# Refactoring Graphs: Assessing Refactoring over Time

Aline Brito, Andre Hora, Marco Tulio Valente
ASERG Group, Department of Computer Science (DCC), Federal University of Minas Gerais, Brazil
{alinebrito, andrehora, mtov}@dcc.ufmg.br

***Abstract*—Refactoring is an essential activity during software evolution. Frequently, practitioners rely on such transformations to improve source code maintainability and quality. As a consequence, this process may produce new source code entities or change the structure of existing ones. Sometimes, the transformations are atomic, i.e., performed in a single commit. In other cases, they generate sequences of modifications performed over time. To study and reason about refactorings over time, in this paper, we propose a novel concept called refactoring graphs and provide an algorithm to build such graphs. Then, we investigate the history of 10 popular open-source Java-based projects. After eliminating trivial graphs, we characterize a large sample of 1,150 refactoring graphs, providing quantitative data on their size, commits, age, refactoring composition, and developers. We conclude by discussing applications and implications of refactoring graphs, for example, to improve code comprehension, detect refactoring patterns, and support software evolution studies.**

***Index Terms*—Refactoring, Refactoring graphs, Mining software repositories, Software evolution.**

## I. Introduction

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, shedding light on aspects such as usage of refactoring engines [1], [2], documentation of refactorings using commit messages [1], motivations for performing refactorings [3]–[5], benefits and challenges of refactoring [6], [7], among many others.

However, *time* seems to be an underinvestigated dimension in refactoring studies. The notable exception are studies on refactoring tactics, particularly on repeated refactoring operations, often called *batch* refactorings. For example, Murphy-Hill *et al.* [1] define batch refactorings as operations that execute within 60 seconds of each another. They report that 40% of refactorings performed using a refactoring tool occur 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*". Bibiano *et al.* [8] investigate the characteristics and impact of batch refactorings on code elements affected by smells. The authors rely on a heuristic to retrieve batches [9], which groups refactorings performed by the same author in a single code element. Thus, their heuristic focus on single methods or classes, most of the cases resulting in batches with a single commit (93%).

Interestingly, in his seminal book on refactoring [10], Fowler 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, i.e., "*refactorings take time [to be concluded]*". However, to our knowledge, refactorings performed over long time windows are not deeply studied by the literature.

Therefore, we propose and evaluate a novel concept, called **refactoring graphs**, to study and reason about refactoring activities over time. In such graphs, the nodes are methods and the edges represent refactoring operations. For example, suppose that a method $foo()$ is renamed to $bar()$. This operation is represented by two nodes, $foo()$ and $bar()$, and one edge connecting them. After this first refactoring, suppose that a method $qux()$ is extracted from $bar()$. As a result, an edge connecting $bar()$ to a new node, representing $qux()$, is also added to the graph. Furthermore, refactoring graphs do not impose time constraints between the represented refactoring operations. In our example, the extract operation, for instance, can be performed months after the rename. Finally, refactoring graphs may also express refactorings performed by different developers. In our example, the rename can be performed by $d_1$ and the extract by another developer $d_2$.

We formalize an algorithm to build refactoring graphs and use it to to extract graphs for 10 well-known and popular open-source Java-based projects. Our goal is to characterize refactoring subgraphs to better understand this scenario. Thus, after removing refactoring graphs coming from a single commit (since our goal is to investigate refactorings over time), we answer five research questions about the following properties:

- Size (RQ1): most refactoring graphs have at most four nodes (85%) and three edges (83%). However, we also found graphs with 57 nodes and 61 edges.
- Commits (RQ2): Most refactoring subgraphs are generated from two or three commits (95%).
- Age (RQ3): The age of the refactoring subgraphs ranges from a few days to weeks or even months. For instance, 67% of the subgraphs have more than one month.
- Refactoring composition (RQ4): Most refactoring subgraphs include more than one refactoring type (72%).
- Developers (RQ5): Most refactoring subgraphs are created by a single developer (60%). However, a relevant amount (40%) is created by multiple developers.

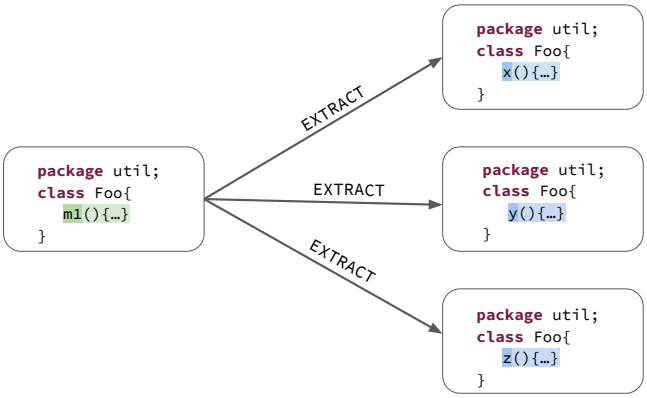


Fig. 1. Refactoring subgraph produced by only one developer

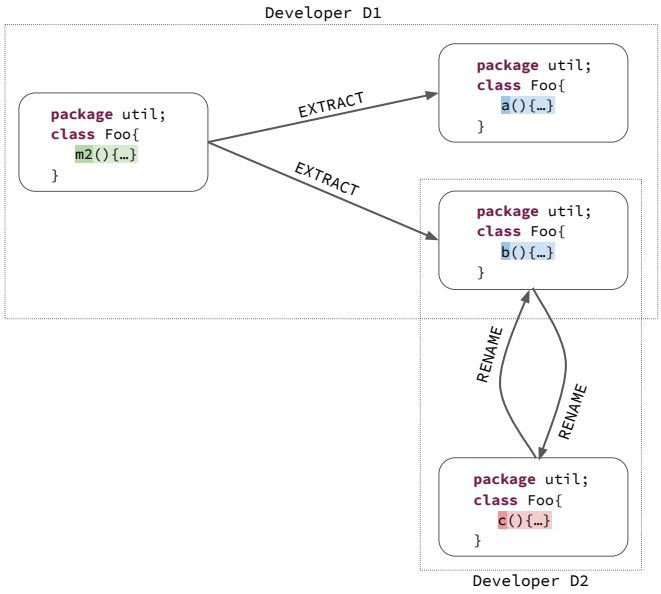


Fig. 2. Refactoring subgraph over time

Our main contributions are threefold. *First*, we propose and formalize the notion of refactoring graphs, which can be used to study and reason about refactorings performed over any time window by multiple developers. *Second*, we reveal several properties of a large sample of 1,150 refactoring graphs extracted for 10 real software projects. *Third*, we discuss several applications and implications of refactoring graphs to expand current refactoring tools, improve code comprehension, detect refactoring patterns, and support software evolution studies.

*Structure:* Section II defines our concept of refactoring graphs. Section III describes the design of our study, while Section IV shows the results. Section V shows an example of a large refactoring subgraph. We discuss the key applications and implications in Section VI. Section VII states threats to validity and Section VIII presents related work. Finally, we conclude the paper in Section IX.

## II. Refactoring Graphs

A *refactoring graph* $G$ is a set of disconnected subgraphs $G' = (V', E')$. Each $G'$ is called a *refactoring subgraph*, with a set of vertices $V'$ and a set of directed edges $E'$. In this way, the history of a software system includes a set of refactoring subgraphs. In refactoring (sub)-graphs, the vertices are the full signature of methods. For instance, we labeled a method $m()$ in class $Foo$ and package $util$ as $util.Foo\#m()$. Finally, the edge indicates the refactoring type (e.g., *move method*) and it also includes meta-data about the operation (e.g., author name and date).

Figure 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 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 code reviewer might have suggested to keep the original name. Thus, the developer undoes the latest refactoring, renaming $c()$ to $b()$ again. In this case, the graph contains refactorings performed by two authors. Besides, there is a cycle when the developer reverts the method to the original name.

As presented in Figure 3, we center our study on eight refactorings at the method level. *Rename* and *move* are the most trivial operations since they involve just changing the method's signature. 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. Extract operations generate new methods in the same class (i.e., they create a new node in our subgraphs). It is possible to extract a method $m()$ or multiple methods $m_i$ from a single method $m1()$. However, as also illustrated in Figure 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 $m_i$. Finally, we consider a refactoring called *extract and move* that extracts a method to a distinct class.

## III. Study Design

### A. Selecting Java Projects

We analyze the characteristics and frequency of refactoring subgraphs in popular software systems. We select 10 popular Java projects in terms of stars on GitHub, since stars is a key metric to reveal the popularity of repositories [11], [12]. We also confine our analysis to projects with more than 1K commits and more than 100 Java files to avoid young and small systems. Table I describes the selected projects, including basic information, such as number of stars, commits, files, contributors, latest version, and

TABLE I
SELECTED JAVA PROJECTS

| Project | Stars | Forks | Commits | Contributors | Java Files | Latest Version | Description |
|---|---|---|---|---|---|---|---|
| Elasticsearch | 44,489 | 14,930 | 48,313 | 1,273 | 11,770 | 7.3.2 | Search engine for cloud systems |
| RxJava | 40,622 | 6,825 | 5,581 | 237 | 1,666 | 3.0.0-RC3 | Library for asynchronous communications |
| Square Okhttp | 34,484 | 7,521 | 4,273 | 189 | 167 | 4.2.0 | HTTP client |
| Square Retrofit | 33,801 | 6,254 | 1,756 | 129 | 241 | 2.6.2 | HTTP client |
| Spring Framework | 32,582 | 21,226 | 19,752 | 396 | 7,203 | 5.2.0 | Framework for web aplications |
| Apache Dubbo | 29,353 | 19,256 | 3,639 | 249 | 1,743 | 2.7.3 | RPC framework |
| MPAndroidChart | 28,647 | 7,424 | 2,018 | 66 | 220 | 3.1.0 | Library to create charts |
| Glide | 27,289 | 5,025 | 2,416 | 102 | 647 | 4.10.0 | Library to load imagens |
| Lottie Android | 26,952 | 4,278 | 1,139 | 76 | 198 | 3.0.7 | Library to parser animations |
| Facebook Fresco | 15,870 | 3,595 | 2,158 | 170 | 985 | 2.0.0 | Library to display images |

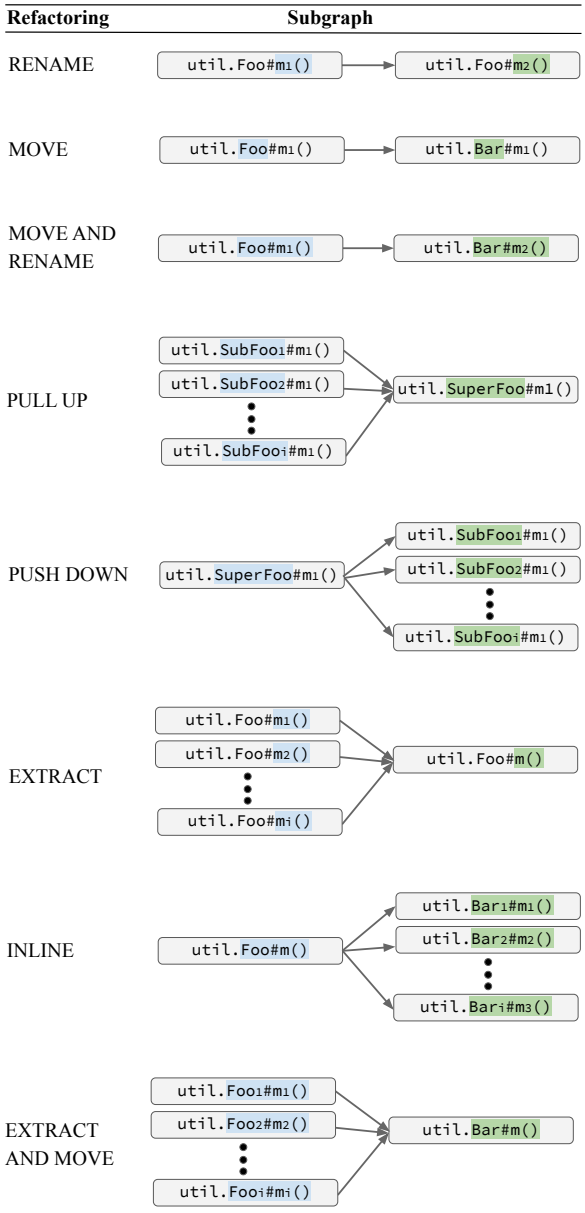


Fig. 3. Example of refactoring subgraphs

description. These projects cover distinct domains, including web development systems and media processing libraries, for example. The most popular project is Elasticsearch (44,489 stars). The number of forks ranges from 3,595 (Facebook Fresco) to 21,226 (Spring Framework). The number of commits ranges from 1,139 (Lottie Android) to 48,313 (Elasticsearch), while the number of contributors varies from 66 (MPAndroidChart) to 1,273 (Elasticsearch). Square Okhttp is the smallest system (167 files); and Elasticsearch is the largest one (11,770 files).

### B. Detecting Refactoring Operations

We use REFDIFF [13] to detect the refactoring operations needed to build 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 level (i.e., rename, move, extract, inline, pull up, and push down, as presented in Figure 3).

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.[1] Additionally, we remove refactorings in packages with the keywords *test(s)*, *example(s)*, and *sample(s)*, since they are not part of the core system.

### C. Building Refactoring Graphs

As mentioned earlier, we identify refactoring subgraphs over time in 10 systems. Algorithm 1 presents the steps to build refactoring graphs. The input comprises a list of refactorings, e.g., $util.Foo\#m()$ moved to $util.Bar\#m()$. First, the algorithm identifies each refactoring $t$ and the two methods involved, $m1$ and $m2$ (line 3). Then, it creates a directed edge representing this refactoring (line 5). Since $V$ and $E$ are sets, each element is represented only one time. The edges are labeled with refactoring's name $t$. The output includes sets of refactoring subgraphs in text format.

Table II presents the frequency of refactoring subgraphs in the analyzed systems. Considering all the projects, we detect a total of 8,926 refactoring subgraphs. Spring Framework

[1]https://git-scm.com/docs/git-log#Documentation/git-log.txt---first-parent

**Algorithm 1:** Building refactoring graphs

**Input:** R (list of refactorings from a system S)
**Output:** DG (refactoring graph)

```
begin
    DG ← ∅, V ← ∅, E ← ∅
    for (m1, m2, t) ∈ R do
        V ← V ∪ {m1, m2}
        E ← E ∪ (m1, m2, t)
    end
    return (V, E)
end
```

has the highest number of subgraphs (3,104), while Square Retrofit has the lowest amount (169). Overall, 87.1% of the refactoring subgraphs comprise a set of operations performed in a single commit. This ratio varies from 69.2% (Glide) to 93.8% (Apache Dubbo). In contrast, 12.9% capture refactorings performed in two or more commits. **In this paper, we assess the 1,150 refactoring subgraphs with number of commits $\geq$ 2, because they are the ones that represent refactoring over time.**

TABLE II
FREQUENCY OF REFACTORING SUBGRAPHS

| Project | Refactoring Subgraphs | | | | |
|---|---|---|---|---|---|
| | All | $len = 1$ | % | $len \geq 2$ | % |
| Elasticsearch | 2,073 | 1,934 | 93.3 | 139 | 6.7 |
| RxJava | 1,073 | 975 | 90.9 | 98 | 9.1 |
| Square Okhttp | 635 | 548 | 86.3 | 87 | 13.7 |
| Square Retrofit | 169 | 135 | 79.9 | 34 | 20.1 |
| Spring Framework | 3,104 | 2,604 | 83.9 | 500 | 16.1 |
| Apache Dubbo | 483 | 453 | 93.8 | 30 | 6.2 |
| MPAndroidChart | 454 | 381 | 83.9 | 73 | 16.1 |
| Glide | 425 | 294 | 69.2 | 131 | 30.8 |
| Lottie Android | 196 | 173 | 88.3 | 23 | 11.7 |
| Facebook Fresco | 314 | 279 | 88.9 | 35 | 11.1 |
| Total | 8,926 | 7,776 | 87.1 | 1,150 | 12.9 |

## IV. RESULTS

### A. (RQ1) What Is the Size of Refactoring Subgraphs?

As presented in Figure 4, most refactoring subgraphs have three vertices (639 occurrences, 56%). The other recurrent cases comprise subgraphs with two (15%) or four vertices (14%). 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%), as shown in Figure 5. MPAndroidChart has the largest subgraph in term of edges. It has 61 edges, most representing *extract and move* operations. Therefore, most subgraphs contain few methods (vertices) and refactoring operations (edges).

Figure 6 shows a real example of a refactoring subgraph from MPAndroidChart, which includes three distinct refactoring operations. In the first commit C1, a developer renamed method $drawYLegend()$ to $drawYLabels()$.[2] In the subsequent operation performed 13 days later, the same developer extracted a new method from $drawYLabels()$ at commit C2.[3] 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.[4]

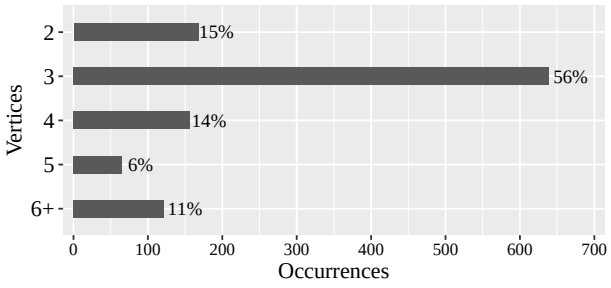


Fig. 4. Number of vertices by refactoring subgraph

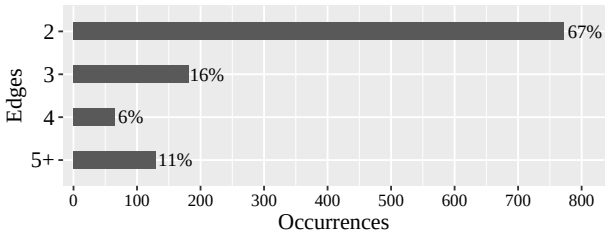


Fig. 5. Number of edges by refactoring subgraph

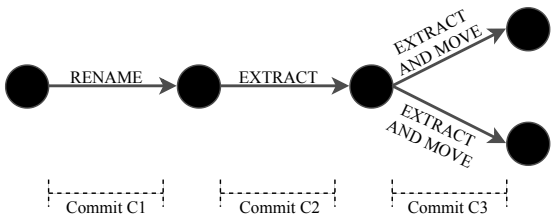


Fig. 6. Example of a refactoring subgraph from MPAndroidChart

*Summary:* Most refactoring subgraphs are small. Among 1,150 samples, most cases comprise subgraphs with the number of vertices ranging from two to four (85%) and the number of edges varying between two and three (83%).

### B. (RQ2) How Many Commits Are in Refactoring Subgraphs?

In this second question, we investigate the number of commits per subgraph. As presented in Figure 7, most cases include subgraphs with two (81%) or three commits (14%). The largest subgraph in terms of commits is again from Square Okhttp (18 commits).

[2]https://github.com/PhilJay/MPAndroidChart/commit/13104b26

[3]https://github.com/PhilJay/MPAndroidChart/commit/063c4bb0

[4]https://github.com/PhilJay/MPAndroidChart/commit/d930ac23

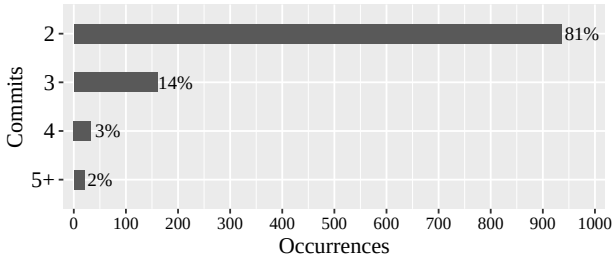


Fig. 7. Number of commits by refactoring subgraph

Figure 8 shows an example from Elasticsearh. In commit C1, a developer moved two methods from class $SocketSelector$ to $NioSelector$.[5] After approximately three months, in commit C2, a second developer extracted duplicated code from three methods to a new method named $handleTask(Runnable)$.[6] 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.

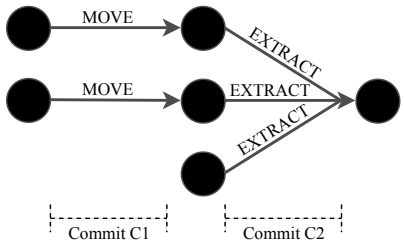


Fig. 8. Example of a refactoring subgraph from Elasticsearch

*Summary:* Most refactoring subgraphs are created in two commits (81%) or in three commits (14%).

### C. (RQ3) What Is the Age of Refactoring Subgraphs?

To assess age, we compute the number of days between the most recent and the oldest commit in a refactoring subgraph. Figure 9 presents the results: we notice that refactoring subgraphs age varies among the projects. Considering the median of the distributions, the youngest subgraphs are found in Lottie Android and RxJava, which have 3 and 3.4 days, respectively. On the other side, the oldest subgraphs are found in Glide (489.8 days), Spring Framework (127.9), and Fresco (192). The other systems have subgraphs with age between 76.7 (Retrofit) and 102.5 days (Dubbo). Regarding the maturity of the target systems, the youngest project is Lottie Android (3 years) while the oldest one is Elasticsearch (9 years). We run the Spearman's test to assess the correlation between the systems age and the median time of their refactoring subgraphs. The correlation coefficient (rho) is 0.067, showing a very weak correlation. In other words, there are subgraphs with different age in both old and young systems. However, the *p-value* is $> 0.001$ due to our small sample size.

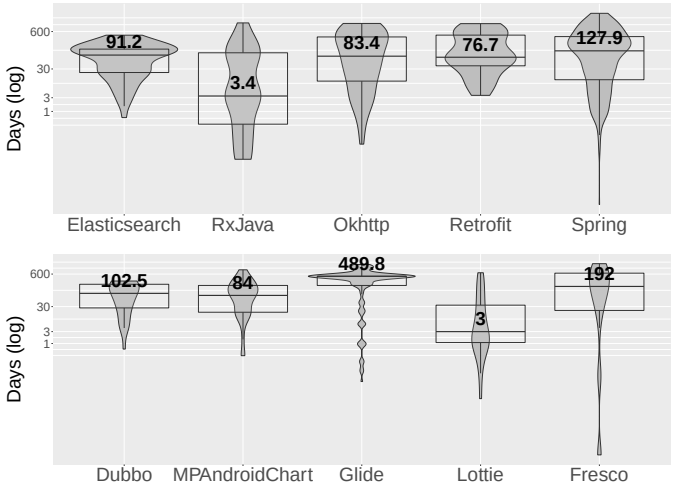


Fig. 9. Age of the refactoring subgraphs

Figure 10 shows an example of a subgraph describing refactorings performed in few days on Spring Framework. In commit C1, a developer renamed method $before(Function)$ to $filterBefore(Function)$.[7] After six days, the same developer reverted the operation in commit C2, renaming $filterBefore(Function)$ to the original name.[8] As a consequence, these modifications created a subgraph with two vertices and two edges.

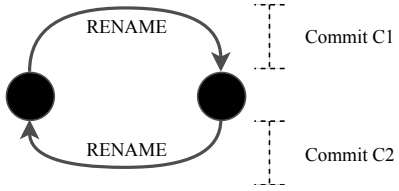


Fig. 10. Example of a refactoring subgraph from Spring Framework

*Summary:* The age of the subgraphs is diverse: while some have few days, the majority of the subgraphs have weeks or even months. For example, 67% of the refactoring subgraphs have more than one month.

### D. (RQ4) Which Refactorings Compose the Refactoring Subgraphs?

First, we present the most common refactoring operations in our sample of 1,150 refactoring subgraphs (Table III). Most cases include *rename method* (21%), *extract and move method* (19%), and *extract method* (17%). By constrast, we detected only 83 occurrences of *move and rename* operations. There are also few inheritance-based refactorings, i.e., *pull up* (330 occurrences) and *push down* (142 occurrences).

[5]https://github.com/elastic/elasticsearch/commit/9ee492a3f07
[6]https://github.com/elastic/elasticsearch/commit/11fe52ad767
[7]https://github.com/spring-projects/spring-framework/commit/794693525f
[8]https://github.com/spring-projects/spring-framework/commit/91e96d8084

TABLE III
FREQUENCY OF REFACTORING OPERATIONS

| Refactoring | Occurrences | % |
|---|---|---|
| Rename | 757 | 21 |
| Extract and move | 685 | 19 |
| Extract | 635 | 17 |
| Move | 579 | 16 |
| Inline | 474 | 13 |
| Pull up | 330 | 9 |
| Push down | 142 | 4 |
| Move and rename | 83 | 2 |
| All | 3,685 | 100 |

Next, we categorize the subgraphs into two groups. The homogeneous group includes subgraphs with a single refactoring operation. In contrast, the heterogeneous category comprises subgraphs with at least two distinct refactoring operations. As presented in Table IV, overall, around 28% of the subgraphs are homogeneous, while 72% are heterogeneous. The results per system follow a similar tendency. Most of the projects have more heterogeneous subgraphs than homogeneous ones; the sole exception is RxJava (57% vs 43%). In addition, as presented in Figure 11, heterogeneous subgraphs often include two distinct refactoring types (84%); in contrast, 12% have three and only 4% have four or more distinct refactoring types.

TABLE IV
HOMOGENEOUS VS HETEROGENEOUS REFACTORING SUBGRAPHS

| Project | Homogeneous | % | Heterogeneous | % |
|---|---|---|---|---|
| Elasticsearch | 43 | 30.9 | 96 | 69.1 |
| RxJava | 56 | 57.1 | 42 | 42.9 |
| Square Okhttp | 22 | 25.3 | 65 | 74.7 |
| Square Retrofit | 12 | 35.3 | 22 | 64.7 |
| Spring Framework | 138 | 27,6 | 362 | 72,4 |
| Apache Dubbo | 6 | 20.0 | 24 | 80.0 |
| MPAndroidChart | 16 | 21.9 | 57 | 78.1 |
| Glide | 19 | 14.5 | 112 | 85.5 |
| Lottie Android | 5 | 21.7 | 18 | 78.3 |
| Facebook Fresco | 6 | 17.1 | 29 | 82.9 |
| All | 323 | 28.1 | 827 | 71.9 |

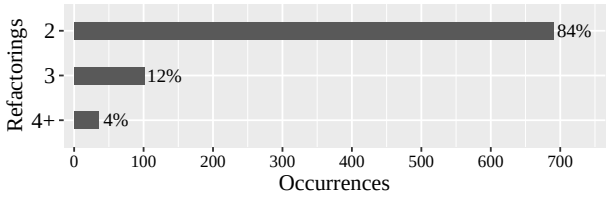


Fig. 11. Number of distinct refactoring operations in heterogeneous subgraphs

Figure 12 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 method fetchDecodedImage(...) from two methods into class $ImagePipeline$.[9] The next operations happened years later when a second developer made two new *extract* operations in commits C2[10] and C3[11].

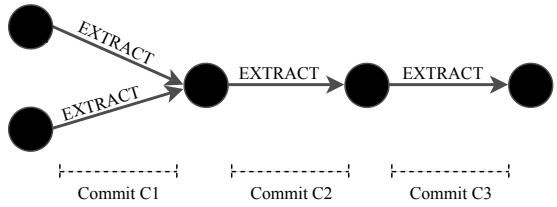


Fig. 12. Example of a homogeneous refactoring subgraph from Facebook Fresco

*Summary:* Most refactoring subgraphs are heterogeneous (71.9%), i.e., they include more than one refactoring type.

### *E. (RQ5) Are the Refactoring Subgraphs Created by the Same or Multiple Developers?*

As the last research 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 V, most subgraphs have a single author (60.3%). As reported in a previous question, the number of commits per subgraph is also small. Thus, we execute Spearman's test to evaluate the correlation between the number of developers and the number of commits for each refactoring subgraph. The correlation coefficient (rho) is 0.244, with a *p-value* $< 0.001$, indicating a weak correlation between these metrics. That is, the higher the number of commits in a subgraph, the higher its amount of developers.

TABLE V
DEVELOPERS OF REFACTORING GRAPHS

| Project | Single dev. | % | Multiple devs. | % |
|---|---|---|---|---|
| Elasticsearch | 32 | 23.0 | 107 | 77.0 |
| RxJava | 88 | 89.8 | 10 | 10.2 |
| Square Okhttp | 32 | 36.8 | 55 | 63.2 |
| Square Retrofit | 14 | 41.2 | 20 | 58.8 |
| Spring Framework | 303 | 60.6 | 197 | 39.4 |
| Apache Dubbo | 17 | 56.7 | 13 | 43.3 |
| MPAndroidChart | 70 | 95.9 | 3 | 4.1 |
| Glide | 116 | 88.5 | 15 | 11.5 |
| Lottie Android | 11 | 47.8 | 12 | 52.2 |
| Facebook Fresco | 10 | 28.6 | 25 | 71.4 |
| All | 693 | 60.3 | 457 | 39.7 |

Figure 14 presents an example of a refactoring subgraph from Square Okhttp. First, in commit C1, a developer D1 renamed three methods from class $OkHttpClient$.[12] Basically, the developer removed the prefix $set$ from their

[9]https://github.com/facebook/fresco/commit/02ef6e0f
[10]https://github.com/facebook/fresco/commit/b76f56ef
[11]https://github.com/facebook/fresco/commit/017c007b
[12]https://github.com/square/okhttp/commit/daf2ec6b9

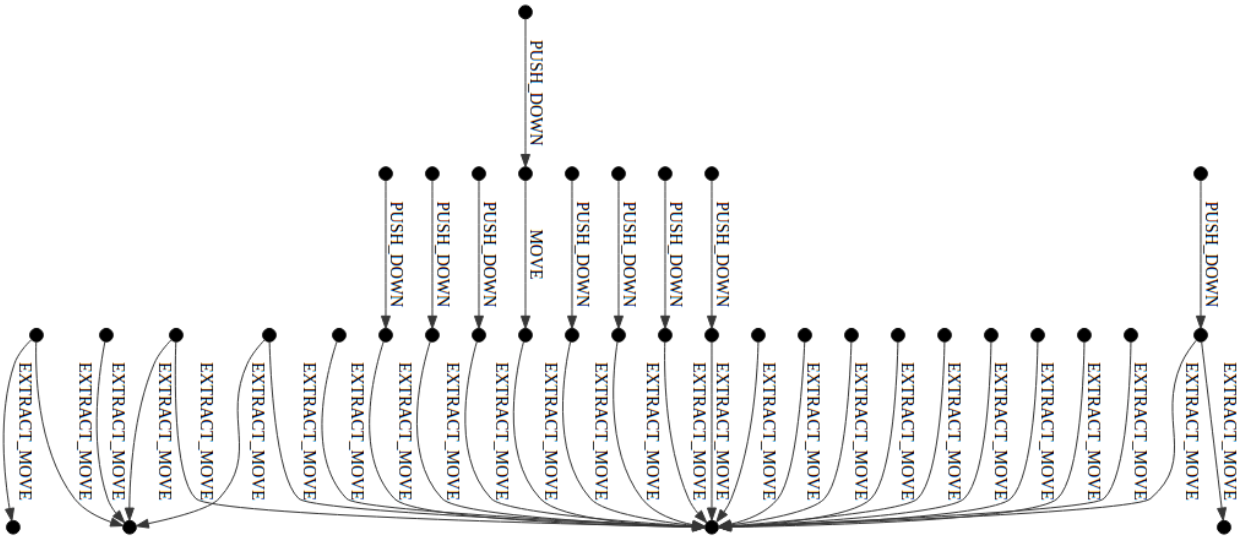


Fig. 13. Example of a large refactoring subgraph from Square Okhttp

names. After 10 months, a second developer D2 removed a duplicate code from these methods, extracting method $checkDuration(...)$.[13] Then, after seven months, D2 moved this method to a new class named $Util$, in commit C3.[14] As a result, these two developers were responsible for a refactoring subgraph with eight vertices and seven edges.

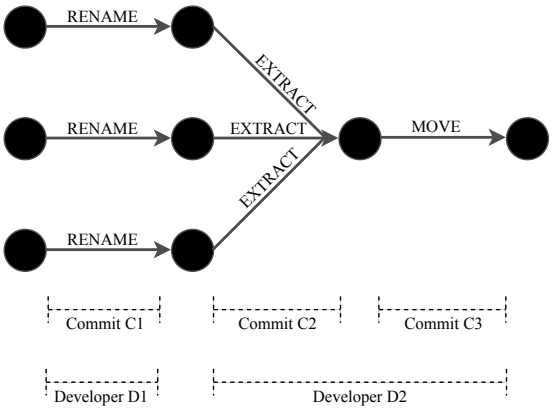


Fig. 14. Example of a refactoring subgraph create by multiple developers from Square Okhttp

*Summary:* Most refactoring subgraphs are created by a single developer (60%). Only 40% have multiple developers.

[13]https://github.com/square/okhttp/commit/c5a26fefd
[14]https://github.com/square/okhttp/commit/a32b1044a

## V. Large Subgraph Example

In this section, we present and discuss an example of a large refactoring subgraph. As we reported in Section IV, most refactoring subgraphs are small, in terms of number of vertices, edges, and commits. For this reason, we only presented small examples when discussing our RQ results. However, we also found graphs describing major refactorings over time, whose presentation we postponed to this section.

Figure 13 shows an example from Square Okhttp. We chose this example because it encompasses different refactoring operations performed over time and it is one of the largest subgraphs from our dataset. This graph has 37 vertices, four commits, and three refactoring operations (*move*, *push down*, and *extract and move*). It was built by multiple developers, over six months. As we can observe, the graph nicely describes an example of code duplication removal. First, a developer performed nine *push down* refactorings to move a method from a superclass to a subclass. Then, a second developer performed 21 *extract method* operations to move the duplicated code to a single method, which has the following code:

```
public int readInt() throws IOException {
  require(4, Deadline.NONE);
  return buffer.readInt();
}
```

Besides that, there are other three *extract method* operations: (i) *readShort()* from a single method (this node has a single incoming edge), (ii) *readByteString()* from four methods, and (iii) *decode()* from a single method. These new methods are presented in the bottom of Figure 13.

## VI. Discussion and Implications

### A. Detecting Refactoring over Time

Several tools and techniques are proposed in the literature to detect refactoring operations, for instance, Refactoring Crawler [14], RefFinder [15], Refactoring Miner [3], [5], and, more recently, RefDiff [13] and RMiner [16]. In common, those approaches only detect *atomic* refactoring, i.e., operations that happen in a single commit and performed by a single developer. In contrast, our approach, refactoring graph, focuses on the detection of refactoring over time, i.e., operations over multiple commits and performed by multiple developers. Moreover, differently from the *batch* refactoring [1], [8], [9], our approach is not constrained by the amount of developers nor to a time window. Indeed, we found refactoring subgraphs with age ranging from weeks to months and created by multiple developers. *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.*

### B. 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 [17]. However, if the reviewed change is large and complex, this task becomes challenging [17]. To alleviate this issue, refactoring-aware code review tools were proposed [18]–[20] to better understand changes mixed with refactoring. 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 14, 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 understating and improvement.*

### C. Detecting Refactoring Patterns and Smells

Frequent refactoring subgraphs may indicate common refactoring patterns over time. In contrast, infrequent refactoring subgraphs that are variations of the pattern may suggest the presence of "refactoring smells" that deserve to be fixed. For example, suppose the refactoring subgraph shown in Figure 2 is frequent: a developer extracted two methods from $m2()$, which are named $a()$ and $b()$; then, $b()$ was renamed to $c()$, finally, $c()$ was renamed back to $b()$. In this case, if we find a single refactoring subgraph that does not include the last renaming, this may suggest that the developer forgot to perform the undo rename in one single case. In this sense, refactoring subgraphs can be used to spot bad smells, which are only visible because refactoring subgraphs provide the big picture of the refactoring. Indeed, this is a topic that we aim to deep assess in further research, possibly with the support of techniques to mine graphs [21]–[23]. *Thus, refactoring graphs can foment the detection of refactoring anomalies over time and drive future research agenda on refactoring patterns.*

### D. Understanding and Assessing Software Evolution

During software evolution, developers often perform refactoring operations. Consequently, the link between methods may be lost [24]. 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 [25]–[29], code evolution visual supporting [30], [31], bug introducing change detection [32]–[36], 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 [24], [25]. 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 $c()$. *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.*

## VII. Threats to Validity

*Generalization of the results.* We analyzed 1,150 refactoring subgraphs from 10 popular and open source Java systems. Therefore, our dataset is built over credible and real-world software systems. 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. Besides that, we focus our study on eight refactorings at method level. 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 operation because it is the sole refactoring detection tool that is multi language, working for Java, JavaScript, and C. It is also extensible to other programming languages. Thus, as we plan to extend this research to cover other programming languages than Java, REFDIFF was the proper solution. In addition to be multi language, REFDIFF accuracy is quite high. REFDIFF's authors provide two evaluations of their tool [13]. In the first evaluation, it achieved an overall F-measure of 96.8% (precision: 100%; recall: 93.9%). In the second evaluation, REFDIFF's authors analyzed 102 real refactoring instances. In this case, it achieved an overall F-measure of 89.3% (precision: 85.4%; recall: 93.6%). Recently, Tsantalis *et al.* [16] proposed the refactoring detection tool RMINER. When considering all refactoring operations, RMINER has an F-measure of 92%

(precision: 98%; recall: 87%) improving on REFDIFF's overall accuracy. However, RMINER works only for Java projects.

*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, we removed constructor methods (vertices) from our analysis because they include mostly initialization settings, and do not have behavior as conventional methods. We also removed some very specific cases of refactoring (edges) in which REFDIFF reported false positives in inner classes or same method. However, these cases are not likely to affect our results because they only represent a fraction (3.5%) of the refactoring operations. 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.

*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 developers. Thus, our results can include the same developer committing with different email addresses. But, we already found that most cases are subgraphs created by a single developer.

## VIII. Related Work

Refactoring is an usual practice during software evolution and maintenance. Constantly, developers refactor the source code for different purposes [3], [37]. For this reason, several studies concentrate on this research field [1], [7], [8], [13], [14], [16], [38]–[49]. Among the empirical studies, some research focus on set of related refactoring. Specifically, these studies analyze *batch refactorings* [1], [8], [9], [50]–[52]. Murphy *et al.* [1] 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 investigate about sets of refactorings operations executed in 60 seconds of each another, 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 baches involve a short period, the study does not investigate refactorings operations that occur in different moments over time.

In another context, Bibiano *et al.* [8] 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 *RMiner* tool to detect 13 well-know refactorings [16], resulting in 24,893 operations. Then, the authors applied a heuristic to compute batch refactorings, i.e., set of related refactorings [9]. 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, the subgraphs involve refactoring over time (i.e., more than one commit), including subgraphs by multiples developers and different code elements.

Other studies also discuss the impact of batches to eliminate code smells, proposing approaches to reuse or suggest sets of related refactoring operations [51]–[53]. Thus, they do not focus on sequences of refactoring operations over time. Fowler [10] 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: the age of the refactoring subgraphs is diverse, ranging from days to weeks or even months.

Hora *et al.* [24] analyze untracked changes during software development. The authors show that refactorings invalidate several tracking strategies to evaluate system evolution. As in our study, 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). In other words, 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 our study, the goal is to investigate refactorings performed over long time windows; we do not concentrate on tracked modifications on source code.

Meananeatra [54] 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. As in our study, 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.

## IX. Conclusion

In this paper, we proposed refactoring graphs, a novel approach to assess refactoring operations over time. We analyzed 10 popular Java systems from which 1,150 refactoring subgraphs were extracted. We then investigate five research questions to evaluate the following properties of refactoring graphs: size, commits, age, composition, and developers. We summarize our findings as follows:

- 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 (95%).
- Refactoring subgraphs span from few days to months.
- Refactoring graphs are often heterogeneous, that is, they are composed by several types of refactoring.
- Refactoring graphs are mostly created by a single developer (60%).

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, only possible to be spotted over time; and (iv) we state the importance of refactoring graphs to resolve method names and support software evolution studies.

Further studies can consider other popular programming languages and ecosystems; refactoring graphs based on class and package level as well as other refactoring types at method level; and also novel approaches to complement existing tools and techniques that focus on *atomic* refactorings.

## Acknowledgments

This research is supported by grants from FAPEMIG, CNPq, and CAPES.

## References

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

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

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

[4] D. Mazinanian, A. Ketkar, N. Tsantalis, and D. Dig, "Understanding the use of lambda expressions in Java," *Programming Languages*, vol. 1, no. 85, pp. 85:1–85:31, 2017.

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

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

[7] M. Kim, T. Zimmermann, and N. Nagappan, "An empirical study of refactoring challenge and benefits at Microsoft," *Transactions on Software Engineering*, vol. 40, no. 7, pp. 633–649, 2014.

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

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

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

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

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

[13] D. Silva and M. T. Valente, "RefDiff: Detecting refactorings in version histories," in *14th International Conference on Mining Software Repositories (MSR)*, pp. 1–11, 2017.

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

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

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

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

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

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

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

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

[22] A. Inokuchi, T. Washio, and H. Motoda, "An apriori-based algorithm for mining frequent substructures from graph data," in *4th Principles and Practice of Knowledge Discovery in Databases (PKDD)*, pp. 13–23, 2000.

[23] M. Kuramochi and G. Karypis, "Frequent subgraph discovery," in *1st International Conference on Data Mining (ICDM)*, pp. 313–320, 2001.

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

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

[26] F. Rahman and P. Devanbu, "Ownership, experience and defects: a fine-grained study of authorship," in *33rd International Conference on Software Engineering (ICSE)*, 2011.

[27] A. Meneely and O. Williams, "Interactive churn metrics: socio-technical variants of code churn," *Software Engineering Notes*, vol. 37, no. 6, 2012.

[28] D. Spinellis, "A repository of Unix history and evolution," *Empirical Software Engineering*, vol. 22, no. 3, pp. 1372–1404, 2017.

[29] L. Hattori and M. Lanza, "Mining the history of synchronous changes to refine code ownership," in *6th International Working Conference on Mining Software Repositories (MSR)*, 2009.

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

[31] V. U. Gómez, S. Ducasse, and T. D'Hondt, "Visually characterizing source code changes," *Science of Computer Programming*, vol. 98, no. P3, pp. 376–393, 2015.

[32] S. Kim, T. Zimmermann, K. Pan, and E. J. J. Whitehead, "Automatic identification of bug-introducing changes," in *21st International Conference on Automated Software Engineering (ASE)*, 2006.

[33] T. Zimmermann, S. Kim, A. Zeller, and E. J. Whitehead, Jr., "Mining version archives for co-changed lines," in *3rd International Workshop on Mining Software Repositories (MSR)*, 2006.

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

[35] T.-H. Chen, M. Nagappan, E. Shihab, and A. E. Hassan, "An empirical study of dormant bugs," in *11th Working Conference on Mining Software Repositories (MSR)*, 2014.

[36] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu, "On the naturalness of buggy code," in *38th International Conference on Software Engineering (ICSE)*, 2016.

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

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

[39] B. Lin, C. Nagy, G. Bavota, and M. Lanza, "On the impact of refactoring operations on code naturalness," in *26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 594–598, 2019.

[40] J. Kim, D. Batory, D. Dig, and M. Azanza, "Improving refactoring speed by 10x," in *38th International Conference on Software Engineering (ICSE)*, pp. 1145–1156, 2016.

[41] G. Szke, C. Nagy, R. Ferenc, and T. Gyimthy, "Designing and developing automated refactoring transformations: An experience report," in *23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pp. 693–697, 2016.

[42] G. Bavota, A. D. Lucia, M. D. Penta, R. Oliveto, and F. Palomba, "An experimental investigation on the innate relationship between quality and refactoring," *Journal of Systems and Software*, vol. 107, no. C, pp. 1–14, 2015.

[43] G. Bavota, B. De Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, and O. Strollo, "When does a refactoring induce bugs? an empirical study," in *12th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 104–113, 2012.

[44] D. Dig and R. Johnson, "How do APIs evolve? a story of refactoring," in *22nd International Conference on Software Maintenance (ICSM)*, pp. 83–107, 2005.

[45] B. Shen, W. Zhang, H. Zhao, G. Liang, Z. Jin, and Q. Wang, "IntelliMerge: A refactoring-aware software merging technique," *Programming Languages*, vol. 3, no. 170, pp. 170:1–170:28, 2019.

[46] R. Terra, M. T. Valente, S. Miranda, , and V. Sales, "JMove: A novel heuristic and tool to detect move method refactoring opportunities," *Journal of Systems and Software*, vol. 138, pp. 19–36, 2018.

[47] E. L. G. Alves, M. Song, and M. Kim, "RefDistiller: A refactoring aware code review tool for inspecting manual refactoring edits," in *22nd International Symposium on Foundations of Software Engineering (FSE)*, pp. 751–754, 2014.

[48] Y. Lin, X. Peng, Y. Cai, D. Dig, D. Zheng, and W. Zhao, "Interactive and guided architectural refactoring with search-based recommendation," in *24th International Symposium on Foundations of Software Engineering (FSE)*, pp. 535–546, 2016.

[49] O. Chaparro, G. Bavota, A. Marcus, and M. D. Penta, "On the impact of refactoring operations on code quality metrics," in *30th International Conference on Software Maintenance and Evolution (ICSME)*, pp. 456–460, 2014.

[50] E. Fernandes, "Stuck in the middle: Removing obstacles to new program features through batch refactoring," in *41st International Conference on Software Engineering: Companion Proceedings (ICSE)*, pp. 206–209, 2019.

[51] D. Tenorio, A. C. Bibiano, and A. Garcia, "On the customization of batch refactoring," in *3rd International Workshop on Refactoring (IWOR)*, pp. 13–16, 2019.

[52] E. Fernandes, A. Uchôa, A. C. Bibiano, and A. Garcia, "On the alternatives for composing batch refactoring," in *3rd International Workshop on Refactoring (IWOR)*, pp. 9–12, 2019.

[53] H. C. Jiau, L. W. Mar, and J. C. Chen, "OBEY: Optimal batched refactoring plan execution for class responsibility redistribution," *Transactions on Software Engineering*, vol. 39, no. 9, pp. 1245–1263, 2013.

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