engineering (MDPLE)*. 37–48.

[22] Michele Lanza and Radu Marinescu. 2010. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems* (1st ed.). Springer Publishing Company, Incorporated.

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

[24] Kui Liu, Dongsun Kim, Tegawendé F. Bissyandé, Taeyoung Kim, Kisub Kim, Anil Koyuncu, Suntae Kim, and Yves Le Traon. 2019. Learning to Spot and Refactor Inconsistent Method Names. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. IEEE Press, Piscataway, NJ, USA, 1–12. https://doi.org/10.1109/ICSE.2019.00019

[25] Isela Macia. 2013. *On The Detection Of Architecturally Relevant Code Anomalies In Software Systems*. Ph.D. Dissertation. Pontifical Catholic University of Rio de Janeiro.

[26] Isela Macia, Roberta Arcoverde, Alessandro Garcia, Christina Chavez, and Arndt von Staa. 2012. On the Relevance of Code Anomalies for Identifying Architecture Degradation Symptoms. *Proceedings of the 16th European Conference on Software Maintenance and Reengineering* (2012), 277–286. https://doi.org/10.1109/CSMR.2012.35

[27] Mehran Mahmoudi, Sarah Nadi, and Nikolaos Tsantalis. 2019. Are Refactorings to Blame? An Empirical Study of Refactorings in Merge Conflicts. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 151–162.

[28] Leandra Mara, Gustavo Honorato, Francisco Dantas Medeiros, Alessandro Garcia, and Carlos Lucena. 2011. Hist-Inspect: A Tool for History-Sensitive Detection of Code Smells. In *Proceedings of the 10th International Conference on Aspect-oriented Software Development Companion (AOSD '11)*. ACM, New York, NY, USA, 65–66. https://doi.org/10.1145/1960314.1960335

[29] Panita Meananeatra. 2012. Identifying Refactoring Sequences For Improving Software Maintainability. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM Press, New York, New York, USA, 406–409. https://doi.org/10.1145/2351676.2351760

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

[31] E. Murphy-Hill and A. P. Black. 2008. Refactoring Tools: Fitness for Purpose. *IEEE Software* 25, 5 (Sep. 2008), 38–44. https://doi.org/10.1109/MS.2008.123

[32] E. Murphy-Hill, C. Parnin, and A. P. Black. 2012. How We Refactor, and How We Know It. *IEEE Transactions on Software Engineering* 38, 1 (2012), 5–18. https://doi.org/10.1109/TSE.2011.41

[33] Mel Ó Cinnéide and Paddy Nixon. 2000. 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)*. 1–6.

[34] Mark O'Keeffe and Mel Ó Cinnéide. 2008. Search-based Refactoring: An Empirical Study. *J. Softw. Maint. Evol.* 20, 5 (Sept. 2008), 345–364. https://doi.org/10.1002/smr.v20:5

[35] William F. Opdyke. 1992. *Refactoring Object-oriented Frameworks*. Ph.D. Dissertation. Champaign, IL, USA. UMI Order No. GAX93-05645.

[36] Ali Ouni, Marouane Kessentini, Mel Ó Cinnéide, Houari Sahraoui, Kalyanmoy Deb, and Katsuro Inoue. 2017. MORE: A multi-objective refactoring recommendation approach to introducing design patterns and fixing code smells. *Journal*

of Software: Evolution and Process 29, 5 (2017), e1843.

[37] Ali Ouni, Marouane Kessentini, and Houari Sahraoui. 2013. Search-based refactoring using recorded code changes. In *17th European Conference on Software Maintenance and Reengineering (CSMR)*. 221–230.

[38] 2020 Replication Package. 2020. https://figshare.com/s/81f7973d07ceb7e4796c.

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

[40] E. Piveta, J. Araujo, M. Pimenta, A. Moreira, P. Guerreiro, and R. T. Price. 2008. Searching for Opportunities of Refactoring Sequences: Reducing the Search Space. In *2008 32nd Annual IEEE International Computer Software and Applications Conference*. 319–326. https://doi.org/10.1109/COMPSAC.2008.119

[41] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim. 2010. Template-Based Reconstruction of Complex Refactorings. In *Proceedings of IEEE International Conference on Software Maintenance*. 1–10. https://doi.org/10.1109/ICSM.2010.5609577

[42] Jacek Ratzinger, Thomas Sigmund, and Harald C Gall. 2008. On The Relation of Refactorings and Software Defect Prediction. In *Proceedings of the International Workshop on Mining Software Repositories*. ACM Press, New York, New York, USA, 35–38. https://doi.org/10.1145/1370750.1370759

[43] Veselin Raychev, Max Schäfer, Manu Sridharan, and Martin Vechev. 2013. Refactoring with synthesis. *ACM SIGPLAN Notices* 48, 10 (2013), 339–354.

[44] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. 2016. Why We Refactor? Confessions of GitHub Contributors. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 858–870. https://doi.org/10.1145/2950290.2950305

[45] Gábor Szőke, Gábor Antal, Csaba Nagy, Rudolf Ferenc, and Tibor Gyimóthy. 2017. Empirical study on refactoring large-scale industrial systems and its effects on maintainability. *Journal of Systems and Software* 129 (2017), 107–126.

[46] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. 2018. Ten years of JDeodorant: Lessons learned from the hunt for smells. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 4–14.

[47] Nikolaos Tsantalis, Matin Mansouri, Laleh M. Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and Efficient Refactoring Detection in Commit History. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA, 483–494. https://doi.org/10.1145/3180155.3180206

[48] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. 2015. When and Why Your Code Starts to Smell Bad. In *Proceedings of the 37th International Conference on Software Engineering (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 403–414.

[49] Carmine Vassallo, Giovanni Grano, Fabio Palomba, Harald C. Gall, and Alberto Bacchelli. 2019. A large-scale empirical exploration on refactoring activities in open source software projects. *Science of Computer Programming* 180 (2019), 1 – 15. https://doi.org/10.1016/j.scico.2019.05.002

[50] Aiko Yamashita and Leon Moonen. 2013. Exploring the Impact of Inter-Smell Relations on Software Maintainability: An Empirical Study. *Proceedings of the International Conference on Software Engineering* (2013), 682–691. https://doi.org/10.1109/ICSE.2013.6606614

[51] Aiko Yamashita and Leon Moonen. 2013. To What Extent can Maintenance Problems be Predicted by Code Smell Detection? An Empirical Study. *Information and Software Technology* 55, 12 (2013), 2223–2242. https://doi.org/10.1016/j.infsof.2013.08.002

[52] Young Seok Yoon and Brad A. Myers. 2015. Supporting Selective Undo in a Code Editor. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 223–233. http://dl.acm.org/citation.cfm?id=2818754.2818784

[53] Trevor J. Young. 2005. *Using AspectJ to build a software product line for mobile devices*. Ph.D. Dissertation. https://doi.org/10.14288/1.0051632