UNIVERSIDADE FEDERAL DE MINAS GERAIS
Instituto de Ciências Exatas
Programa de Pós-Graduação em Ciência da Computação

Aline Norberta de Brito

Refactoring Graphs: Reasoning about Refactoring over Time

Belo Horizonte
2023

Aline Norberta de Brito

Refactoring Graphs: Reasoning about Refactoring over Time

Final Version

Thesis presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Doctor in Computer Science.

Advisor: Marco Tulio de Oliveira Valente
Co-Advisor: Andre Cavalcante Hora

Belo Horizonte
2023

Brito, Aline Norberta de.

B862r Refactoring graphs [recurso eletrônico]: reasoning about refactoring over time / Aline Norberta de Brito – 2023.
1 recurso online (131 f. il, color.) : pdf.

Orientador: Marco Tulio de Oliveira Valente.
Coorientador: André Cavalcante Hora.
Tese (Doutorado) - Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação.
Referências: f. 122 -131.

1. Computação – Teses. 2. Engenharia de software – Teses 3. Grafos de refatoração – Teses. 4. Mineração de repositórios de software – Teses. I. Valente, Marco Tulio de Oliveira. II. Hora, André Cavalcante. III. Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação. IV. Título.

CDU 519.6*32(043)

Ficha catalográfica elaborada pela bibliotecária Belkiz Inez Rezende Costa
CRB 6/1510 Universidade Federal de Minas Gerais - ICEx

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Refactoring Graphs: Reasoning about Refactoring over Time

ALINE NORBERTA DE BRITO

Tese defendida e aprovada pela banca examinadora constituída pelos Senhores:

[illegible]

Prof. Marco Túlio de Oliveira Valente - Orientador
Departamento de Ciência da Computação - UFMG

André Cavalcante Hora

Prof. André Cavalcante Hora - Coorientador
Departamento de Ciência da Computação - UFMG

[illegible]

Profa. Elisa Yumi Nakagawa
Instituto de Ciências Matemáticas e de Computação - USP

Fernando J. Castor de Lima Filho

Prof. Fernando José Castor de Lima Filho
Centro de Informática - UFPE

Eduardo Figueiredo

Prof. Eduardo Magno Lages Figueiredo
Departamento de Ciência da Computação - UFMG

[illegible]

Profa. Tayana Uchôa Conte
Instituto de Computação - UFAM

Belo Horizonte, 10 de março de 2023.

Acknowledgments

Este manuscrito é o resultado de uma jornada de muito aprendizado, esforço e dedicação. Gostaria de agradecer a todos que fizeram parte desta etapa tão importante na minha vida.

Inicialmente, gostaria de agradecer a Deus, por permitir a conclusão desta etapa, e aos meus familiares, pelo incentivo durante este curso de doutorado. Especialmente, gostaria de agradecer aos meus pais, Bernadete e José Norberto, por todo apoio recebido, e ao meu irmão, Rodrigo, pela amizade, conselhos, e presença durante este período.

Gostaria de expressar também minha sincera gratidão ao meu orientador, Prof. Marco Tulio Valente, que se tornou um grande mentor na minha vida e carreira. Agradeço também ao meu coorientador, Prof. André Hora, cujo suporte e apoio foi fundamental para conclusão desta etapa.

Aos meus amigos, agradeço pela amizade, apoio e por todos os momentos felizes proporcionados. Especialmente, agradeço algumas amigas muito especiais que estiveram presentes no decorrer deste período.

Agradeço também aos meus colegas do ASERG e do DCC/UFMG, que fizeram esta jornada ser mais leve e produtiva. Obrigada pelas conversas, suporte e amizade durante este período. Agradeço também aos meus colegas de trabalho, professores e alunos, pela contribuição para o meu crescimento pessoal e profissional.

Aos membros da banca examinadora, agradeço pela disponibilidade em participar deste trabalho.

Por fim, gostaria de agradecer também ao DCC/UFMG, CNPq, CAPES, e FAPEMIG, pelo suporte financeiro, logístico e profissional.

“*Education is the most powerful weapon which you can use to change the world.*”
(Nelson Mandela)

Resumo

Frequentemente, desenvolvedores refatoram o código, criando novas entidades ou alterando estruturas existentes. Algumas vezes, estas operações são realizadas em um curto período. Em outros casos, as operações geram sequências de modificações realizadas ao longo do tempo, um cenário que usualmente não é considerado na literatura. Neste contexto, o principal objetivo deste trabalho é caracterizar e compreender refatorações realizadas ao longo do tempo. Para tanto, propõe-se uma nova abstração denominada grafos de refatoração (*refactoring graphs*). Neste modelo baseado em grafos, os vértices representam métodos ou funções e as arestas referem-se às operações de refatoração. Esta pesquisa está organizada em três grandes trabalhos. Inicialmente, define-se a abstração proposta, descrevendo os principais elementos e provendo um conjunto de *scripts* que permite a detecção de grafos de refatoração em sistemas de *software* reais. Em seguida, na segunda unidade de trabalho, caracteriza-se aproximadamente 1,5 mil grafos de refatoração provenientes de projetos Java e JavaScript populares hospedados no GitHub. Os resultados confirmam que as refatorações não são apenas operações únicas, mas também sequências de transformações realizadas ao longo do tempo. Além disso, um estudo qualitativo é realizado, no qual contactou-se os desenvolvedores responsáveis por subgrafos que descrevem grandes operações, visando identificar as suas principais motivações. Por fim, na última unidade de trabalho, explora-se aplicações para a abstração proposta na tese. Inicialmente, avalia-se operações realizadas por alunos de uma disciplina de Engenharia de Software, contando com os grafos de refatoração para compreender e inspecionar as operações. Logo, assume-se a perspectiva de professor que almeja entender atividades práticas neste contexto. Além disso, propõe-se um catálogo de refatorações comumente realizadas ao longo do tempo, denominadas refatorações compostas. Os grafos de refatoração são utilizados para documentar e ilustrar instâncias do catálogo detectadas em um oráculo com centenas de refatorações e no histórico de projetos populares hospedados no GitHub.

Palavras-chave: Refatoração, Grafos de Refatoração, Mineração de Repositórios de Software, Evolução de Software.

Abstract

Frequently, practitioners refactor their code, producing new entities or changing the structure of existing ones. Sometimes, these transformations are performed in a constrained time frame. In other cases, they generate sequences of modifications performed over a long time period, a scenario not usually considered in the literature. In this context, the main goal of this Ph.D. thesis is to characterize and understand refactoring operations performed over time. For this purpose, we introduce a novel abstraction for reasoning about refactorings, named refactoring graphs. In this graph-based abstraction, nodes represent methods or functions, and edges refer to refactoring operations. We organize the research into three major working units. We start by defining the proposed abstraction, describing the elements and providing a set of scripts to detect refactoring graphs in real-world projects. Then, in the second working unit, we characterize about 1.5K refactoring subgraphs from popular Java and JavaScript projects hosted on GitHub. The results confirmed our hypothesis that refactorings are not only sole operations. There are also sequences of transformations performed over time. Besides, we also perform a qualitative study, in which we contact developers responsible by subgraphs describing large operations, aiming to identify their main reasons to perform such operations. Finally, in the last working unit, we explore applications of the proposed graph-based model. First, we evaluate refactoring tasks performed by undergraduate students from a Software Engineering course, relying on refactoring graphs to understand and inspect the operations. In other words, we assume a professor's perspective who needs to understand practical exercises on refactoring. We also propose a catalog of common refactorings performed over time, which we decide to call composite refactorings. We rely on refactoring graphs to document and illustrate instances of composites detected in an oracle with hundreds of operations and in the history of popular projects hosted on GitHub.

Keywords: Refactoring, Refactoring Graphs, Mining Software Repositories, Software Evolution.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

This chapter introduces this Ph.D. thesis. We start by stating our problem and motivation in Section 1.1. Section 1.2 details our objectives, goals, and intended contributions, while Section 1.3 shows our current publications. Finally, we present the outline of this thesis in Section 1.4.

1.1 Problem and Motivation

Refactoring is a key activity to preserve and evolve the internal design of software systems. Due to the importance of the practice in modern software development, there is a large body of papers and studies about refactoring, focusing on aspects such as the usage of refactoring engines [70, 71], documentation of refactorings using commit messages [70], motivations for performing refactorings [65, 86, 94], benefits and challenges of refactoring [56, 57], among others.

Despite the extensive literature on this subject, *time* seems to be an under-investigated dimension in refactoring studies, which usually consider refactoring as individual and atomic operations. The notable exception involves studies on refactoring tactics, particularly on repeated refactoring operations, often called *batch* refactorings. For example, Murphy-Hill et. al. [70] define batch refactorings as operations that execute within 60 seconds of each other. They report that 40% of refactorings performed using a refactoring tool occurs in batches, i.e., programmers repeat refactorings. But the authors also mention that "*the main limitation of [our] analysis is that, while we wished to measure how often several related refactorings are performed in sequence, we instead used a 60-second heuristic*". As a second example, Bibiano et. al. [12] investigate the characteristics and impact of batch refactorings on program elements affected by smells. The authors rely on a heuristic to retrieve batches [26], which they define as sequences of refactorings performed by the same author in a single code element. Thus, even this extended heuristic focuses on single methods or classes, most of the cases resulting in batches with a single commit (93%).

Interestingly, in his seminal book on refactoring, Fowler [35] dedicates a chapter—co-authored with Kent Beck—to *big refactorings*. They claim that when studied individually refactorings do not provide a whole picture of the "game" played by developers when improving software design. Literally, the author states that"*refactorings take time [to be concluded]*". Fowler reinforces this observation in the second edition of his book by discussing *long-term* refactorings [36], which may require an effort of weeks to complete.

However, understanding and reasoning about refactorings performed over time is a nontrivial task, for the main reasons:

- Modern detection tools report refactoring at a fine granularity level, restricting the operation to a short time and scope. For example, RefactoringMiner [95, 96] and RefDiff [85, 87] detect refactorings as independent changes, which are restricted and confined to a single commit. As a consequence, it is not trivial to retrieve sequences of refactoring operations performed over days or weeks.

- Developers widely rely on source code history to understand the evolution of code or elements [59]. However, refactoring operations may create ruptures in such histories, interfering to track the changes [38, 43, 49, 83]. For example, when we move a method `m()` from class `Foo` to class `Bar`, GitHub diff shows the addition of a new method and removal of the previous one. However, the operation does not introduce a new program element: it is only a "fork" in the method history due to the refactoring.

- Catalogs of refactorings often focus on single transformations [35, 36]. We still need reference guides and standardized vocabulary, which can be used to guide developers when performing and documenting sequences of operations over time. We can observe this problem in a commit from `Robovm`, a compiler for Java bytecode.[1] In this commit, for instance, the developer performed 29 extractions to create the method `has(CFString)` [93, 96]. The developer may not understand the whole scenario by inspecting individual parts. Therefore, ideally, these operations could be clustered under a single composite refactoring, for example, named Method Composition. In other words, in many programming contexts, it is better to reason and look at these refactorings as a single transformation than as disconnected ones. An analogy can be made with the chemistry concepts of atoms, which are basic units of matter that compose molecules.

In summary, focusing on independent refactoring operations does not cover the whole scenario, since refactoring performed over time is a relevant aspect that affects several algorithms, tools, approaches, and software tasks as well [30, 43, 72, 90].

[1]`https://github.com/robovm/robovm/commit/bf5ee44b`

> To our knowledge, refactorings performed over long time windows are not deeply studied by the literature.

1.2 Goals and Contributions

As previously mentioned, we still need to better understand and document refactoring operations performed over time. Also, it is necessary to provide abstractions to support a broad comprehension of refactoring. Hence, our main goal in this thesis is described as follows:

> We provide an in-depth study on refactoring operations performed over time. For this purpose, we propose, define, and explore an abstraction to support practitioners and researchers when understanding, documenting, and visualizing refactoring operations that are not restricted to a single period.

To achieve this objective, we divided the thesis into three working units:

1. Initially, we proposed a new abstraction named *refactoring graphs*, aspiring to assess and understand refactoring operations over time.

2. In the second working unit, we explored the characteristics of refactoring operations performed over time, relying on *refactoring graphs* to conduct this exploration.

3. The results from the second working unit confirmed that refactorings are not independent operations, inspiring two further studies involving software understanding and documentation. First, we performed a study relying on *refactoring graphs* to evaluate and visualize refactoring tasks performed by undergraduate students from a Software Engineering course. Then, we used *refactoring graphs* to document a catalog of sequences of refactoring operations which we decided to call composite refactorings.

We summarize each work and highlight their contribution in the remainder of this section.

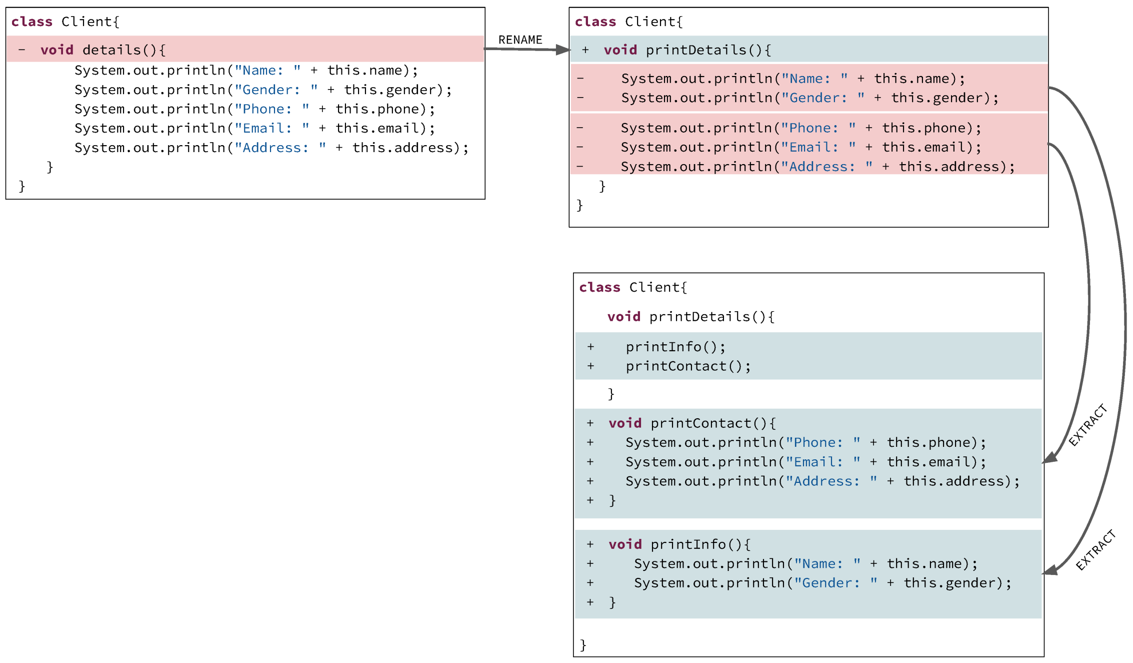
1.2.1 Definition of Refactoring Graphs

In this thesis, we initially proposed an abstraction named *refactoring graphs*, aiming to study and reason about refactoring operations over time.

In such graphs, the nodes are methods or functions, and the edges represent refactoring operations. For example, suppose that method `details()` is renamed to `printDetails()` in class `Client`, as shown in Figure 1.1. This operation is represented by two nodes and one edge connecting them.[2] After this first refactoring, suppose that two methods called `printInfo()` and `printContact()` are extracted from `printDetails()`. As a result, edges connecting `printDetails()` to new nodes are also added to the graph, representing `printInfo()` and `printContact()`.

Therefore, these operations generate a refactoring graph with four nodes and three edges, as presented in Figure 1.2. It is worth noting that, *refactoring graphs* do not impose time constraints on the represented refactoring operations. In our example, the rename operation, for instance, can be performed months after the extract operations. Also, it does not include details about the order of the refactoring operations. However, it is possible to retrieve this information from the metadata in the edge. Finally, *refactoring graphs* may include refactorings performed by different developers. In our example, the rename operation can be performed by d_1 and the extract by another developer d_2.

Figure 1.1: Example of refactoring operations



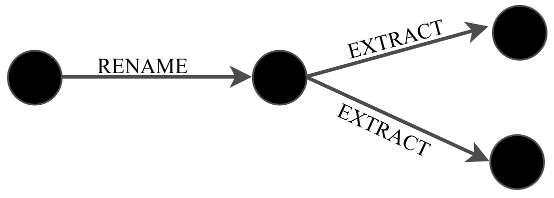
```
class Client{
-  void details(){
        System.out.println("Name: " + this.name);
        System.out.println("Gender: " + this.gender);
        System.out.println("Phone: " + this.phone);
        System.out.println("Email: " + this.email);
        System.out.println("Address: " + this.address);
    }
}
```

```
class Client{
 +  void printDetails(){
 -      System.out.println("Name: " + this.name);
 -      System.out.println("Gender: " + this.gender);
 -      System.out.println("Phone: " + this.phone);
 -      System.out.println("Email: " + this.email);
 -      System.out.println("Address: " + this.address);
    }
}
```

```
class Client{
    void printDetails(){
 +     printInfo();
 +     printContact();
    }
 +  void printContact(){
 +     System.out.println("Phone: " + this.phone);
 +     System.out.println("Email: " + this.email);
 +     System.out.println("Address: " + this.address);
 +  }
 +  void printInfo(){
 +      System.out.println("Name: " + this.name);
 +      System.out.println("Gender: " + this.gender);
 +  }
}
```

[2]After a rename operation, for instance, developers must update the code to use the new method signature. However, the traceability of these changes is outside the scope of the proposed graph-based abstraction.

Figure 1.2: Example of a refactoring graph



In summary, in this first working unit, the main contribution concerns the proposal and definition of this graph-based model by conducting two main activities:

- We formalized the proposed abstraction, i.e., we defined how to represent nodes and edges. We start by providing an overview, using examples to clarify the graph model. Then, we define the graph-based model by describing the elements and steps to build such graphs.
- We implemented a set of scripts to mine and visualize refactoring subgraphs, aiming to provide tool support for investigating refactoring practices beyond single and independent operations.

1.2.2 Characterization of Refactoring Graphs

In the second working unit, we seek to empirically study and characterize refactoring operations performed over time, relying on *refactoring graphs* for this purpose. Particularly, we focused on refactoring operations performed on popular GitHub projects. Besides Java projects, we also study refactorings in JavaScript systems, aiming to fill the gap in studies about refactoring in dynamic programming languages. In this working unit, we are providing the following main results and contributions:

- We revealed several characteristics of a large sample of refactoring operations performed over time, extracted from 20 real-world software projects. Specifically, we investigated six characteristics of 1,525 refactoring subgraphs from well-known and popular Java and JavaScript projects: size, number of commits, time span, homogeneity, ownership, and common patterns. The results confirmed our hypothesis that refactorings are not only sole operations but also sequences of operations performed over time. Indeed, refactorings over time are very common, though most refactoring graphs are small in terms of the number of refactoring operations.
- As a complementary perspective to the quantitative analysis, we performed a survey with authors of large refactoring subgraphs. In this study, we asked about their

motivations for performing the refactoring operations. Most developers mentioned large refactorings were performed to improve design, facilitate the implementation and maintenance of new features, or bug fixing.

- We discussed several applications of refactoring graphs. In particular, we mention that they can be a key abstraction for improving current refactoring tools, which focus on single refactoring operations. We also discussed the applications for refactoring graphs in other scenarios, such as code visualization and comprehension.
- Finally, we designed and implemented a web application to easily visualize refactoring graphs, which is publicity available at `https://refactoring-graph.github.io`

1.2.3 Applications of Refactoring Graphs

Using a canonical example proposed by Fowler [35], we also investigated how refactoring graphs can help program comprehension. Particularly, we assumed the perspective of educators who need to understand refactoring operations performed by students. We used refactoring graphs to visualize, understand, and evaluate the refactorings performed by 46 undergraduate students of a Software Engineering course.

Specifically, we invited students to perform refactoring tasks under two distinct scenarios. First, the students followed a list of explicit and well-defined guidelines, i.e., refactoring tasks with precise instructions about the piece of code that should be refactored. Then, they also performed refactoring tasks following flexible and open guidelines. The refactoring tasks generate a large refactoring subgraph with 24 nodes and 26 edges.

Then, we relied on two strategies to evaluate the results. In the case of explicit guidelines, we used a similarity metric to compare our ground-truth subgraph with the students' subgraphs. For the second part, flexible guidelines, we performed a manual inspection by verifying entity names and operations.

1.2.4 Catalog of Composite Refactorings

In previous working units, we mined hundreds of refactoring graphs. Among the results, we noticed recurring sequences of refactoring operations to compose or decompose a program element. For instance, we noticed that developers frequently perform multiple

Extract Method operations from distinct methods, aiming to generate a single one. However, the most popular refactoring catalog—proposed by Fowler [35, 36]—does not document these observed groups of transformations. Instead, it provides examples and mechanics to perform mostly single refactoring operations. Therefore, in this final working unit, we proposed a refactoring catalog, and we decided to call them composite refactorings [62, 89, 96]. In this context, we provide the following contributions:

- We introduced a catalog, which includes eight instances of sequences of refactoring operations to compose or decompose program elements. As usual in refactoring catalogs, we defined the involved refactoring types and their mechanics. We also relied on *refactoring graphs* to document and visualize instances of the composite refactorings from relevant software systems.

- We implemented a set of scripts to detect the composite refactorings in this catalog. The scripts provide visual outputs based on refactoring graphs.

- We characterized a large sample of composite refactorings in two datasets. We first mined composites from a well-known refactoring oracle [93, 95, 96], which includes hundreds of atomic refactoring instances. Particularly, we show that about 60% of such refactorings are part of composite instances. Then, as a complementary analysis, we mined composite refactorings in the history of popular GitHub projects.

1.3 Publications

The results of this thesis generated the following publications:

- **SANER'20** Brito, A., Hora, A., and Valente, M. T. Refactoring graphs: Assessing refactoring over time. In *27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 367–377, 2020. doi: 10.1109/SANER48275.2020.9054864. (**Chapter 3**)

- **EMSE'21** Brito, A., Hora, A., and Valente, M. T. Characterizing refactoring graphs in Java and JavaScript projects. *Empirical Software Engineering*, 26, 2021. doi: 10.1007/s10664-021-10023-3. (**Chapter 3**)

- **CibSE'22** Brito, A., Hora, A., and Valente, M. T. Understanding refactoring tasks over time: A study using refactoring graphs. In *25th Ibero-American Conference on Software Engineering (CIbSE)*, pages 330–344, 2022. doi: 10.5753/cibse.2022.20982. (**Chapter 4**)

- **JSEP'23** Brito, A., Hora, A., and Valente, M. T. Towards a catalog of composite refactorings. *Journal of Software: Evolution and Process*, e2530, 2023. doi: 10.1002/smr.2530. (**Chapter 5**)

The following publication represents an earlier research effort during this Ph.D.:

- **EMSE'20** Brito, A., Valente, M. T., Xavier, L., and Hora, A. You broke my code: Understanding the motivations for breaking changes in APIs. *Empirical Software Engineering*, 25:1458–1492, 2020. doi: 10.1007/s10664-019-09756-z.

1.4 Thesis Outline

We organize this thesis as follows:

Chapter 2 comprises the state of the art. We separate related work into four categories, involving batch and composite refactorings, catalogs, refactoring comprehension, and refactoring practices during software evolution. We highlight the main differences between such works and ours. We also briefly discuss the tool that is used to detect refactoring operations in this thesis.

Chapter 3 provides basic concepts on *refactoring graphs*, which is the abstraction proposed in this thesis to assess refactorings over time. We also detail the elements and steps to build this graph-based abstraction. Next, we present a study aiming to understand, characterize, and compute the occurrences of refactoring graphs in two programming languages: Java and JavaScript. We describe the study methodology as well as its findings based on a sample of approximately 1.5K refactoring subgraphs.

Chapter 4 details a first application of refactoring graphs, where we explore how refactoring graphs can assist educators in Software Engineering courses. Specifically, we used the proposed abstraction to understand refactoring operations performed by undergraduate students.

Chapter 5 shows a second application of refactoring graphs, inspired by insights from previous analyses described in Chapter 3. Specifically, we introduce a catalog of composite refactorings, i.e., sequences of operations to compose or decompose a source program element. We also rely on our graph-based model to document and illustrate the instances of the catalog.

Chapter 6 summarizes the conclusions we leveraged throughout this thesis. It also outlines some ideas we find interesting to investigate in the future.

Chapter 2

Background & Related Work

In this chapter, we present background information and work related to this PhD thesis. First, we introduce RefDiff, the tool that we use to detect refactoring operations (Section 2.1). Next, we present studies related to refactoring operations (Section 2.2). Particularly, we discuss two central types of studies: on *batch* refactorings and on *composite* refactorings. Moreover, we also present in this chapter an overview of other works, highlighting their differences to this thesis content. Specifically, we report field studies on refactoring comprehension (Section 2.3) by considering the educational and developer perspective. We also discuss catalogs of refactoring (Section 2.4), contrasting with the study reported in Chapter 5. Finally, we describe other studies on refactoring (Section 2.5), which are related with the studies described from Chapters 3 to 5. We conclude this chapter with general statements about the discussed topics (Section 2.6).

2.1 Detecting Refactoring Operations

RefDiff is a tool to detect refactoring operations [85, 87]. The current version is based on the *Code Structure Tree* (CST), which provides a language-agnostic representation of the source code. As a consequence, it is possible to detect refactorings in multiple languages. In this thesis, we concentrate on two programming languages supported by the tool: Java and JavaScript. Also, we focus on operations at the method or function level. Table 2.1 lists the refactorings detected by RefDiff at these levels. As we can notice, both languages have known refactorings, comprising *extract* and *inline* operations, as well as changes in method's signature (i.e., *rename* and *move*).

Since JavaScript is a prototype-based language, there are no inheritance-based refactorings (i.e., *pull up* and *push down*). In additon, JavaScript systems usually contain large files that are composed of several nested elements. For this reason, many refactorings occur on a single file. RefDiff reports these cases as internal operations. Figure 2.1 shows an example of an *internal move* operation. In this case, the developer moved function f_1

from f_a to f_b. However, both functions are located in a single file.

Table 2.1: Function and method level refactorings detected by RefDiff

Language	**Refactoring**
Java	rename method, move method, move and rename method, inline method, extract method, extract and move method, push down method, pull up method
JavaScript	rename function, move function, move and rename function, inline function, extract function, extract and move function, internal move function, internal rename and move function

Figure 2.1: Example of an *internal move* operation in JavaScript (strikethrough text represents deleted line and the symbol "+" denotes added lines of code)

```
function fa() {
~~  f1 = () => {~~
~~      ...~~
~~  }~~
  f2 = () => {
  ...
  }
  return {~~f1~~, f2};
}

function fb() {
+ f1 = () => {
+     ...
+ }
+ return {f1};
}
```

2.2 Batch and Composite Refactorings

There are two central types of studies regarding groups of related refactoring operations, studies on *batch* refactorings and studies on *composite* refactorings. Batch refactorings refer to a set of single refactoring operations, which are then grouped considering criteria such as time [69, 70], version system [26], and developers [12, 26]. As mentioned by Cedrim *et. al.* [26], *"the way the batches are synthesized is open-ended, i.e., different developers can have different views of how to create a batch"*. Similarly, composites are defined as sequences of atomic refactoring operations [62, 89, 96]. This concept is explored in contexts like domain specific languages for describing refactoring [62] and code smells [13, 89].

Murphy-Hill et. al. [70] analyzed four datasets from different sources, all of these including metadata about the usage of Eclipse IDE. For instance, the dataset named *Everyone* contains Eclipse refactoring commands used by developers. Based on these datasets, the authors discuss usage and configurations of refactoring tools, frequency of refactoring operations, and commit messages. They also investigated refactorings operations executed in 60 seconds, which are named *batches.* The authors state that the some refactorings types are more common in batches, such as *rename*, *introduce a parameter*, and *encapsulate field.* Besides that, about 47% of refactorings performed using a refactoring tool happen in batches. However, the batches involve a short period: the study does not investigate refactorings operations that occur in different moments over time.

In another context, Bibiano et. al. [12] point out that sets of related refactorings can solve problems due to code smells. The authors studied 54 GitHub projects and three closed systems. First, they used RefactoringMiner tool to detect 13 known refactorings [95], resulting in 24,893 operations. Then, the authors applied a heuristic to compute batch refactorings, i.e., set of related refactorings [26]. The heuristic includes two main requirements do retrieve a batch refactoring: (i) there are more than two refactoring operations in a single entity and (ii) the operations are from a single developer. The results are 4.607 batch refactorings. Next, the authors used another tool and scripts to identify more than 41K code smell occurrences in these systems. Finally, the authors computed the effect of batch refactorings to remove code smells. The main results show that most batches have only one commit (93%) and two refactoring types. Also, the authors state that batches have a negative or neutral effect on code smells (81%). However, the authors focus on code smells and operations performed by a single developer. In our study described in Chapter 3, the subgraphs involve refactoring over time (i.e., more than one commit), including subgraphs by multiples developers and different code elements.

Sousa *et. al.* [89] originally defined composite refactorings as *"two or more interrelated refactorings that affect one or more elements"*. The detection of composite refactoring relies on three distinct heuristics. The first heuristic–called *element-based*–combines single refactoring operations by the scope. The scope can be, for instance, a single class. For example, the authors show a composite refactoring from this category, which includes the movement of attributes, movement of methods, and extract superclass operations. The *commit-based* heuristic links refactoring operations performed in a single commit. Finally, the third heuristic—named *range-based*—connects refactorings by location (e.g., if a refactoring crosscuts two classes named C_1 and C_2, both are part of the location). As a consequence, an instance of a composite can include mixed operations at distinct levels, i.e, classes, attributes, and methods. In summary, the study considers some criteria to cluster composites, also reusing previous heuristics [12, 26]. However, they do not introduce and document a catalog of composite refactorings (as we do in Chapter 5) and a significant part of the study investigates the relevance of composites for removing code smells. Although we are reusing the definition, in this PhD thesis, we explore another perspective. Specifically, in Chapter 5, our key goal is to propose and document a catalog of composite refactorings. Moreover, we also show the importance of composite refactorings by mining and characterizing their occurrence in two datasets: a sample with hundreds of confirmed single refactoring operations and the history of ten known open-source projects. For this purpose, we rely on *refactoring graphs* to document and visualize instances of the composite refactorings in real-world scenarios.

There are also studies focusing on subcategories of composite refactorings. For example, “incomplete composites” [11], i.e., when the composite refactoring *“is not able to entirely remove a smelly structure”*.

Fowler [35] mention a similar term called *big refactoring*. The author points out that some refactorings are atomic, i.e., they are finished in a few minutes. By contrast, there are big refactorings, which are performed during months or years. We reinforce this observation in Chapter 3: the time span of the refactoring subgraphs is diverse, ranging from days to weeks or even months.

2.3 Refactoring Comprehension

Several studies aiming to understand refactoring activities. A significant part of them focuses on developers’ perception of refactoring. In this case, the goal involves understanding refactoring activities by investigating, for example, benefits and challenges [56, 57], merge conflicts [64], motivations to refactor a source code [75, 76, 86, 99], association with technical debt [50], and refactoring opportunities [25].

Silva et. al. [86] performed *firehouse* interviews to understand the reasons behind refactoring operations in GitHub projects. Based on 195 developers’ answers, the authors found 44 reasons to refactor methods and attributes in Java. As in our study reported in Chapter 3, the authors contacted GitHub developers by email and used *thematic analysis* to examine the responses [29]. Five refactoring instances are also in our study: *extract method*, *move method*, *inline method*, *pull up method*, and *push down method*. Besides that, there are related motivations in our category *improve code design* (e.g., the movement of elements to an appropriate container). However, in our research presented in Chapter 3, we investigate sets of refactoring operations that generate large subgraphs in Java and JavaScript systems. This is different from the mentioned study, which focuses on motivations behind refactorings performed in a single commit and Java projects. That is, in our study, we explore another perspective, centering on a large set of refactoring activities over time in distinct software ecosystems.

A recent study also assesses motivations behind refactoring instances [75]. The authors conducted quantitative and qualitative research on a large scale by analyzing refactoring activities in 150 GitHub projects. In the quantitative part, the authors discuss metrics involving code quality (e.g., number of elements, the coupling between classes), code smells, and process-related factors (e.g., number of commits in releases, number of fixed bugs). The qualitative results extend the catalog proposed by Silva *et al.* [86], adding 26 new ones. The motivations are based on discussions in 551 pull requests, as well as comments in the related commits. In Chapter 3, our category *improve code design* is inspired by a core theme proposed by this research, involving the improvement of encapsulation and maintainability. Besides, our category *"fix bugs or improve existing features"* also incorporates another theme, which is called *"Prevent Bugs"*. Interestingly, the main authors' findings point out that 52% of the cases, the discussions do not focus on a particular refactoring, i.e., the developers mention a combination of refactoring operations. However, the study focuses only on operations mentioned in pull requests and Java projects.

Lastly, the improvement of existing features is also reported in a recent study about refactoring operations in the code review process [73]. Similar to our results and previous researches [74, 75, 86], the authors mention the occurrence of refactoring operations associated with feature maintenance or bug fixing. The authors also reinforce the idea that refactoring is not a sole operation by investigating sequences in code reviews. The main findings point to Extract methods occurring with other refactoring types in the Java ecosystem. In Chapter 3, we used the *Gspan* algorithm to investigate refactoring patterns in the subgraphs [102]. However, in our study, the most recurrent pattern in Java refers to successive *rename* operations, occurring in 153 subgraphs. Our results also suggest that patterns do not necessarily occur between reviews. That is, refactoring patterns can happen in a single commit, i.e., atomic subgraphs.

As previously mentioned, most studies focus on open-source developers' perspectives. On the educational side, research on refactoring covers distinct scenarios. Demeyer et. al. [31] propose refactoring examples for students. López et. at. [63] propose activities to teach refactoring in Computer Science. The study suggests a set of tasks, such as reading texts, comprehension of refactoring catalogs, and practical exercises. Other studies discuss approaches and lessons to promote refactoring practices [34, 46, 92]. There are also tools to improve the student's perception of refactoring [3]. However, the mentioned studies do not consider graph-based abstractions to evaluate refactoring activities over time.

Keuning et. al. [54] performed a study with 133 students from the second year of a Software Engineering course. The experiment uses a tutoring system, which contains a set of refactoring exercises and provides feedback during the refactoring tasks. Specifically, there is an option to check the student progress, and a second functionality to get hints (i.e., to diagnose and provide suggestions). The experiment happened during one hour, including 15 minutes to discuss code quality and demonstration of the tool, 30 minutes to perform the refactoring tasks, and 15 minutes to answer a survey about the experiment. As in our study reported in Chapter 4, the authors pointed to a high rate of refactoring tasks performed successfully. The results also show that students frequently used the features to get hints and visualize their progress. The students' mistakes were extracted from log entries of the system. However, most of them are outside the scope of refactoring. For example, the authors report compiler errors due to language unsupported constructs, runtime errors, and failed tests. Besides, each of the five exercises involved an isolated issue. For example, two tasks are from a website that contains a list of code problems. In Chapter 4, we use a canonical refactoring example (proposed by Fowler [35], with 18 refactoring tasks) and leverage *refactoring graphs* to assess refactoring operations performed over time.

Shamsa et. al. [2] conducted a study with 23 students. Each group performed a kind of project scheme in a real-world system. The first scheme contains a set of software changes, ending with refactoring tasks. In contrast, the second scheme starts with refactoring issues. The authors reported better results in the second one. However, the study does not concentrate exactly on refactoring comprehension. The authors focus on the main differences by performing refactoring operations before and after functional improvements of the system. Besides, the tasks are performed by two or three students. Thus, different students' perceptions of refactoring can influence the results. In Chapter 4, each result refers to a single student. In addition, our guidelines are based on a known refactoring example.

Karac et. al. [53] investigated the impact of task granularity on Test-Driven Development (TDD). Overall, the study encompasses five main steps. For example, there are steps to implement a feature and the respective tests. The last step involves refactoring operations to improve code quality. The study involves 52 students, most of them are novice programs. The tasks rely on programming exercises, which are essentially algorithmic. A group of students performed decomposed tasks. In contrast, the second group received tasks without decomposition in small steps. Among the main results, the authors argue about the importance of breaking larger work into smaller ones, mainly for novice practitioners. However, the study concentrates on the granularity of issues, and the refactoring operations are just a step of the experiment.

2.4 Catalog of Refactorings

Catalogs constitute a reference guide for communication between practitioners since they standardize a common refactoring vocabulary. Developers frequently rely on such documents to learn and perform refactoring operations. Fowler's book includes a popular and widely used catalog of refactoring operations [35, 36]. However, in his seminal book, most examples and mechanics describe how to perform single and independent refactoring operations. The recent version includes new composite refactorings, such as Inline Class and Collapse Hierarchy.[1] In the case of Inline Class, we eliminate a class by moving all elements to distinct ones. Therefore, it is a subcategory of Class Decomposition, which is described in Chapter 5. However, the current refactoring detection tools do not support this composite [6, 22, 85, 87, 95, 96]. In Collapse Hierarchy, we eliminate subclasses by moving all elements to the superclass. Therefore, Composite Pull Up Method can be a part of this operation, which is another instance from our catalog. The current version of RefactoringMiner detects this refactoring [96]. However, it is not properly explored in the literature. For example, the oracle used in Chapter 5 includes only a single instance of Collapse Hierarchy.

Recently, Bibiano et. al. [13] investigated *"complete composites"*, i.e., sets of refactoring operations that remove the whole occurrence of four code smell types. The study includes 618 complete composites formed by known refactoring operations. Differently from our study, the identification of composite refactorings relies on a range-based heuristic defined in previous studies [89], which groups refactorings affecting the same location. The authors also present a catalog of complete composites to remove code smells. This catalog includes five complete composites, which are sequences of Move Method or Extract Method operations. For example, their catalog focuses on the removal of Long Method (i.e., large and complex methods) and Feature Envy (i.e., a method that uses several methods from a distinct class). For each instance, the authors discuss side-effects, i.e., when a composite removes a target code smell but introduces other ones. Among the five complete composites from their catalog, three instances refer to extract operations to remove long methods. However, in the first case, the extraction contributes to introducing a Feature Envy. The second one does not reduce the method's size, i.e., it is necessary to perform new extract operations to remove the smell. Finally, the third instance introduces a long parameter list before the extraction. In Chapter 5, we propose a catalog of eight types of composite refactoring, which are formed by distinct refactoring types. We cluster refactoring operations by considering the source or target code elements. In other words, our scripts identify sequences of single refactoring operations to compose or decompose a

[1]`https://refactoring.com/catalog`

source code element, regardless of the presence of a code smell. In fact, there are several reasons to refactor a given source code element, which do not necessarily involve smell removal [75, 86]. We rely on *refactoring graphs* to document and visualize instances of the composites in relevant software systems.

Tsantalis et. al. [96] also present a brief discussion regards composites. The authors introduce a new version of RefactoringMiner, which detects Extract Class—also defined in Fowler's catalog [35, 36]. According to the authors, composite refactorings *"are composed of basic ones"*. Therefore, Extract Class matches this concept, since it comprises a set of Move Method and Move Field operations aiming to generate a new class. In our catalog (Chapter 5), Class Decomposition can include a set of move operations to a new class or existing one. However, the focus refers to the decomposition of the source class.

2.5 Other Studies on Refactoring

Hora et. al. [49] analyze untracked changes during software development. The authors show that refactorings invalidate several tracking strategies to evaluate system evolution. As in this thesis, they represent evolutionary changes as graphs. In this case, each node refers to a class or a method, and the edges indicate tracked changes (i.e., entities that keep their names after a modification) and untracked changes (i.e., entities that change their names after a refactoring). That is, a graph represents traceable changes or alterations that split the entity's history. The results point up to 21% of the changes at the method level and up to 15% at the class level are untraceable. By contrast, in Chapter 3, the goal is to investigate refactorings performed over long time windows; we do not concentrate on tracked modifications on source code.

Meananeatra [66] also reports changes during software evolution as graphs. However, the study concentrates on refactoring sequences to remove *long methods*. The author proposes an approach based on two main criteria to detect an optimal set of refactorings. An optimal refactoring sequence centers on four metrics: number of removed bad smells, size of the refactoring sequence, number of the affected code elements, and the maintainability value (i.e., analyzability, changeability, stability, and testability). The technique represents candidate refactoring sequences as graphs. In this case, a graph contains a root node representing the original method version with smells. Each new node denotes a new method version after a refactoring operation. Similar to *refactoring graphs*, the edges refer to refactorings. By contrast, the nodes represent the same method before and after the changes. Each path in the graph is a candidate refactoring sequence, which can meet the selection criteria. Thus, the study does not focus on real refactorings over time.

Instead, the graph model represents steps to decompose a long method.

Finally, we reinforce findings from a recent study that points out a significant rate of multiple extractions to decompose methods in a single commit [48]. The authors show that Extract Method operations are frequently performed by developers, who create methods for distinct purposes, such as testing, validation, and setup. However, the study does not document a catalog of composite refactorings. Its goal is to characterize method extractions, for example, their content, size, and degree. Similarly, in our study described in Chapter 5, Method Decomposition and Method Composition—composites formed by Extract Method and Extract and Move Method operations—are among the top-3 most frequent composites. In the study in the wild (Section 5.6), for example, we detected 1,265 occurrences. Among them, 275 composites (21.7%) involving extractions are performed over multiple commits.

2.6 Final Remarks

In this chapter, we present background information and work related to the major themes of this thesis. We started by describing RefDiff, the tool used to detect refactoring operations. Specifically, we introduced a brief discussion regarding the employed refactorings and main characteristics of the tool. Then, we discussed two field studies involving batches and composite refactorings by highlighting their differences with *refactoring graphs*. Finally, we also mentioned other works related to this thesis content, such as software evolution, catalogs of refactoring, and software comprehension.

Overall, we show the pertinence of studying refactorings performed over time. We do not cover the whole scenario by concentrating on independent operations. Refactoring performed over time may affect algorithms, tools, approaches, and maintenance tasks. There are investigations on sets of refactoring operations, aiming to better understand new perspectives of refactoring. However, to the best of our knowledge, refactorings performed over a long time are not deeply investigated by the literature. It is also possible to notice the need for approaches and abstractions to track, understand, visualize and document refactoring operations in such scenarios.

Chapter 3

Defining and Characterizing Refactoring Graphs

In this chapter, **we introduce the formal definition of refactoring graphs** by defining the graph elements and formalizing the steps to build them. We also describe the results of a exploratory study, which intends to understand, characterize, and compute the frequency of the proposed abstraction.

This chapter is organized as follows. Section 3.1 introduces the definition of refactoring graphs. Section 3.2 presents a quantitative study. In this case, we extracted graphs for 20 well-known and popular open-source Java and JavaScript projects, investigating six characteristics: size, number of commits, time span, homogeneity, ownership, and patterns. Among the main findings, we observed that most refactoring subgraphs are small. However, we also notice subgraphs describing large refactoring operations. Therefore, we also performed a qualitative analysis, which is described in Section 3.3. Specifically, we contacted the authors of large refactoring subgraphs, asking for the motivations behind their operations. We discuss the key applications and implications in Section 3.4, while Section 3.5 states threats to validity. Lastly, we conclude this chapter in Section 3.6.

3.1 Definition and Examples

3.1.1 Overview

A *refactoring graph* RG is a set of disconnected subgraphs $G' = (V', E')$. Each G' is named a *refactoring subgraph*, with a set of vertices V' and a set of directed edges E'. The history of a software system includes a set of refactoring subgraphs. In refactoring (sub)-graphs, the vertices are the full signatures of methods or functions. For instance, in Java projects, we labeled a method $m()$ in class *Foo* and package *util* as *util.Foo#m()*. Since Java is a strongly typed programming language, the signature also includes the type of the parameters. For example, we label the same method m as *util.Foo#m(String)* wherever it requires a string type parameter. In JavaScript graphs, this procedure is not practical since it is an untyped language. Thus, we labeled the vertices utilizing the file name. For example, *util.Bar.js.C#f1* represents a function $f1$ in class C, file *Bar.js*, and directory *util*. Finally, the edges indicate the refactoring type (e.g., *move method*) and they also include meta-data about the operation (e.g., author name and date).

Figure 3.1: Refactoring subgraph produced by only one developer

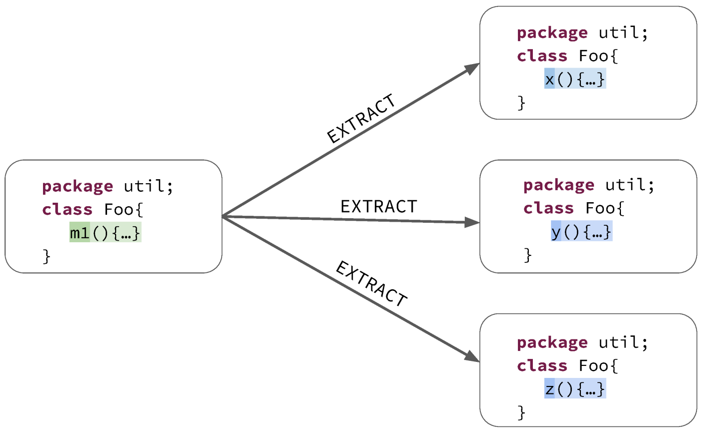


Figure 3.2: Refactoring subgraph over time

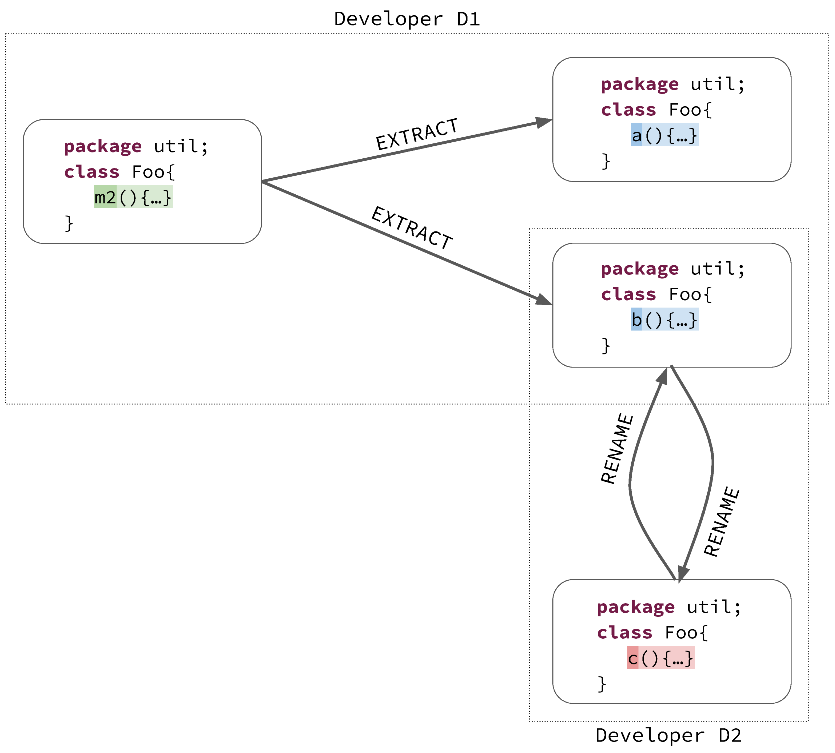
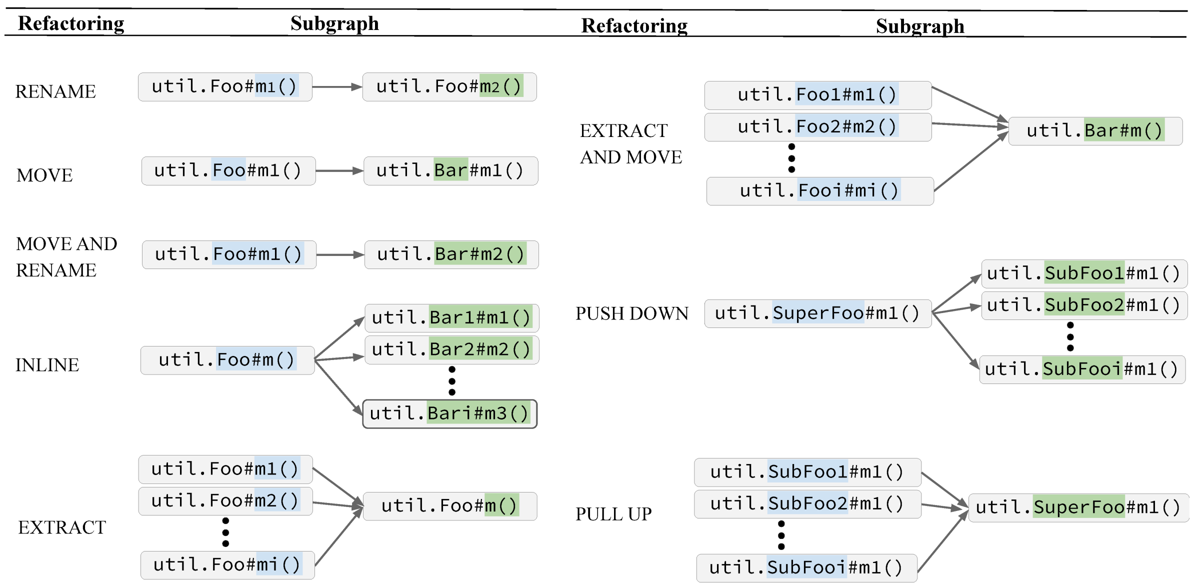


Figure 3.1 shows an example of a *refactoring graph*. A developer extracted three methods from $m1()$, which are named $x()$, $y()$, and $z()$. The edges refer to the refactoring operation. It is worth noting that a refactoring graph can include refactorings performed by multiple developers. For instance, Figure 3.2 illustrates a second example, where a developer $D1$ extracted two methods from $m2()$, which are named $a()$ and $b()$. Then, a second developer $D2$ renamed $b()$ to $c()$. After that, a reviewer might have suggested to keep the original name. Thus, the developer undid the latest refactoring, renaming $c()$ to $b()$ again. In this case, the graph contains refactorings performed by two authors. Besides, a cycle is created when the developer reverts the method to the original name.

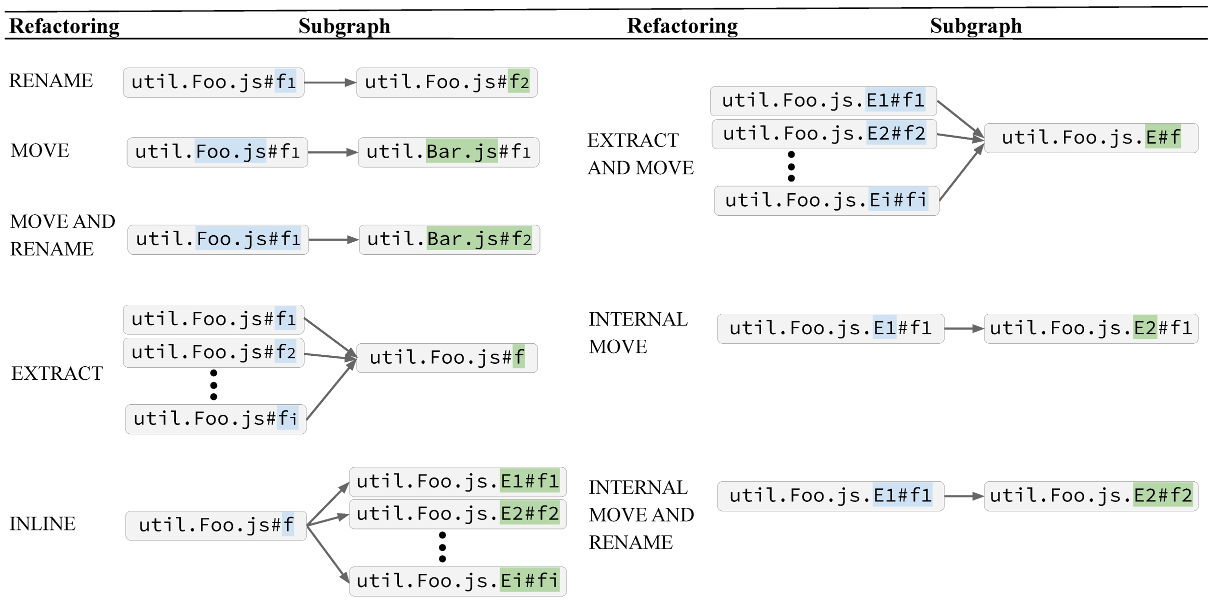
As presented in Figure 3.3, in the case of Java, we center our study on eight distinct refactorings at the method level. *Rename* and *move* are the most trivial operations since they involve just changing the method's signature. Extract operations generate new methods in the same class (i.e., they create a new node in our subgraphs). It is also possible to extract a method $m()$ or multiple methods m_i from a single method $m1()$. Furthermore, as illustrated in Figure 3.3, it is possible to extract $m()$ from multiple methods m_i. In this case, the extracted code is duplicated in each method m_i. *Inline method* is a dual operation, involving the removal of trivial elements and replacement of the respective calls by their content. As in the case of *extract*, we can inline a method $m()$ in multiple methods m_i. We also studied a refactoring called *extract and move* that extracts a method to another class. Finally, inheritance-based refactorings comprise the movement of one or more methods to supertypes or subtypes (i.e., *pull up* and *push down*). For example, a *pull up* moves methods from subclasses to a superclass.

Figure 3.3: Example of refactoring subgraphs (Java)



Similar refactorings apply to functions in JavaScript. As shown in Figure 3.4, in JavaScript, there are also internal operations, i.e., refactorings performed in a single file.

Figure 3.4: Example of refactoring subgraphs (JavaScript)



3.1.2 Definition

Suppose a software project with a history of *Commits*. Using a refactoring detection tool, we first iterate over *Commits* to identify a set of *Refactorings*, such that:

$$Refactorings = \{ref_1, ref_2, ref_3 \dots ref_n\}$$

Each refactoring *ref* is defined by a tuple:

$$ref = (source, target, refType)$$

Specifically, for each *ref*, *source* represents a method before applying the refactoring *ref*, *target* represents a method created by performing *ref* or an existing method impacted by *ref*,[1] and *refType* represents the refactoring type associated with *ref* such that: *refType* $\in \{rename, move, move\_rename, inline, extract, extract\_move, push\_down, pull\_up, internal\_move, internal\_rename\_move\}$.

Refactorings were performed over a set of *Methods*.[2] Each *meth* $\in$ *Methods* includes information about the usual container structure, such that:

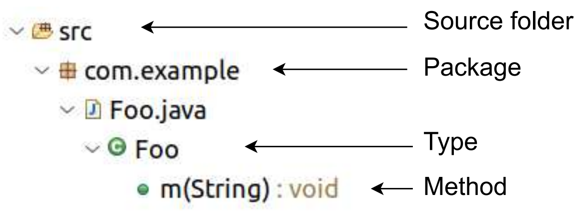
$$meth = (src, pkg, type, sig)$$

[1]For example, for Inline Method operations, the targets are existing methods, which receive the content of the Inlined Method.

[2]It is also straightforward to extend our model to support other code elements, such as packages and classes.

meth is a unique identifier referring to the localization of a method at a commit, before or after a refactoring operation, respectively. The identifier includes source folder (*src*), package (*pkg*), and the type declaration that the method belongs to (*type*), followed by the method signature (*sig*). Figure 3.5 shows an example, in which *meth* = (`src`, `com.example`, `Foo`, `m(String)`). In the case of nested code elements, all types in the hierarchy are part of the identifier. For example, in Figure 3.5, if method `m(String)` is located in an inner class `Bar`, both types are part of the identifier (i.e, *meth* = (`src`, `com.example`, `Foo.Bar`, `m(String)`). The same applies to a prototype-based language, such as JavaScript, in which nested elements compose the complete path of a code element.

Figure 3.5: Example of a method structure in Java



In this context, the triples *(source, target, refType)* result in a refactoring graph *RG*, which includes a set of vertices V and a set of directed edges E, such that:

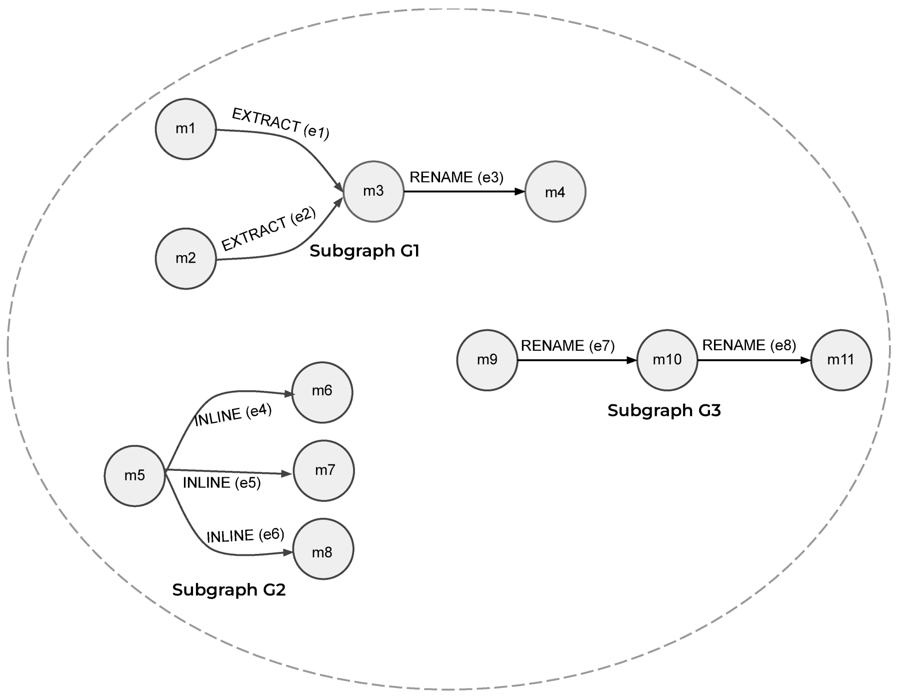
$$V = \{meth \in Methods \mid \{\exists\ (meth, \_, \_) \in Refactorings\}\ or\ \{\exists\ (\_, meth, \_) \in Refactorings\}$$

$$E = \{(source, target) \in Methods \text{ x } Methods \mid \exists (source, target, \_) \in Refactorings\}$$

In fact, a refactoring graph *RG* is a set of disconnected subgraphs *G*. Figure 3.6 shows as example a refactoring graph *RG* that is formed by three subgraphs, G_1, G_2, and G_3. Subgraph G_1 has four methods—m_1, m_2, m_3, and m_4—connected by three refactorings. Specifically, there are two Extract Method operations, whose result is a new method m_3. Then, m_3 was renamed to m_4. In this case, G_1 is a disconnected subgraph because there are no other refactorings (i.e, edges) in subgraphs G_2 and G_3 that "connect" to G_1's methods. Subgraph G_2 represents an example of Inline Method operations, which move the body of a trivial method m_5 to three existing methods (m_6, m_7, and m_8). All methods represented in G_2 are not involved in other refactoring operations over the commit's history. Finally, subgraph G_3 represents a scenario where a developer performed two subsequent Rename Method operations.

The algorithm to build a refactoring graph has four main steps and it requires as input a list of the refactorings performed in the commits of a given system. First, for each refactoring, the algorithm identifies the triples *(source, target, refType)*, i.e., the two methods involved and the refactoring type. Then, it creates two vertices, representing the elements before and after the refactoring operation. In the third step, it creates a directed edge representing this refactoring. The edges are labeled with refactoring's name and with information about the refactoring operation. The final output consists of a refactoring graph, with its subgraphs.

Figure 3.6: Example of refactoring graph (RG) formed by three subgraphs (G_1, G_2, G_3)



$V = \{m_1, m_2, m_3, m_4, m_5, m_6, m_7, m_8, m_9, m_{10}, m_{11}\}$
$E = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8\}$

3.2 Quantitative Study

In this study, our *goal* is to quantitatively analyze refactoring in multiple programming languages with the *purpose* of understanding and characterizing refactoring activities performed over time. The *context* of the study consists of approximately 1.5K refactoring subgraphs from 20 Java and JavaScript open-source projects, ten in each language. . Since refactoring graphs are a novel abstraction, we see value in starting by shedding light on several of their properties. In other words, before performing a qualitative study with developers (Section 3.3), we found it important to mine the maximum amount of data and information about such graphs. Specifically, we address the following research questions, aiming to investigate seven properties: refactorings over time, size, number of commits, time span, homogeneity, ownership, and patterns.[3]

[3]We also designed and implemented a web application to easily visualize the results `https://refactoring-graph.github.io`

- (RQ_0) *How many refactoring operations generate subgraphs over time?* Most studies concentrate on refactorings performed in a single commit [5, 32, 51, 86]. For this reason, the rationale of this preliminary research question is to assess the prevalence of the key practice we investigate in our study, i.e., refactorings that are spread over multiple commits.

- (RQ_1) *What is the size of refactoring subgraphs?* We are interested in investigating the size of refactoring subgraphs, in terms of number of vertices and edges. This investigation may provide insights about the impact of refactorings in the design/architecture of the studied systems.

- (RQ_2) *How many commits are represented in refactoring subgraphs?* Each commit can contribute to one or more refactoring in a refactoring subgraph. Therefore, our objective is to investigate how refactoring subgraphs increase over time. This investigation complements the perspective of previous studies, which rely on refactoring operations detected in a single commit or in a short time interval [70, 89].

- (RQ_3) *What is the time span of refactoring subgraphs?* We investigate the lifetime of subgraphs, i.e., the interval between the first and the latest refactoring operation in a subgraph. For example, this investigation might also provide insights about large and long-running changes in the design/architecture of the studied systems.

- (RQ_4) *Which are the most common refactoring operations in refactoring subgraphs?* In this RQ, we discuss the most recurring refactoring types that occur over time, complementing the panorama of studies that report the frequency of single-commit operations [75, 86, 87, 96]. We also analyze the homogeneity of refactoring subgraphs. In other words, we investigate the frequency of subgraphs formed by the same or distinct refactoring types.

- (RQ_5) *Are refactoring subgraphs created by the same or by multiple developers?* The rationale of this research question is to investigate whether refactoring operations over time are performed by distinct developers. That is, we aim to assess whether refactoring operations over time are concentrated on single developers or spread over multiple ones.

- (RQ_6) *What are the most common refactoring subgraphs?* This research question provides an overview of recurrent graphs in distinct projects, i.e., refactoring graph patterns that occur frequently in our dataset.

3.2.1 Study Design

Selecting Projects

In this chapter, we analyze the characteristics and frequency of refactoring subgraphs in popular Java and JavaScript systems. We used the following criteria for selecting the projects for each programming language. First, the projects should be among the top-100 GitHub repositories in terms of stars, since stars is a key metric to reveal the popularity of repositories [15, 88]. Second, the project should have more than 1K commits (in order to remove recent systems with a short history of refactoring activity). Finally, the project should be a software system. Thus, we removed, for example, code samples (such as iluwatar/java-design-patterns)[4] and JavaScript style guides (such as airbnb/javascript).[5] Table 3.1 describes the selected projects, including basic information, such as number of stars, commits, files, contributors, and short description. These projects cover distinct domains, including web development systems and media processing libraries, for example.

Table 3.1: Selected projects (Java and JavaScript)

Project	**Stars**	**Com.**	**Cont.**	**Files**	**Bran.**	**Desc.**
Elasticsearch	44,489	48,313	1,273	11,770	master	Search engine
RxJava	40,622	5,581	237	1,666	3.x	Event-based lib.
Square Okhttp	34,484	4,273	189	167	master	HTTP client
Square Retrofit	33,801	1,756	129	241	master	HTTP client
Spring Framework	32,582	19,752	396	7,203	master	Web framework
Apache Dubbo	29,353	3,639	249	1,743	master	RPC framework
MPAndroidChart	28,647	2,018	66	220	master	Chart lib.
Glide	27,289	2,416	102	647	master	Image lib.
Lottie Android	26,952	1,139	76	198	master	Animation lib.
Facebook Fresco	15,870	2,158	170	985	master	Image lib.
Vue	163,721	3,099	293	432	dev	UI framework
React	148,441	13,231	1,383	1,378	master	UI library
Parcel	35,651	1,891	233	1,618	v2	Files bundler
Hexo	30,371	3,259	145	272	master	Blog framework
Leaflet	27,805	6,843	643	141	master	Maps lib.
Quill	26,386	5,199	120	89	develop	Text editor
Request	24,553	2,270	286	74	master	HTTP client
Nylas Mail	24,529	6,116	89	120	master	Mail app
Select2	24,415	2,607	442	230	develop	Selector lib.
Carbon	24,061	1,411	125	92	master	Screenshot app

[4]https://github.com/iluwatar/java-design-patterns
[5]https://github.com/airbnb/javascript

Detecting Refactoring Operations

As mentioned in Chapter 2, we use RefDiff [85, 87] to detect the refactoring operations represented in refactoring graphs. RefDiff identifies refactorings between two versions of a git-based project. In our study, we focus on well-known refactoring operations detected by RefDiff at the method or function level, as presented in Figures 3.3 and 3.4 (Section 3.1.1). RefDiff works by comparing each commit with its previous version in history. To avoid analyzing commits from temporary branches, we focus on the main branch evolution. Particularly, we use the command `git log –first-parent` to get the list of commits of each project.[6] Additionally, we remove refactorings in packages that are not part of the core system. For Java projects, we remove refactorings from packages with the keywords *test(s)*, *example(s)*, and *sample(s)*. In JavaScript, we also filter other keywords. For instance, we discarded refactorings from the package *dist*, since it is frequently used to store source code for distribution. Other cases are specific from a single JavaScript system. For example, in Vue, we remove refactorings from *packages/vue-server-renderer* since the documentation mentions: "*This package is auto-generated*".[7]

Building Refactoring Graphs

We identify refactoring subgraphs over time in 20 systems. Table 3.2 presents the frequency of refactoring subgraphs for each Java project, and Table 3.3 presents the results for JavaScript. Considering both languages, we detect a total of 11,341 refactoring subgraphs. In the case of Java, we detect 9,200 subgraphs, whereas 2,141 for JavaScript.

Table 3.2: Frequency of refactoring subgraphs (Java)

Project	**Refactoring Subgraphs**				
	All	*commit* = 1	%	*commit* ≥ 2	%
Elasticsearch	2,150	1,971	91.7	179	8.3
RxJava	1,120	1,034	92.3	86	7.7
Square Okhttp	650	563	86.6	87	13.4
Square Retrofit	182	148	81.3	34	18.7
Spring Framework	3,206	2,705	84.4	501	15.6
Apache Dubbo	486	452	93.0	34	7.0
MPAndroidChart	453	380	83.9	73	16.1
Glide	441	296	67.1	145	32.9
Lottie Android	197	174	88.3	23	11.7
Facebook Fresco	315	279	88.6	36	11.4
All	9,200	8,002	87.0	1,198	13.0

[6]`https://git-scm.com/docs/git-log#Documentation/git-log.txt---first-parent`
[7]`https://github.com/vuejs/vue/tree/dev/packages/vue-server-renderer`

Table 3.3: Frequency of refactoring subgraphs (JavaScript)

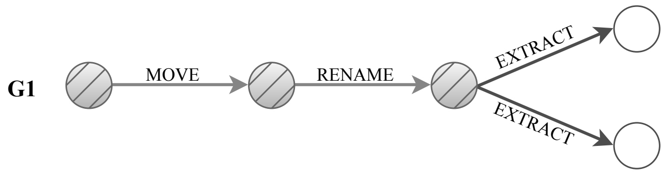
Project	**Refactoring Subgraphs**				
	All	*commit* = 1	%	*commit* ≥ 2	%
Vue	281	218	77.6	63	22.4
React	843	737	87.4	106	12.6
Parcel	108	96	88.9	12	11.1
Hexo	196	168	85.7	28	14.3
Leaflet	268	206	76.9	62	23.1
Quill	217	197	90.8	20	9.2
Request	59	43	72.9	16	27.1
Nylas Mail	72	67	93.1	5	6.9
Select2	69	63	91.3	6	8.7
Carbon	28	19	67.9	9	32.1
All	2,141	1,814	84.7	327	15.3

Spring Framework has the highest number of subgraphs (3,206), while Square Retrofit has the lowest number (182). Overall, 87% of the refactoring subgraphs comprise operations performed in a single commit. This ratio varies from 67.1% (Glide) to 93% (Apache Dubbo). The results follow a similar trend in JavaScript systems. The percentage of single-commit subgraphs ranges from 67.9% (Carbon) to 93.1% (Nylas Mail).
From RQ1 to RQ5, we assess 1,525 subgraphs with number of commits ≥ 2, because they are the ones that represent refactorings over time.

Mining Frequent Graphs

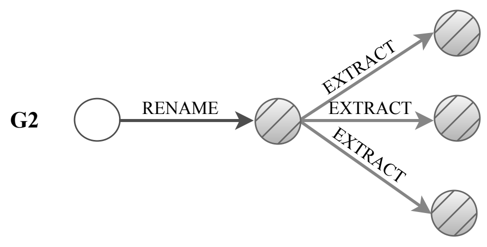
In our last research question (RQ6), we investigate frequent graphs, i.e., graph patterns that occur frequently in our dataset. For this analysis, we use *GSpan*, a well-known algorithm that identifies subgraphs whose incidence is greater than a given support [61, 102]. Figure 3.7 shows a simple example of graph pattern (gray nodes and edges). For instance, suppose that *GSpan* reports the operation *move* method followed by a *rename* method as a pattern that occurs repeatedly in our dataset. As we can notice, $G1$ contains this pattern (grey vertices), which refers to two distinct commits over time.

Figure 3.7: Example of over time graph patterns



$G2$ illustrates a second example, a pattern with three *extract* method operations, as shown in Figure 3.8. However, in this case, there are two possible situations: (i) the three *extract* operations were performed in a single commit, or (ii) the *extract* operations were performed in multiple commits over time.

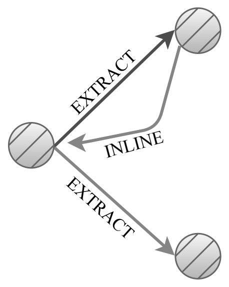
Figure 3.8: Example of possibly atomic graph patterns



Therefore, there are two categories of refactoring patterns: *possibly atomic* refactoring patterns and *over time* refactoring patterns. *Over time* patterns represent frequent refactorings performed in distinct commits (e.g., $G1$). In contrast, *possibly atomic* patterns can be detected in single or multiple commits. In other words, we cannot safely infer they include refactorings over time (e.g., $G2$).[8] Besides that, *GSpan* can report more than one pattern in the same subgraph. For instance, the algorithm can identify a pattern with two *extract* operations and a second pattern with three *extract* operations in $G2$.

Finally, it is also worth noting that refactorings graphs might have cycles, as in the example of Figure 3.9. In this subgraph, the *extract* refactorings were performed in the same commit. After that, in a second commit, one of the *extract* was reverted using an *inline* operation. If we do not take precaution, *GSpan* might detect the following pattern in this graph: *inline* → *extract* (assuming this pattern also happens in other subgraphs). However, this is a misleading pattern, since the *inline* happened before the *extract*. As the reader might have already concluded, misleading patterns are only possible when at least one edge is part of a cycle. For this reason, in order to answer RQ6, we implemented a script to identify and remove subgraphs with cycles from our dataset. As a result, we discarded 289 subgraphs in Java (3%) and 47 subgraphs in JavaScript (2%).

Figure 3.9: Example of an misleading refactoring graph pattern (*inline* → *extract*)



[8]GSpan output does not include information about the edges, such as commit or date. The algorithm only reports the occurrence of a pattern in a set of subgraphs. As a consequence, for graph patterns involving a single element (i.e., refactoring from the same source or refactoring to the same target), it is not possible to infer they include refactorings over time.

Since patterns can be detected in any number of commits (i.e., even in a single commit), in RQ6, we do not separate the dataset by the number of commits. *As a result, in this RQ, we assess 8,911 subgraphs in Java,*[9] *and 2,094 subgraphs in JavaScript.*[10] We fixed *support* = 13 (Java) and *support* = 8 (JavaScript). We set these thresholds due to execution time that would allow us to classify the retrieved graphs as patterns. The threshold for JavaScript is lower because the number of graphs we mined for this language is also lower.

Overview of Data Collection and Analysis

Table 3.4 presents an overview of the dataset we use to address the research questions. RQ_0 provides an introductory analysis, considering the frequency of multiple-commits operations in the subgraphs over time. From RQ_1 to RQ_5, we work on the same sample, which includes 1,525 refactoring subgraphs over time (1,198 Java and 327 JavaScript). In the case of RQ6, we consider all subgraphs without cycles to investigate refactoring graph patterns.

Table 3.4: Numbers of the quantitative study

Description	**RQs**	**All**	**Java**	**JS**
All refactoring operations	RQ_0	15,945	13,162	2,783
All refactoring subgraphs	RQ_0	11,341	9,200	2,141
Ref. subgraphs ($commit \geq 2$)	RQ_1 to RQ_5	1,525	1,198	327
Ref. subgraphs without cycles	RQ_6	11,005	8,911	2,094

3.2.2 Results

(RQ0) How Many Refactoring Operations Generate Subgraphs over Time?

In this first research question, we provide an overview of the refactoring operations in our sample. Specifically, we discuss how many refactorings result in subgraphs over time. As presented in Table 3.5, for Java, 29.3% of the operations are part of a refactoring subgraph over time (3,853 occurrences).

In the case of JavaScript, this rate is 32.4% (902 occurrences), as shown in Table 3.6. Interestingly, in three projects, more than 50% of the detected refactorings correspond to edges of subgraphs overtime: Carbon (53.7%), Request (60.2%), and Glide (56%).

[9] All 9,200 subgraphs presented in Table 3.2 minus the 289 subgraphs with cycles.
[10] All 2,141 subgraphs presented in Table 3.3 minus the 47 subgraphs with cycles.

Table 3.5: Frequency of refactoring operations in subgraphs (Java)

Project	**Refactoring Operations**				
	All	**Atomic**	%	**Over time**	%
Elasticsearch	2,969	2,394	80.6	575	19.4
RxJava	1,421	1,235	86.9	186	13.1
Square Okhttp	1,147	694	60.5	453	39.5
Square Retrofit	249	164	65.9	85	34.1
Spring Framework	4,640	3,071	66.2	1,569	33.8
Apache Dubbo	596	489	82.0	107	18.0
MPAndroidChart	720	423	58.8	297	41.3
Glide	734	323	44.0	411	56.0
Lottie Android	288	209	72.6	79	27.4
Facebook Fresco	398	307	77.1	91	22.9
All	13,162	9,309	70.7	3,853	29.3

Table 3.6: Frequency of refactoring operations in subgraphs (JavaScript)

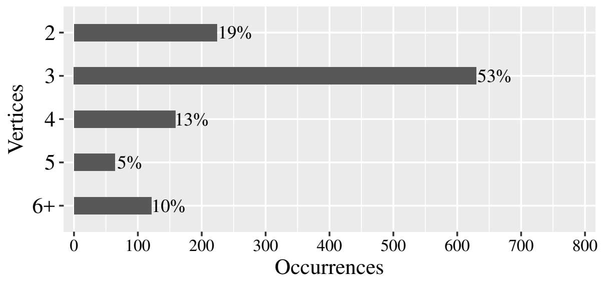
Project	**Refactoring Subgraphs**				
	All	**Atomic**	%	**Over time**	%
Vue	394	221	56.1	173	43.9
React	1,029	759	73.8	270	26.2
Parcel	130	99	76.2	31	23.8
Hexo	264	178	67.4	86	32.6
Leaflet	376	216	57.4	160	42.6
Quill	255	206	80.8	49	19.2
Request	108	43	39.8	65	60.2
Nylas Mail	88	73	83.0	15	17.0
Select2	98	67	68.4	31	31.6
Carbon	41	19	46.3	22	53.7
All	2,783	1,881	67.6	902	32.4

Summary of RQ0: In both languages, Java and JavaScript, about 30% of the refactoring operations are part of a refactoring subgraph over time.

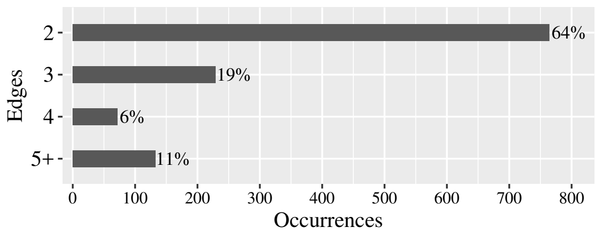
(RQ1) What is the Size of Refactoring Subgraphs?

As presented in Figure 3.10, in Java, most refactoring subgraphs have three vertices (630 occurrences, 53%). The other recurrent cases comprise subgraphs with two (19%) or four vertices (13%). Square Okhttp holds the largest subgraph regarding the number of vertices (57), which are most related to *inline* operations. Concerning the number of edges, most subgraphs have two (67%) or three edges (16%). MPAndroidChart has the largest subgraph in terms of edges. It has 61 edges, most representing *extract and move* operations. Therefore, most subgraphs contain few methods (vertices) and refactoring operations (edges).

Figure 3.10: Size of refactoring subgraphs (Java)



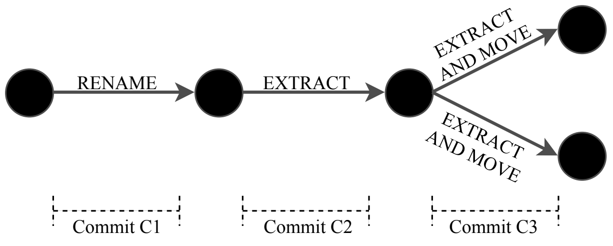
(a) Number of vertices



(b) Number of edges

Figure 3.11 shows a real-world example of a refactoring subgraph from MPAndroid-Chart, which includes three distinct refactoring operations. In the first commit $C1$, a developer renamed method *drawYLegend()* to $drawYLabels()$.[11] In the subsequent commit performed 13 days later, the same developer extracted a new method from $drawYLabels()$ at commit C2.[12] Two days after the second operation, in commit C3, he made new extractions from $drawYLabels()$ to another class, creating a subgraph with five vertices and four edges.[13]

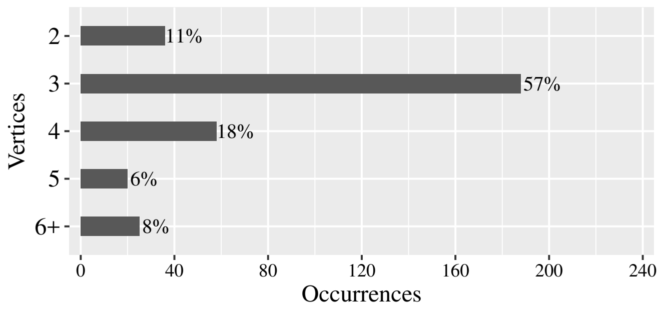
Figure 3.11: Example of a refactoring subgraph from MPAndroidChart (Java)



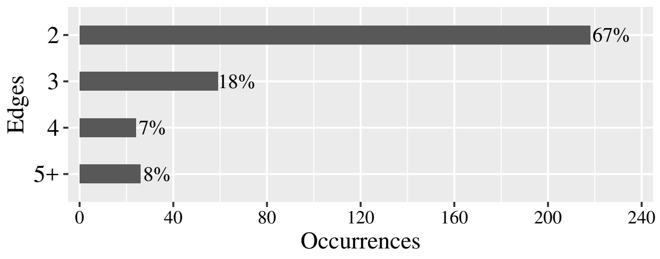
In the case of JavaScript, most subgraphs also have three vertices (57%), as shown in Figure 3.12. Other common cases refer to subgraphs with two (11%) or four vertices (18%). Regarding the number of edges, the subgraphs also are small, 92% of them involve up to four edges.

[11]https://github.com/PhilJay/MPAndroidChart/commit/13104b26
[12]https://github.com/PhilJay/MPAndroidChart/commit/063c4bb0
[13]https://github.com/PhilJay/MPAndroidChart/commit/d930ac23

Figure 3.12: Size of refactoring subgraphs (JavaScript)



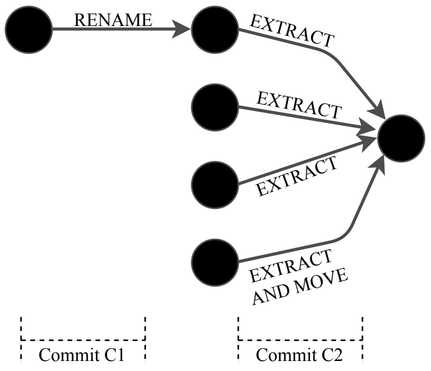
(a) Number of vertices



(b) Number of edges

Figure 3.13 presents an example of a refactoring subgraph from Quill, which includes five edges and three distinct refactoring operations. In commit C1, a developer renamed function *formatCursor* to *format*.[14] Seven months later, in commit C2, the same developer made four extract operations to function *isEnable*, aiming the removal of a single duplicated line.[15]

Figure 3.13: Example of a refactoring subgraph from Quill (JavaScript)



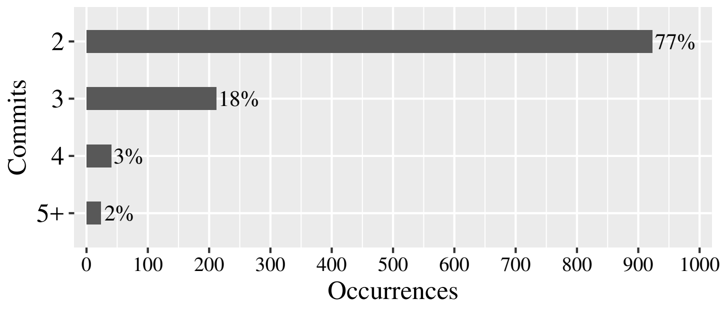
[14]`https://github.com/quilljs/quill/commit/aee9b867`
[15]`https://github.com/quilljs/quill/commit/e1d76d9f`

Summary of RQ1: In both languages, most refactoring subgraphs are small. Among 1,198 Java samples, most cases comprise subgraphs with the number of vertices ranging from two to four (85%) and two or three edges (83%). JavaScript reveals a similar result, most refactoring subgraphs have up to four vertices (86%) and three edges (85%). However, the presence of large subgraphs is not negligible.

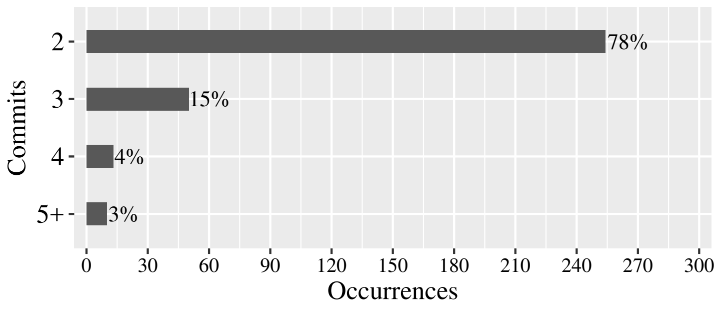
(RQ2) How Many Commits are Represented in Refactoring Subgraphs?

In this second question, we investigate the number of commits per subgraph. As presented in Figure 3.14, most cases include subgraphs with two or three commits. In Java, 95% of subgraphs (1,135 occurrences) are created from up to three commits. The largest subgraph in terms of commits is again from Square Okhttp (18 commits). Similarly, in JavaScript, 93% of subgraphs (304 occurrences) also comprise two or three commits.

Figure 3.14: Number of commits by refactoring subgraph



(a) Java

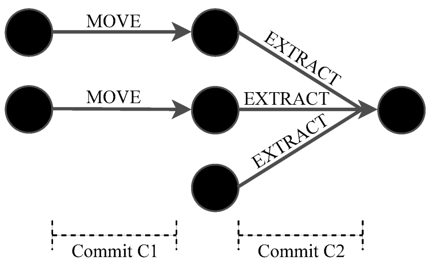


(b) JavaScript

Figure 3.15 shows an example from Elasticsearh. In commit C1, a developer moved two methods from class *SocketSelector* to *NioSelector*.[16] After approximately three months, in commit C2, a second developer extracted duplicated code from three methods to a new method named *handleTask*(*Runnable*).[17] Among the source methods, two methods are the ones moved early. As a consequence, these two commits create a refactoring subgraph with six vertices and five edges.

[16] `https://github.com/elastic/elasticsearch/commit/9ee492a3f07`
[17] `https://github.com/elastic/elasticsearch/commit/11fe52ad767`

Figure 3.15: Example of a refactoring subgraph from Elasticsearch (Java)



Summary of RQ2: Most refactoring subgraphs are created from few commits, e.g., 95% of Java subgraphs and 93% of JavaScript subgraphs are created from at most three commits.

(RQ3) What is the Time Span of Refactoring Subgraphs?

To assess interval, we compute the number of days between the most recent and the oldest operation in a subgraph. Figure 3.16 presents the results for Java. Considering the median of the distributions, the youngest subgraphs are found in Lottie Android and RxJava, which are 3 and 3.4 days, respectively. On the other side, the oldest subgraphs are found in Glide (489.8 days), Spring Framework (121.9), and Fresco (167.8). The other systems have subgraphs with time span between 45.4 (Elasticsearch) and 84 days (MPAndroidChart). Regarding the maturity of the target systems, the youngest project is Lottie Android (3 years) while the oldest one is Elasticsearch (9 years).

Figure 3.16: Timespan of the refactoring subgraphs (Java)

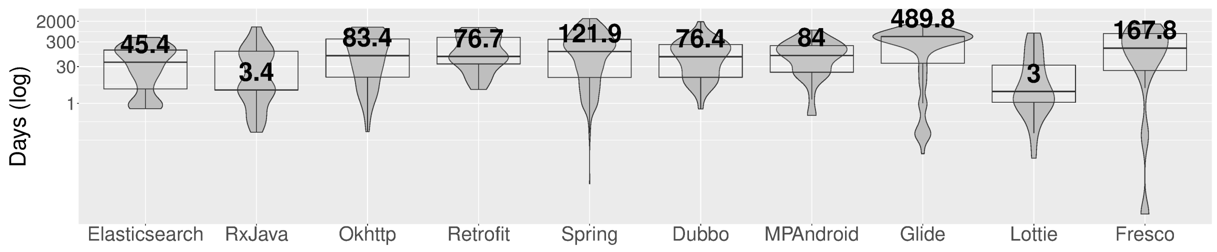


Figure 3.17 presents the distribution of period in JavaScript. Considering the median, the youngest subgraphs are from Carbon and Nylas Mail, with approximately 25 days. In contrast, there are also older subgraphs. For instance, in Hexo, the median is around four years. Thus, the time span of refactoring subgraphs also diverse in JavaScript.

Figure 3.18 shows an example of a subgraph describing refactorings performed in few days on Spring Framework. In commit C1, a developer renamed method $before(...)$ to $filterBefore(...)$.[18] After six days, the same developer reverted the operation in commit C2, renaming $filterBefore(...)$ to the original name.[19]

[18]https://github.com/spring-projects/spring-framework/commit/794693525f
[19]https://github.com/spring-projects/spring-framework/commit/91e96d8084

Figure 3.17: Timespan of the refactoring subgraphs (JavaScript)

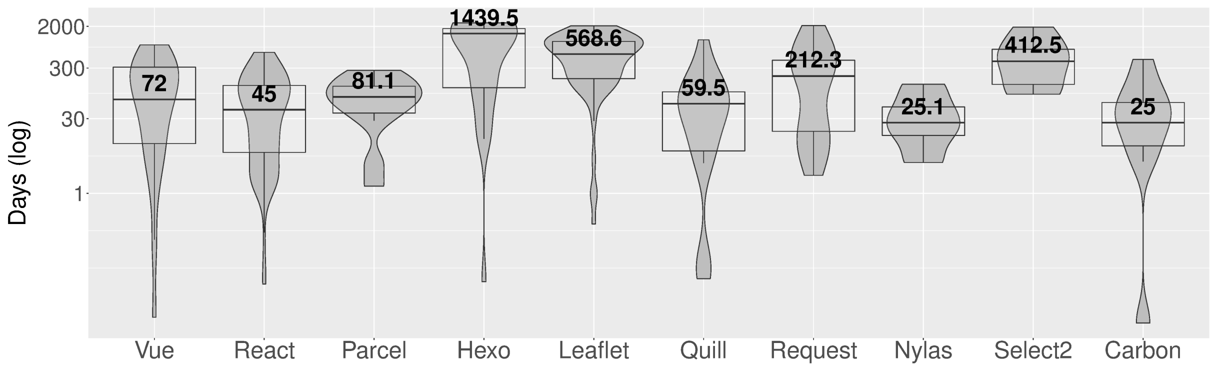


Figure 3.18: Example of a refactoring subgraph from Spring Framework (Java)

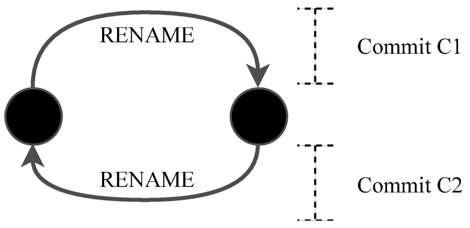
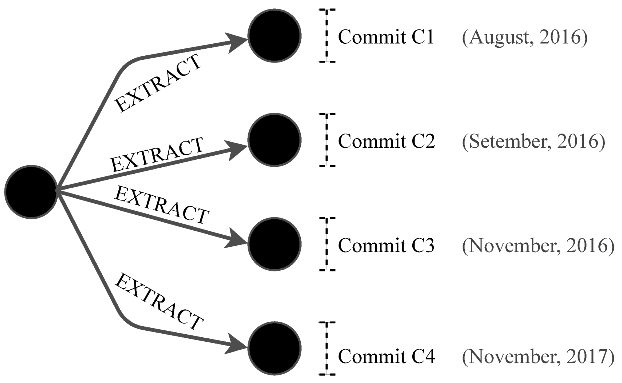


Figure 3.19 presents a second example, a subgraph with more than one year in Vue. The first operation occurs in August 2016, in commit C1, when a developer extracts a function from *createElm*.[20] The same developer performs more three operations during 15 months, extracting functions *createChildren*,[21] *createComponent*,[22] and *isUnknownElement*.[23]

Figure 3.19: Example of a refactoring subgraph from Vue (JavaScript)


EXTRACT
EXTRACT
EXTRACT
EXTRACT
Commit C1 (August, 2016)
Commit C2 (Setember, 2016)
Commit C3 (November, 2016)
Commit C4 (November, 2017)

[20]`https://github.com/vuejs/vue/commit/351aef3c`
[21]`https://github.com/vuejs/vue/commit/7a2c9867`
[22]`https://github.com/vuejs/vue/commit/de7764a3`
[23]`https://github.com/vuejs/vue/commit/df82aeb0`

Summary of RQ3: The period captured by the refactoring subgraphs is diverse. While some have few days, the majority of the subgraphs has weeks or even months. For example, for approximately 60% of the refactoring subgraphs in both languages (Java and JavaScript), the interval between the most recent and the oldest operation is more than one month.

(RQ4) Which are the Most Common Refactorings in Refactoring Subgraphs?

Table 3.7 presents the most common refactorings in Java. Most cases include *rename* (20%), *move* (18%), and *extract and move method* (17%). By constrast, we detected only 92 occurrences of *move and rename* operations. There are also few inheritance-based refactorings, i.e., *pull up* (369 occurrences) and *push down* (148 occurrences).

Table 3.7: Frequency of refactoring operations (Java)

Refactoring	**Occurrences**	**%**
Rename	752	20
Move	677	18
Extract and move	673	17
Extract	653	17
Inline	489	13
Pull up	369	10
Push down	148	4
Move and rename	92	2
All	3,853	100

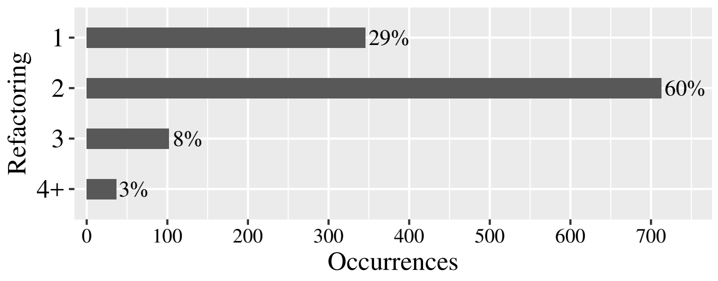
We also divided our sample of 1,198 subgraphs into two groups. The *homogeneous* group includes subgraphs with a single refactoring operation. In contrast, the *heterogeneous* group comprises subgraphs with at least two distinct refactoring operations. As presented in Table 3.8, around 28.9% of the subgraphs are homogeneous, while 71.1% are heterogeneous.

Table 3.8: Homogeneous vs heterogeneous refactoring subgraphs (Java)

Project	**Homogeneous**	**%**	**Heterogeneous**	**%**
Elasticsearch	63	35.2	116	64.8
RxJava	45	52.3	41	47.7
Square Okhttp	22	25.3	65	74.7
Square Retrofit	12	35.3	22	64.7
Spring Framework	140	27.9	361	72.1
Apache Dubbo	8	23.5	26	76.5
MPAndroidChart	16	21.9	57	78.1
Glide	29	20.0	116	80.0
Lottie Android	5	21.7	18	78.3
Facebook Fresco	6	16.7	30	83.3
All	346	28.9	852	71.1

The results per system follow a similar tendency. Most of the projects have more heterogeneous subgraphs than homogeneous ones; the sole exception is RxJava (52.3% vs 47.7%). In addition, as presented in Figure 3.20, heterogeneous subgraphs often include two distinct refactoring types (60%); in contrast, 8% have three and only 3% have four or more distinct refactoring types.

Figure 3.20: Number of distinct refactorings by subgraph (Java)



As shown in Table 3.9, in JavaScript, 76% of the refactorings refer to *extract*, *move*, and *rename* operations. There are also 88 occurrences of *internal move* operations, that is, the movement of nested functions into a single file. Among the 902 refactorings, 628 cases (69.6%) denote to heterogeneous subgraphs, which is the largest group, as presented in Table 3.10. Besides that, as shown in Figure 3.21, heterogenous subgraphs frequently include two distinct refactoring operations, following the same tendency of Java subgraphs.

Table 3.9: Frequency of refactoring operations (JavaScript)

Refactoring	**Occurrences**	**%**
Extract	238	26
Move	234	26
Rename	214	24
Internal move	88	10
Inline	53	6
Extract and move	29	3
Move and rename	36	4
Internal mode and rename	10	1
All	902	100

Figure 3.22 shows an example of a homogeneous subgraph from Facebook Fresco. In this case, the subgraph represents four *extract* operations performed over time. First, in commit C1, a developer extracted $fetchDecodedImage(...)$ from two methods into class $ImagePipeline$.[24] The next operations happened years later when a second developer made two new *extract* operations in commits C2[25] and C3[26].

[24] https://github.com/facebook/fresco/commit/02ef6e0f
[25] https://github.com/facebook/fresco/commit/b76f56ef
[26] https://github.com/facebook/fresco/commit/017c007b

Table 3.10: Homogeneous vs heterogeneous refactoring subgraphs (JavaScript)

Project	**Homogeneous**	**%**	**Heterogeneous**	**%**
Vue	21	33.3	42	66.7
React	44	41.5	62	58.5
Parcel	3	25.0	9	75.0
Hexo	4	14.3	24	85.7
Leaflet	27	43.5	35	56.5
Quill	3	15.0	17	85.0
Request	10	62.5	6	37.5
Nylas Mail	1	20.0	4	80.0
Select2	2	33.3	4	66.7
Carbon	3	33.3	6	66.7
All	118	36.1	209	63.9

Figure 3.21: Number of distinct refactorings by subgraph (JavaScript)

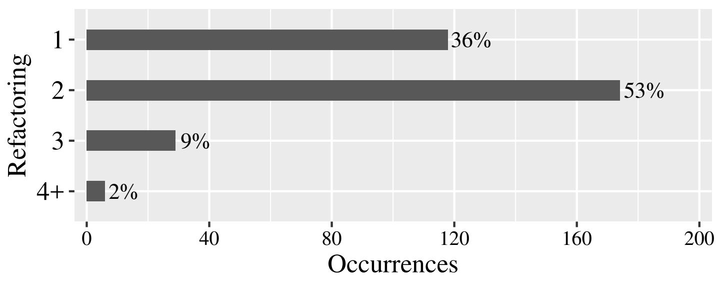
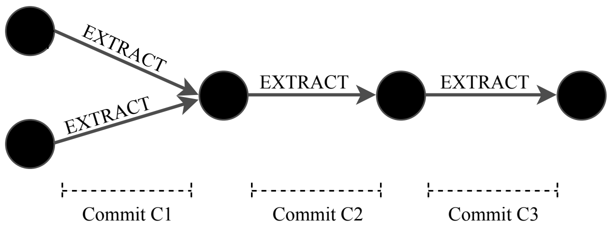


Figure 3.22: Example of a homogeneous refactoring subgraph from Facebook Fresco (Java)

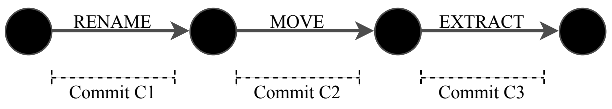


As a second example, we present a heterogenous subgraph from Parcel in Figure 3.23. In this case, a single developer performed three distinct operations in nine months by renaming function *resolveModule* to *resolveAsset*,[27] moving it to another file,[28] and extracting function *getLoadedAsset*.[29]

Summary of RQ4: Most refactoring subgraphs are heterogeneous, e.g., 71.1% of Java subgraphs and 63.9% of JavaScript subgraphs include more than one refactoring type.

[27] `https://github.com/parcel-bundler/parcel/commit/38d4a830`
[28] `https://github.com/parcel-bundler/parcel/commit/e4cee192`
[29] `https://github.com/parcel-bundler/parcel/commit/dd3ea464`

Figure 3.23: Example of a heterogenous refactoring subgraph from Parcel (JavaScript)



(RQ5) Are Refactoring Subgraphs Created by the Same or by Multiple Developers?

In this question, we separate the refactoring subgraphs into two groups. The first group includes subgraphs with refactoring operations performed by a single developer. The second category is the opposite; it holds subgraphs by multiple developers. As presented in Table 3.11, in Java, most subgraphs have a single author (61.4%). It is also possible to notice a similar tendency in JavaScript, i.e., 203 subgraphs (62.1%) include refactoring operations performed by a sole developer, as shown in Table 3.12.

Table 3.11: Developers by refactoring graphs (Java)

Project	**Single dev.**	**%**	**Multiple devs.**	**%**
Elasticsearch	67	37.4	112	62.6
RxJava	77	89.5	9	10.5
Square Okhttp	32	36.8	55	63.2
Square Retrofit	14	41.2	20	58.8
Spring Framework	309	61.7	192	38.3
Apache Dubbo	20	58.8	14	41.2
MPAndroidChart	70	95.9	3	4.1
Glide	125	86.2	20	13.8
Lottie Android	11	47.8	12	52.2
Facebook Fresco	11	30.6	25	69.4
All	736	61.4	462	38.6

Table 3.12: Developers by refactoring graphs (JavaScript)

Project	**Single dev.**	**%**	**Multiple devs.**	**%**
Vue	41	65.1	22	34.9
React	55	51.9	51	48.1
Parcel	9	75.0	3	25.0
Hexo	10	35.7	18	64.3
Leaflet	56	90.3	6	9.7
Quill	20	100.0	0	0.0
Request	4	25.0	12	75.0
Nylas Mail	1	20.0	4	80.0
Select2	3	50.0	3	50.0
Carbon	4	44.4	5	55.6
All	203	62.1	124	37.9

Figure 3.24 presents an example of a refactoring subgraph from Square Okhttp. First, in commit C1, developer D1 renamed three methods from class *OkHttpClient*.[30] Basically, the developer removed the prefix *set* from their names. After 10 months, a second developer D2 removed duplicated code from these methods, extracting method *checkDuration*(...).[31] Then, after seven months, D2 moved this method to a new class named *Util*, in commit C3.[32] As a result, these two developers are responsible for a refactoring subgraph with eight vertices and seven edges. Figure 3.25 shows an opposite scenario, a subgraph from Facebook React, which was created by a single developer. After performing five *inline* operations,[33] the developer renamed a function, adding the prefix *deprecated*.[34]

Figure 3.24: Example of a refactoring subgraph created by multiple developers in Square Okhttp (Java)

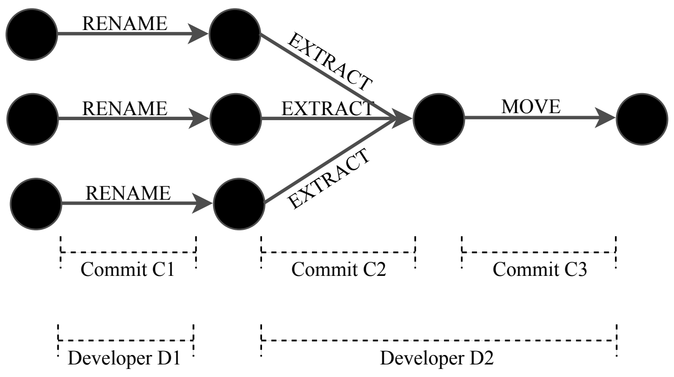
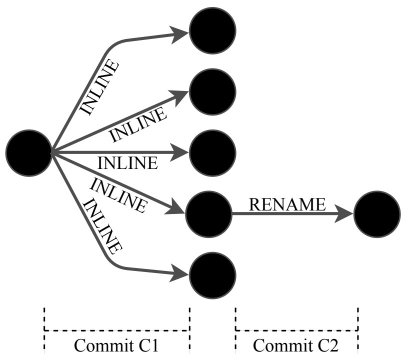


Figure 3.25: Example of a refactoring subgraph created by a single developer in Facebook React (JavaScript)



[30] https://github.com/square/okhttp/commit/daf2ec6b9
[31] https://github.com/square/okhttp/commit/c5a26fefd
[32] https://github.com/square/okhttp/commit/a32b1044a
[33] https://github.com/facebook/react/commit/50988911
[34] https://github.com/facebook/react/commit/9fe10312

Summary of RQ5: Most refactoring subgraphs are created by a single developer, e.g., only 38.6% of Java subgraphs and 37.9% of JavaScript subgraphs have multiple developers.

(RQ6) What are the Most Common Refactoring Subgraphs?

In this last research question, we mine frequent refactoring patterns. Specifically, we search for patterns that occur frequently in our dataset.

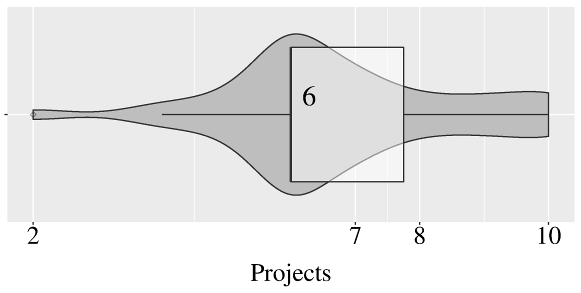
As presented in Table 3.13, in Java, we detect a total of 38 patterns using *GSpan* [102]. Most cases refer to *over time* patterns (60.5%, 23 occurrences), i.e., patterns that happen over multiple commits. In contrast, 15 patterns (39.5%) refer to *possibly atomic* patterns, that is, they can happen in single or multiple commits.

Table 3.13: Refactoring patterns

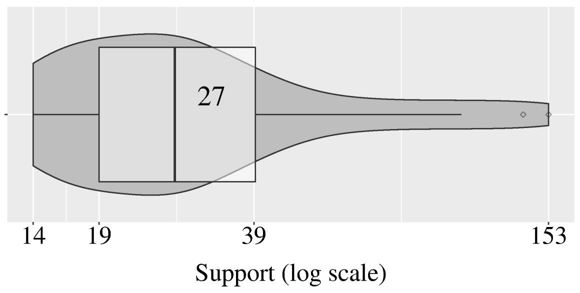
Vertices	**Java Patterns**			**JavaScript Patterns**		
	Over Time	**P. Atomic**	**All**	**Over Time**	**P. Atomic**	**All**
3	23	10	33	11	3	14
4	0	4	4	0	1	1
5	0	1	1	0	0	0
All	23	15	38	11	4	15

Figure 3.26 shows the distribution of the 38 patterns by the number of distinct projects and their support in Java. Interestingly, four patterns appear in all studied systems. Furthermore, 75% of the patterns occur in up to eight projects, and support values range from 14 to 153. In JavaScript, *GSpan* reports 15 patterns, 11 of then in the *over time* category (73%). Figure 3.27 presents the distribution of the detected patterns. The support median is 18, varying from 8 to 50.

Figure 3.26: Patterns distribution (Java)



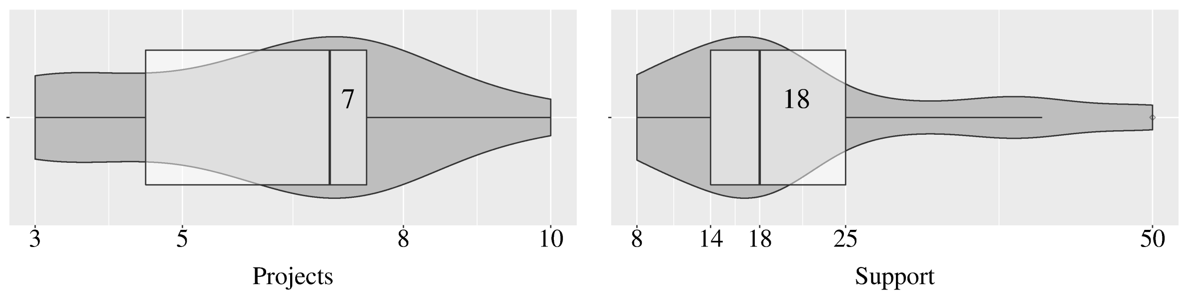
(a) Distinct projects by pattern



(b) Support by pattern

In the remainder of the section, we provide an analysis of refactoring patterns considering their number of vertices. As shown in Table 3.13, this number ranges from three to five vertices.

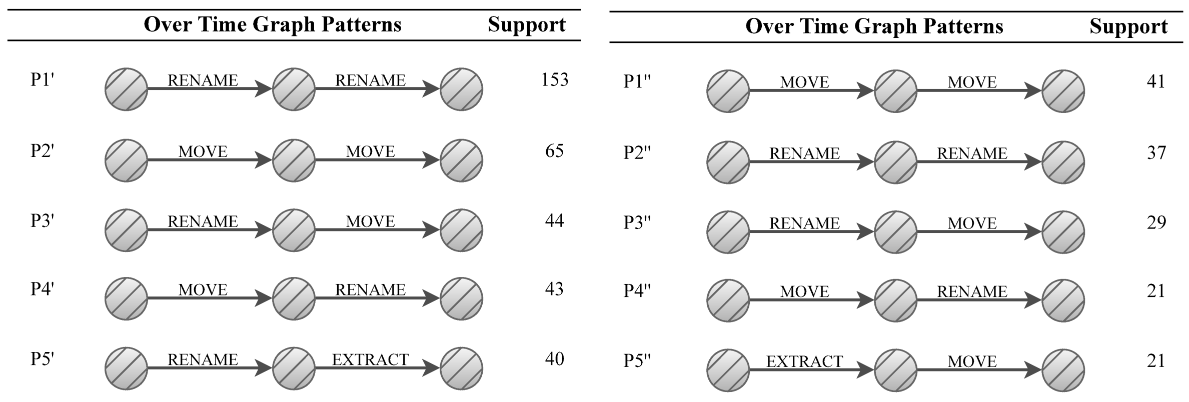
Figure 3.27: Patterns distribution (JavaScript)



(a) Distinct projects by pattern (b) Support by pattern

Refactoring graph patterns with three vertices: As we can observe in Table 3.13, in Java, all *over time* patterns have three vertices. Figure 3.28 shows the top-5 over time patterns in terms of support. Interestingly, the most recurrent patterns are homogeneous, that is, they refer to successive rename operations ($P1'$, 153 occurrences) and move operations ($P2'$, 65 occurrences). In fact, $P1'$ appears in all studied Java systems.

Figure 3.28: Top-5 over time graph patterns



	Over Time Graph Patterns	Support
P1'	RENAME → RENAME	153
P2'	MOVE → MOVE	65
P3'	RENAME → MOVE	44
P4'	MOVE → RENAME	43
P5'	RENAME → EXTRACT	40

	Over Time Graph Patterns	Support
P1"	MOVE → MOVE	41
P2"	RENAME → RENAME	37
P3"	RENAME → MOVE	29
P4"	MOVE → RENAME	21
P5"	EXTRACT → MOVE	21

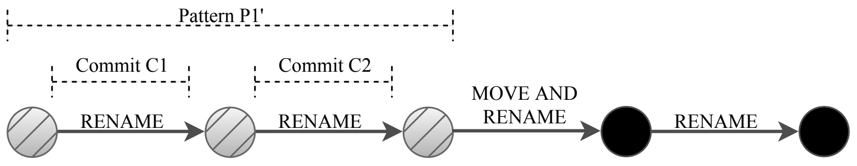
(a) Java (b) JavaScript

Figure 3.29 presents a subgraph from Glide with pattern $P1'$. A single developer performed the operations that represent the over time pattern in commits $C1$ and $C2$. First, he renamed *buildStreamOpener* to *buildStreamLoader*.[35] The developer repeated the same operation ten days later, replacing the prefix *build* by *get* in the method' name.[36]

In the case of JavaScript, support values are lower due to the sample size. However, the results show a similar tendency. All *over time* patterns have three vertices, as shown in Table 3.13. Besides, as presented in Figure 3.28, the top-2 patterns are homogeneous.

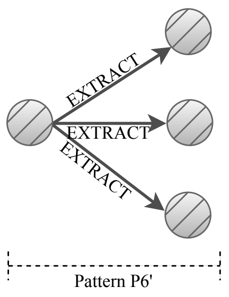
[35]https://github.com/bumptech/glide/commit/6bbe4343c
[36]https://github.com/bumptech/glide/commit/c572847b4

Figure 3.29: Example of a refactoring graph pattern from Glide (Java, 153 occurrences)



Refactoring graph patterns with four vertices: In both languages, all patterns with four vertices belong to the *possibly atomic* group. Figure 3.30 presents an example from Spring Framework. This graph describes multiple extract operation from method *processConstraintViolations*(...) to three methods.[37] This pattern occurs in 19 subgraphs in our dataset.

Figure 3.30: Example of a possibly atomic graph pattern from Spring Framework (Java, 19 occurrences)



Refactoring graph patterns with five vertices: In Java, the sole graph pattern occurs in 16 subgraphs and it includes four *inline* operations. Figure 3.31 shows a refactoring subgraph from RxJava with this pattern ($P7'$). In this subgraph, the *inline* operations involve the removal of method *threadPoolForComputation*, and replacement of the respective calls in six methods.[38] There are no occurrences of patterns with five vertices in JavaScript.

Summary of RQ6: In Java, the top-3 over time patterns are *rename* → *rename* (153 occurrences), *move* → *move* (65), and *rename* → *move* (44). In JavaScript, the top-3 over time patterns are *move* → *move* (41 occurrences), *rename* → *rename* (37), and *rename* → *move* (29).

[37] `https://github.com/spring-projects/spring-framework/commit/c43acd7675`
[38] `https://github.com/ReactiveX/RxJava/commit/320495fde`

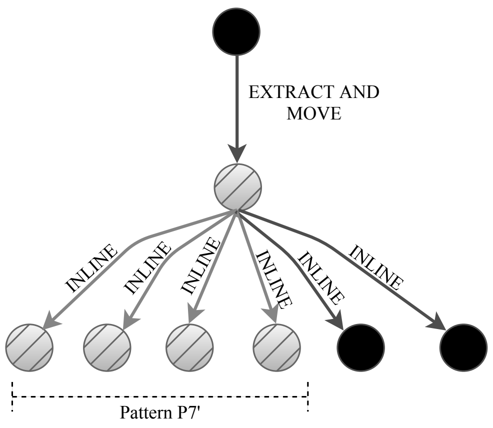


Figure 3.31: Example of a refactoring graph pattern from RxJava (Java, 16 occurrences)

3.3 Qualitative Study

As we reported in Section 3.2.2, most subgraphs are small in terms of their number of vertices, edges, and commits. For this reason, we showed small examples when discussing our quantitative RQ results. However, we also found subgraphs describing major refactoring operations. Therefore, the *goal* of this second study is to qualitatively analyze such subgraphs, with the *purpose* of investigating the motivation behind large refactoring operations performed over time. Specifically, we conducted a survey with the developers responsible for these refactorings. The *context* of the study consists of nine developers' feedback about 66 refactoring operations from eight subgraphs. These subgraphs represent the top-1% largest graphs in our dataset, by number of vertices.

3.3.1 Survey Design

Selecting Refactoring Subgraphs

We started by selecting the top-1% subgraphs by the number of vertices per programming language. In this way, for Java, we picked subgraphs with at least seven vertices, resulting in 132 instances. In the case of JavaScript, the top-1% refer to 27 subgraphs with at least six vertices. For both languages, we ordered the subgraphs by the number of vertices and we executed the following steps for each one:

1. We identified the authors of the commits associated with the subgraph. If one of the developers selected in this step was previously contacted, we also discarded her. Our

goal is to avoid sending more than one email per developer, reducing the perception of our survey as spam.

2. In this last step, we manually inspected the selected subgraphs to confirm whether the edges and vertices refer to true positives operations. As a result, we cleaned the subgraphs by removing false positive edges. Lastly, after those filtering steps, we contacted the authors.

We manually inspected 50 subgraphs (33 in Java and 17 in JavaScript), comprising 16 distinct projects.[39] In Java, the 33 subgraphs refer to 557 refactorings, which were detected by RefDiff in 120 commits, as shown in Table 3.14. Overall, the tool presents a high precision: 486 out of 557 (87%) refactorings are true positives. For instance, the precision for *extract and move* method is 93%, which is the most frequent refactoring operation (243 occurrences).

Table 3.14: Precision (Java)

Refactoring	**#**	**TP**	**FP**	**Prec.**	**Commit**	**Proj.**	**Subgraphs**
Extract and move	243	226	17	0.93	43	7	22
Inline	117	81	36	0.69	21	5	12
Extract	90	80	10	0.89	31	5	18
Push down	38	30	8	0.79	6	4	6
Move	24	24	0	1.00	15	7	12
Rename	23	23	0	1.00	15	6	9
Pull up	19	19	0	1.00	3	2	3
Move and rename	3	3	0	1.00	3	2	2
All	557	486	71	0.87	120	8	33

We followed the same steps in JavaScript by inspecting 133 refactoring operations in 60 distinct commits, as presented in Table 3.15. We notice that the overall precision is also high (93%). For instance, the most common refactoring operation is *extract* function (79 occurrences), whose the precision is 97%.

Contacting Developers

From July to August 2020, we sent emails to 62 developers asking for the motivations behind the refactoring subgraphs (see the template in Figure 3.32). In the emails, we added a short description of our research goals and a screenshot of the subgraph they are responsible for. In this case, the developer may be responsible for the whole subgraph or a piece of it. We also implemented a web app to navigate the graph structures, i.e., by using this app, our survey participants could check the vertices names, edges, and commits. Therefore, we included a link to the surveyed subgraphs in the survey message, as in the following example from Elasticsearch:

[39]`https://github.com/alinebrito/refactoring-graphs-thesis`

Table 3.15: Precision (JavaScript)

Refactoring	**#**	**TP**	**FP**	**Prec.**	**Commit**	**Proj.**	**Subgraphs**
Extract	79	77	2	0.97	36	8	16
Move	19	19	0	1.00	9	5	7
Internal move	9	9	0	1.00	3	3	3
Rename	9	9	0	1.00	8	6	8
Extract and move	11	4	7	0.36	4	3	3
Move and rename	4	4	0	1.00	3	2	3
Inline	1	1	0	1.00	1	1	1
Internal move and rename	1	1	0	1.00	1	1	1
All	133	124	9	0.93	60	8	17

`https://refactoring-graph.github.io/#/elastic/elasticsearch/713`

Figure 3.32: Email sent to the authors of refactoring subgraphs

Dear [developer name],

I'm a Ph.D. student at UFMG, Brazil, investigating large refactoring operations.

I'm using a graph structure to represent such refactorings (the vertices are methods and the edges are refactoring operations performed on them).

I found that you performed the following "big refactoring" on [repository/project]:

[screenshot of the subgraph]

To better navigate in this graph (e.g. see the vertices names), you can also check:
https://refactoring-graph.github.io/#/repository/project/subgraphID

Could you please describe **why these refactorings were performed**?

Table 3.16 summarizes the numbers and statistics about this qualitative study, as previously described in this section. We received nine answers, which represents a response ratio of 15%. Each of them corresponds to the developer's motivation to perform a set of refactorings. In a single case, the developer did not remember the motivation to perform the refactorings because it involved old commits. Overall, the answers are from relevant open-source developers. For example, we received replies from developers working in VMware, Elasticsearch, and Square. Besides, seven developers are among the top-10 contributors in the studied systems. In summary, our qualitative study contains answers from 66 refactorings instances represented in seven refactoring subgraphs. We used labels D1 to D9 to designate the developers and their responses and labels G1 to G7 to indicate the subgraphs.

Table 3.16: Numbers of the qualitative study

Large refactoring subgraphs sent to authors	50
Inspected refactoring operations (edges)	690
Emails sent to author	62
Received answers	9
Response ratio	15%

3.3.2 Survey Results

As presented in Table 3.17, the survey answers suggest two major reasons behind large refactoring subgraphs. In the following paragraphs, we explain and provide examples for each motivation.

Table 3.17: Reasons to perform large refactoring subgraphs

Motivation	**Subgraphs**	**Refactorings**
Fix bugs or improve existing features	5	35
Improve code design	2	30
Unclear	1	1

Improve code design: With 30 edges and two subgraphs, this category was inspired by a recent theme proposed in the literature [75]. Essentially, it groups large refactoring operations to improve maintainability or encapsulation. As examples, we have the following answers from two authors of the same subgraph, which is shown in Figure 3.33.[40] In the first answer, D_2 performed two refactoring operations by extracting a function and moving it to a distinct file. Similarly, D_3 also moved a function. In their answers, the developers emphasized their major motivation was to improve the code design:

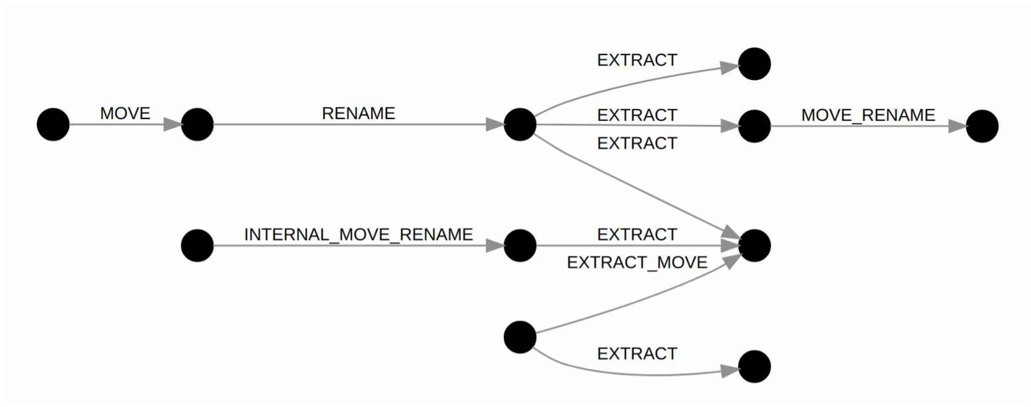
"Specifically in the case of [Function Name] all of the code was in a single file. The first step toward making it more maintainable is by reducing scope, also known as encapsulation. (...) I moved [Function Name] out, and a bunch of other functions into separate modules in order to reduce scope, or at least try to minimize it (...)" (D_2, 2 refactorings in subgraph G_1)

"It was a large file. It is easier to maintain by separating in several components (...)" (D_3, 1 refactoring in subgraph G_1)

Figure 3.34 shows a second example in this category.[41] In this case, the author of one *move* operation and 26 *extract and move* operations points that the major reason was to migrate parts of the code to the appropriate container:

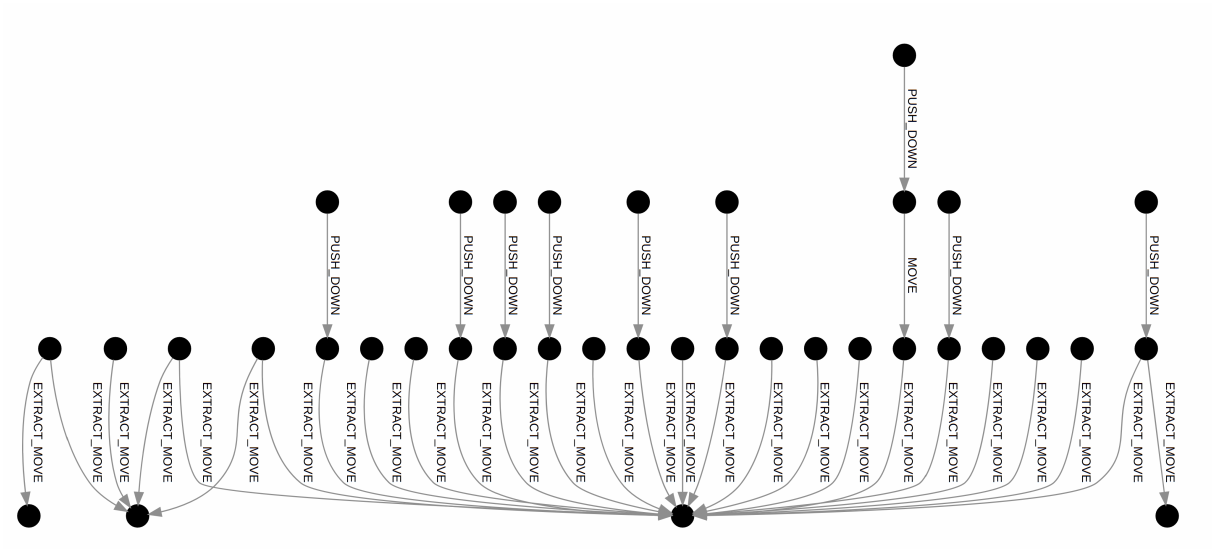
[40] `https://refactoring-graph.github.io/#/request/request/0`
[41] `https://refactoring-graph.github.io/#/square/okhttp/485`

Figure 3.33: Example of a large subgraph from Request (G_1, JavaScript)



"Most of the refactorings here move code that's logically related to also be physically related." (D_6, 27 refactorings in subgraph G_4)

Figure 3.34: Example of a large subgraph from Square Okhttp (G_4, Java)

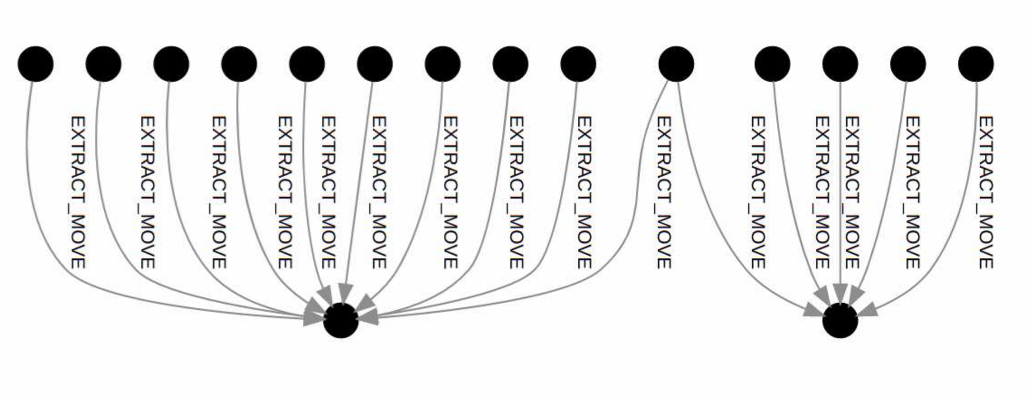


Fix bugs or improve existing features: In five answers (56%), developers essentially mention opportunistic refactorings performed during changes to fix bugs or improve features, which are also reported in a recent study [73]. This category includes 35 refactorings located in five distinct subgraphs. As a first example, we show an answer related to several *extract and move* operations performed to create two methods, as represented in the subgraph in Figure 3.35.[42] D_5 explains his motivation was to improve the usage of events subscription feature:

"I did those to make sure that empty/error cases use the right objects and call the right methods everywhere they are needed. In addition, they would now indicate in the original method that there are no extra actions intended to be performed on those code paths." (D_5, 15 refactorings in subgraph G_3)

[42]https://refactoring-graph.github.io/#/ReactiveX/RxJava/784

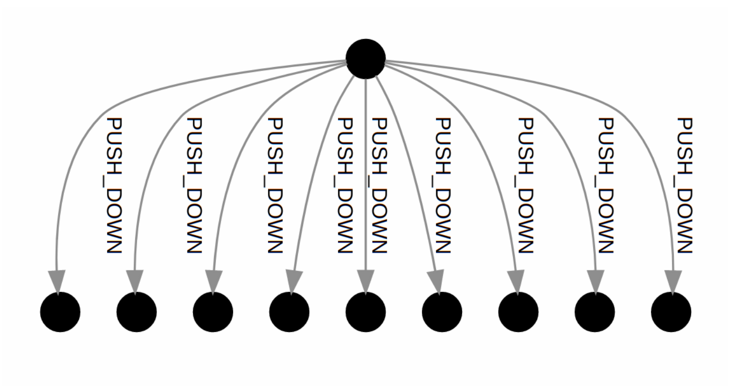
Figure 3.35: Example of a large subgraph from RxJava (G_3, Java)



D_8 also points to the maintainability of a feature by pushing down a method to nine subclasses, as presented in Figure 3.36.[43] In this example, the goal is to support a non-mutable communication option:

"We have a concept in [Project Name] used for reading/writing objects when forming requests/responses for inter-node communication. That concept originally depended on using default constructors, with mutable members (...) In order to allow non mutable state in these requests/responses, we changed this model (...) I found there were many layers at the top of the hierarchy of classes that were no longer needed (...) The change referenced here was to remove the [Method Name] from base classes that no longer contained any logic." (D_8, 9 refactorings in subgraph G_6)

Figure 3.36: Example of a large subgraph from Elasticsearch (G_6, Java)

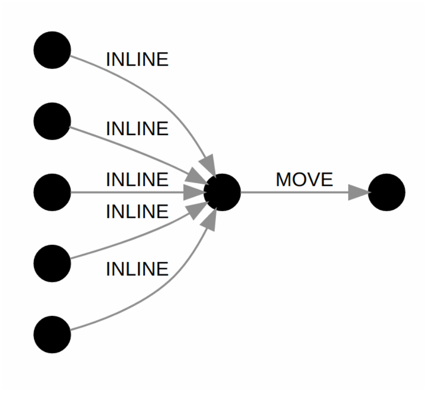


As a last example involving fixing an existing thread-related bug, we show D_7's answer. In this case, the developer performed the refactorings to provide a safe mode to instantiate a class, generating the subgraph in Figure 3.37:[44]

[43]https://refactoring-graph.github.io/#/elastic/elasticsearch/308
[44]https://refactoring-graph.github.io/#/spring-projects/spring-framework/2820

"We pushed everything from the front-facing API class (...) that enabled us to call the existing [Class Name] thread safe because each use of it would now create and use a new instance (...) Prior to the change if two threads had the same [Class Name] and called parse at the same time, I think it would get into a mess." (D_7, 6 refactorings in subgraph G_5)

Figure 3.37: Example of a large subgraph from Spring Framework (G_5, Java)



Finally, in two answers, the motivation is also related to fixing bugs:

"(...) I centralized some repeated code around timeouts and fixed a bug where it wasn't cleared properly." (D_1, 3 refactorings in subgraph G_1)

"I was doing closure elimination and memory leakage fix in the two refactoring (...)" (D_4, 2 refactorings in subgraph G_2)

3.4 Discussion and Implications

Refactoring over time & programming languages. In this chapter, we analyzed refactoring graphs in two different programming languages: JavaScript and Java. These languages have distinct styles. Java is a strongly-typed and object-oriented programming language, while JavaScript is an interpreted and dynamic language. Despite their distinct properties, our results regarding refactoring operations over time are similar in both languages, as summarized in Table 3.18. For example, in both languages, most subgraphs are small (RQ1) and heterogeneous (RQ4). On the other hand, there is a significant variation in the absolute number of detected refactoring subgraphs. We found 1,198 subgraphs over time in Java and 327 subgraphs in JavaScript. However, considering the relative rate, the results remain similar (13% in Java, 15% in JavaScript).

Detecting refactorings over time. Several tools and techniques are proposed in the literature to detect refactoring operations, such as Refactoring Crawler [33], RefFinder [55], Refactoring Miner [86, 94], and, more recently, RefDiff [85] and RefactoringMiner [95, 96]. In common, those approaches only detect *atomic* refactorings, i.e., operations that happen in a single commit and performed by a single developer. However, as presented in Section 3.2, there is a significant rate of refactoring operations spreading over multiple commits (RQ0). In contrast, our approach, refactoring graphs, focuses on the detection of refactorings over time, i.e., operations over multiple commits and performed by multiple developers. Moreover, differently from the *batch* refactoring [12, 26, 70], our approach is not constrained by the number of developers nor to a time window. Indeed, we found refactoring subgraphs with time span ranging from weeks to months (RQ3) and created by multiple developers (RQ5). *Therefore, we contribute to the refactoring literature with a novel approach to detect and explore refactoring operations in a broader perspective to complement existing tools and techniques.* In addition, these tools do not cluster refactoring operations performed in multiple steps. For example, suppose a developer extracted class *Foo* from class *Bar* in commit C_1. In this case, the tool used in this chapter detects an Extract Class, since the refactoring generates a new entity. However, if she keeps moving methods from *Bar* to *Foo* in the next commits, the tool does not group these operations. Instead, it reports them as isolated move operations. *Therefore, we also envision studies on new strategies to cluster or group related refactorings performed in multiple steps.* Besides, it would be interesting to evaluate the impact of such “missing” operations in the results and findings of previous empirical studies that relied on *atomic* refactoring detection tools [5, 12, 16, 49, 73, 89, 98].

Refactoring comprehension and improvement. When performing code review, developers often adopt diff tools to better understand code changes, and decide whether they will be accepted or not. In this process, developers may also look for defects and code improvement opportunities [9]. However, if the reviewed change is large and complex, this task becomes challenging [9]. To alleviate this issue, refactoring-aware code review tools were proposed [23, 39, 40, 45] to better understand changes mixed with refactorings. Refactoring graphs can contribute to handle this issue by providing navigability at method level. That is, a code reviewer may navigate back in a method to reason how a similar change was performed. For example, in Figure 3.24, a code reviewer may investigate whether all methods were properly renamed in the past, before accepting commit C3. *Thus, refactoring graphs can be integrated to code review tools to better support code understanding and improvement.*

Table 3.18: Summary of refactoring graphs properties

RQ	Description	Java	JavaScript
-	Refactoring subgraphs over time	1,198 subgraphs (13%)	327 subgraphs (15%)
-	Level	Method	Function
RQ_0	Refactoring operations over time	3,853 (29%) operations are part of subgraphs over time	902 (32%) operations are part of subgraphs over time
RQ_1	Size in vertices	Most subgraphs are small (median = 3)	Most subgraphs are small (median = 3)
RQ_1	Size in edges	Most subgraphs are small (median = 2)	Most subgraphs are small (median = 2)
RQ_2	Number of commits	Most subgraphs are created from at most three commits (1,135 occurrences, 95%)	Most subgraphs are created from at most three commits (304 occurrences, 93%)
RQ_3	Time span	64% have more than one month	67% have more than one month
RQ_4	Refactoring types	Most subgraphs represent *rename method* (20%), *move method* (18%), and *extract and move method* (17%)	Most subgraphs represent *extract function* (26%), *move function* (26%), and *rename function* (24%)
RQ_4	Homogeneity	Most subgraphs are heterogeneous (852 occurrences, 71%)	Most subgraphs are heterogeneous (209 occurrences, 64%)
RQ_5	Ownership	Most subgraphs are created by a single developer (736 occurrences, 61%)	Most subgraphs are created by a single developer (203 occurrences, 62%)
RQ_6	Refactoring Patterns	The top-3 over time patterns are *rename* → *rename* (153 occurrences), *move* → *move* (65), and *rename* → *move* (44)	The top-3 over time patterns are *move* → *move* (41 occurrences), *rename* → *rename* (37), and *rename* → *move* (29)

Detecting refactoring patterns and smells. In our qualitative study, we investigated subgraphs describing large refactoring operations (RQ1). As we can notice, these subgraphs may represent the improvement of pieces of code. For instance, Figure 3.34 shows a large subgraph from our dataset. Among the refactoring instances, there are 21 *extract method* operations, generating a single method with two lines of code. This method is represented as a node in the subgraph (in the bottom), which is the node with the highest *in-degree*, i.e., the highest number of edges coming to it. Therefore, it may indicate a pattern to move a specific duplicated code to an appropriate container. In addition, there is an interesting question in this context: *could the developer extract these two lines from another part of the project?* In other words, *should the graph have more edges?* In the same way, a high *out-degree* of a node, i.e., a high number of edges leaving it, can suggest an anomaly on a method. For example, Figure 3.19 shows a subgraph with four *extract* operations from a single method. In this case, it is probably a frequent behavior during a method evolution, since in RQ6, we identify refactoring graph patterns that are formed by three *extract* operations (Figure 3.30). However, a method which is decomposed several times over time (i.e., high *out-degree*) can reveal a code design problem. *Thus, refactoring graphs can foment the detection of refactoring anomalies over time and drive future research agenda on refactoring patterns.*

Understanding and assessing software evolution. During software evolution, developers often perform refactoring operations. Consequently, the link between methods may be lost [49]. For example, if a method $a()$ is renamed to $b()$ and then extracted to $c()$, it becomes quite hard to trace $a()$ to $c()$, and vice versa. This has several implications to software evolution research, particularly on studies that assess multiple code versions, such as code authorship detection [7, 44, 67, 77, 91], code evolution visual supporting [41, 42], bug introducing change detection [27, 58, 78, 79, 104], to name a few. In practice, these studies often rely on tools provided by Git and SVN, such as `git blame` and `svn blame`, which show what revision and author last modified each line of a file. However, this process is sensitive to refactoring operations [7, 49]. As Git and SVN tools cannot track fine-grained refactoring operations, particularly at method level, these approaches may miss relevant data. For instance, in the aforementioned example, it would be not possible to detect that method $c()$ was originated in method $a()$. Consequently, we would be not able to find the real creator of method $c()$ nor the developer who introduced a bug on $c()$. As shown in Section 3.2, most subgraphs are small (RQ1) and have few commits (RQ2), suggesting that the whole history of the elements may contain a few ruptures due to refactoring. However, it still may reflect a significant impact on the retrieval of source code changes [43, 49]. *With refactoring graphs, we are able to resolve method names over time, thus, software evolution studies can benefit as more precise tools can be created on the top.*

3.5 Threats to Validity

Generalization of the results. We analyzed 1,525 refactoring subgraphs from 20 popular and open-source Java and JavaScript systems. Therefore, our dataset is built over credible and real-world software systems. Our qualitative study reinforces recent results about motivations to refactor a source code [73, 75, 86], which were reported in another contexts. Also, the motivations are based on answers from relevant contributors to the open-source community. Despite these observations, our findings—as usual in empirical software engineering—may not be directly generalized to other systems, particularly commercial, closed source, and the ones implemented in other languages than Java and JavaScript. Finally, we focus our study on eight refactorings at method level (Java) and eight refactorings at function level (JavaScript). Thus, other refactoring types can affect the size of subgraphs. We plan to extend this research to cover software systems implemented in other programming languages and refactorings at class level.

Adoption of RefDiff. We adopted RefDiff to detect refactoring operations because it is the sole refactoring detection tool that is multi-language, working for Java, JavaScript, C, and Go [22, 87]. It is also extensible to other programming languages. In this chapter, we concentrated on Java and JavaScript systems. Thus, as we planned to extend this research to cover other programming languages than Java, RefDiff was the proper solution. Besides, despite being multi-language, RefDiff accuracy is quite high. For example, in the current version [87], the authors provide an evaluation of the tool for three languages: Java (precision: 96.4%; recall: 80.4%), JavaScript (precision: 91%; recall: 88%), and C (precision: 88%; recall: 91%). The recent evaluation for Go reports 92% of precision and 80% of recall [22]. In our dataset, the tool also presents a high precision for Java (557 refactoring instances; precision: 87%) and JavaScript (133 refactoring instances; precision: 93%). Recently, Tsantalis *et al.* [95, 96] proposed the refactoring detection tool RefactoringMiner. In the current version [96], RefactoringMiner has a precision of 99.6% and recall of 94%, improving on RefDiff's overall accuracy. However, RefactoringMiner works only for Java projects. Finally, RefDiff detects refactorings using a generic data structure called Code Structure Tree (CST). The generation of this data structure for JavaScript relies on a simplified call graph due to the dynamic nature of the language. This might result in a higher rate of false negatives. However, the authors mention the tool "*works well even when the information encoded in the CST is not completely precise*" [87].

Building refactoring graphs. When creating the refactoring graphs, we cleaned up our data (i.e., vertices and edges) to keep only meaningful subgraphs. For instance, in Java, we removed constructor methods (vertices) from our analysis because they include mostly initialization settings, and do not have behavior as conventional methods. In JavaScript, we removed refactorings in anonymous functions, i.e., functions without a name, since it is necessary to generate the vertices in the refactoring subgraphs. We also removed some very specific cases of refactoring (edges) in which RefDiff reported operations in same element. However, these cases are not likely to affect our results because they only represent a fraction of the refactoring operations. For example, RefDiff detected 89% of the removed operations in anonymous functions in only two systems (Facebook React, 85 occurrences; Hexo, 82 occurrences). Finally, the refactoring subgraphs can include unintentional operations (e.g., reverted commits by automatic deployment systems). To mitigate this threat, we focus our study on the main branch evolution to avoid experimental or unstable versions. Additionally, our results can miss refactoring operations that have not been merged on the main branch. However, as mentioned in previous studies [49], this strategy provides a safe overview of the system, avoiding refactorings performed in experimental code. Also, the qualitative study confirmed the selected branches are active ones. For example, developers mentioned large refactoring operations to implement features or improve code design in commits from these branches.

Detection of developers. In RQ5, we investigate the number of developers per refactoring subgraphs. We used the email available on `git log` to distinguish the author of the commits. Thus, our results can include, for example, the same developer committing with different email addresses. But, we already found that most cases are subgraphs created by a single developer.

Large refactoring graphs motivations. In the qualitative study, we manually inspected the refactoring subgraphs. Although this inspection might be an error-prone task, it was carefully performed during about a month. Furthermore, we did not receive complaints from the survey participants about false positives that were not detected in this analysis. Our analysis is also publicly available.[45]

3.6 Final Remarks

In this chapter, we present refactoring graphs, an approach to assess refactoring operations over time. We analyzed 1,525 refactoring subgraphs from 20 popular systems and two programming languages, Java and JavaScript. We then investigate seven research questions to evaluate the following properties of refactoring graphs: operations over time, size, commits, time span, homogeneity, ownership, and patterns. In both languages, the results suggest a similar tendency. We summarize our findings as follows:

- Approximately 30% of refactoring operations are part of a refactoring subgraph over time.
- The majority of the refactoring subgraphs are small (four nodes and three edges). However, there also outliers with dozens of nodes and edges.
- Most refactoring subgraphs have up to three commits.
- Refactoring subgraphs span from few days to months.

[45]`https://github.com/alinebrito/refactoring-graphs-thesis`

- Refactoring graphs are often heterogeneous, that is, they are composed by several types of refactoring.

- Refactoring graphs are mostly created by a single developer.

In the last research question, we mine graph patterns in approximately 9k subgraphs in Java and 2k subgraphs in JavaScript. Our results point to recurring graph patterns over time formed by two edges (e.g., successive rename operations). As a complementary perspective, we also perform a qualitative study with large refactoring subgraphs from our dataset, i.e., subgraphs with several vertices and edges. We contacted the developers, asking for the motivation for their operations. Considering nine developers' answers, 66 refactoring instances, and seven subgraphs, our results suggest that large refactoring subgraphs are motivated by well-know maintenance activities, involving the improvement of code design, fixing bugs, or the improvement of features. However, it is also important to mention that a single graph may include multiple of such motivations.

Based on our findings, we provided further discussion and implications to our study. Particularly, (i) we discuss our contributions regarding refactoring tools as a novel approach to explore refactoring operations in a broader perspective; (ii) we argue that refactoring graphs can be integrated to code review tools to better support code comprehension; (iii) we claim that refactoring graphs can play a role on the detection of refactoring patterns and anomalies; and (iv) we state the importance of refactoring graphs to resolve method names and support software evolution studies.

Chapter 4

Refactoring Comprehension Using Refactoring Graphs

In the previous chapter, we conducted an empirical study to formalize and characterize refactoring graphs in relevant systems. Our ultimate goal was to show the frequency and main aspects of refactoring graphs in the wild. In the current chapter, we aim to present **a first application of refactoring graphs**. Specifically, we assume a professor' perspective inspecting refactoring tasks performed by 46 Computer Science students. In this scenario, we used refactoring graphs to assist the understanding and visualization of the refactoring operations. The students refactored a Video Store System, which is a canonical example proposed by Fowler [35] to introduce refactoring concepts.

This chapter is organized as follows. Section 4.1 introduces the relevance of conducting this study. Section 4.2 includes details about the Video Store System and Section 4.3 shows the representation of refactoring operations as graphs. Section 4.4 describes the design of our study, while Section 4.5 shows the results. We discuss the implications in Section 4.6, and Section 4.7 states threats to validity. Finally, we conclude the chapter in Section 4.8.

4.1 Motivation

A large number of studies investigate refactoring practices in the last 30 years [1]. Many of those studies focus on refactoring comprehension, for example, assessing motivations, benefits, impact, and challenges of refactoring operations [57, 75, 82]. There are also approaches and tools to understand source code affected by refactoring [23, 43, 87, 96]. However, typically, prior literature does not rely on abstractions to extract, visualize, and understand complex refactoring tasks performed over time, which is a relevant aspect for researchers and practitioners [4, 14].

Therefore, in this chapter, we explore *refactoring graphs* to visualize, interpret, and evaluate refactoring tasks performed over time. For this purpose, we rely on a canonical refactoring example: a Video Store System, proposed by Fowler [35]. This example is used by Fowler to present and discuss refactoring operations, practices, and benefits. Specifically, we invited 46 students from a Software Engineering undergraduate course to perform refactoring operations in the Video Store System under two distinct scenarios. The first one includes a list of *explicit* and well-defined guidelines, i.e., refactoring tasks with detail instructions about the piece of code that should be refactored and the expected operations. The second scenario comprises *flexible* and open guidelines, in which we asked the students to implement the Template Method design pattern [37].

After that, we generated *refactoring graphs* to describe the refactorings performed by each student. Particularly, our guidelines generate a large refactoring graph with 24 vertices and 26 edges.

To better understand the tasks performed by the students, we first rely on a graph-based metric. Specifically, for *explicit guidelines*, we used a similarity metric called *edit distance* to compare each student's graph with our ground truth [81]. This strategy allowed us to automate the analysis of the tasks performed by the students and to rapidly conclude whether they matched our proposed ground truth. It also allowed us to rapidly identify and analyze the divergences between the student's solution and the ground truth. In the case of *flexible guidelines*, we performed a visual inspection by navigating in the graph-based structure to verify, for example, refactoring operations, sequence of operations, and affected elements. Our key results are summarized as follows:

Explicit Guidelines: When following *explicit guidelines*, most students performed refactoring tasks successfully (93.5%). The few mistakes refer to refactorings performed in multiples commits.

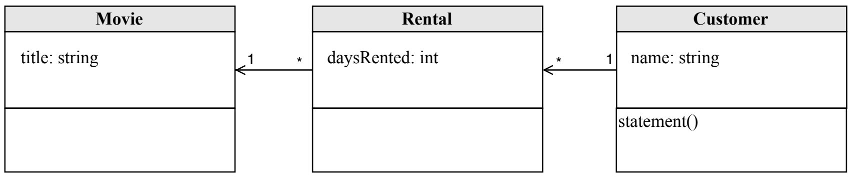
Flexible Guidelines: When following *flexible guidelines*, less students implemented the design pattern successfully (70%). Particularly, a significant part of the students faced difficulties identifying the appropriate refactorings operations.

Contributions: First, we show that *refactoring graphs* can be used to understand, evaluate, and visualize refactoring tasks. In some cases, this analysis can be automated using standard graph metrics, such as *edit distance*. Second, we complement research on refactoring practices, mainly on the educational side, by exploring the student's understanding of refactoring.

4.2 Video Store System

In the study described in this chapter, the students followed the steps proposed by Fowler [35] to refactor a well-known example designed to illustrate the benefits and the mechanisms of refactoring: a system to calculate and print clients' charges at a Video Store.[1] Basically, the system prints the movies rented by a client with prices and the number of rented days. The price depends on the movie's category (i.e., regular, children, and new releases) and the number of rented days. Figure 4.1 presents the initial class diagram of the system, which contains three classes:

Figure 4.1: Class diagram of video store system (*initial version*)

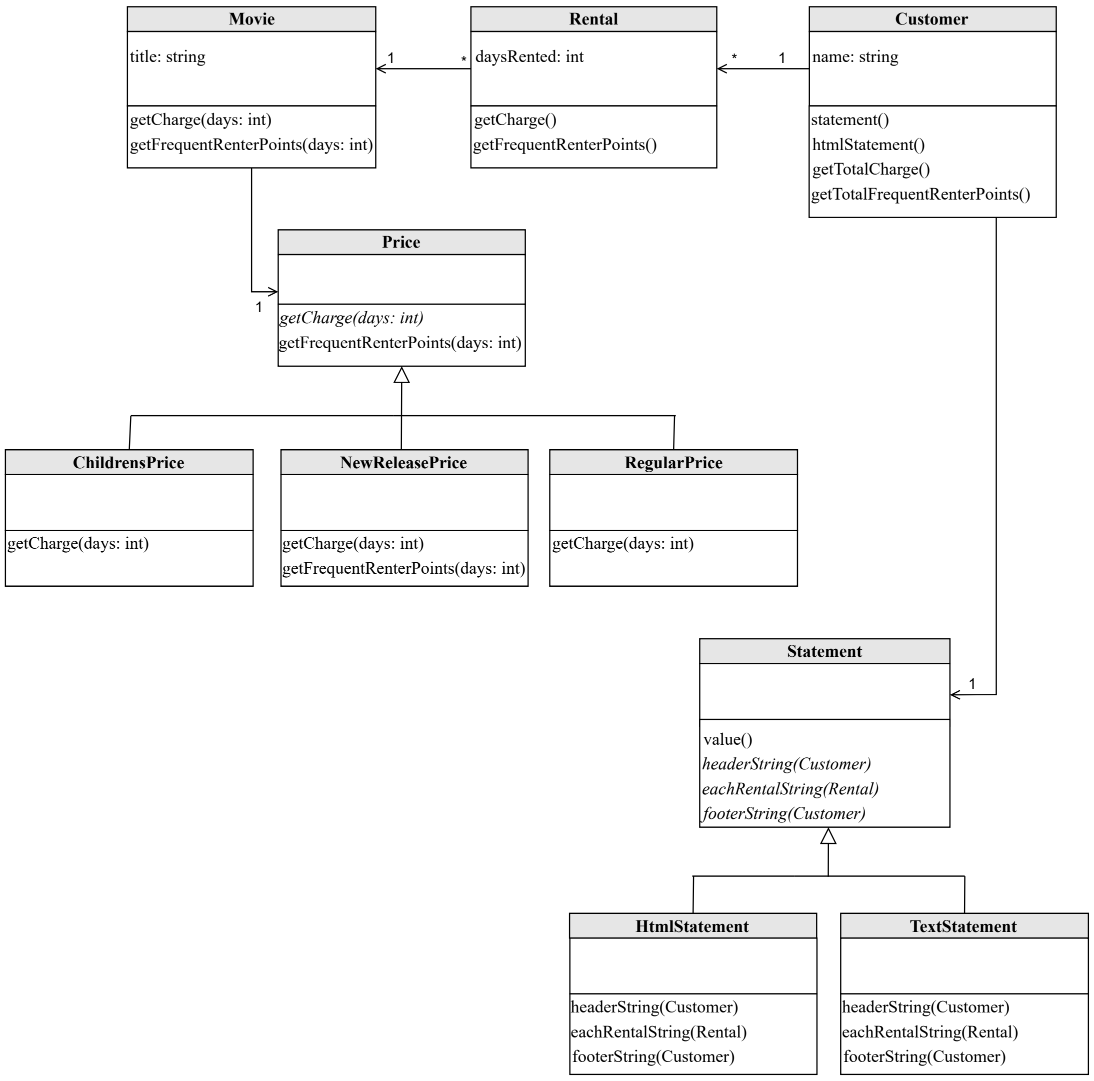


- *Movie*: class that represents a movie with the respective data, i.e., title, price, and category.
- *Rental*: class that represents a client renting a movie. It includes information about the movie and the number of rented days.
- *Customer*: class to represent the store' clients. It comprises information as the client's name and a list of rented movies. There is also a method, named `statement()`, which prints the statement in text format.

Figure 4.2 presents the final class diagram of the system after applying the refactoring operations. The new code design is a consequence of several refactoring operations to improve existing code and incorporate a new feature. Among these refactorings, most cases aim to decompose the large method `statement()` by extracting and moving distinct pieces of the code. Other refactorings generate the class `Price` and their subclasses in seven main steps, as proposed by the State design pattern. In this case, the goal is to remove a conditional statement that determines the price of each video category. There are also classes to print the statement in different formats (i.e., `HTMLStatement` and `TextStatement`), as proposed by the Template Method pattern.

[1]In the 2nd edition of his book, Fowler changed the system to JavaScript. However, we decided to use the original system since it is still widely known.

Figure 4.2: Class diagram of video store system (*final version*)

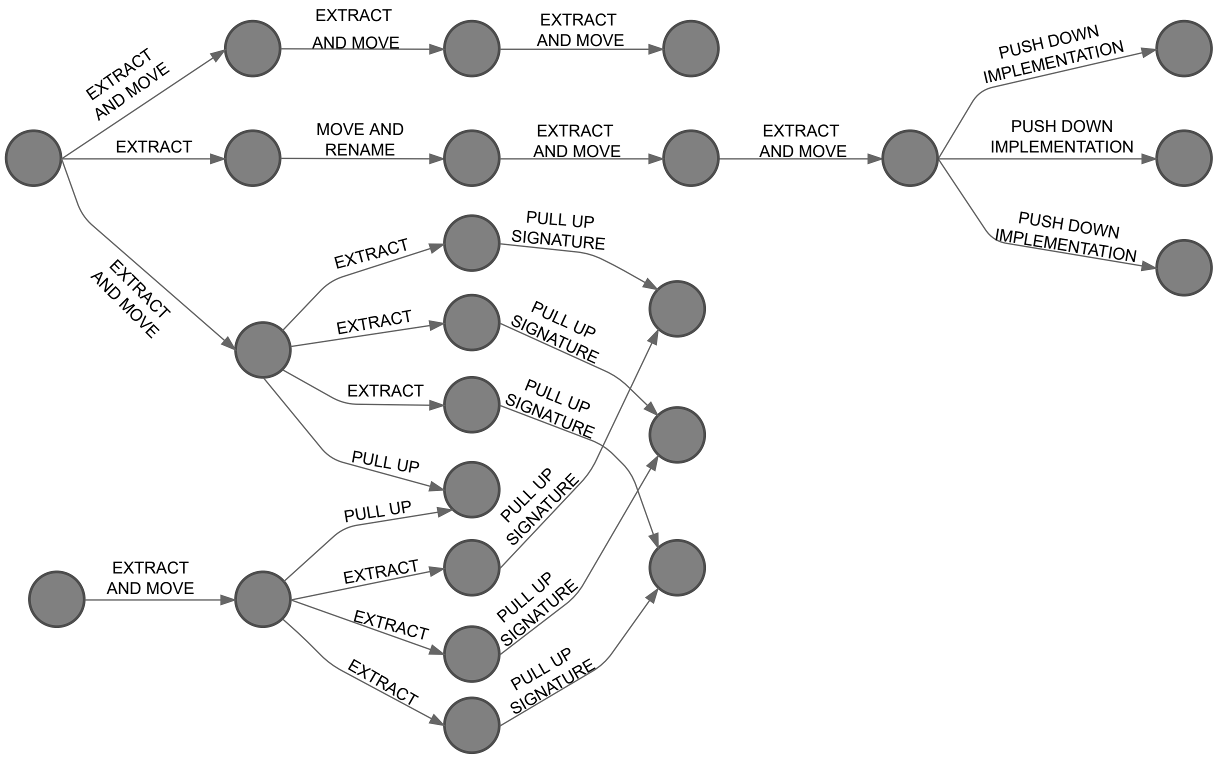


4.3 Refactoring Subgraph from Video Store System

We use refactoring graphs to visualize and understand the refactoring tasks performed by the students in our experiment. Specifically, we rely on the definition that are described in Chapter 3. Therefore, in this study, the vertices represent the full signature of methods. The edges refer to the refactoring type (e.g., extract method), and it includes additional information about the operation (e.g., author name, date, and commit).

In this context, following all the steps to refactor the Video Store System creates a large subgraph with 24 vertices and 26 edges, as presented in Figure 4.3. As we can notice, 14 edges refer to *extract* operations, generating one or more methods, i.e., new vertices in the subgraph. The subgraph also includes two *pull up* methods, six *pull up signatures* (when only the signature is pulled up, preserving the method body in the subclass), and three *push down implementations* (when the signature is kept in the superclass, as an abstract method). There is a single operation called *move and rename* method, which moves a method to a distinct class, changing its name.

Figure 4.3: Refactoring subgraph from Video Store System (*ground-truth*)



4.4 Study Design

4.4.1 Study Participants

We perform our study with 46 undergraduate students in Computer Science (10 women and 36 men), which we label from $S01$ to $S46$. Specifically, they are students concluding a Software Engineering course. In this course, they have studied topics such as refactoring, testing, design patterns, and clean code. The students could decline participation in the study. Also, we inform the participants that the results could be reported exclusively in an anonymous format. The period to perform the proposed refactoring exercise was one week. The results do not include three students who completed the set of tasks in less than ten minutes.

4.4.2 Refactoring Tasks & Research Questions

Each student received the three initial classes—`Movie`, `Rental`, and `Customer`—and the instructions to perform the refactorings. We separate the study into two main parts: *explicit* and *flexible* guidelines.

The first part refers to *explicit guidelines*, in which the students received explicit information about the expected refactoring tasks. Specifically, these guidelines indicate the piece of code that should be refactored, the refactoring type, and the signature of the new methods. There are 15 main steps. Three of them introduce new methods or classes, and the other cases involve refactoring operations to make the source code cleaner or to apply changes in the inheritance hierarchy. There is also large refactoring to replace a conditional by polymorphism, which results in a new `Price` hierarchy. Each step finishes with a commit indicating the conclusion of a refactoring task.

After finishing the first part, there are two similar methods in class `Customer`, `statement()` and `htmlStatement()`, which print the statement in ASCII and HTML formats, respectively. Therefore, in the second part of the study, we described to the students the problems of this design structure. For example, it is necessary to duplicate the code to print the statement in other formats, such as CSV, JSON, etc. To mitigate this issue, Fowler proposes the usage of a Template Method. To implement this pattern, it is necessary to extract and move a set of similar methods to new classes. Then, multiple extract operations are needed to make the methods identical. As a final operation, a set of pull up operations should be performed to generate the template code.

Therefore, the second part includes a set of *flexible guidelines*. We invite the students to implement the Template Method pattern to eliminate duplicated code as we previously described. Different from the *explicit guidelines*, we do not indicate operations or details about the piece of code that should be refactored. Each student received only high-level instructions. Specifically, they should create a template method by refactoring the source code. For all tasks, there are no restrictions regarding the use of automatic refactoring tools, which are provided, for example, by IDEs.

We report each guideline results in distinct research questions, which are described in Section 4.5. Specifically, we present the findings from *explicit guidelines* in RQ1, and the results from *flexible guidelines* in RQ2. We use refactoring graphs as an abstraction to understanding and visualizing the sequence of refactoring operations performed by the students. In RQ1, we rely on a graph similarity metric to detect results that diverge from our ground truth [81]. In RQ2, we perform a manual inspection of the refactoring subgraphs created from the refactorings performed by the students.

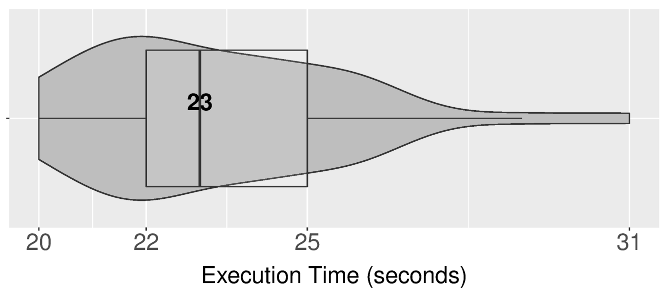
4.4.3 Detecting Refactoring Operations

We used RefDiff to identify the refactorings performed by the students [87]. As mentioned in a previous section, our study concentrates on six distinct refactoring operations over methods (i.e., *extract*, *extract and move*, *move and rename*, *push down implementation*, *pull up*, and *pull up signature*). As described in Chapter 2, RefDiff is a tool that detects refactorings by analyzing the commits of git-based projects. Therefore, for the 46 projects (i.e., one project per participant), we iterate in the list of commits, comparing each commit with its parent one.

4.4.4 Execution Time

We also computed the time to generate the refactoring graphs. To this purpose, we used a Core i7-8550U workstation with 16 GB of RAM, 5,400 RPM HDD, and Ubuntu 18.04. The process to build our ground truth executed in 18 seconds. As we can observe in Figure 4.4, regarding the students' projects, the execution times are also low, ranging from 20 to 31 seconds, with a median of 23 seconds. These runtime measures consider the whole process, which includes getting the list of commits, detection of refactoring operations by RefDiff, and generation of refactoring subgraphs. In total, we were able to generate all refactoring subgraphs (of all students) in approximately 18 minutes.

Figure 4.4: Distribution of the time to generate refactoring graphs per student



4.5 Results

In this section, we discuss our results. We divide the discussion in two parts: refactorings performed using *explicit guidelines* and refactorings performed using *flexible guidelines*. We rely on refactoring graphs to interpret the refactoring tasks performed by the students.

4.5.1 (RQ1) How do Students Apply Refactorings When Following Explicit Guidelines?

The instructions from the *explicit guidelines* generate ten edges in the expected refactoring subgraph (top of Figure 4.3). We compare this subgraph with our ground truth, reporting the number of extra and missing edges. We also assess the similarity by computing the subgraph edit distance. This distance is defined as the minimum number of addition, deletion, and replacement of vertices or edges that makes the student's subgraph identical to the ground-truth graph. As shown in Table 4.1, only three students made mistakes (6.5%). The other 43 students (93.5%) completed the tasks successfully, i.e., the distance to the ground truth is zero.

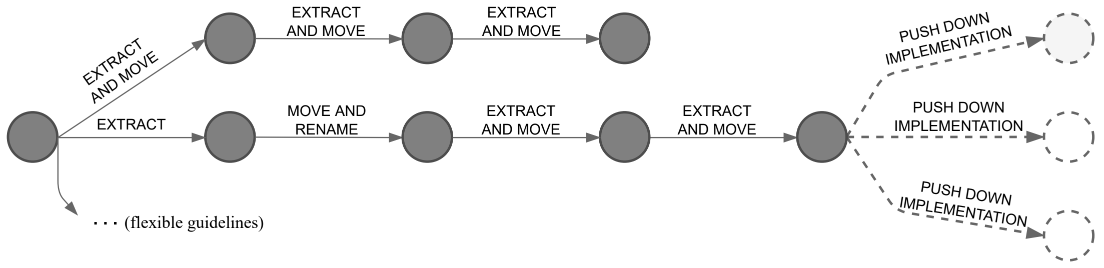
Table 4.1: Subgraphs with mistakes in the explicit guidelines

Student	**Edges**			**Distance**
	Total	**Extra**	**Absent**	
S02	7	-	3	6
S22	9	-	1	1
S44	12	2	-	2

These three students only made minor errors. For example, in the sixth refactoring task, the students should extract and move the method `Rental.getFrequent-RenterPoints()` from `Customer.statement()`. However, $S22$ performed this refactoring in two steps. In the first commit, the student copied the code from `statement()` to `getFrequentRenterPoints()`. In other words, the student created duplicated code. However, in a second commit, the student removed the duplication, replacing the code with a call to the new method.

There is a similar mistake in $S02$'s subgraph. In the thirteenth refactoring task, the students received instructions to change method `Price.getCharge()` by pushing down the implementation to three subclasses. But, $S02$ also performed this refactoring in two steps, creating duplicated code. In the first commit, the student copied the code to the subclasses. In a subsequent commit, the student updated the superclass, keeping only the method signature. Since the student did not perform the three *push down implementations* in a single commit, our refactoring mining tool had not detected the refactorings. As a result, the distance between the refactoring graphs is six. In other words, it is necessary to add three edges and three vertices to make the subgraph identical to the ground truth, as shown in Figure 4.5.

Figure 4.5: Example of a student's refactoring subgraph. Dashed lines represent missing edges and vertices (*explicit guidelines*, S02).



Finally, $S44$ reverted a refactoring task, which generates two extra edges in the subgraph. In this case, the reason is that the student created the new method with only a piece of the expected code. Therefore, $S44$ reverted the operation and after that performed it correctly.

Summary: When following explicit guidelines on refactoring operations, 93.5% of the students perform the refactoring tasks successfully. The few mistakes refer to refactorings performed in multiple commits. To evaluate the tasks, we used a similarity metric to compare each student's subgraph with the ground truth. We were able to obtain each assessment in a few milliseconds.

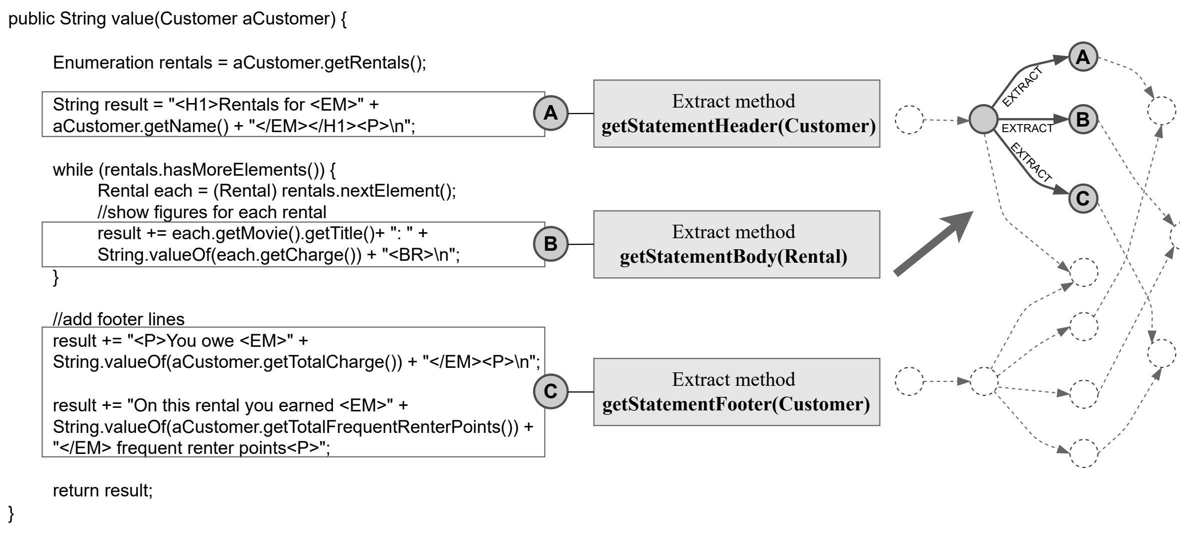
4.5.2 (RQ2) How do Students Apply Refactorings When Following Flexible Guidelines?

The *flexible guidelines* can be finished with three refactoring tasks. These tasks refer to the implementation of the Template Method pattern, which generates 16 edges in the ground truth refactoring subgraph (bottom of Figure 4.3). For this study part, we performed a qualitative analysis by manually inspecting the 46 subgraphs and the source code to investigate the students' strategies. In the following paragraphs, we describe the results. Overall, 32 students (70%) performed the *flexible guidelines* successfully.

Task 1. In the first task, the students received instructions to create the superclass `Statement`, as well as the respective subclasses (i.e., `TextStatement` and `HtmlStatement`). Next, they were instructed to extract and move method `statement()` to subclass `TextStatement`, which must contain the code to print the statement in ASCII format. Similarly, it is necessary to extract and move method `htmlStatement()` to subclass `HtmlStatement`, which contains the code to print the statement in HTML format. The two moved methods should have the name `value()`. Most students performed these steps correctly, generating two edges in the subgraph (44 students, 96%). In two cases, minor mistakes refer to unfinished refactorings.

Task 2. In the second task, we invited the students to perform all refactoring operations in order to make the two methods identical, i.e., `HtmlStatement.value()` and `TextStatement.value()`. For each method, we expect at least three extract operations, as presented in Figure 4.6. The first operation extracts a method to build the statement's header, returning the client's name in a specific format (HTML or ASCII). The second refactoring generates a method to retrieve the movie's title and charge. Finally, the third extracted method returns the statement's footer, which contains information about the price and the client's renter points. In Figure 4.6, vertices A, B, and C indicate the operations to decompose `HtmlStatement.value()`. Similarly, there are three extract operations to decompose method `TextStatement.value()`.

Figure 4.6: Refactoring operations to decompose method value() in subclass HtmlStatement (*flexible guidelines*, *task 2*)



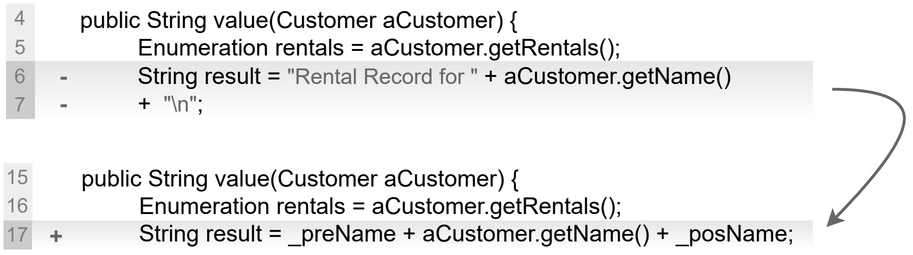
```
public String value(Customer aCustomer) {

    Enumeration rentals = aCustomer.getRentals();

    String result = "<H1>Rentals for <EM>" +
    aCustomer.getName() + "</EM></H1><P>\n";

    while (rentals.hasMoreElements()) {
        Rental each = (Rental) rentals.nextElement();
        //show figures for each rental
        result += each.getMovie().getTitle()+ ": " +
        String.valueOf(each.getCharge()) + "<BR>\n";
    }

    //add footer lines
    result += "<P>You owe <EM>" +
    String.valueOf(aCustomer.getTotalCharge()) + "</EM><P>\n";

    result += "On this rental you earned <EM>" +
    String.valueOf(aCustomer.getTotalFrequentRenterPoints()) +
    "</EM> frequent renter points<P>";

    return result;
}
```

A
Extract method getStatementHeader(Customer)
B
Extract method getStatementBody(Rental)
C
Extract method getStatementFooter(Customer)
EXTRACT
EXTRACT
EXTRACT
A
B
C

We detect that 33 students (72%) performed the expected operations, i.e., they decomposed the methods by extracting three new ones in each subclass. Interestingly, five students (11%) used a different strategy. They divided the statement footer into two methods. The first method returns the price and the second one returns the client's renter points. As a consequence, these students created eight vertices in the subgraph (i.e, four *extract* operations in each subclass).

Eight students (17%) did not perform extract operations to remove the code duplication. Among these cases, five students (11%) added fields. For example, *S*22 used fields to customize the strings, as presented in Figure 4.7. The values of the fields are defined in the constructors. For instance, field `_preName` receives the value *"<H1>Rentals for <EM>"* in subclass `HtmlStatement`. The same field receives the value *"Rental Record for"* in the subclass `TextStatement`. Finally, *S*17 only added empty methods, and two students did not make the methods identical.

Figure 4.7: Textual diff produced by GitHub. S22 added two fields (_preName and _posName) in the header line (*explicit guidelines*, *task 3*)



```
    public String value(Customer aCustomer) {
        Enumeration rentals = aCustomer.getRentals();
-       String result = "Rental Record for " + aCustomer.getName()
-       + "\n";

    public String value(Customer aCustomer) {
        Enumeration rentals = aCustomer.getRentals();
+       String result = _preName + aCustomer.getName() + _posName;
```

Task 3. Assuming the students performed the refactoring tasks as expected, we should have in this step two identical methods `value()` in `TextStatement` and `HtmlStatement`. As a final operation, we requested the students to remove this duplicated code. It is expected at least six *pull up signatures* and two *pull up* methods in this last task, which generates eight edges in the subgraph. Among the 46 students, 74% completed this task successfully (34 occurrences).

Summary: When following flexible guidelines for refactoring operations, most students implemented the *Template Method* pattern successfully (70%). Most mistakes refer to students who faced problems identifying the appropriate operations. Refactoring graphs assisted in visualization and understanding the sequence of refactoring tasks performed by the students.

4.6 Discussion

First, among the most recurring students' mistakes, there are refactoring operations completed in two steps (i.e., two commits). Refactorings performed along multiple commits represent a threat to current studies on refactoring practices. Particularly, the ones based on tools such as RefactoringMiner [86, 95, 96] and RefDiff [22, 85, 87]. These tools detect refactorings by computing the "diff" between one commit and its parent. For example, if we assume three sequential commits $C1$, $C2$, and $C3$, the tools will miss a refactoring that starts in $C1$, but that is only finished in $C2$, as we observed when manually analyzing the results of both RQs. Since studies on refactoring rely on the default configuration of RefactoringMiner and RefDiff [5, 12, 16, 49, 73, 89], they do not consider refactorings performed over multiple commits. Therefore, we also envision research on new heuristics and techniques to detect refactoring in multiples commits, as well as the investigation of their impact on mining software studies.

Second, several well-known graph algorithms can be used with refactoring graphs, such as algorithms to mine graph patterns and metrics [61, 81, 102]. For example, in the first research question, we used a similarity metric called *graph edit distance* to compare our ground-truth subgraph with the students' subgraphs. Refactoring graphs also allow easy navigation on refactoring operations. We can inspect, for example, entity names, visualize sequences of refactoring tasks, and identify missing and extra operations (i.e., edges and vertices). In this context, researchers can rely on refactoring graphs to perform empirical studies on complex and large refactoring practices over time, which is a hurdle recently pointed in the literature [4].

4.7 Threats to Validity

Refactoring tasks. First, the communication among the students is a possible threat in our study. However, the participants received instructions to avoid sharing their source code. To mitigate this threat, we removed outliers from our results. The period to perform the proposed refactoring exercise was one week. For the *explicit guidelines* (RQ1), about 75% of the students complete the tasks up to 198 minutes (i.e., approximately three hours), with a median of 106 minutes. In case of *flexible guidelines* (RQ2), 75% of the students spent about one hour to conclude the three refactoring tasks (i.e., 65 minutes), with a median of 44 minutes. The highest intervals probably refer to students that paused the refactoring tasks for a while, returning on a distinct day. Considering the distribution of the time, we eliminated the projects of three students, who performed the *explicit* or *flexible* guidelines in less than ten minutes since this is not a viable time to conclude the refactoring tasks. Second, the refactoring tasks include examples and discussions, which are spread over eleven chapters of Fowler's book. In other words, it is not trivial to obtain the answers directly from the book. Still, the usage of Fowler's book by students is a possible threat. We relied on two different strategies to alleviate this threat, i.e., creating not only *explicit* but also *flexible* (or open) guidelines. All students followed these guidelines, and we obtained different results. Particularly, when following the flexible guidelines, a significant part of them faced problems performing the refactoring tasks. Third, the Video Store System may represent a nonessential software project nowadays. We decided to keep this system to preserve consistency with Fowler's catalog, which is widely used in Software Engineering courses. In future studies, we plan to use the new book version [36], which relies on JavaScript and adaption of the system to another context.

Detection of refactoring operations. We removed a single refactoring operation and two projects due to compilation errors or false positives. These cases represent a small piece of the sample. Thus, it is not likely to influence our results. In addition, we used RefDiff to detect the refactorings [87]. We adopted this tool due to its support to multiple programming languages. Therefore, it is possible to replicate the study with the refactorings described in the last edition of Fowler's book, which relies on JavaScript [36]. It is also possible to adapt the guidelines to other programming languages, such as *C* [87] and *Go* [22].

Student's mistakes. We used *refactoring graphs* as a visual instrument to understand and evaluate the refactoring operations performed by the students. Also, we used a similarity metric called *graph edit distance* to compare our ground-truth subgraph with the students' subgraphs. For this purpose, we rely on a well-known Python library that implements this similarity algorithm.[2] However, our results also rely on the manual inspection of the commits of each student. Thus, we reinforce their subjective nature. The refactoring operations performed by the students (and our analysis) are publicly available at: `github.com/alinebrito/refactoring-graphs-thesis`

Generalization of the results. Our guidelines relied on the Java programming language and on a canonical refactoring example to introduce refactoring concepts, i.e., the Video Store System proposed by Fowler [35]. Thus, as unusual in empirical software studies, the results can not be generalized to other scenarios, such as different refactoring tasks or programming languages.

Sample size and students' experience. The 46 participants might not be a representative sample. However, they have previously studied several topics in the Software Engineering course (e.g., refactoring, testing, design patterns, and clean code).

[2]networkx.org/documentation/stable/reference/algorithms/similarity.html

4.8 Final Remarks

In this chapter, we explore the usage of *refactoring graphs* to represent, visualize, and assess large refactoring tasks over time. For this purpose, we invited 46 undergraduate students from a Software Engineering course to refactor a well-known example proposed by Fowler, a Video Store System [35]. Each student received two groups of refactoring tasks. The first group included explicit guidelines, i.e., the precise indication of the piece of code to be refactored and the expected operation. In contrast, the second group received flexible instructions. Specifically, in this second case, we invited the students to implement a design pattern without details about the piece of code to be refactored.

The refactoring tasks create a large refactoring subgraph with 24 vertices and 26 edges. The genereration of all refactoring graphs demanded about 18 minutes, i.e, a median of 23 seconds per student. After building such graphs, we assessed the refactoring tasks using graph-based metrics and performing visual inspections. We summarize our findings as follows:

- When following *explicit guidelines*, most students performed refactoring tasks successfully (93.5%). The few mistakes refer to refactorings performed in multiples commits.
- When following *flexible guidelines*, most students implemented the proposed design pattern successfully (70%). However, a significant number of students faced problems identifying the appropriate refactoring operations.

Based on our results, we provided implications for different scenarios. Specifically, we discuss our contributions regarding graph-based abstractions to support comprehension of large and complex refactoring operations. Finally, we argue about a possible thread in studies based on state-of-the-art mining tools [22, 85, 87, 95, 96], which do not detect refactoring performed in multiples commits. The dataset is publicly available at: `https://github.com/alinebrito/refactoring-graphs-thesis`

Chapter 5

A Catalog of Composite Refactorings

In Chapter 3, we observe recurring cases of refactoring operations by inspecting *refactoring subgraphs*. For example, developers decompose a method by performing multiple extractions. We see value investigating such scenarios, since it suggests that some operations are frequently performed over time. Therefore, we rely on the GSpan algorithm [61, 102] to mine graph patterns in a dataset of 11,005 GitHub projects.

We do not identify large graph patterns due to the size of the refactoring subgraphs. There are also irrelevant patterns, such as successive Rename operations. However, we notice some instances that reinforce our insight. For example, there are recurring cases of Inline operations to decompose a trivial method. Therefore, inspired by previous results and perceptions, **we rely on refactoring graphs to document a catalog of high-level code transformations**. We focus on sequences of operations to compose or decompose a program element, which we call *composite refactorings* [89].

To show the relevance of the proposed catalog, we mine instances of composites in a representative refactoring oracle with hundreds of confirmed single refactoring operations. Next, to complement this first study, we search for composites in the full history of ten well-known open-source projects. We rely on the refactoring graphs model to represent and document instances from this catalog.

This chapter is organized as follows. Section 5.1 introduces the relevance of conducting this study. Section 5.2 shows a first example of composite refactoring, relying on refactoring graphs to illustrate it. Section 5.3 shows how to represent composites relying on refactoring graphs, while Section 5.4 shows a first example of composite refactoring. We describe the results of the oracle study in Section 5.5, while Section 5.6 includes results in the wild, covering the full evolution history of ten popular open-source projects. The results are then discussed in Section 5.7. Section 5.8 states threats to validity. Finally, we conclude this chapter in Section 5.9.

5.1 Introduction

Aiming to promote and facilitate the dissemination of refactoring practice among developers, refactoring operations are usually documented in catalogs, like the one proposed by Fowler [35, 36] in his seminal book. In this catalog, Fowler provides detailed documentation about dozens of refactorings, providing a name for each refactoring, describing the mechanics required to perform the source code transformation, and also giving illustrative examples of the proposed refactorings. However, most refactorings described in Fowler's catalog are restricted in time and scope. Particularly, they are described as source code transformations that can be performed by a single developer, in a short time frame (time constraint) and by impacting a limited number of program elements (scope constraint). This understanding of refactoring is also assumed by modern refactoring detection tools, such as RefactoringMiner [6, 95, 96] and RefDiff [22, 85, 87]. Indeed, these tools report refactorings at a very fine granularity level. For example, suppose that a given method `m()` is implemented in classes `A1`, `A2`,..., `An`. Then, suppose that a Pull Up refactoring is performed to move the replicated method to a superclass B. These tools report this refactoring as a sequence of the following unrelated operations:

```
Pull up: A1.m() to B
Pull up: A2.m() to B
Pull up: A3.m() to B
...
```

However, we claim that the best output would be reporting a single composite refactoring operation:

```
Pull up: A1.m(), A2.(), A3.m(), ..., to B
```

This is just a trivial example of composite refactoring (in Section 5.4 we provide a more complex example). Indeed, composite refactorings were previously defined by Sousa et al. [89] as *"two or more interrelated refactorings that affect one or more elements"*. However, in their work, the authors focused on the role played by composite refactorings when removing code smells. In other words, they do not explore, document, and illustrate a comprehensive catalog of composite refactorings.

In this capther, we initially describe eight composite refactorings in abstract terms. Then, we look for instances of these catalog in two relevant samples, which are described bellow. As usually in catalogs, we also include illustrative examples.

Oracle study. In the first study, we mine composites in a large and representative refactoring oracle commonly used in the literature [93, 96]. Specifically, we look for occurrences of the refactorings in our catalog among 1,7K confirmed single refactoring instances listed in this oracle. We identify that a significant rate of 60.5% of the refactorings of interest in this oracle are part of composite operations. We also characterize the detected composites, under dimensions such as size and scope.

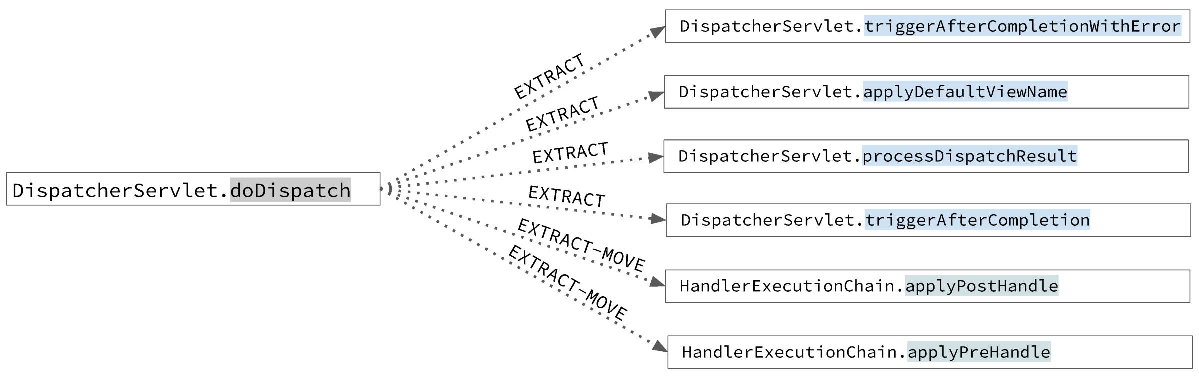
Study in the wild. In the oracle study, we rely on a sample that includes selected refactorings from distinct projects. Therefore, as a complementary analysis, we also look for composite refactorings in the full history of ten popular GitHub projects. As a result, we were able to identify and characterize 2,886 instances of composite refactorings.

Contributions: Based on the results described in this chapter, our contributions are threefold: (1) we propose a comprehensive catalog of composite refactorings; (2) we explore a second application of refactoring graphs, by using our graph-based abstraction to illustrate and document the proposed catalog; and (3) we characterize a large sample of composite refactorings performed in real-world software projects, providing a new viewpoint of a well-known refactoring oracle.

5.2 An Example of Composite Refactoring

We rely on *refactoring graphs* to illustrate the concept of composite refactorings. Figure 5.1 shows an example from Spring that is based in our graph-based abstraction.

Figure 5.1: Example of composite refactoring from *Spring Framework*. Method *doDispatch* was decomposed by applying six Extract Method refactorings


DispatcherServlet.doDispatch
EXTRACT
EXTRACT
EXTRACT
EXTRACT
EXTRACT-MOVE
EXTRACT-MOVE
DispatcherServlet.triggerAfterCompletionWithError
DispatcherServlet.applyDefaultViewName
DispatcherServlet.processDispatchResult
DispatcherServlet.triggerAfterCompletion
HandlerExecutionChain.applyPostHandle
HandlerExecutionChain.applyPreHandle

As we can see, method `doDispatch(...)` was decomposed by performing six Extract Method refactorings. Moreover, in two cases the Extract was followed by a Move

Method. These operations were performed in a single commit.[1]

When used in a context like this one, a refactoring detection tool, such as RefDiff [22, 85, 87] or RefactoringMiner [95, 96], detects these six single refactorings independently. However, it would be interesting to detect a high-level refactoring operation, i.e., a composite refactoring grouping the six transformations. As we detailed in the Section 5.4, we call Method Decomposition refactoring, this particular coarse-grained operation.

Techniques that may benefit from the detection of independent refactorings (like code visualization [4, 17, 68], code review [8, 9, 40, 73, 80], code authorship [7, 44], bug-introducing detection [58, 78], refactoring-aware tools [23, 84], software mining approaches [43, 47, 49, 90], to name a few) may also benefit from the detection of composite refactorings. As refactoring detection is the basis of such techniques, composite refactorings would bring to light novel operations not restricted to time and scope, therefore, better representing the actual source code changes.

Before presenting our catalog, it is important to mention that composite refactorings are not limited to a single commit [89]. For example, as stated by Fowler in the new version of his book on refactoring [36], there are also *long-term* refactorings "*that can take a team weeks to complete*".

5.3 Representing Composites with Refactoring Graphs

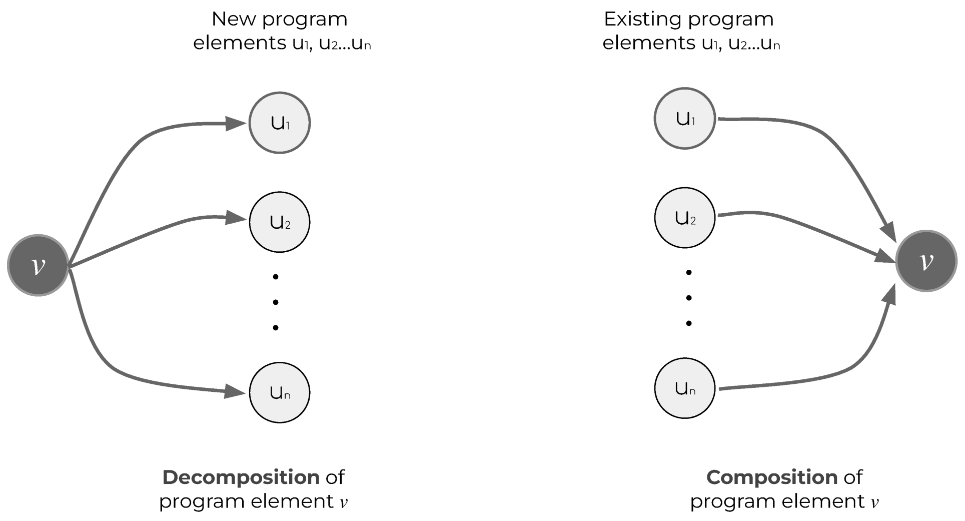
In the previous example (Figure 5.1), we show a composite refactoring that includes nodes representing methods before or after the source code transformation, according to the definition of refactoring graphs that are described in Chapter 3. However, we also extend our definition to classes and fields. Specifically, we include new types of nodes in the graph model, which represents the complete path of these code elements.

In case of composite refactoring representation, there are two typical graph-based models, which refer to a set of operations to compose or decompose program elements, as shown in Figure 5.2. For cases involving decomposition of an element, each subgraph includes a node v that represents the source element, which has *in-degree* $d^{-}(v) = 0$ and *out-degree* $d^{+}(v) \geqslant 2$. In other words, two or more edges are leaving from this single node. The set of vertices u_1, u_2, ..., u_n represent the new program elements after applying the refactoring operation, which are reachable only from node v. In an opposite scenario involving composition of program elements, the set of vertices u_1, u_2, ..., u_n refer to

[1]github.com/spring-projects/spring-framework/commit/3642b0f3

the elements affected by refactoring operations, which were performed to compose the program element represented by the node v. In this context, $d^-(v) \geqslant 2$ and $d^+(v) = 0$, i.e., there are two or more edges reaching to v. Finally, the edges hold metadata regarding the refactoring operations, as previously defined in Chapter 3.

Figure 5.2: Representation of composite refactorings as *refactoring graphs*. Node v denotes a program element (class or method) decomposed or composed by a performing a set of refactoring operations


New program elements u1, u2...un
Existing program elements u1, u2...un
v
u1
u2
un
v
u1
u2
un
Decomposition of program element v
Composition of program element v

5.4 Catalog of Composite Refactorings

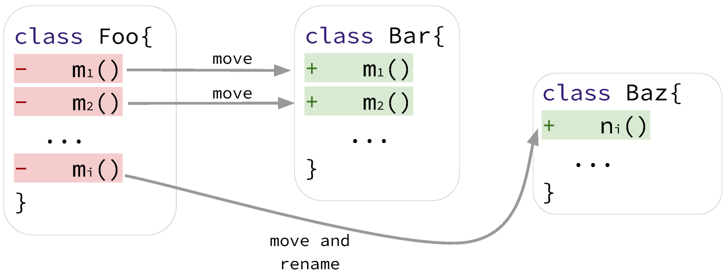
In this section, we introduce the proposed catalog of composite refactorings. As customary in refactoring catalogs, we describe the proposed refactoring types and their mechanics. We also present an abstract example of each composite refactoring, using the graph-based abstraction described in Chapter 3. There are two main groups of refactorings: (i) to decompose program structures (five composite refactorings), and (ii) to create program structures (three composite refactorings).

5.4.1 Class Decomposition

Motivation: According to Fowler [35, 36], during software evolution we might need to *"move elements around"*, aiming to improve modularity and cohesion and reduce coupling. Specifically, single Move Method operations should be performed *"when classes have too much behavior or when classes are collaborating too much and are too highly coupled"*. However, the overall solution may not be restricted to a single refactoring operation. Instead, we might need to move more than one method from a single source class. In this case, we say we performed a composite refactoring named Class Decomposition.

Mechanics: Figure 5.3 shows an example, in which class Foo lost multiple methods to classes `Bar` and `Baz`. The target class can be existing or new. Also, the Move operations can be followed by a Rename operation. In all cases, the final goal is to decompose the source class and make its implementation more cohesive. It is also worth noting that our definition does not require all move operations to be performed in a single commit. In other words, they can be spread over time, in multiple commits.

Figure 5.3: Class Decomposition

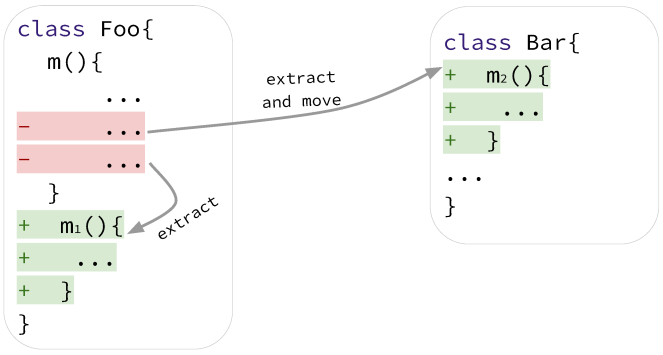


5.4.2 Method Decomposition

Motivation: We perform Extract Method operations when *"you have to spend effort looking at a fragment of code and figuring out what it's doing"* [35, 36]. In other words, Fowler advocates the improvement of understandability as the main reason to perform method extractions. However, the solution does not need to be limited to a sole operation. We could perform a sequence of two or more Extract Method operations over a single method. As a result, it generates a simpler one. Evidently, these refactorings also generate new methods. However, the goal is still the decomposition of the source method. In this case, we say we performed a composite refactoring named Method Decomposition.

Mechanics: Figure 5.4 shows an abstract example, in which methods $m_1()$ and $m_2()$ were extracted from method `m()`. After the extractions, the new methods can be moved to a distinct class, as happened with $m_2()$. As usual, the operations can be performed in one or multiple commits.

Figure 5.4: Method Decomposition

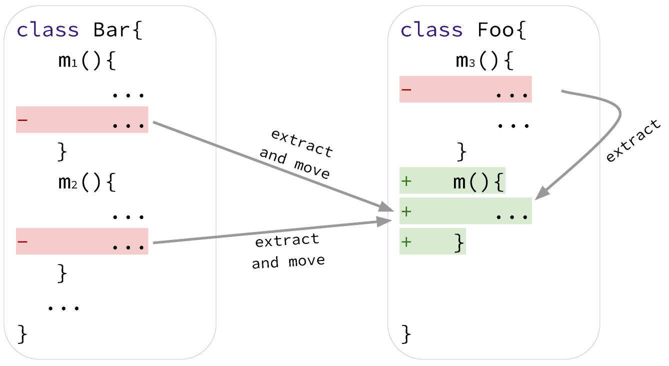


5.4.3 Method Composition

Motivation: Extractions also can be performed to promote reuse and to remove duplication [35, 36, 86]. Particularly, in such cases, we have similar fragments of code scattered over multiple locations. Therefore, a single Extract Method operation does not eliminate the duplicated issue. Instead, it may be necessary to apply multiple extractions to remove the duplicated code, generating a new method. In this case, we say we performed a composite refactoring named Method Composition.

Mechanics: Two or more Extract Method operations are performed over duplicated code, as illustrated in Figure 5.5. This code is then removed and a new method is created, with the previously duplicated code. The operations also can be followed by Move Method operations, i.e., the new method is placed in a distinct class.

Figure 5.5: Method Composition

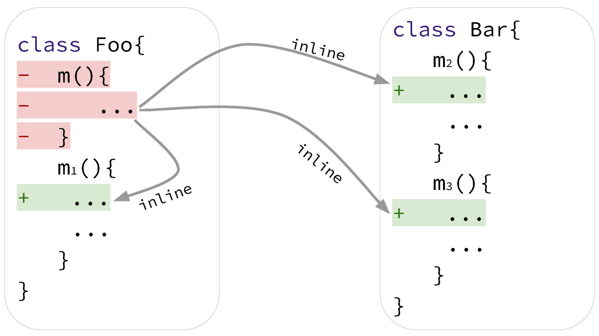


5.4.4 Composite Inline Method

Motivation: Inline Method—as originally proposed in Fowler's catalog—is reported as the opposite operation of Extract Method. The author suggests applying a set of Inline Method operations to remove trivial methods [36]. However, Inline Method is usually detected as a single operation by current refactoring detection tools [87, 96]. That is, such tools report independent Inline operations, even when they are part of the same group of operations. Therefore, we decided to include this refactoring in our catalog, since it matches our criteria for composite refactorings and is not properly explored and detected by current tools.

Mechanics: We expand a (simple) method body in its call sites, as shown in Figure 5.6. Then, we remove the source method. The calls may be located in methods from distinct classes.

Figure 5.6: Composite Inline Method



```
class Foo{
-   m(){
-         ...
-   }
    m1(){
+     ...
      ...
    }
}
```

```
class Bar{
    m2(){
+     ...
      ...
    }
    m3(){
+     ...
      ...
    }
}
```

5.4.5 Composite Pull Up Method

Motivation: Fowler also points to the need to move up or down methods in inheritance hierarchies [36]. In this context, we apply sequences of Pull Up Method to create a single and more general method in the superclass, therefore achieving code reuse. As in the case of Inline, we decided to include this refactoring in our catalog mainly because Pull Up operations are reported as individual and independent operations by current refactoring detection tools.

Mechanics: This operation refers to sequences of transformations performed to move methods from subclasses to their superclass. For example, consider a class `SuperFoo` with subclasses `SubFoo1`, `SubFoo2`, and `SubFoo3`, as presented in Figure 5.7. Suppose that a Pull Up operation is applied to move method `m()` from these subclasses to the superclass. Usually, this operation occurs in a single commit. First, a developer copies the method `m()` to the superclass, which can be an existing or new one. After that, the method is removed from the subclasses. In this context, the following three messages are issued by RefactoringMiner [96]:

```
Pull Up Method public m() : void from class SubFoo1
to public m() : void from class SuperFoo
```

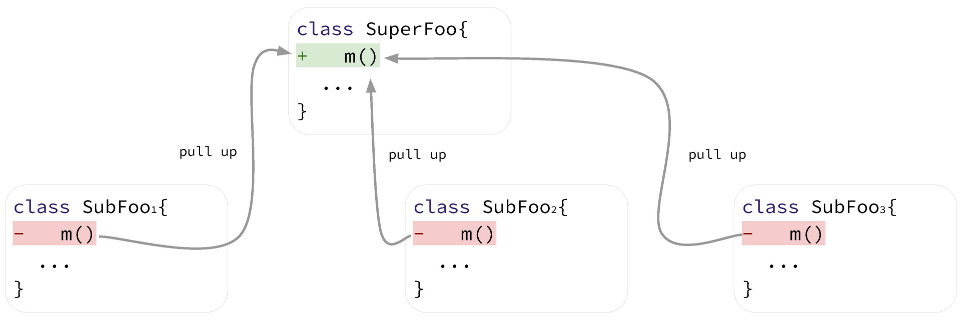
```
Pull Up Method public m() : void from class SubFoo2
to public m() : void from class SuperFoo
```

```
Pull Up Method public m() : void from class SubFoo3
to public m() : void from class SuperFoo
```

However, since essentially they are part of the same composite refactoring, we claim these operations should have been reported using a single and comprehensive message, such as:

```
Pull Up method public m() : void
From: SubFoo1, SubFoo2, and SubFoo3
To: public m() : void in SuperFoo
```

Figure 5.7: Composite Pull Up Method

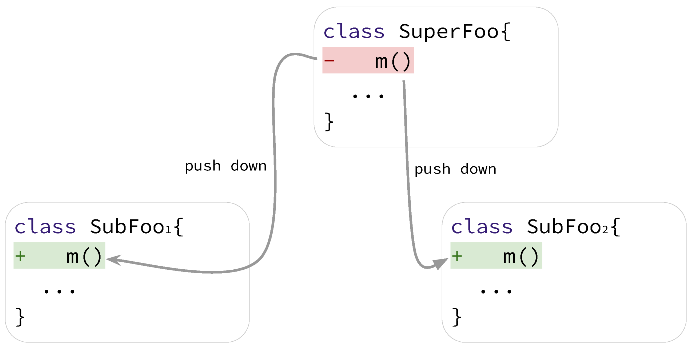


5.4.6 Composite Push Down Method

Motivation: As an opposite scenario, we perform Push Down Method when a method is needed only in a few subclasses [35, 36]. Therefore, this refactoring promotes inheritance simplification. This operation—also present in Fowler's catalog—matches our criteria for composite refactoring. However, as in the case of Pull Up and Inline, it is reported as independent operations by current refactoring mining tools.

Mechanics: This operation moves a given method from the superclass to particular subclasses, as presented in Figure 5.8. After that, the method is removed from the superclass.

Figure 5.8: Composite Push Down Method

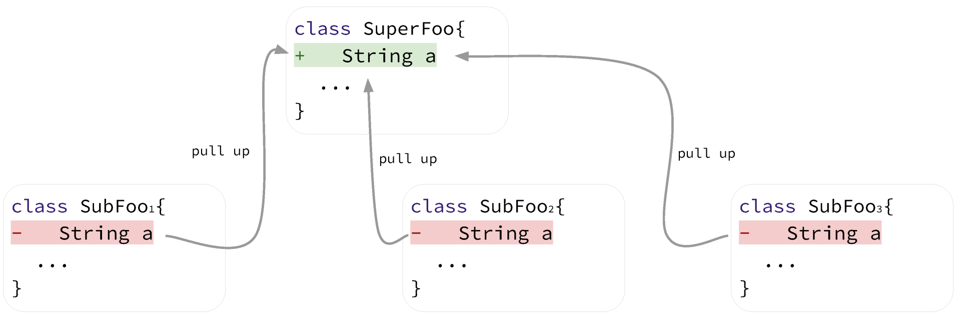


5.4.7 Composite Pull Up Field

Motivation: Often, we have duplicate data in inheritance hierarchies, for example, fields used for a similar purpose in distinct subclasses. In this case, we can perform a sequence of Pull Up Field to create a single one in the superclass, aiming to promote reuse [36]. Therefore, this operation also corresponds to our criteria for composite refactoring and we say we performed a Composite Pull Up Field.

Mechanics: First, we declare the field in the superclass. Then, we remove the declaration in the subclasses, as shown in Figure 5.9. This operation can also be preceded by Rename Field, aiming to standardize the names before the movement to the superclass.

Figure 5.9: Composite Pull Up Field

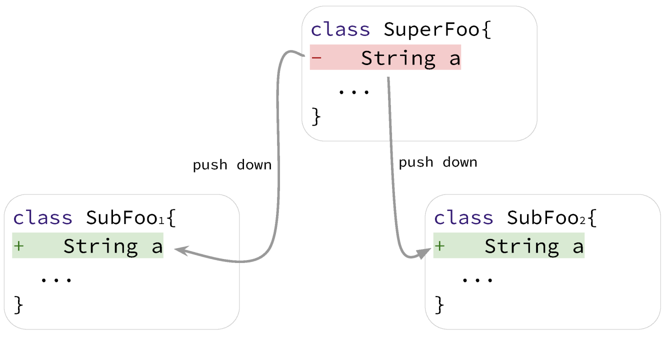


5.4.8 Composite Push Down Field

Motivation: Similar to Push Down Method, the goal involves moving data from a superclass to specific subclasses [36]. When this operation contemplates a sequence of fields movements, we say we performed a Composite Push Down Field.

Mechanics: First, we declare the field in the required subclasses. Then, we remove the declaration in the superclass, as shown in Figure 5.10.

Figure 5.10: Composite Push Down Field



5.4.9 A Final Note on Completeness

The original refactoring catalog proposed by Fowler has dozens of refactorings. Therefore, the catalog of composites described in this section is much smaller (eight composites). On the one hand, this difference is expected because composites are coarse-grained and complex source code transformations, composed by atomic refactorings. On the other hand, it is also important to acknowledge that we do not claim on the completeness of the proposed catalog. Indeed, our central intention is to provide a comprehensive, well-documented, and easy to understand initial list of composite refactorings. In future studies, we can extend the list to include other types of composites. For example, most current instances are refactorings at the method level, since they are among the most frequent code elements affected by such operations [87, 96]. However, it is possible to extend the catalog by including, for example, operations to compose or decompose packages.

5.5 A First Oracle of Composite Refactoring

To investigate whether the proposed composite refactorings occur in real-world projects, we initially search for composite refactorings in one of the most representative refactoring oracles in the literature, curated by Tsantalis and other researchers [93, 95, 96]. This oracle has been expanded over the years. The latest version includes more than 14K refactoring operations from 185 Java projects. The oracle instances were validated by multiple authors and/or well-known tools. In other words, it is a trustworthy dataset for studying refactoring practices.

5.5.1 Study Design

Research Questions Assessment

We propose two research questions:

(RQ1) What are the Most Common Composite Refactorings in the Oracle? In the current version of the oracle, refactoring operations are reported as individual (i.e., non-composite) ones. Thus, in this first RQ, our goal is to explore the oracle data from a new perspective, looking for occurrences of composite refactorings. In other words, we aim to provide a new oracle view, which is not based on individual refactoring operations. For this purpose, we first compute the frequency (i.e., the number of occurrences) of each composite instance.

(RQ2) What are the Characteristics of Composite Refactorings in the Oracle? The rationale of this second research question is to understand the main characteristics of the composite refactorings detected in RQ1. Therefore, for each composite instance, we compute

information such as its scope (i.e., location of the entities before and after a refactoring operation) and size (i.e., number of individual refactoring operations).

Dataset

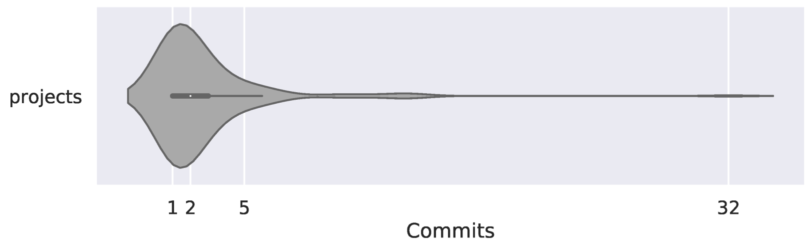
In January 2022, we retrieved the most recent oracle version. Then, **we selected only refactoring operations that could be part of composite operations**.[2] For example, the original oracle includes refactorings such as Move Attribute and Rename Method, which are not related at all with the composites described in Section 5.4. As presented in Table 5.1, our oracle sample includes 1,725 individual refactoring instances. Most instances are Extract Method (976 occurrences) and Move Method operations (227 occurrences).

Table 5.1: Selected refactoring operations in the oracle

Operation	**Projects**	**Commits**	**Occurrences**	**%**
Extract Method	140	329	976	56.6
Move Method	53	73	227	13.2
Inline Method	48	64	127	7.4
Move and Rename Method	29	35	116	6.7
Extract and Move Method	29	35	114	6.6
Pull Up Method	24	28	74	4.3
Push Down Method	10	11	30	1.7
Pull Up Field	14	14	36	2.1
Push Down Field	11	11	25	1.4
All	166	450	1,725	100

These operations are detected in 450 commits from 166 projects, such as `Infinispan` (a tool for storing, managing, and processing data)[3] and `Gradle` (a build automation tool).[4] Figure 5.11 shows the distribution of the number of selected commits per project. As we can observe, the median is two commits, while the 90th percentile is about five commits. In the case of 78 projects (47%), there are only instances from a single commit. In other words, the oracle sample does not include the whole project's history.

Figure 5.11: Distribution of commits per project (oracle)



[2]By construction, the discarded refactorings cannot be part of the composites included in our catalog. However, we acknowledge they can be part of future composites (in this case, therefore we will need to update the current oracle).

[3]`https://github.com/infinispan/infinispan`

[4]`https://github.com/gradle/gradle`

Detecting Composite Refactorings

We implement a set of scripts to detect the composites described in Section 5.4. Their input comprises a list of individual refactoring operations. Basically, these scripts operate by searching for clusters of refactoring operations R_1, R_2,.., R_n that can be replaced by a single composite refactoring *CR*. Therefore, we iterate over the list of refactorings detected in a system, grouping operations by considering the criteria described in Table 5.2.

Table 5.2: Conditions to cluster two refactoring operations (r_1 and r_2) into a composite

Composite	**Condition**
Method Composition	$signature(r1.target) = signature(r2.target) \wedge type(r1.target) = type(r2.target) \wedge (r1.refType, r2.refType) \in \{extract, extract\_move\}$
Method Decomposition	$signature(r1.source) = signature(r2.source) \wedge type(r1.source) = type(r2.source) \wedge (r1.refType, r2.refType) \in \{extract, extract\_move\}$
Class Decomposition	$type(r1.source) = type(r2.source) \wedge (r1.refType, r2.refType) \in \{move, move\_rename\}$
Composite Inline Method	$signature(r1.source) = signature(r2.source) \wedge type(r1.source) = type(r2.source) \wedge (r1.refType, r2.refType) \in \{inline\}$
Composite Pull Up Method	$signature(r1.target) = signature(r2.target) \wedge type(r1.target) = type(r2.target) \wedge (r1.refType, r2.refType) \in \{pull\_up\}$
Composite Push Down Method	$signature(r1.source) = signature(r2.source) \wedge type(r1.source) = type(r2.source) \wedge (r1.refType, r2.refType) \in \{push\_down\}$
Composite Pull Up Field	$name(r1.target) = name(r2.target) \wedge type(r1.target) = type(r2.target) \wedge (r1.refType, r2.refType) \in \{pull\_up\}$
Composite Push Down Field	$name(r1.source) = name(r2.source) \wedge type(r1.source) = type(r2.source) \wedge (r1.refType, r2.refType) \in \{push\_down\}$
$r.source$	program element before a refactoring operation r
$r.target$	program element after a refactoring operation r
$r.refType$	type of a refactoring operation r
$signature(m)$	signature of a method m
$name(f)$	name of a field f
$type(e)$	type of a program element e

For Method Decomposition, Class Decomposition, Composite Push Down Method, Composite Push Down Field, and Composite Inline Method (i.e., operations that break down code elements), we search for groups of refactorings that have as source the same code element. For Composite Pull Up Method, Composite Pull Up Field, and Method Composition), we look for refactorings that have as target the same code element.

Moreover, the source and target checking vary according to each composite refactoring. For composites at the level of methods, we verify the signature and the class. For example, for Method Composition, we group refactorings r_1 and r_2 into the same composite whenever the signature of the target methods are the same (i.e., $signature(r1.target) = signature(r2.target)$) and the target methods are in the same class (i.e., $type(r1.target) = type(r2.target)$). We also check the respective refactoring types. In the case of Composite Pull Up Field and Composite Push Down Field, i.e, composites at the level of fields, we verify the field's name and their respective class. Finally, for Class Decomposition, we group Move Method operations that originated from the same class (i.e., $type(r1.source) = type(r2.source)$).

It is also important to mention that our criteria for grouping refactoring operations do not include time constraints. Therefore, two or more refactorings can be part of the same composite, even though they were performed in distinct periods over the system's history. We made this decision motivated by two considerations. First, it is not trivial to set a threshold for the duration of the composites. Second, because our main goal is to propose a catalog of composites, as well as to mine and analyze examples of these refactorings, even if they were performed in long time intervals.

After their execution, our scripts produce a list of composite refactorings, including a textual data and a visualization based on *refactoring graphs*. To validate the results, we manually inspected a sample of composite refactorings from the oracle. Specifically, we execute the following steps for each composite type:

1. We selected a random sample of four instances (of each composite).

2. For each selected instance:

   - We carefully analyzed the respective refactorings in the oracle, verifying whether the operations are correct. In other words, we check the refactoring type, source, and target, as well as basic information such as project name and commit.

   - In the last step, we verify if there are missing refactorings. In other words, we check if there are operations in the oracle that should be a part of the selected composite. For example, in `neo4j`, we detected a Method Composition that creates the method `createCountsTracker()` in class `CountsComputerTest`.[5] For this case, we verify if there are extractions to the same target that were not properly detected by our scripts.

We manually inspected 28 composites, since for Composite Push Down Method and Composite Push Down Field, we detected only four instances in the oracle. Table 5.3 summarizes the results. The size of the selected composites ranges from 2 to 39 refactoring operations, covering 160 refactorings from the oracle. Overall, we did not identify errors

[5]`github.com/alinebrito/composite-refactoring-catalog/blob/main/results/oracle/neo4j/neo4j/results/composition_extract_method/view/subgraph_atomic_4.md`

by inspecting the sample of composite refactorings. In other words, we do not detect absent refactoring operations, i.e, operations that were not clustered correctly by our scripts. The scripts and inspected sample are publicly available at:

`https://github.com/alinebrito/refactoring-graphs-thesis`

Table 5.3: Inspected sample of composite refactorings (Oracle)

Composite	**Instances**	**Refactorings**
Method Composition	4	48
Method Decomposition	4	22
Class Decomposition	4	52
Inline Method	4	9
Pull Up Method	4	11
Push Down Method	2	4
Composite Pull Up Field	4	10
Composite Push Down Field	2	4
All	28	160

5.5.2 Results

(RQ1) What are the Most Common Composite Refactorings in the Oracle?

Among 1,725 single refactoring operations, an impressive number of 1,043 (60.5%) are part of composite refactorings, as presented in Table 5.4. For example, 537 Extract Method or Extract and Move Method are part of Method Composition instances, which is the most frequent case. There are also significant rates of Method Decomposition (125 occurrences, 34.1%) and Class Decomposition (55 occurrences, 15%). However, composite refactorings in the inheritance hierarchy are infrequent. For example, there are only 15 composite refactorings formed by Push Down Method and Pull Up Method operations (4.1%). Also, there are a few occurrences of composites at the field level.

Overall, we detected 366 composite refactorings over 81 distinct projects. In 39 projects (48.1%), there is only a single composite instance. We identify most cases in a project called `Robovm`—127 instances grouping 389 single refactoring operations. Interestingly, this project also includes the largest composite refactoring instance, involving the composition of a method `has(...)`, which was created as the result of 30 Extract Method operations.[6] The new method contains only a single line of code:

[6] `https://github.com/robovm/robovm/commit/bf5ee44b`

Table 5.4: Frequency of composite refactorings (Oracle)

Name	**Projects**	**Composites**		
		Operations	**Occurrences**	**%**
Method Composition	37	537	142	38.8
Method Decomposition	37	295	125	34.1
Class Decomposition	37	277	55	15.0
Composite Inline Method	11	48	21	5.7
Composite Pull Up Method	7	33	13	3.6
Composite Push Down Method	2	4	2	0.6
Composite Pull Up Field	4	15	6	1.6
Composite Push Down Field	1	4	2	0.6
All	81	1,043	366	100

```
public boolean has(CFString key) {
    return data.containsKey(key);
}
```

Therefore, this particular case of Method Composition was performed to remove code duplication (in this case, represented by a single line of code). It is worth mentioning that in the original oracle, this information was diluted over 30 individual and disconnected refactoring operations. By contrast, in our oracle view, they are represented by a single composite refactoring.

Summary of RQ1: Out of 1,725 single refactoring operations, approximately 60% are part of composite refactorings. We detected the instances of composite refactoring in 81 projects. The most recurring cases are Method Composition (142 occurrences, 38.8%), Method Decomposition (125, 34.1%), and Class Decomposition (55, 15%).

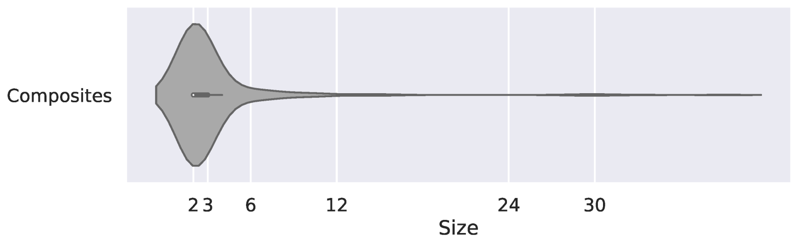
(RQ2) What are the Characteristics of Composite Refactorings in the Oracle?

We also investigate the main characteristics of the composite refactorings detected in RQ1 in terms of size and scope. Regarding their size, i.e., the number of refactoring operations, most instances are small, as expected. As we can observe in Figure 5.12, about 84% of the detected composite refactorings have up to three refactoring operations (308 occurrences). The values range from 2 to 39 refactoring operations per composite.

Next, we detail the results for the most important composites:

Class Decomposition. Among the 55 instances of Class Decomposition, 61.8% refers to classes losing up to two methods (29 occurrences) or three methods (5 occurrences). However, this category includes one of the largest composite refactorings in the oracle,

Figure 5.12: Distribution of the size of composite refactorings (Oracle)

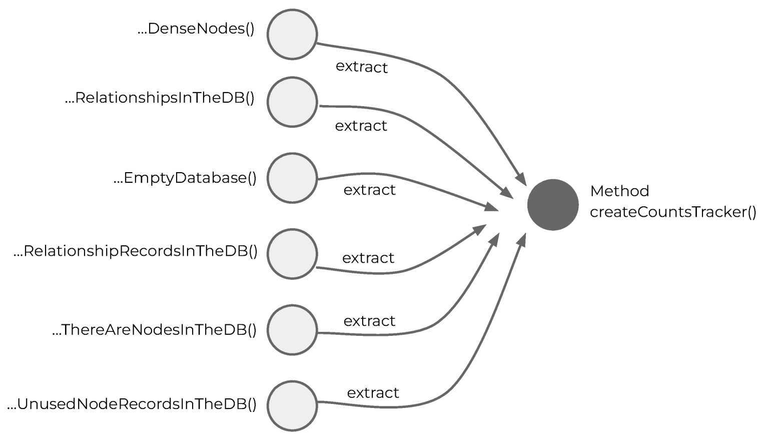


where a developer from `Graphhopper` decomposed a class by moving 39 methods.[7] Interestingly, in the commit message, the developer added a brief description regarding the motivation, which is related to a well-known design principle (use composition instead of inheritance [37, 97]):

"Refactoring of [class name]: use composition instead of inheritance"

Method Composition. The size of the 142 instances of Method Composition varies from 2 to 31 operations, with a median of two operations per composite. Furthermore, most Method Composition instances are *intra-class* (121 occurrences, 85%), i.e., the source methods are located in the same class of the target method. Figure 5.13 shows an example from `Neo4j`, where a developer extracted a method called `createCountsTracker()` from six methods.[8] All refactorings happened in the scope of the same class `CountsComputerTest`. However, in the original oracle, these refactorings are reported as six distinct and unrelated operations. Finally, in 10 cases (7%), the composites are *inter-class*, i.e., developers compose methods by "merging" pieces of code coming from distinct classes. The remaining are *mixed* Method Decomposition, including the two categories.

Figure 5.13: Example of Method Composition from Neo4j (Oracle)


...DenseNodes()
...RelationshipsInTheDB()
...EmptyDatabase()
...RelationshipRecordsInTheDB()
...ThereAreNodesInTheDB()
...UnusedNodeRecordsInTheDB()
extract
extract
extract
extract
extract
extract
Method
createCountsTracker()

Method Decomposition. 95% of the instances of Method Composition (119 occurrences), which were detected in 37 projects, have up to three operations. The values range from

[7]`https://github.com/graphhopper/graphhopper/commit/7f80425b`
[8]`https://github.com/neo4j/neo4j/commit/5fa74fbb`

2 to 15 operations, distributed among *intra-class* cases (91%), *inter-class* cases (3%), and *mixed* ones (6%).

Other cases. In the oracle, there are only 127 occurrences of Inline Method. Consequently, we also found a few cases of composite inlines (21 instances, 5.7%), including at most four operations. The same applies to composite refactorings over inheritance hierarchies: Composite Pull Up Method (13 instances, 3.6%), Composite Push Down Method (2 instances, 0.6%), Composite Pull Up Field (6 instances, 1.6%), and Composite Push Down Field (2 instances, 0.6%).

Summary of RQ2: Most composite refactorings are small, including up to three operations. However, we also detect large instances, for example, 30 Extract Method operations to compose a single method. Regarding the scope of the operations, most Method Composition and Method Decomposition are *intra-class.* In other words, developers usually extract multiple methods to the current classes.

5.6 Composite Refactoring in the Wild

In the first study, we look for composites in a well-known oracle. However, the refactoring instances selected for this oracle do not cover the complete history of each project. In other words, the oracle used in Section 5.5 only contains selected refactoring instances. Therefore, we might have missed operations in the reported composite refactorings simply because they were not selected for inclusion in the oracle. To tackle this issue, we decided to perform a complementary study, in which we search for composite refactorings in the complete history of ten popular GitHub-based projects.[9]

5.6.1 Study Design

Research Questions Assessment

As in the study described in Section 5.5, we propose two research questions:

[9]Since operations at the field level are infrequent, and it is also unsupported by the current RefDiff tool version, we decide not to include them in this complementary study.

(RQ3) What are the Most Common Composite Refactorings in the Wild? Similarly to RQ1, we assess the frequency of each composite refactoring, but now in 10 popular GitHub projects.

(RQ4) What are the Characteristics of Composite Refactorings in the Wild? Similarly to RQ2, the rationale of this research question is to shed light on the main characteristics of composite refactorings while considering the complete development history of 10 projects.

Dataset

To answer the proposed research questions, we relied on a set of real-world and popular projects. Specifically, we selected the top-10 Java projects on GitHub, ordered by their number of stars. We adopted this criterion because stars is a relevant metric to identify popular repositories [15, 88]. Moreover, in our sample, we only include projects that are software systems. For example, despite having a high number of stars, we did not include *kdn251/interviews* (a guide for interviews),[10] and *iluwatar/java-design-patterns* (a set of code samples).[11] Table 5.5 describes the selected projects, including basic information, such as number of stars, commits, contributors, and short descriptions. The selected projects are from distinct domain areas, including web frameworks and animation libraries.

Table 5.5: Selected Java projects

Project	**Stars**	**Comm.**	**Contr.**	**Short Description**
Spring Boot	56,717	33,692	831	Support framework
Elasticsearch	56,081	60,227	1,651	Analytics engine
RxJava	45,055	5,921	278	Event-based library
Spring Framework	43,943	22,728	551	Support framework
Google Guava	42,045	5,609	265	Core Java libraries
Square Retrofit	38,539	1,902	158	HTTP client
Apache Dubbo	35,968	4,848	349	RPC framework
MPAndroidChart	33,811	2,070	69	Chart library
Lottie Android	31,612	1,321	106	Rendering library
Glide	31,578	2,592	131	Image library

Detecting Composite Refactorings

To detect composite refactorings, we need first to identify single refactoring operations. For this purpose, we used RefDiff, a well-known multi-language refactoring tool [85, 87], which is described in Chapter 2. As usual in git-based mining tools, RefDiff detects refactorings by comparing a commit with its parent commit. To facilitate the usage of

[10]`https://github.com/kdn251/interviews`
[11]`https://github.com/iluwatar/java-design-patterns`

the tool, we first implemented a set of scripts that automate tasks such as downloading GitHub projects and retrieving the list of commits from the default branch. The scripts then rely on RefDiff to detect single refactoring operations. They also automatically exclude refactorings in non-core packages, such as *"test"* and *"sample"*. The final step concerns the detection of the composites defined in our catalog, which are represented as *refactoring graphs.*

5.6.2 Results

(RQ3) What are the Most Common Composite Refactorings in the Wild?

As presented in Table 5.6, we identify 2,886 occurrences of composite refactorings. Most cases refer to Class Decomposition (957 occurrences, 33.2%), i.e., 957 classes and interfaces have lost multiple methods. The values range from 8 classes in Lottie Android to 280 classes in Elasticsearch.

Table 5.6: Frequency of composite refactorings (in the wild)

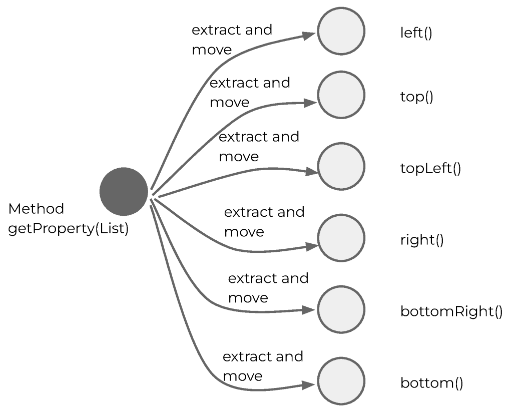
Name	**Wild**		**Oracle**	
	Occur.	**%**	**Occur.**	**%**
Class Decomposition	957	33.2	55	15.0
Method Decomposition	683	23.7	125	34.1
Method Composition	582	20.2	142	38.8
Composite Pull Up Method	450	15.6	13	3.6
Composite Inline Method	129	4.5	21	5.7
Composite Push Down Method	85	2.8	2	0.6
Composite Pull Up Field	-	-	6	1.6
Composite Push Down Field	-	-	2	0.6
All	2,886	100	366	100

Moreover, about 32% of the composites are from Elasticsearch, a popular search engine.[12] In this project, we detect 921 composites grouping 3,310 single refactoring operations. Among them, most cases refer to Class Decomposition (280 occurrences, 30.4%).

There is also a significant number of Method Decomposition (683 occurrences, 23.7%), such as in the example of Figure 5.14. In this case, method `getProperty(List)` lost multiple pieces of code, after a developer performed six Extract and Move operations in a single commit.

[12]`https://github.com/elastic/elasticsearch`

Figure 5.14: Example of Method Decomposition in Elasticsearch



Interestingly, all extracted methods were moved to the same class `GeoBoundingBox`. In the commit description,[13] the maintainer points out the intention to centralize related logic:

"A lot of this logic can be centralized instead of having separated efforts to do the same things"

Summary of RQ3: In our extended dataset, the most common composite refactorings are Class Decomposition (957 occurrences, 33.2%); Method Decomposition (683 occurrences, 23.7%); and Method Composition (582 occurrences, 20.2%). There are also a few occurrences of composite refactorings related to inheritance, i.e., Composite Pull Up Method and Composite Push Down Method.

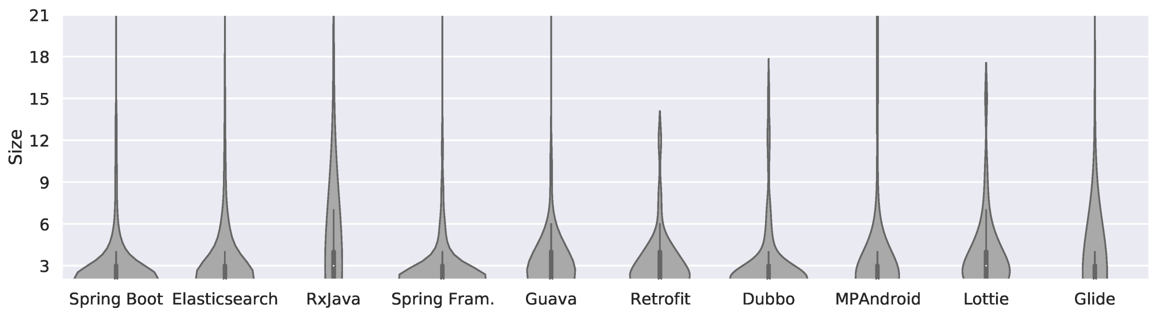
Comparison with the oracle results (RQ1): In Table 5.6, we also report the results obtained with the oracle sample, aiming to facilitate comparison. As we can notice, the frequency of composites follows a similar tendency, i.e., the top-3 cases are exactly the same: Class Decomposition, Method Decomposition, and Method Composition. However, in the oracle, the order is the reverse (e.g., Method Composition is the most frequent composite).

(RQ4) What are the Characteristics of Composite Refactorings in the Wild?

Regarding their size—as measured by the number of single refactorings in each composite—most instances in the extended dataset are also small. Figure 5.15 presents the size distribution per project, after removing outliers, since they tend to distort the plot's aspect. In all projects, the median is two or three operations. However, there are also large composites, for example, the largest case includes dozens of operations, in which several methods were moved from a single class.[14] In the following paragraphs, we detail the characteristics and give examples of each composite refactoring.

[13]https://github.com/elastic/elasticsearch/commit/769650e0
[14]https://github.com/ReactiveX/RxJava/commit/10325b90

Figure 5.15: Distribution of the size of composite refactorings per project



Class Decomposition. Figure 5.16 summarizes the size results of Class Decomposition. As we can notice, most composites of this type are small. About 62% of the cases involve up to three operations, such as in the example in Figure 5.17. In this example, a class of Lottie Android lost three methods in two commits. However, there are also large instances. For example, in Google Guava one developer moved each method from class `EmptyImmutableMap` to a distinct one, i.e., he performed a composite refactoring composed of ten operations.[15]

Figure 5.16: Number of operations by composite refactoring (Class Decomposition)

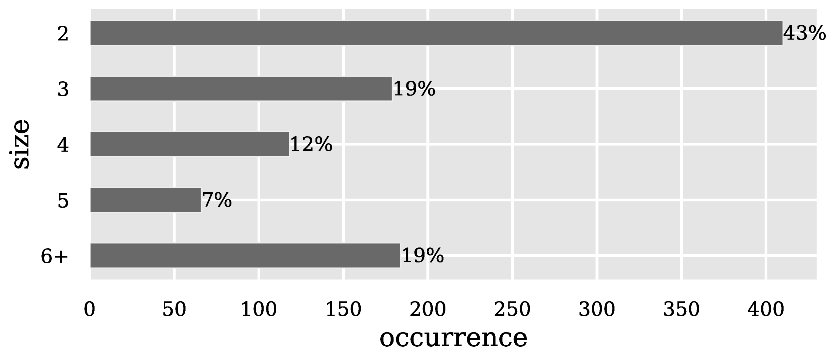
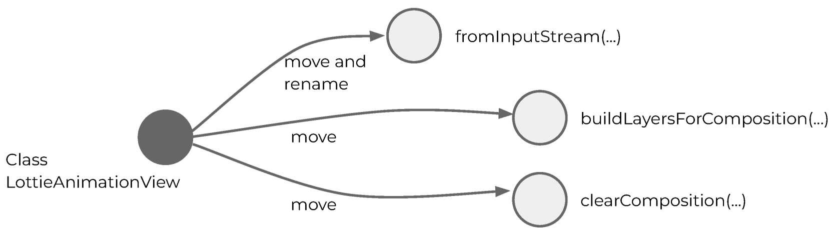


Figure 5.17: Example of Class Decomposition in Lottie Android



Method Decomposition. As in the study described in Section 5.5, we also separate the composites into *intra-class* (i.e., extractions to the same class of the fragmented method),

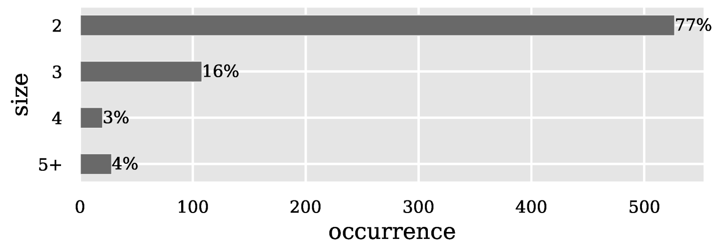
[15] `https://github.com/google/guava/commit/d8f98873`

inter-class (i.e., when the extracted methods are moved to distinct classes), and *mixed* (i.e., both cases), as shown in Table 5.7. As we can observe, most extractions are in the *intra-class* category (317 occurrences, 46%). However, another significant part of the results are *inter-class* (238 occurrences, 35%), i.e., all extracted methods are kept in the current class. We also investigate the number of extractions, i.e., the size of the Method Decomposition instances. As presented in Figure 5.18, most cases refer to methods decomposed using two Extract operations (527 occurrences, 77%) or three operations (108 occurrences, 16%).

Table 5.7: Characteristics of Method Decomposition (in the wild)

Project	**Occur.**	**Intra-class**		**Inter-class**		**Mixed**	
		Occur.	**%**	**Occur.**	**%**	**Occur.**	**%**
Spring Boot	148	107	72	21	14	20	14
Elasticsearch	234	75	32	118	50	41	18
RxJava	5	2	40	3	60	0	0
Spring Framework	152	77	51	38	25	37	24
Guava	10	3	30	5	50	2	20
Retrofit	6	2	33	2	33	2	33
Dubbo	51	22	43	20	39	9	18
MPAndroidChart	35	5	14	25	71	5	14
Lottie Android	14	5	36	5	36	4	29
Guide	28	19	68	1	4	8	29
All	683	317	46	238	35	128	19

Figure 5.18: Number of operations by composite refactoring (Method Decomposition)



Method Composition. Among 582 instances of this composite refactoring, frequently, the extracted code is moved to distinct classes (345 occurrences, 59%), i.e., they are *inter-class*, as shown in Table 5.8. Figure 5.19 shows the results considering the size: as we can observe, most cases involve up to three operations (467 occurrences, 80%). Dubbo includes an outlier, in which a developer extracted a utility method called `isEmptyMap(Map)` from seven other methods.[16] The extracted method has the following code:

[16]`https://github.com/apache/dubbo/commit/458a4504`

```
public static boolean isEmptyMap(Map map) {
    return map == null || map.size() == 0;
}
```

Figure 5.19: Number of operations by composite refactoring (Method Composition)

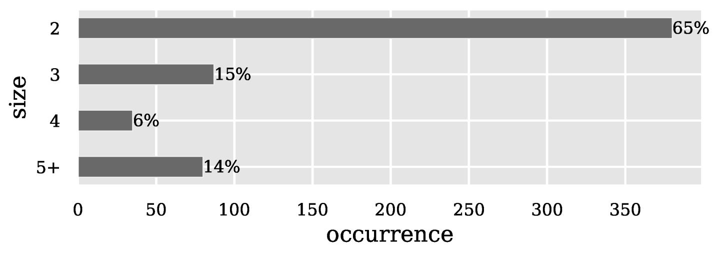


Table 5.8: Characteristics of Method Composition (in the wild)

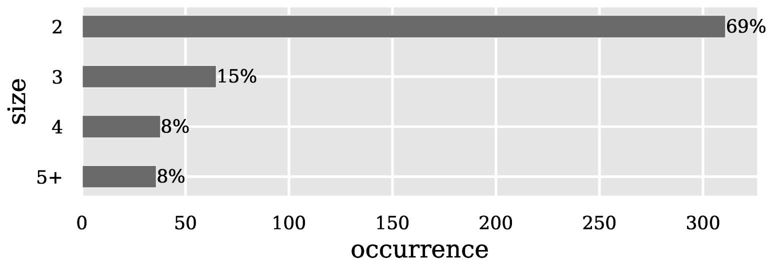
Project	**Occur.**	**Intra-class**		**Inter-class**		**Mixed**	
		Occur.	**%**	**Occur.**	**%**	**Occur.**	**%**
Spring Boot	79	34	43	42	53	3	4
Elasticsearch	219	49	22	161	74	9	4
RxJava	8	2	25	6	75	0	0
Spring Framework	151	67	44	66	44	18	12
Guava	17	5	29	10	59	2	12
Retrofit	5	1	20	4	80	0	0
Dubbo	39	20	51	15	38	4	10
MPAndroidChart	30	5	17	22	73	3	10
Lottie Android	12	1	8	10	83	1	8
Guide	22	13	59	9	41	0	0
All	582	197	34	345	59	40	7

Composite Pull Up Method. In this category, the number of operations follows the same tendency detected in RQ2, e.g., most cases comprise two (311 occurrences, 69%) or three operations (65 occurrences, 15%), as reported in Figure 5.20. However, 8% of the occurrences have five or more operations. Spring Boot includes an example, in which a developer moved method `matches(...)` from five subclasses to the superclass `SpringBootCondition`.[17] In the commit description, the developer mentioned his intention, which relates to the improvement of the inheritance hierarchy:

"Create common [name class] base class... This removes the need for [class name] and simplifies many of the existing condition implementations."

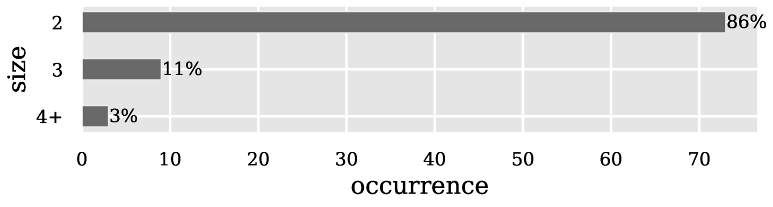
[17]`https://github.com/spring-projects/spring-boot/commit/840fdeb5`

Figure 5.20: Number of operations by composite refactoring (Composite Pull Up Method)



Composite Push Down Method. Figure 5.21 presents the results regarding the size of this type of composite refactoring. Overall, most cases comprise operations to move a method to at most three subclasses (82 occurrences, 97%).

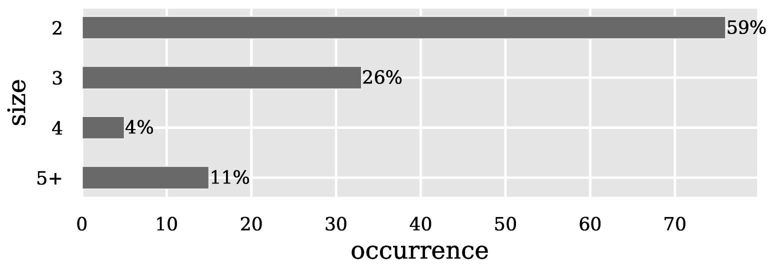
Figure 5.21: Size of composite refactorings (Composite Push Down Method)



Composite Inline Method. Regarding the number of affected elements, most Composite Inline operations involve two or three operations (109 occurrences, 85%), as presented in Figure 5.22. Elasticsearch includes a large instance, in which a developer removed method `cast(Input, Output)` by performing 23 Inline Method operations.[18] In the commit description, the developer explained his motivation in the following way:

"Remove [functionality name] from [class name] as mutable state... this is no longer necessary as each cast is only used directly in the semantic pass after its creation..."

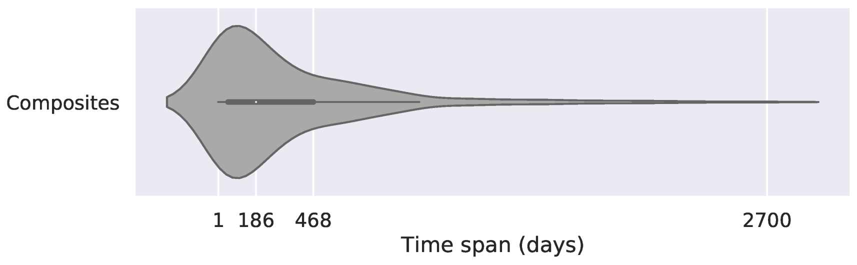
Figure 5.22: Size of composite refactorings (Composite Inline Method)



[18]`https://github.com/elastic/elasticsearch/commit/022d3d7d`

Time span of composites. In the oracle study (Section 5.5), we detect a single composite performed over multiple commits. However, in the wild study, a significant part of the composites are performed over time (448 instances, 15.5%). In these cases, we also assess time span by computing the number of days between the most recent and the oldest operation in a composite. Figure 5.23 shows the distribution of the results. As we can notice, there are composites performed in a single day (3.3%, 15 composites), but also there are composites performed over months. Among the 448 composite instances, 75% are performed up to 468 days (about 15 months), with a median time span of 186 days (approximately six months). The 90th percentile is 835 days. However, it is difficult to generalize these results. For example, open source projects are subjected to multiple periods of inactivity [28].

Figure 5.23: Distribution of the time span of composite refactorings performed over multiple commits (wild, 448 instances)



Summary of RQ4: In our extended dataset, most composite refactorings are also small, i.e., they are formed by two or three operations (2,258 composites, 78%). Regarding the scope of the operations, most Method Decomposition are *intra-class* (46%), while most Method Composition are *inter-class* (59%). In other words, when decomposing methods, developers usually extract them to their current classes. In contrast, when removing code duplication, developers frequently extract methods from multiple classes.

Comparison with the oracle results (RQ2): In the study described in Section 5.5, we also investigate composite characteristics. Regarding the size, the results are similar. For example, most composites have up to three operations (84% in the oracle vs 78% in the extended dataset). The notable difference between both datasets refers to composites over multiple commits. The oracle only contains selected refactoring instances, i.e., it does not cover the whole projects' history. Due to this fact, we identified a single composite over time, i.e., a composite performed over more than one commit. In contrast, in RQ4, we detect 448 composites spread over two or more commits (15.5%). Finally, regarding the refactorings' scope, the results also follow a similar tendency. For example, most Method Decomposition operations are *intra-class* in both samples. However, in the oracle, there is a higher frequency of *intra-class* operations (91% in the oracle vs 46% in the extended dataset).

5.7 Discussion and Implications

In this chapter, we proposed a catalog of eight composite refactorings, i.e., refactoring operations composed of simple code transformations. We used the refactoring graphs to illustrate and document the catalog. Three of these refactorings—Class Decomposition, Method Decomposition, and Method Composition—are new, in the sense they are not documented in Fowler's catalog. The other refactorings—Pull Up Method, Push Down Method, Pull Up Field, Push Down Field, and Inline Method—are also described in Fowler's catalog. However, we decided to include them in our catalog for two key reasons: (a) they imply the realization of multiple source code transformations that affect multiple program elements; (b) they are not properly detected by refactoring detection tools, such as RefactoringMiner [96] and RefDiff [87]. However, to avoid potential conflicts, for these instances, we added the prefix "Composite" in their names. Regarding their popularity, the three new composites—Class Decomposition, Method Decomposition, and Method Composition—represent about 77% of the results in the wild study (Section 5.6). In the oracle study, 88% of the instances refer to these new cases (Section 5.5). These values are highlighted in Table 5.6.

Essentially, the main contribution of our study is the catalog; the set of scripts to identify the described composite refactorings; and a new perspective of the well-known refactoring oracle proposed by Tsantalis and other researchers [93, 96].

We claim this contribution can have two practical implications. First, as usual, our catalog highlights the importance and existence of composite refactorings. In other words, a catalog is a fundamental artifact to promote and disseminate the usage of composite refactorings among software practitioners. In fact, our studies showed that developers rely on composite refactorings during maintenance tasks. Therefore, the catalog and oracle can contribute to increasing the usage and application of such refactorings.

As a second practical implication, we showed that composite refactorings are not properly identified by refactoring detection tools, such as RefactoringMiner [6, 95, 96] and RefDiff [22, 85, 87]. Typically, these tools detect the parts of composite refactorings as independent operations. For this reason, we decided to implement a set of scripts to detect the eight composite refactorings in our catalog. Consequently, we also claim the concept of composite refactoring can be used to improve the results of empirical software engineering studies on refactoring practices. Finally, our scripts and catalog can also help to improve the user experience provided by refactoring-aware code review tools [23], by supporting the detection of refactorings at a higher abstraction level.

5.8 Threats to Validity

Generalization of results. We characterized composite refactorings in terms of size and location. Our findings are based on a relevant oracle of refactoring operations and ten real-world Java systems hosted on GitHub. However, they—as common in empirical software engineering—cannot be generalized to other scenarios, such as closed software systems or other programming languages.

Catalog of composite refactorings. Our catalog includes eight composite refactorings, which describe a sequence of operations to compose or decompose source code elements. We acknowledge that the current version of our catalog is not complete and final. However, any refactoring catalog can increase over time due to new insights, research, and development demands. For example, the first catalog proposed by Fowler has 68 refactoring operations [35]. After 18 years, in the second edition of his book, he introduced fifteen new refactorings [36]. We also followed the idea of Fowler's book [35, 36], using a single and popular programming language to guide the documentation and to provide illustrative examples. As mentioned by the author is "*better to use a single language so they can get used to a consistent form of expression*".[19] In fact, we plan to extend our study in the future, by including, for example, composite refactorings at the package level. We also intend to explore other programming languages and refactoring types.

Detection of single refactorings. Before detecting composite refactorings, we first need to identify single operations. In our first study—described in Section 5.5—we rely on a well-known refactoring oracle, in which single refactoring instances were validated by multiple authors or tools [93, 96]. Therefore, our results are based on a trustworthy sample. For the second study—described in Section 5.6—we rely on RefDiff [87] to mine refactorings in ten popular projects. According to recent results, the precision of RefDiff is high, reaching 96.4% for Java [87]. In this second study, we also cleaned up the dataset, for example, we remove packages that are not part of the core system (e.g., *test*, *docs*, *sample*), and we removed constructors since they are essentially initialization structures. Finally, as natural during software evolution, commits can include temporary or unintentional operations, such as reverted commits due to test fails and experimental code. To mitigate this threat, we focus only on the main branch evolution.

Detection of composite refactorings. Regarding composite detection, we implement a set of scripts, as described in Section 5.5.1. The input comprises a list of single refactoring operations, including details such as path, refactoring type, and entities names. A possible threat is the possibility of errors in the implementation of our tool and parsers.

[19] `https://martinfowler.com/articles/refactoring-2nd-ed.html`

For the oracle analysis, we extract this information from textual data. We also rely on well-known Python libraries to mitigate this threat, e.g., retrieving the data by regex expression. Also, we inspected a sample of 28 composite refactorings to check the results (see details in Section 5.5.1), when we did not identify any error in the process of clustering refactoring operations as composites. Our verification included 160 single refactoring operations from the oracle created and curated by Tsantalis et al. [93, 95, 96]. Finally, we are making publicly available the datasets and scripts used to detect composite refactorings.

5.9 Final Remarks

We used *refactoring graphs* to document a catalog of composite refactorings. According to our definition, a composite refactoring can be spread in multiple commits. Our catalog includes eight instances that describe sequences of operations that compose or decompose program elements: Method Composition, Method Decomposition, Class Decomposition, Composite Pull Up Method, Composite Push Down Method, Composite Pull Up Field, Composite Push Down Field and Composite Inline Method.

In order to show that the proposed refactorings occur in real-world scenarios, we searched for occurrences of each instance in two datasets. First, we focus on a well-known refactoring oracle. In this first study, we identify that about 60% of the selected sample is part of a higher-level composite refactoring. Then, we mine the history of ten popular GitHub projects, in which we detected 2,886 instances of composite refactorings. The scripts are publicly available at `https://github.com/alinebrito/refactoring-graphs-thesis`

Chapter 6

Conclusion

This chapter concludes the thesis by discussing the main contributions and prospecting future works. Specifically, in Section 6.1, we present an overview of this thesis content. We summarize the results and highlight the main contributions in Section 6.2 while discussing future research in Section 6.3.

6.1 Thesis Recapitulation

Refactoring is a common and indispensable practice during software evolution. Developers frequently refactor their code for different proposals [70, 75, 86]. As a consequence, a large number of studies investigate refactoring practices in the last 30 years [1]. However, typically, prior literature does not rely on abstractions to extract, visualize, and understand refactoring operations performed over time, which is a relevant aspect for researchers and practitioners [4, 14]. In this context, we provided in this thesis a set of three major studies where we extensively analyzed scenarios involving refactoring practices over time. For this purpose, we relied on *refactoring graphs*, the proposed graph-based model.

In Chapter 2, we started by reinforcing the importance of refactoring practices on software development and evolution. Despite the relevance and vast literature on the subject, we pointed out possible gaps referring to refactoring operations performed over time. Essentially, we discussed the need for approaches and abstractions to describe not only single, but also sequential refactorings as well. Finally, we also confronted this thesis content with studies that investigate sets of refactorings. We compared our results with works on batches, composites, and refactoring comprehension.

Next, in Chapter 3, we initially defined *refactoring graphs*, with the purpose to understand refactoring operations performed over time. We also reported a large-scale investigation, in which we mine for occurrences of refactoring graphs in relevant Java and JavaScript projects. Overall, we observed that most instances are small, including up to four operations. However, we also identified large subgraphs, involving dozens of operations. Aiming to investigate such cases, we complemented our study with a qualitative analysis by asking developers about the reasons behind their operations. The results from the qualitative part reinforce findings from the recent literature, i.e., developers frequently refactor their code to fix bugs or improve existing features, and code design improvement. Regarding other characteristics, most refactoring graphs have up to three commits, span from few days to months, and they are heterogeneous, i.e., involving distinct types of refactoring. Therefore, we show the importance of investigating refactorings from another perspective, i.e., beyond individual operations. We also reinforced the usefulness of the proposed graph-based abstraction to visualize and mine refactorings performed over time.

Finally, in the subsequent chapters, we investigated applications for refactoring graphs. In other words, we conducted studies using our graph based-model as a key data structure to visualize, understand, and represent refactoring operations. First, in Chapter 4, we relied on refactoring graphs to evaluate refactoring tasks performed by undergraduate students from a Software Engineering course. Specifically, we performed a manual inspection and also used a graph-based metric to compute similarity. Therefore, we explored a first application to our model. It may be a useful data structure to understand and visualize refactoring tasks. Next, in Chapter 5, we investigated a second application for refactoring graphs by documenting sequences of refactorings performed over a given program element, i.e., a catalog of composite refactorings. We discussed the relevance of catalogs of refactorings to guide developers and research. Also, we point out limitations of existing catalogs and tools, which most cases focus on single refactoring operations. In general, we obtained interesting results. We show, for example, that a significant rate of operations identified as singles ones by refactoring detection tools, are actually composite operations. We also mined dozens of instances of composites in projects hosted on GitHub.

6.2 Contributions

We summarize our contributions as follows:

- We proposed a **new abstraction named refactoring graphs**, aiming to support studies on refactoring performed over time. Large, complex, and sequential refactoring tasks are a challenge reported by the recent literature [4]. We claim that the proposed graph-based model is a useful data structure for studies in such scenarios.

- Using our graph-based model, we quantitatively and qualitatively **characterized a large sample of approximately 1.5K sequences of refactoring operations performed over time** in two popular programming languages, Java and JavaScript. We showed characteristics referring to size, location, timespan, and graph patterns. We also investigate the main reasons behind the large graphs from our sample. The results suggest that there is another perspective, which does not involve only single refactoring operations. Moreover, we also reinforce the relevance of refactoring graphs to mine and represent operations performed over time.

- We **extend the existing datasets of refactoring with new instances or perspectives**. Specifically, we provide a new evaluation of the precision of RefDiff [85, 87], which is the tool we used to detect refactoring operations. The evaluation relies on real-world Java and JavaScript open-source projects, comprising 690 single refactoring instances. We also show a new perspective to one of the most relevant refactoring oracles, which has been curated by Tsantalis and other researchers [93, 95, 96]. In this case, we show that there is a significant rate of composite refactorings in this dataset, which are reported as single ones.

- We **complemented research on refactoring practices**, relying on refactoring graphs to investigate distinct fields of study. First, we complement studies on sequences of refactoring operations by proposing a catalog of composite refactorings. Refactoring catalogs are relevant instruments to guide developers and researchers. We mined real-world instances of the proposed catalog and we also reported implications regarding single and composite refactorings. Second, we complement research on the educational side by conducting studies with undergraduate students on a Software Engineering course. In this case, we discussed the need for approaches to teach, visualize, and evaluate refactoring practices. Third, we characterized a large sample of refactoring operations performed over time. We mined hundreds of sequences of refactoring operations, showing that when we focus on single operations, we do not contemplate the complete scenario.

- We provided a **research compendium** that is publicly available at `https://github.com/alinebrito/refactoring-graphs-thesis`. It includes a collection of scripts, metadata, and documentation referring to the results reported in this Ph.D. thesis, allowing, for instance, to detect refactoring graphs and visualize the results reported over the chapters.

- We designed and implemented **a web application to easily visualize refactoring graphs**: `https://refactoring-graph.github.io`. Therefore, we provide a visual option to support the comprehension of the results reported in this thesis. Essentially, the application allows us to visualize the refactoring graphs reported in Chapter 3.

6.3 Future Work

Throughout this thesis, we defined and characterized the proposed graph-based abstraction named *refactoring graphs*. We also presented applications involving the graph model. Specifically, we performed a study using refactoring graphs to understand refactoring tasks performed by students, and also a second investigation using refactoring graphs to document a catalog of composite refactorings. During the investigations, we envision new fields of study contemplating sequences of refactoring operations. We prospect about these topics in the following paragraphs.

Tactical Forking. Usually, companies work with multiple teams. Sometimes, they operate in the same core system, which is part of multiple systems in the company. This practice requires intense coordination. For example, developers could frequently deal with blocking contributions or issues due to changes performed by another team. Aiming to mitigate this effort during the software evolution, they can rely on Tactical Forkings.[1] Tactical Forking consists of creating an independent fork. In other words, the team decouples a version of the core system from the codebase and other copies. Then, they perform constant refactoring operations to eliminate and adapt the code to their personal needs. The goal is to eliminate or decompose unnecessary parts of the software system, creating a simpler and more compact one. This strategy can also support the migration from monolithic architectures to microservices. In this context, a vital part of the work involves applying sequences of refactoring operations after the fork from the codebase, which might be composite

[1] `faustodelatog.wordpress.com/2020/10/16/tactical-forking`

refactorings. However, to the best of our knowledge, guidelines to perform tactical forkings are nor deeply explored in the literature. In this way, as future work, we suggest an extension of the catalog of composite refactorings proposed in Chapter 5, focusing on sequences of operations common when performing tactical forkings.

Exploring Other Research Methodologies. This thesis includes qualitative analysis aiming to understand and show the relevance of refactoring graphs. Specifically, in Chapter 3, we sent emails to developers asking about the motivations behind large refactoring subgraphs. Also, in Chapter 4, we used refactoring graphs to visualize refactoring tasks performed by students, in which we show a first application for the proposed graph-based model. However, other methodologies could have been used in these studies, gathering more complete responses and new perspectives. For example, it is possible to conduct semi-structured interviews with developers, educators, and students regarding the use of refactoring graphs to understand complex refactoring tasks.

Refactoring understanding and visualization. In Chapter 4, we explored the usage of refactoring graphs to represent, visualize, and assess large refactoring tasks over time. Specifically, we investigate refactoring operations performed by undergraduate students from a Software Engineering course. We argue that the proposed graph-based model may be used to provide easy navigation on large refactoring operations, since it is possible, for example, to inspect entity names, visualize sequences of refactoring tasks, and organize multiple operations. In fact, visualization tools and techniques are frequently reported as useful by practitioners [4, 10, 24, 60, 68, 100]. The literature also shows that cognitive information is helpful in educational activities [68, 100, 101, 103]. In other words, visual perception improves code comprehension tasks. In the particular context of refactoring, for example, mining and managing complex refactoring talks is a hurdle recently pointed out in the literature [4].

Ecosystems and Programming Languages. Most studies on refactorings focus on the Java programming language, ranging from empirical investigations [12, 75, 86, 89] to refactoring tools [16, 85, 96]. However, the software industry has been adopting other popular languages, such as JavaScript, Python, and Go.[2] As a consequence, we observe an effort in recent tools and techniques to support refactoring performed in multiple programming languages [6, 22, 87]. We notice the same in academia. For example, the current version of Fowler's book uses JavaScript to describe the catalog of refactoring operations [36]. In this PhD thesis, specifically in Chapter 3, we explore refactoring performed over time in Java and JavaScript, com-

[2]`survey.stackoverflow.co/2022`

plementing this gap in research using dynamic and untyped programming languages. Still, we envision new studies to complement the perspectives of this thesis content. Therefore, we suggest investigating applications of refactoring graphs beyond Java, as well as studies about the frequency and impact of refactoring performed over time in distinct environments.

Source Code Histories. Source code history is a relevant resource to investigate refactoring development practices, understand the evolution of a piece of code or element [43], and it is also used by software mining tools and techniques [30, 72, 90]. In current version control systems (VCS), such as `git`, it may be harder to understand performed changes, since they trace and show the modifications at the line level. Sometimes, refactoring also impacts the process by breaking the code element history. For example, a Move Method operation makes tracking more difficult, since a GitHub diff does not precisely identify the movement. It reports the removal of a method and the addition of a new one. Due to this fact, recent studies introduce new strategies to track code changes. Among them, there are `CodeShovel` [43] and `FinerGit` [47], which track the changes at the method level. In this thesis, we envision the use of our graph-based model to overcome a limitation on these approaches. Specifically, the use of refactoring graphs to solve issues regarding the creation of method histories as a sequential line. `CodeShovel` and `FinerGit` do not cover the whole scenario. For example, it is possible to extract multiple methods from a single one. Therefore, for multiple instances, the history of a method is not a sequential line. Notice that this is not a particular behavior of extract operations: as mentioned in Chapter 5, other refactorings also produce multiple elements. Interestingly, a recent tool called `CodeTracker` [52] solved this issue by relying on a graph-based design, which reinforces our observations. However, it is still possible to improve these solutions, considering other fine-grained code histories, such as refactoring at field level and internal refactorings.

Changes in code blocks. The graph model defined in Chapter 3 focuses on method level, i.e., refactoring operations whose the target and the source are methods. We also discussed the extension to other code elements, such classes and fields. In fact, it was performed when documenting a catalog of composite refactorings in Chapter 5. However, refactoring graphs do not provide a useful representation to internal operations, i.e., refactorings inside the same code element. For instance, Replace Temp with Query—a refactoring cataloged in Fowler's books—involves extracting a method and replacing a temporary variable by calling an extracted method. Similarly, other changes happen inside methods, involving mixed internal and external movements. For example, the current version of RefactoringMiner [96] detects refactorings involving loops. Therefore, we also suggest investigations regarding new graph-based designs or structures to represent these categories of code changes.

Bibliography

[1] Abid, C., Alizadeh, V., Kessentini, M., Ferreira, T., and Dig, D. 30 years of software refactoring research: A systematic literature review. *ArXiv*, abs/2007.02194, 2020.

[2] Abid, S., Abdul Basit, H., and Arshad, N. Reflections on teaching refactoring: A tale of two projects. In *20th ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE)*, page 225–230, 2015.

[3] Agrahari, V. and Chimalakonda, S. Refactor4green: A game for novice programmers to learn code smells. In *42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 324–325, 2020.

[4] AlOmar, E., Mkaouer, M., and Ouni, A. *Knowledge Management in the Development of Data-Intensive Systems*, chapter Mining and Managing Big Data Refactoring for Design Improvement: Are We There Yet? Auerbach Publications, 2021.

[5] AlOmar, E. A., Mkaouer, M. W., and Ouni, A. Toward the automatic classification of self-affirmed refactoring. *Journal of Systems and Software (JSS)*, 171:110821, 2021.

[6] Atwi, H., Lin, B., Tsantalis, N., Kashiwa, Y., Kamei, Y., Ubayashi, N., Bavota, G., and Lanza, M. PYREF: Refactoring detection in python projects. In *21st IEEE International Working Conference on Source Code Analysis and Manipulation,Engineering Track (SCAM)*, pages 136–141, 2021.

[7] Avelino, G., Passos, L., Hora, A., and Valente, M. T. A novel approach for estimating truck factors. In *24th International Conference on Program Comprehension (ICPC)*, pages 1–10, 2016.

[8] Bacchelli, A. and Bird, C. Expectations, outcomes, and challenges of modern code review. In *35th International Conference on Software Engineering (ICSE)*, pages 712–721, 2013.

[9] Bacchelli, A. and Bird, C. Expectations, outcomes, and challenges of modern code review. In *35th International Conference on Software Engineering (ICSE)*, pages 712–721, 2013.

[10] Bassil, S. and Keller, R. K. Software visualization tools: survey and analysis. In *9th International Workshop on Program Comprehension (IWPC)*, pages 7–17, 2001.

[11] Bibiano, A., Soares, V., Coutinho, D., Fernandes, E., Correia, J., Santos, K., Oliveira, A., Garcia, A., Gheyi, R., BaldoinoFonseca, Ribeiro, M., Silva, C., and Oliveira, D. How does incomplete composite refactoring affect internal quality attributes. In *28th International Conference on Program Comprehension (ICPC)*, pages 149–159, 2020.

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

[13] Bibiano, A. C., Assunção, W., Coutinho, D., Santos, K., Soares, V., Gheyi, R., Garcia, A., Fonseca, B., Ribeiro, M., Daniel Oliveira, C. B., Marques, J. L., and Oliveira, A. Look ahead! revealing complete composite refactorings and their smelliness effects. In *37th International Conference on Software Maintenance and Evolution (ICSME)*, pages 298–308, 2021.

[14] Bogart, A., AlOmar, E. A., Mkaouer, M. W., and Ouni, A. Increasing the trust in refactoring through visualization. In *42nd International Conference on Software Engineering Workshops (ICSEW)*, pages 334–341, 2020.

[15] Borges, H., Hora, A., and Valente, M. T. Understanding the factors that impact the popularity of GitHub repositories. In *32nd International Conference on Software Maintenance and Evolution (ICSME)*, pages 334–344, 2016.

[16] Brito, A., Xavier, L., Hora, A., and Valente, M. T. APIDiff: Detecting API breaking changes. In *25th International Conference on Software Analysis, Evolution and Reengineering (SANER), Tool Track*, pages 507–511, 2018.

[17] Brito, A., Hora, A., and Valente, M. T. Refactoring graphs: Assessing refactoring over time. In *27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 367–377, 2020. doi: 10.1109/SANER48275.2020.9054864.

[18] Brito, A., Valente, M. T., Xavier, L., and Hora, A. You broke my code: Understanding the motivations for breaking changes in APIs. *Empirical Software Engineering*, 25:1458–1492, 2020. doi: 10.1007/s10664-019-09756-z.

[19] Brito, A., Hora, A., and Valente, M. T. Characterizing refactoring graphs in Java and JavaScript projects. *Empirical Software Engineering*, 26, 2021. doi: 10.1007/s10664-021-10023-3.

[20] Brito, A., Hora, A., and Valente, M. T. Understanding refactoring tasks over time: A study using refactoring graphs. In *25th Ibero-American Conference on Software Engineering (CIbSE)*, pages 330–344, 2022. doi: 10.5753/cibse.2022.20982.

[21] Brito, A., Hora, A., and Valente, M. T. Towards a catalog of composite refactorings. *Journal of Software: Evolution and Process*, e2530, 2023. doi: 10.1002/smr.2530.

[22] Brito, R. and Valente, M. T. RefDiff4Go: Detecting refactorings in Go. In *14th Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS)*, pages 101–110, 2020.

[23] Brito, R. and Valente, M. T. RAID - Refactoring aware and intelligent diffs. In *29th International Conference on Program Comprehension (ICPC)*, pages 265–275, 2021.

[24] Brito, R., Brito, A., Brito, G., and Valente, M. T. GoCity: Code city for Go. In *26th International Conference on Software Analysis, Evolution and Reengineering (SANER), Tool Track*, pages 649–653, 2019.

[25] Catolino, G., Palomba, F., Tamburri, D. A., Serebrenik, A., and Ferrucci, F. Refactoring community smells in the wild: The practitioner's field manual. In *42nd International Conference on Software Engineering: Companion Proceedings (ICSE)*, pages 25–34, 2020.

[26] Cedrim, D. *Understanding and improving batch refactoring in software systems.* PhD thesis, PUC-Rio, 2018.

[27] Chen, T.-H., Nagappan, M., Shihab, E., and Hassan, A. E. An empirical study of dormant bugs. In *11th Working Conference on Mining Software Repositories (MSR)*, pages 82–91, 2014.

[28] Coelho, J., Valente, M. T., Milen, L., and Silva, L. L. Is this GitHub project maintained? measuring the level of maintenance activity of open-source projects. *Information and Software Technology*, 1:1–35, 2020.

[29] Cruzes, D. S. and Dyba, T. Recommended steps for thematic synthesis in software engineering. In *5th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 275–284, 2011.

[30] da Costa, D. A., McIntosh, S., Shang, W., Kulesza, U., Coelho, R., and Hassan, A. E. A framework for evaluating the results of the SZZ approach for identifying bug-introducing changes. *Transactions on Software Engineering*, 43(7):641–657, 2017.

[31] Demeyer, S., Van Rysselberghe, F., Girba, T., Ratzinger, J., Marinescu, R., Mens, T., Du Bois, B., Janssens, D., Ducasse, S., Lanza, M., Rieger, M., Gall, H., and El-Ramly, M. The LAN-simulation: a refactoring teaching example. In *8th International Workshop on Principles of Software Evolution (IWPSE)*, pages 123–131, 2005.

[32] Di Penta, M., Bavota, G., and Zampetti, F. On the relationship between refactoring actions and bugs: A differentiated replication. In *28th European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE)*, pages 556–567, 2020.

[33] Dig, D., Comertoglu, C., Marinov, D., and Johnson, R. Automated detection of refactorings in evolving components. In *20th European Conference on Object-Oriented Programming (ECOOP)*, pages 404–428, 2006.

[34] El-Ramly, M. Experience in teaching a software reengineering course. In *28th International Conference on Software Engineering (ICSE)*, page 699–702, 2006.

[35] Fowler, M. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[36] Fowler, M. *Refactoring: improving the design of existing code*. Addison-Wesley, 2018.

[37] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[38] Ge, X., Sarkar, S., Witschey, J., and Murphy-Hill, E. Refactoring-aware code review. In *Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 71–79, 2017.

[39] Ge, X., Sarkar, S., and Murphy-Hill, E. Towards refactoring-aware code review. In *7th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pages 99–102. ACM, 2014.

[40] Ge, X., Sarkar, S., Witschey, J., and Murphy-Hill, E. Refactoring-aware code review. In *Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 71–79, 2017.

[41] Gómez, V. U., Ducasse, S., and D'Hondt, T. Visually supporting source code changes integration: the Torch dashboard. In *17th Working Conference on Reverse Engineering (WCRE)*, pages 55–64, 2010.

[42] Gómez, V. U., Ducasse, S., and D'Hondt, T. Visually characterizing source code changes. *Science of Computer Programming*, 98(P3):376–393, 2015.

[43] Grund, F., Chowdhury, S., Bradley, N., Hall, B., and Holmes, R. CodeShovel: Constructing method-level source code histories. In *43rd International Conference on Software Engineering: Companion Proceedings (ICSE)*, pages 1510–1522, 2021.

[44] Hattori, L. and Lanza, M. Mining the history of synchronous changes to refine code ownership. In *6th International Working Conference on Mining Software Repositories (MSR)*, pages 141–150, 2009.

[45] Hayashi, S., Thangthumachit, S., and Saeki, M. Rediffs: Refactoring-aware difference viewer for Java. In *20th Working Conference on Reverse Engineering (WCRE)*, pages 487–488, 2013.

[46] Hebig, R., Ho-Quang, T., Jolak, R., Schröder, J., Linero, H., Ågren, M., and Maro, S. H. How do students experience and judge software comprehension techniques? In *28th International Conference on Program Comprehension (ICPC)*, page 425–435, 2020.

[47] Higo, Y., Hayashi, S., and Kusumoto, S. On tracking Java methods with git mechanisms. *Journal of Systems and Software (JSS)*, 165, 2020.

[48] Hora, A. and Robbes, R. Characteristics of method extractions in java: A large scale empirical study. *Empirical Software Engineering*, 25:1798–1833, 2020.

[49] Hora, A., Silva, D., Robbes, R., and Valente, M. T. Assessing the threat of untracked changes in software evolution. In *40th International Conference on Software Engineering (ICSE)*, pages 1102–1113, 2018.

[50] Iammarino, M., Zampetti, F., Aversano, L., and Penta, M. D. Self-admitted technical debt removal and refactoring actions: Co-occurrence or more? In *35th International Conference on Software Maintenance and Evolution (ICSME)*, pages 186–190, 2019.

[51] Jiang, Y., Liu, H., Niu, N., Zhang, L., and Hu, Y. Extracting concise bug-fixing patches from human-written patches in version control systems. In *43rd International Conference on Software Engineering (ICSE)*, pages 686–698, 2021.

[52] Jodavi, M. and Tsantalis, N. Accurate method and variable tracking in commit history. In *30th Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE)*, pages 183–195, 2022.

[53] Karac, E. I., Turhan, B., and Juristo, N. A controlled experiment with novice developers on the impact of task description granularity on software quality in test-driven development. *IEEE Transactions on Software Engineering (TSE)*, pages 1–16, 2019.

[54] Keuning, H., Heeren, B., and Jeuring, J. Student refactoring behaviour in a programming tutor. In *20th Koli Calling International Conference on Computing Education Research (Koli Calling)*, pages 1–10, 2020.

[55] Kim, M., Gee, M., Loh, A., and Rachatasumrit, N. Ref-finder: a refactoring reconstruction tool based on logic query templates. In *8th International Symposium on Foundations of software engineering (FSE)*, pages 371–372, 2010.

[56] Kim, M., Zimmermann, T., and Nagappan, N. A field study of refactoring challenges and benefits. In *20th International Symposium on the Foundations of Software Engineering (FSE)*, pages 50:1–50:11, 2012.

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

[58] Kim, S., Zimmermann, T., Pan, K., and Whitehead, E. J. J. Automatic identification of bug-introducing changes. In *21st International Conference on Automated Software Engineering (ASE)*, pages 81–90, 2006.

[59] Kononenko, O., Baysal, O., and Godfrey, M. W. Code review quality: How developers see it. In *38th International Conference on Software Engineering (ICSE)*, pages 1028–1038, 2016. doi: 10.1145/2884781.2884840.

[60] Koschke, R. Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey. *Journal of Software Maintenance and Evolution: Research and Practice (JSME)*, 15(2):87–109, 2003.

[61] Leung, C. Technical notes on extending gSpan to directed graphs. Technical report, Singapore Management University, 2010.

[62] Li, H. and Thompson, S. A domain-specific language for scripting refactorings in erlang. In *15th Fundamental Approaches to Software Engineering (FASE)*, pages 501–515, 2012.

[63] López, C., Alonso, J. M., Marticorena, R., and Maudes, J. M. Design of e-activities for the learning of code refactoring tasks. In *2014 16th International Symposium on Computers in Education (SIIE)*, pages 35–40, 2014. doi: 10.1109/SIIE.2014.7017701.

[64] Mahmoudi, M., Nadi, S., and Tsantalis, N. Are refactorings to blame? an empirical study of refactorings in merge conflicts. In *26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 151–162, 2019.

[65] Mazinanian, D., Ketkar, A., Tsantalis, N., and Dig, D. Understanding the use of lambda expressions in Java. *Programming Languages*, 1(85):85:1–85:31, 2017.

[66] Meananeatra, P. Identifying refactoring sequences for improving software maintainability. In *27th International Conference on Automated Software Engineering (ASE)*, pages 406–409, 2012.

[67] Meneely, A. and Williams, O. Interactive churn metrics: socio-technical variants of code churn. *Software Engineering Notes*, 37(6), 2012.

[68] Merino, L., Ghafari, M., Anslow, C., and Nierstrasz, O. A systematic literature review of software visualization evaluation. *Journal of Systems and Software (JSS)*, 144:165–180, 2018.

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

[70] Murphy-Hill, E., Parnin, C., and Black, A. P. How we refactor, and how we know it. In *31st International Conference on Software Engineering (ICSE)*, pages 287–297, 2009.

[71] Negara, S., Chen, N., Vakilian, M., Johnson, R. E., and Dig, D. A comparative study of manual and automated refactorings. In *27th European Conference on Object-Oriented Programming (ECOOP)*, pages 552–576, 2013.

[72] Neto, E. C. and da Costa andUirá Kulesza, D. A. The impact of refactoring changes on the SZZ algorithm: An empirical study. In *25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 380–390, 2018.

[73] Paixao, M., Uchôa, A., Bibiano, A. C., Oliveira, D., Garcia, A., Krinke, J., and Arvonio, E. Behind the intents: An in-depth empirical study on software refactoring in modern code review. In *17th International Conference on Mining Software Repositories (MSR)*, pages 125–136, 2020.

[74] Palomba, F., Zaidman, A., Oliveto, R., and Lucia, A. D. An exploratory study on the relationship between changes and refactoring. In *25th International Conference on Program Comprehension (ICPC)*, pages 176–185, 2017.

[75] Pantiuchina, J., Zampetti, F., Scalabrino, S., Piantadosi, V., Oliveto, R., Bavota, G., and Penta, M. D. Why developers refactor source code: A mining-based study. *ACM Transactions on Software Engineering and Methodology*, 37(4):1–32, 2020.

[76] Peruma, A., Mkaouer, M., Decker, M., and Newman, C. An empirical investigation of how and why developers rename identifiers. In *2nd International Workshop on Refactoring (IWoR)*, pages 26–33, 2018.

[77] Rahman, F. and Devanbu, P. Ownership, experience and defects: a fine-grained study of authorship. In *33rd International Conference on Software Engineering (ICSE)*, pages 491–500, 2011.

[78] Rahman, F., Posnett, D., Hindle, A., Barr, E., and Devanbu, P. BugCache for inspections: hit or miss? In *19th International Symposium on the Foundations of Software Engineering (FSE)*, pages 322–331, 2011.

[79] Ray, B., Hellendoorn, V., Godhane, S., Tu, Z., Bacchelli, A., and Devanbu, P. On the naturalness of buggy code. In *38th International Conference on Software Engineering (ICSE)*, pages 428–439, 2016.

[80] Sadowski, C., Söderberg, E., Church, L., Sipko, M., and Bacchelli, A. Modern code review: A case study at Google. In *40th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 181–190, 2018.

[81] Sanfeliu, A. and Fu, K.-S. A distance measure between attributed relational graphs for pattern recognition. *IEEE Transactions on Systems, Man, and Cybernetics (SMC)*, SMC-13(3):353–362, 1983.

[82] Sellitto, G., Iannone, E., Codabux, Z., Lenarduzzi, V., Lucia, A. D., Palomba, F., and Ferrucci1, F. Toward understanding the impact of refactoring on program comprehension. In *29th International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 731–742, 2022.

[83] Shen, B., Zhang, W., Zhao, H., Liang, G., Jin, Z., and Wang, Q. IntelliMerge: A refactoring-aware software merging technique. *Programming Languages*, 3(170): 170:1–170:28, 2019.

[84] Shen, B., Zhang, W., Zhao, H., Liang, G., Jin, Z., and Wang, Q. IntelliMerge: A refactoring-aware software merging technique. *Programming Languages*, 3(170): 170:1–170:28, 2019.

[85] Silva, D. and Valente, M. T. RefDiff: Detecting refactorings in version histories. In *14th International Conference on Mining Software Repositories (MSR)*, pages 269–279, 2017.

[86] Silva, D., Tsantalis, N., and Valente, M. T. Why we refactor? Confessions of GitHub contributors. In *24th International Symposium on the Foundations of Software Engineering (FSE)*, pages 858–870, 2016.

[87] Silva, D., da Silva, J. P., Santos, G., Terra, R., and Valente, M. T. Refdiff 2.0: A multi-language refactoring detection tool. *IEEE Transactions on Software Engineering*, 47(12):2786–2802, 2021.

[88] Silva, H. and Valente, M. T. What's in a GitHub star? Understanding repository starring practices in a social coding platform. *Journal of Systems and Software*, 146:112–129, 2018.

[89] Sousa, L., Cedrim, D., Garcia, A., Oizumi, W., Bibiano, A. C., Oliveira, D., Kim, M., and Oliveira, A. Characterizing and identifying composite refactorings: Concepts, heuristics and patterns. In *17th International Conference on Mining Software Repositories (MSR)*, page 186–197, 2020.

[90] Spadini, D., Aniche, M., and Bacchelli, A. PyDriller: Python framework for mining software repositories. In *26th Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE)*, pages 908–911, 2018.

[91] Spinellis, D. A repository of Unix history and evolution. *Empirical Software Engineering*, 22(3):1372–1404, 2017.

[92] Stoecklin, S., Smith, S., and Serino, C. Teaching students to build well formed object-oriented methods through refactoring. In *38th Technical Symposium on Computer Science Education (SIGCSE)*, pages 145–149, 2007. doi: 10.1145/1227310.1227364.

[93] Tsantalis, N., Mansouri, M., Eshkevari, L., Mazinanian, D., and Ketkar, A. Refactoring oracle. `http://refactoring.encs.concordia.ca/oracle`, 2022. Online; accessed January 2022.

[94] Tsantalis, N., Guana, V., Stroulia, E., and Hindle, A. A multidimensional empirical study on refactoring activity. In *23th Conference of the Center for Advanced Studies on Collaborative Research (CASCON)*, pages 132–146, 2013.

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

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

[97] Valente, M. T. *Engenharia de Software Moderna: Princípios e Práticas para Desenvolvimento de Software com Produtividade.* Independente, 2022.

[98] Vassallo, C., Grano, G., Palomba, F., Gall, H., and Bacchelli, A. A large-scale empirical exploration on refactoring activities in open source software projects. *Science of Computer Programming*, 180:1–15, 2019.

[99] Wang, Y. What motivate software engineers to refactor source code? evidences from professional developers. In *International Conference on Software Maintenance (ICSM)*, pages 413–416, 2009.

[100] Wettel, R., Lanza, M., and Robbes, R. Software systems as cities: a controlled experiment. In *33rd International Conference on Software Engineering (ICSE)*, pages 551–560, 2011.

[101] Xie, S., Kraemer, E., and Stirewalt, R. E. K. Empirical evaluation of a uml sequence diagram with adornments to support understanding of thread interactions. In *15th International Conference on Program Comprehension (ICPC)*, pages 123–134, 2007.

[102] Xifeng Yan and Jiawei Han. gSpan: graph-based substructure pattern mining. In *2nd International Conference on Data Mining (ICDM)*, pages 721–724, 2002.

[103] Yi Ding, Yongmin Hang, Gang Wan, and Shuiyan He. Application of software visualization in programming teaching. In *9th International Conference on Computer Science Education (ICCSE)*, pages 803–806, 2014.

[104] Zimmermann, T., Kim, S., Zeller, A., and Whitehead, E. J., Jr. Mining version archives for co-changed lines. In *3rd International Workshop on Mining Software Repositories (MSR)*, pages 72–75, 2006.