Noname manuscript No.
(will be inserted by the editor)

Characterizing Refactoring Graphs in Java and JavaScript Projects

Aline Brito · Andre Hora · Marco Tulio Valente

Received: date / Accepted: date

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, we rely on refactoring graphs. Using this abstraction, we provide quantitative and qualitative investigation on 20 popular open-source Java and JavaScript-based projects. After eliminating trivial graphs, we characterize a large sample of 1,525 refactoring graphs, providing quantitative data on their size, commits, age, refactoring composition, ownership, operations over time, and refactoring graph patterns. Besides, we contact the authors of subgraphs describing large refactoring operations to understand the reasons behind their operations. We conclude by discussing applications and implications of refactoring graphs, for example, to improve code comprehension, detect refactoring patterns, and support software evolution studies.

Keywords Refactoring · Refactoring graphs · Mining software repositories · Software evolution

1 Introduction

Refactoring is a key activity to preserve and improve the internal design of software systems. Due to the importance of the practice in modern software

Aline Brito
Department of Computer Science, UFMG, Brazil, E-mail: alinebrito@dcc.ufmg.br

Andre Hora
Department of Computer Science, UFMG, Brazil, E-mail: andrehora@dcc.ufmg.br

Marco Tulio Valente
Department of Computer Science, UFMG, Brazil, E-mail: mtov@dcc.ufmg.br

development, there is a large body of studies about refactoring, shedding light on aspects such as usage of refactoring engines [52, 53], documentation of refactorings using commit messages [52], motivations for performing refactorings [49, 64, 73], benefits and challenges of refactoring [42, 43], among 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.* [52] 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 [15], which are groups of 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%). However, to our knowledge, refactorings performed over long time windows were not deeply studied by the literature.

Therefore, in a previous conference paper [10], 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 operation by another developer d_2.

Quantitative study: We formalize an algorithm to build refactoring graphs and use it to extract graphs for 20 well-known and popular open-source Java and JavaScript projects. In this first study, our goal is to characterize refactoring subgraphs. Thus, we answer seven research questions about the following properties:

1. *Refactorings over time*: In both languages, approximately 30% of refactoring operations are part of a refactoring subgraph over time.
2. *Size*: Most refactoring subgraphs are small. In Java, most cases refer to subgraphs with up to four vertices (85%) and three edges (83%). Similarly, in JavaScript, most refactoring subgraphs have up to four nodes (86%)

and three edges (85%). However, we also found subgraphs due to large refactoring operations (e.g., subgraphs with more than 30 vertices).
3. *Commits*: Most refactoring subgraphs are generated from two or three commits, e.g., 95% of Java subgraphs and 93% of JavaScript subgraphs.
4. *Age*: The age of the refactoring subgraphs ranges from a few days to weeks or even months. For instance, in both languages, approximately 60% of the subgraphs have more than one month.
5. *Homogeneity*: 71% of Java subgraphs and 64% of JavaScript subgraphs include more than one refactoring type.
6. *Ownership*: In both languages, about 60% of the refactoring subgraphs are created by a single developer.
7. *Patterns*: The most recurring *over time* patterns of refactoring graphs have two edges. For example, in Java, the most recurrent case refer to successive rename operations, i.e., *rename* → *rename* (153 occurrences). In JavaScript, this is the second most recurrent pattern, appearing in 37 subgraphs in our dataset.

Qualitative study: In our quantitative study, we observed that most refactoring subgraphs are small. However, we also notice subgraphs describing large refactoring operations. Thus, in this second study, we selected and manually inspected 50 large refactoring subgraphs in terms of vertices. Then, we contacted the authors of these refactoring instances, asking for the motivations behind their operations. Based on these developers' feedback, our results suggest that large subgraphs relate to two major reasons: *improving code design* and *fixing bugs or improving existing features.*

Paper extension: This paper is an extension of a previous study [10]. We expand this former work in the following major points:

1. We perform a novel analysis on JavaScript systems (the former paper only included Java) and extend all RQs with refactoring graphs mined for this language.
2. We propose a new research question (RQ6) where we assess refactoring graph patterns, and an introductory research question (RQ0) about refactorings that are spread over multiple commits.
3. We perform a novel qualitative analysis by applying a survey with developers, aiming to understand the motivations behind large refactoring subgraphs.
4. We designed and implemented a web application to easily visualize refactoring graphs.[1] Also, we provide scripts to automatically visualize refactoring subgraphs for Java and JavaScript projects hosted on GitHub.[2]
5. We provide a new evaluation of the precision of RefDiff [63, 65], which is the tool we used to detect refactoring operations. The evaluation relies on real-world Java and JavaScript open-source projects, increasing the existing datasets with new refactoring instances.

[1] `https://refactoring-graph.github.io`

[2] `https://github.com/alinebrito/refactoring-graph-generator`

Structure: Section 2 introduces REFDIFF, which is the tool used to detect refactoring operations. Section 3 defines our concept of refactoring graphs. Section 4 describes the design of our quantitative study and results. Section 5 presents the second study, based on a survey with authors of large refactoring subgraphs. We discuss the key applications and implications in Section 6. Section 7 states threats to validity and Section 8 presents related work. Finally, we conclude the paper in Section 9.

2 RefDiff Tool

REFDIFF [63, 65] is a tool to detect refactoring operations. 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 our study, we concentrate on two programming languages supported by the tool: Java and JavaScript.[3] We selected these languages due to their popularity. For example, they were pointed to amongst the most adopted and loved programming languages by developers worldwide.[4] Besides that, most refactoring research in the literature discuss refactoring practices only in Java [8, 57, 64, 67]. Therefore, by studying JavaScript, we attempted to contribute with a refactoring study that also considered interpreted, dynamic, and very popular programming languages. Also, we focus on method or function level operations since refactoring operations frequently affect these elements [36, 63, 75]. Table 1 lists the refactorings detected by REFDIFF at these elements. As we can notice, both languages have well-know refactorings, comprising *extract* and *inline* operations, as well as changes in method's signature (i.e., *rename* and *move*).

Table 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

Since JavaScript is a dynamic language, inheritance-based refactorings are not detected in this language (i.e., *pull up* and *push down*). In additon, JavaScript-based systems usually contain large files that are composed of several nested elements. For this reason, many refactoring occur on a single file. REFDIFF reports these cases as internal operations. Listing 1 shows an example

[3] In Section 5.1.1, we detail the results of a precision analysis of RefDiff: Java (87%) and JavaScript (93%)

[4] `https://insights.stackoverflow.com/survey/2020`

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.

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


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

Listing 1 Example of an *internal move* operation in JavaScript (strikethrough text represents deleted line and the symbol "+" denotes added lines of code)

3 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 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 practicable 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 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 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, is created a cycle when the developer reverts the method to the original name.

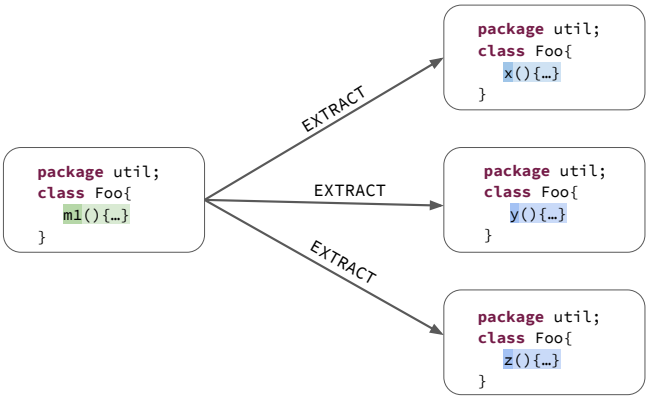


Fig. 1 Refactoring subgraph produced by only one developer

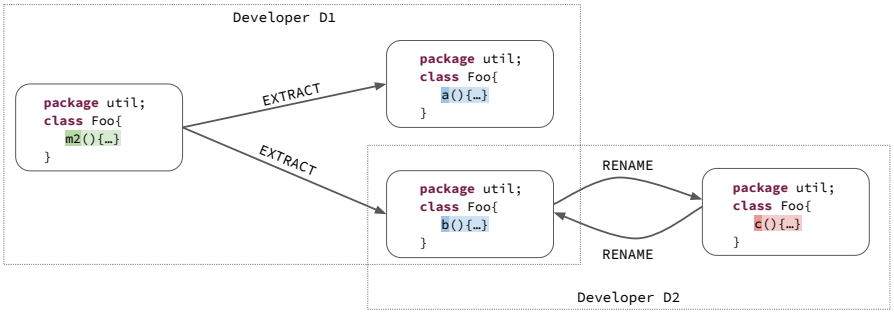


Fig. 2 Refactoring subgraph over time

As presented in Figure 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, 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 super-

types or subtypes (i.e., *pull up* and *push down*). For example, a *pull up* moves methods from subclasses to a superclass.

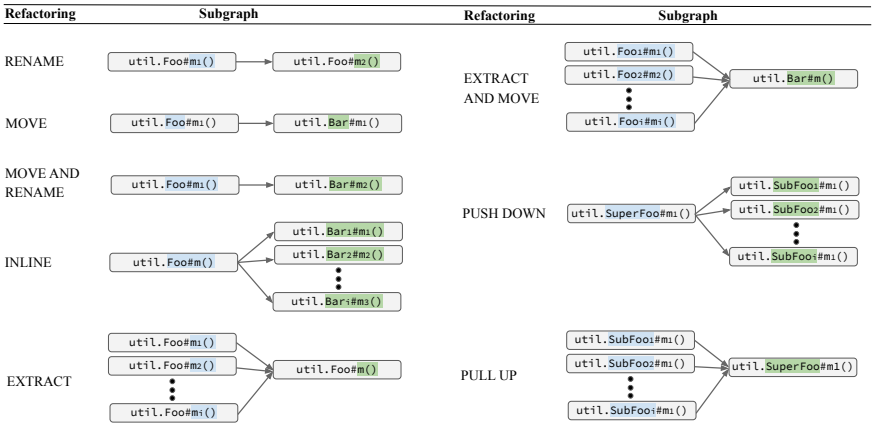


Fig. 3 Example of refactoring subgraphs (Java)

Similar refactorings apply to functions in JavaScript. As shown in Figure 4, in JavaScript, there are also internal operations, i.e., refactorings performed in a single file.

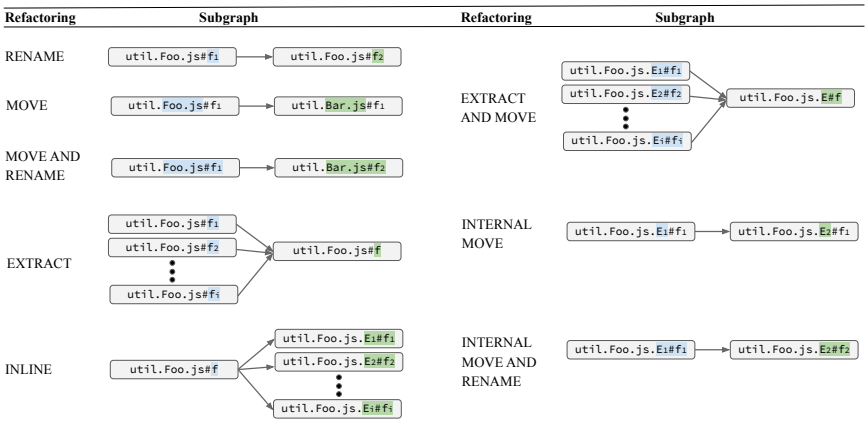


Fig. 4 Example of refactoring subgraphs (JavaScript)

4 Quantitative Study: Characterizing Refactoring Graphs

4.1 Study Design

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. 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 5), 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, age, homogeneity, ownership, and patterns.

- (RQ_0) *How many refactoring operations generate subgraphs over time?* Most studies concentrate on refactorings performed in a single commit [1, 20, 38, 64]. 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 [52, 67].
- (RQ_3) *What is the age 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 [57, 63, 64, 74]. 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.

4.1.1 Selecting Projects

In this paper, 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 [9, 66]. 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)[5] and JavaScript style guides (such as airbnb/javascript).[6] Table 2 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 2 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

[5] `https://github.com/iluwatar/java-design-patterns`

[6] `https://github.com/airbnb/javascript`

4.1.2 Detecting Refactoring Operations

As mentioned in Section 2, we use REFDIFF [63, 65] 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 and 4. 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.[7] 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*".[8]

4.1.3 Building Refactoring Graphs

As mentioned earlier, we identify refactoring subgraphs over time in 20 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 the sets of refactoring subgraphs.

Algorithm 1: Building refactoring graphs

```
   Input: R (list of refactorings from a system S)
   Output: DG (refactoring graph)
1  begin
2  |  DG ← ∅, V ← ∅, E ← ∅
3  |  for (m1, m2, t) ∈ R do
4  |  |  V ← V ∪ {m1, m2}
5  |  |  E ← E ∪ (m1, m2, t)
6  |  end
7  |  return (V, E)
8  end
```

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

[8] https://github.com/vuejs/vue/tree/dev/packages/vue-server-renderer

Table 3 presents the frequency of refactoring subgraphs for each Java project, and Table 4 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 Frequency of refactoring subgraphs (Java)

Project	**Refactoring Subgraphs**				
	All	$commit = 1$	%	$commit \geq 2$	%
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

Table 4 Frequency of refactoring subgraphs (JavaScript)

Project	**Refactoring Subgraphs**				
	All	$commit = 1$	%	$commit \geq 2$	%
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 amount (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.*

4.1.4 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 [45, 78]. Figure 5 shows a simple example of graph pattern. 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.

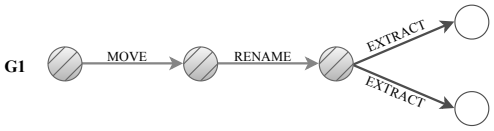


Fig. 5 Example of over time graph patterns

$G2$ illustrates a second example, a pattern with three *extract* method operations, as shown in Figure 6. 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.

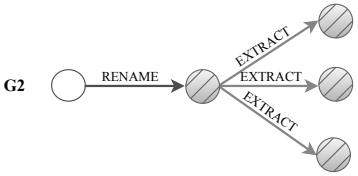


Fig. 6 Example of possibly atomic graph patterns

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., $G2$).[9] 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 7. In this subgraph, the *extract* refactorings

[9] 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.

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%).

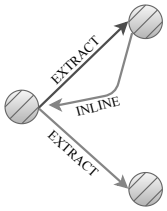


Fig. 7 Example of an misleading refactoring graph pattern (*inline* → *extract*)

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,*[10] *and 2,094 subgraphs in JavaScript.*[11] We fixed *support* = 13 (Java) and *support* = 8 (JavaScript). We set these thresholds after experiments where we strived to balance two variables: execution time and a reasonable number of occurrences 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.

4.1.5 Overview of Data Collection and Analysis

Table 5 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.

[10] All 9,200 subgraphs presented in Table 3 minus the 289 subgraphs with cycles.
[11] All 2,141 subgraphs presented in Table 4 minus the 47 subgraphs with cycles.

Table 5 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

4.2 Results

4.2.1 (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 6, for Java, 29.3% of the operations are part of a refactoring subgraph over time (3,853 occurrences).

Table 6 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

In the case of JavaScript, this rate is 32.4% (902 occurrences), as shown in Table 7. 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%).

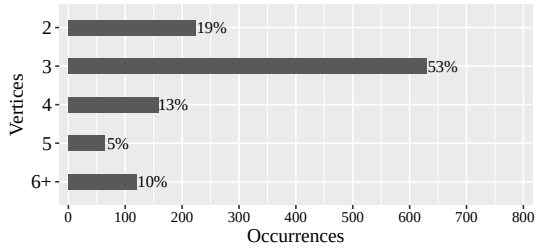
Summary of RQ0: In both languages, Java and JavaScript, about 30% of the refactoring operations are part of a refactoring subgraph over time.

Table 7 Frequency of refactoring operations in subgraphs (JavaScript)

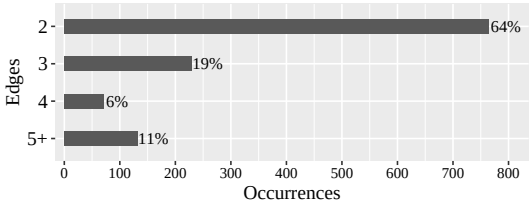
Project	Refactoring Subgraphs				
	All	Atomic	%	Over time	%
Vue	394	221	56.1	173	43.9
React	1,029	759	73.8	270	26.2
Parcel	130	99	76.2	31	23.8
Hexo	264	178	67.4	86	32.6
Leaflet	376	216	57.4	160	42.6
Quill	255	206	80.8	49	19.2
Request	108	43	39.8	65	60.2
Nylas Mail	88	73	83.0	15	17.0
Select2	98	67	68.4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

4.2.2 (RQ1) What is the Size of Refactoring Subgraphs?

As presented in Figure 8, 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).



(a) Number of vertices



(b) Number of edges

Fig. 8 Size of refactoring subgraphs (Java)

Figure 9 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 $drawYLabels()$.[12] In the subsequent commit performed 13 days later, the same developer extracted a new method from $drawYLabels()$ at commit C2.[13] 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.[14]

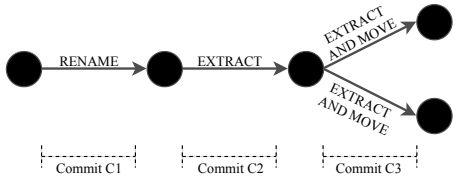
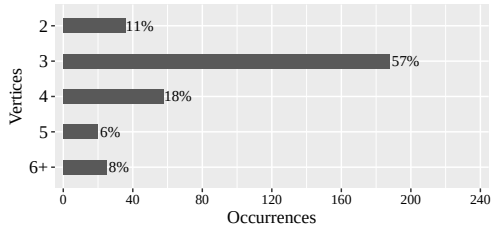
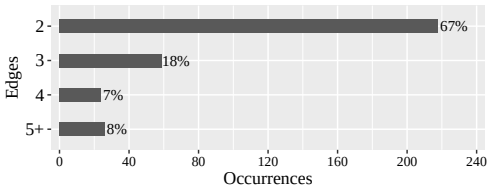


Fig. 9 Example of a refactoring subgraph from MPAndroidChart (Java)

In the case of JavaScript, most subgraphs also have three vertices (57%), as shown in Figure 10. 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.



(a) Number of vertices



(b) Number of edges

Fig. 10 Size of refactoring subgraphs (JavaScript)

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

Figure 11 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 $format$.[15] Seven months later, in commit C2, the same developer made four extract operations to function $isEnable$, aiming the removal of a single duplicated line.[16]

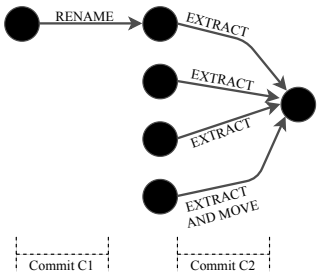


Fig. 11 Example of a refactoring subgraph from Quill (JavaScript)

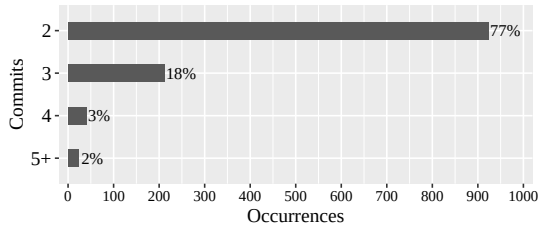
Summary of RQ1: In both languages, most refactoring subgraphs are small. Among 1,198 Java samples, most cases comprise subgraphs with the number of vertices ranging from two to four (85%) and two or three edges (83%). JavaScript reveals a similar result, most refactoring subgraphs have up to four vertices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

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

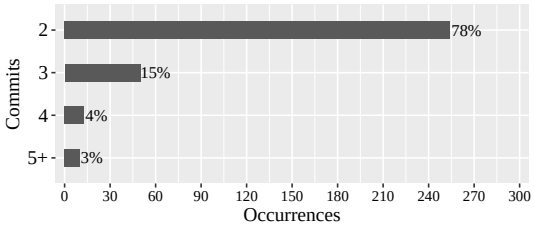
In this second question, we investigate the number of commits per subgraph. As presented in Figure 12, most cases include subgraphs with two or three commits. In Java, 95% of subgraphs (1,135 occurrences) are created from up to three commits. The largest subgraph in terms of commits is again from Square Okhttp (18 commits). Similarly, in JavaScript, 93% of subgraphs (304 occurrences) also comprise two or three commits.

Figure 13 shows an example from Elasticsearh. In commit C1, a developer moved two methods from class $SocketSelector$ to $NioSelector$.[17] After approximately three months, in commit C2, a second developer extracted duplicated code from three methods to a new method named $handleTask(Runnable)$.[18] 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.

[15] https://github.com/quilljs/quill/commit/aee9b867
[16] https://github.com/quilljs/quill/commit/e1d76d9f
[17] https://github.com/elastic/elasticsearch/commit/9ee492a3f07
[18] https://github.com/elastic/elasticsearch/commit/11fe52ad767



(a) Java



(b) JavaScript

Fig. 12 Number of commits by refactoring subgraph

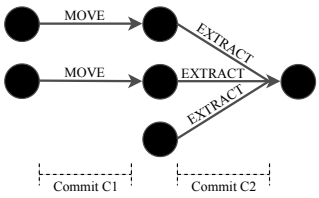


Fig. 13 Example of a refactoring subgraph from Elasticsearch (Java)

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.

4.2.4 (RQ3) What is the Age of Refactoring Subgraphs?

To assess interval, we compute the number of days between the most recent and the oldest commit in a subgraph. Figure 14 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 age 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). 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 0.115, suggesting a negligible correlation [34,66]. In other words, there are subgraphs with different ages in both old and young systems.

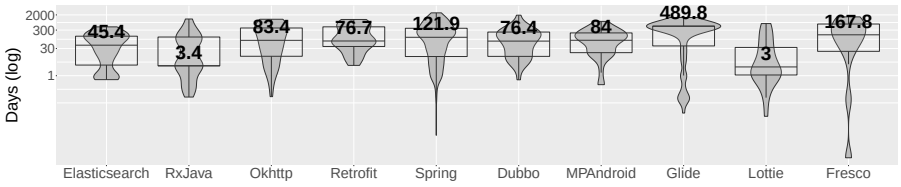


Fig. 14 Age of the refactoring subgraphs (Java)

Figure 15 presents the distribution of age 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 age of refactoring subgraphs also diverse in JavaScript. Spearman's test suggest a moderate correlation in our sample (rho = 0.624). In other words, the older the system, the older its the median time of their refactoring subgraphs.

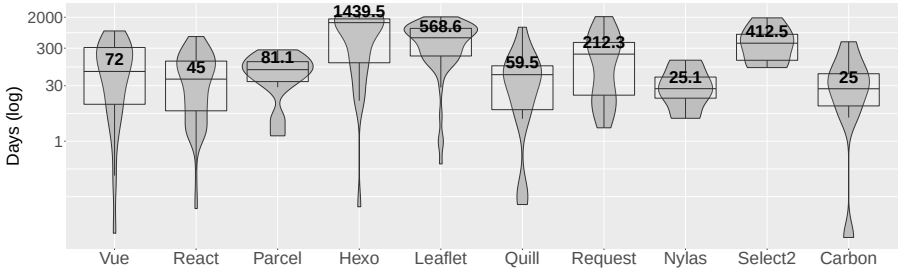


Fig. 15 Age of the refactoring subgraphs (JavaScript)

Figure 16 shows an example of a subgraph describing refactorings performed in few days on Spring Framework. In commit C1, a developer renamed method $before(...)$ to $filterBefore(...)$.[19] After six days, the same developer reverted the operation in commit C2, renaming $filterBefore(...)$ to the original name.[20] Figure 17 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 $createElm$.[21] The same devel-

[19] https://github.com/spring-projects/spring-framework/commit/794693525f
[20] https://github.com/spring-projects/spring-framework/commit/91e96d8084
[21] https://github.com/vuejs/vue/commit/351aef3c

oper performs more three operations during 15 months, extracting functions *createChildren*,[22] *createComponent*,[23] and *isUnknownElement*.[24]

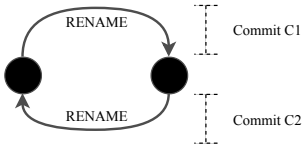


Fig. 16 Example of a refactoring subgraph from Spring Framework (Java)

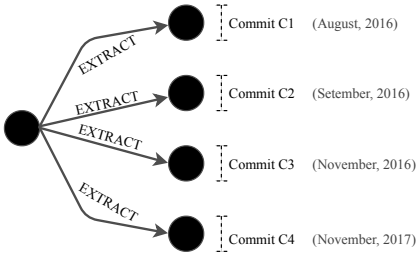


Fig. 17 Example of a refactoring subgraph from Vue (JavaScript)

Summary of RQ3: The age of the refactoring subgraphs is diverse. While some have few days, the majority of the subgraphs has weeks or even months. For example, in both languages (Java and JavaScript), about 60% of the refactoring subgraphs have more than one month.

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

Table 8 presents the most common refactoring operations in Java. Most cases include *rename method* (20%), *move method* (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).

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 9, around 28.9% of the subgraphs are homogeneous, while 71.1% are heterogeneous. The results per

[22] `https://github.com/vuejs/vue/commit/7a2c9867`
[23] `https://github.com/vuejs/vue/commit/de7764a3`
[24] `https://github.com/vuejs/vue/commit/df82aeb0`

Table 8 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

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 18, heterogeneous subgraphs often include two distinct refactoring types (60%); in contrast, 8% have three and only 3% have four or more distinct refactoring types.

Table 9 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

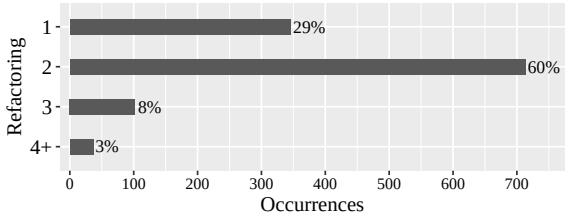


Fig. 18 Number of distinct refactorings by subgraph (Java)

As shown in Table 10, 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 11. Besides that, as shown in Figure 19, heterogenous subgraphs frequently include two distinct refactoring operations, following the same tendency of Java subgraphs.

Table 10 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

Table 11 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

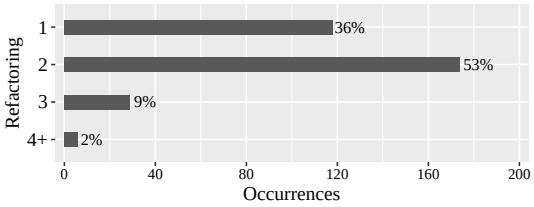


Fig. 19 Number of distinct refactorings by subgraph (JavaScript)

Figure 20 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 $ImagePipeline$.[25] The next operations happened years later when a second developer made two new *extract* operations in commits C2[26] and C3[27].

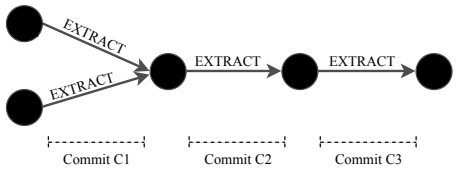


Fig. 20 Example of a homogeneous refactoring subgraph from Facebook Fresco (Java)

As a second example, we present a heterogenous subgraph from Parcel in Figure 21. In this case, a single developer performed three distinct operations in nine months by renaming function $resolveModule$ to $resolveAsset$,[28] moving it to another file,[29] and extracting function $getLoadedAsset$.[30]

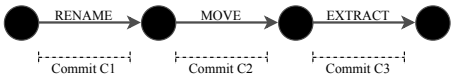


Fig. 21 Example of a heterogenous refactoring subgraph from Parcel (JavaScript)

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.

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

In the fifth 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 12, 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 13.

[25] `https://github.com/facebook/fresco/commit/02ef6e0f`
[26] `https://github.com/facebook/fresco/commit/b76f56ef`
[27] `https://github.com/facebook/fresco/commit/017c007b`
[28] `https://github.com/parcel-bundler/parcel/commit/38d4a830`
[29] `https://github.com/parcel-bundler/parcel/commit/e4cee192`
[30] `https://github.com/parcel-bundler/parcel/commit/dd3ea464`

Table 12 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 13 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 22 presents an example of a refactoring subgraph from Square Okhttp. First, in commit C1, developer D1 renamed three methods from class *OkHttpClient*.[31] Basically, the developer removed the prefix *set* from their names. After 10 months, a second developer D2 removed a duplicated code from these methods, extracting method *checkDuration*(...).[32] Then, after seven months, D2 moved this method to a new class named *Util*, in commit C3.[33] As a result, these two developers are responsible for a refactoring subgraph with eight vertices and seven edges. Figure 23 shows an opposite scenario, a subgraph from Facebook React, which was created by a single developer. After performing five *inline* operations,[34] the developer renamed a function, adding the prefix *deprecated*.[35]

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

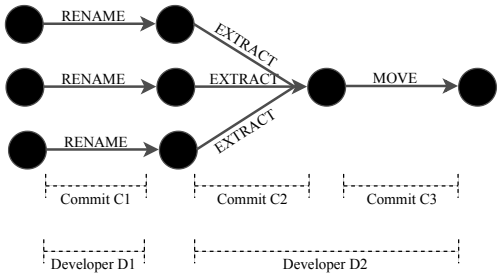


Fig. 22 Example of a refactoring subgraph created by multiple developers in Square Okhttp (Java)

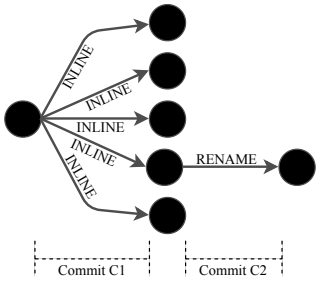


Fig. 23 Example of a refactoring subgraph created by a single developer in Facebook React (JavaScript)

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.

4.2.7 (RQ6) What are the Most Common Refactoring Subgraphs?

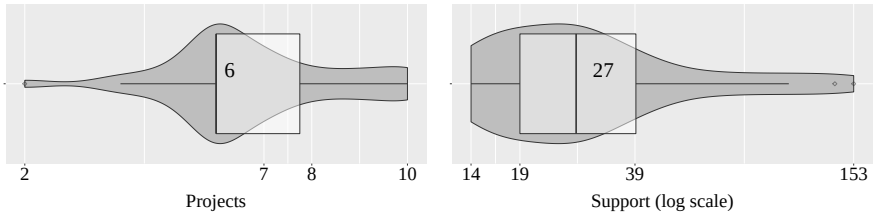
In this last research question, we mine frequent refactoring patterns. Specifically, we search for patterns that occur frequently in our dataset.

As presented in Table 14, in Java, we detect a total of 38 patterns using *GSpan* [78]. 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 14 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	13	2	15

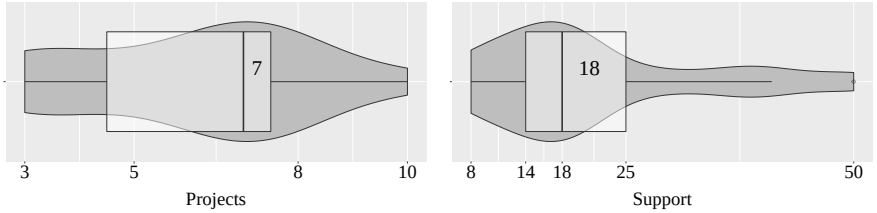
Figure 24 shows the distribution of the 38 patterns by the number of distinct projects and their support in Java. Interestingly, four patterns appear in all studied systems. Furthermore, 75% of the patterns occur in up to eight projects, and support values range from 14 to 153.



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

Fig. 24 Patterns distribution (Java)

In JavaScript, *GSpan* reports 15 patterns, 11 of then in the *over time* category (73%). Figure 25 presents the distribution of the detected patterns. The support median is 18, varying from 8 to 50.



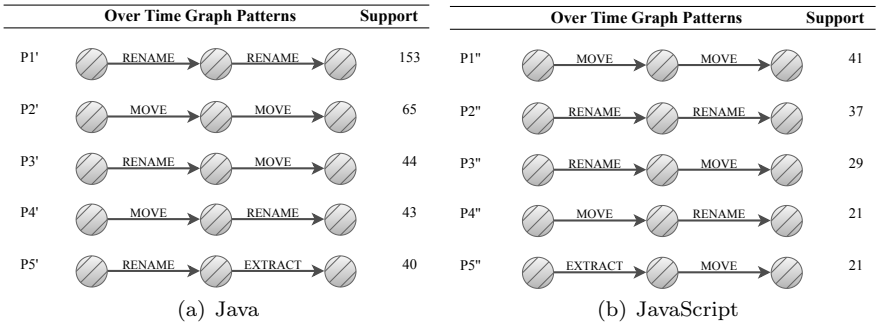
(a) Distinct projects by pattern (b) Support by pattern

Fig. 25 Patterns distribution (JavaScript)

In the remainder of the section, we provide an analysis of refactoring patterns considering their number of vertices. As shown in Table 14, this number ranges from three to five vertices.

Refactoring graph patterns with three vertices: As we can observe in Table 14, in Java, all *over time* patterns have three vertices. Figure 26 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 27 presents a subgraph from Glide with pattern $P1'$. A single developer performed the operations that represent the over time pattern in commits



(a) Java (b) JavaScript

Fig. 26 Top-5 over time graph patterns

$C1$ and $C2$. First, he renamed *buildStreamOpener* to *buildStreamLoader*.[36] The developer repeated the same operation ten days later, replacing the prefix *build* by *get* in the method' name.[37]

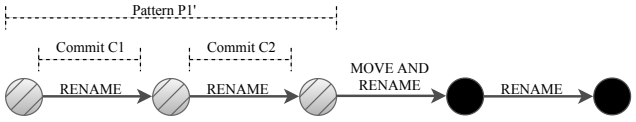


Fig. 27 Example of a refactoring graph pattern from Glide (Java, 153 occurrences)

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 14. Besides, as presented in Figure 26, the top-2 patterns are homogeneous.

Refactoring graph patterns with four vertices: In both languages, all patterns with four vertices belong to the *possibly atomic* group. Figure 28 presents an example from Spring Framework. This graph describes multiple extract operation from method $processConstraintViolations(...)$ to three methods.[38] This pattern occurs in 19 subgraphs in our dataset.

Refactoring graph patterns with five vertices: In Java, the sole graph pattern occurs in 16 subgraphs and it includes four *inline* operations. Figure 29 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.[39] There are no occurrences of patterns with five vertices in JavaScript.

[36] `https://github.com/bumptech/glide/commit/6bbe4343c`
[37] `https://github.com/bumptech/glide/commit/c572847b4`
[38] `https://github.com/spring-projects/spring-framework/commit/c43acd7675`
[39] `https://github.com/ReactiveX/RxJava/commit/320495fde`

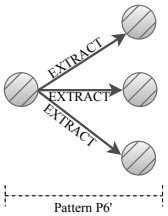


Fig. 28 Example of a possibly atomic graph pattern from Spring Framework (Java, 19 occurrences)

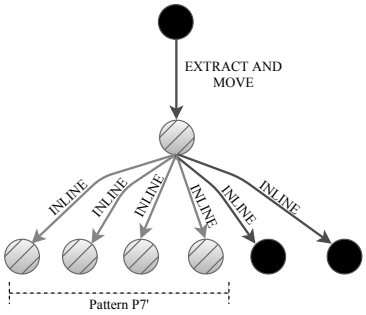


Fig. 29 Example of a refactoring graph pattern from RxJava (Java, 16 occurrences)

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).

5 Qualitative Study: Investigating Large Subgraphs

5.1 Survey Design

As we reported in Section 4, 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.

5.1.1 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.[40] In Java, the 33 subgraphs refer to 557 refactorings, which were detected by RefDiff in 120 commits, as shown in Table 15. 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 15 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 16. 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%.

5.1.2 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 30). In the

[40] `https://docs.google.com/spreadsheets/d/1eBsZW37z1w1dt77S6DIukdgGZF9fndrsVZ2vYyIh5pg`

Table 16 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

emails, we added a short description of our research goals and a screenshot of the subgraph they are responsible for. 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: `https://refactoring-graph.github.io/#/elastic/elasticsearch/713`

> 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**?

Fig. 30 Email sent to the authors of refactoring subgraphs

Table 17 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 17 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%

5.2 Survey Results

As presented in Table 18, the survey answers suggest two major reasons behind large refactoring subgraphs. In the following paragraphs, we explain and provide examples for each motivation.

Table 18 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 [57]. 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 31.[41]

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)

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

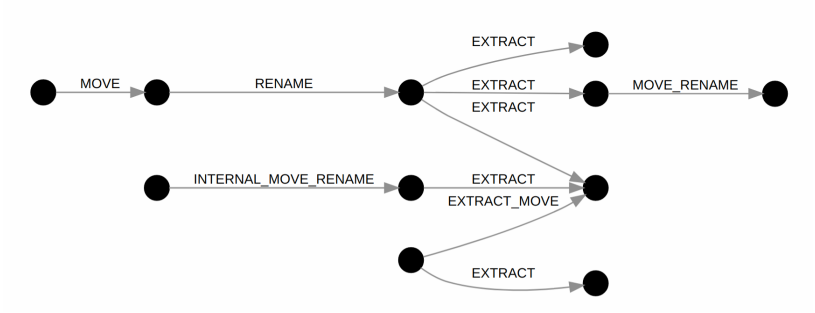


Fig. 31 Example of a large subgraph from Request (G_1, JavaScript)

Figure 32 shows a second example in this category.[42] 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:

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

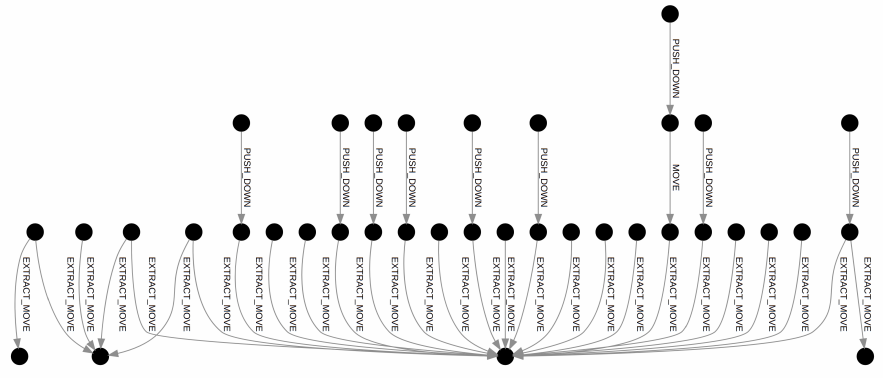


Fig. 32 Example of a large subgraph from Square Okhttp (G_4, Java)

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 [55]. 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 33.[43] D_5 explains his motivation was to improve the usage of events subscription feature:

[42] https://refactoring-graph.github.io/#/square/okhttp/485
[43] https://refactoring-graph.github.io/#/ReactiveX/RxJava/784

"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)

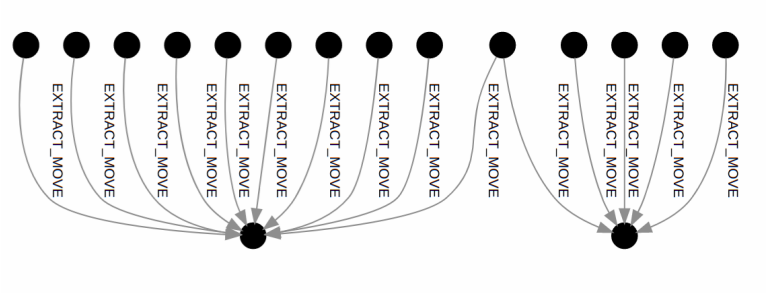


Fig. 33 Example of a large subgraph from RxJava (G_3, Java)

D_8 also points to the maintainability of a feature by pushing down a method to nine subclasses, as presented in Figure 34.[44] 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)

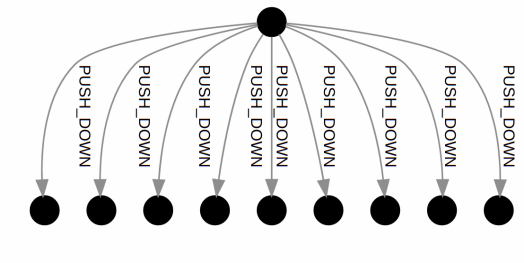


Fig. 34 Example of a large subgraph from Elasticsearch (G_6, Java)

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

[44] `https://refactoring-graph.github.io/#/elastic/elasticsearch/308`

a safe mode to instantiate a class, generating the subgraph in Figure 35:[45]

"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)

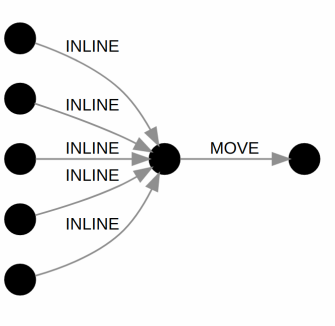


Fig. 35 Example of a large subgraph from Spring Framework (G_5, Java)

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)

6 Discussion and Implications

Refactoring over time & programming languages: In this paper, 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 19. 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).

[45] `https://refactoring-graph.github.io/#/spring-projects/spring-framework/2820`

Table 19 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	Age	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 refactorings over time: Several tools and techniques are proposed in the literature to detect refactoring operations, such as Refactoring Crawler [21], RefFinder [41], Refactoring Miner [64,73], and, more recently, RefDiff [65] and RMiner [74, 75]. 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 4, 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 [8,15,52], our approach is not constrained by the number of developers nor to a time window. Indeed, we found refactoring subgraphs with age 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 refac-*

toring 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 paper 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 [1, 8, 11, 36, 55, 67, 76].

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 [4]. However, if the reviewed change is large and complex, this task becomes challenging [4]. To alleviate this issue, refactoring-aware code review tools were proposed [13, 26, 27, 32] 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 22, 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.*

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 32 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 17 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 28). 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 [36]. 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 [3,31,51,59,69], code evolution visual supporting [28,29], bug introducing change detection [17,44,60,61,79], 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 [3,36]. 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 4, 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 [30,36]. *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.*

7 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 [55,57,64], 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 [12,63]. It is also extensible to other programming languages. In our first study [10], we concentrated on Java systems. In this second study, we include refactoring subgraphs in JavaScript. 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 [63], 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 [12]. 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.* [74, 75] proposed the refactoring detection tool RefactoringMiner. In the current version [74], 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*" [63].

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 [36], 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, the refactoring subgraphs were manually inspected by the first paper's author. 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.[46]

8 Related Work

8.1 Studies on Refactoring Evolution

Refactoring is an usual practice during software evolution and maintenance. Constantly, developers refactor the source code for different purposes [57, 64, 77]. For this reason, several studies concentrate on this research field [2, 5, 6, 8, 16, 21, 22, 35, 40, 43, 46, 47, 52, 62, 70, 72]. Among those, some research focus on assessing sets of related refactoring. Specifically, these studies analyze *batch refactorings* [8,15,23,24,52,71]. Murphy *et al.* [52] 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 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 [75], resulting in 24,893 operations. Then, the authors applied a heuristic to compute batch refactorings, i.e., set of related refactorings [15]. 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

[46] https://docs.google.com/spreadsheets/d/1eBsZW37z1w1dt77S6DIukdgGZF9fndrsVZ2vYyIh5pg

developers and different code elements. A second study reuses the heuristic proposed Bibiano *et al.* [8] and introduces two new ones [67], which are based on refactorings in the same commit and scope of the operations. As in our first study [10], the authors also discuss refactorings properties as the number of commits and refactoring types. However, the study focuses on code smells and a single programming language (Java). Other studies also discuss the impact of batches to eliminate code smells, proposing approaches to reuse or suggest sets of related refactoring operations [7, 24, 39, 71]. Thus, they do not focus on related refactoring operations over time.

In his seminal book on refactoring, Fowler [25] dedicates a chapter—co-authored with Kent Beck—to a similar term called big refactoring. The author points out that most refactorings are atomic, i.e., they are finished in a few minutes. By contrast, big refactorings are performed during months or years. However, in Fowler's book such refactorings are discussed in the context of large modularization performing to improve the architecture of a system.

Hora *et al.* [36] 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). 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 our study, the goal is to investigate refactorings performed over long time windows; we do not concentrate on tracked modifications on source code.

Meananeatra [50] 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.

8.2 Studies on Refactoring Comprehension

The literature proposes several studies on refactoring comprehension. In this case, the goal involves understanding refactoring activities by investigating, for example, benefits and challenges [42, 43], merge conflicts [48], motivations to refactor a source code [57, 58, 64, 77], association with technical debt [37], and refactoring opportunities [14].

Silva *et al.* [64] 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, the authors contacted GitHub developers by email and used *thematic analysis* to examine the responses [18]. 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, 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 this 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 [57]. 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.* [64], adding 26 new ones. The motivations are based on discussions in 551 pull requests, as well as comments in the related commits. 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 [55]. Similar to our results and previous researches [56, 57, 64], 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 RQ6, we used the *Gspan* algorithm to investigate refactoring patterns in the subgraphs [78]. 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.

9 Conclusion

In this paper, 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, age, 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.
- 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.

Further studies can consider refactoring graphs based on class level; novel approaches to complement existing tools and techniques that focus on *atomic*

refactorings; and also other popular programming languages and ecosystems (e.g., the current REFDIFF version also supports languages C and Go [12, 63]). Also, we are planning future studies on using refactoring graphs to track changes at the method level. Specifically, we intend to design and implement an *Application Interface Programming* (API) for incorporating refactoring graphs in software mining and tracing tools [19, 30, 33, 54, 68]. In such future studies and tools, we also point out possible improvements in the current refactoring graph design, such as an alternative design that handles cycles and different presentation layouts to distinguish the temporal distance between edges.

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

References

1. AlOmar, E.A., Mkaouer, M.W., Ouni, A.: Toward the automatic classification of self-affirmed refactoring. Journal of Systems and Software (JSS) **171**, 110821 (2021)
2. Alves, E.L.G., Song, M., Kim, M.: 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)
3. Avelino, G., Passos, L., Hora, A., Valente, M.T.: A novel approach for estimating truck factors. In: 24th International Conference on Program Comprehension (ICPC), pp. 1–10 (2016)
4. Bacchelli, A., Bird, C.: Expectations, outcomes, and challenges of modern code review. In: 35th International Conference on Software Engineering (ICSE), pp. 712–721 (2013)
5. Bavota, G., De Carluccio, B., De Lucia, A., Di Penta, M., Oliveto, R., Strollo, O.: When does a refactoring induce bugs? an empirical study. In: 12th International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 104–113 (2012)
6. Bavota, G., Lucia, A.D., Penta, M.D., Oliveto, R., Palomba, F.: An experimental investigation on the innate relationship between quality and refactoring. Journal of Systems and Software **107**(C), 1–14 (2015)
7. Bibiano, A., Soares, V., Coutinho, D., Fernandes, E., Correia, J., Santos, K., Oliveira, A., Garcia, A., Gheyi, R., BaldoinoFonseca, Ribeiro, M., Silva, C., Oliveira, D.: How does incomplete composite refactoring affect internal quality attributes. In: 28th International Conference on Program Comprehension (ICPC), pp. 149–159 (2020)
8. Bibiano, A.C., Garcia, E.F.D.O.A., Kalinowski, M., Fonseca, B., Oliveira, R., Oliveira, A., Cedrim, D.: A quantitative study on characteristics and effect of batch refactoring on code smells. In: 13th International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 1–11 (2019)
9. Borges, H., Hora, A., Valente, M.T.: Understanding the factors that impact the popularity of GitHub repositories. In: 32nd International Conference on Software Maintenance and Evolution (ICSME), pp. 334–344 (2016)
10. Brito, A., Hora, A., Valente, M.T.: Refactoring graphs: Assessing refactoring over time. In: International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 367–377 (2020)
11. Brito, A., Xavier, L., Hora, A., Valente, M.T.: APIDiff: Detecting API breaking changes. In: 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), Tool Track, pp. 507–511 (2018)
12. Brito, R., Valente, M.T.: RefDiff4Go: Detecting refactorings in Go. In: 14th Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS), pp. 101–110 (2020)
13. Brito, R., Valente, M.T.: RAID - Refactoring aware and intelligent diffs. In: 29th International Conference on Program Comprehension (ICPC), pp. 265–275 (2021)

14. Catolino, G., Palomba, F., Tamburri, D.A., Serebrenik, A., Ferrucci, F.: Refactoring community smells in the wild: The practitioner's field manual. In: 42nd International Conference on Software Engineering: Companion Proceedings (ICSE), pp. 25–34 (2020)
15. Cedrim, D.: Understanding and improving batch refactoring in software systems. Ph.D. thesis, PUC-Rio (2018)
16. Chaparro, O., Bavota, G., Marcus, A., Penta, M.D.: On the impact of refactoring operations on code quality metrics. In: 30th International Conference on Software Maintenance and Evolution (ICSME), pp. 456–460 (2014)
17. Chen, T.H., Nagappan, M., Shihab, E., Hassan, A.E.: An empirical study of dormant bugs. In: 11th Working Conference on Mining Software Repositories (MSR) (2014)
18. Cruzes, D.S., Dyba, T.: Recommended steps for thematic synthesis in software engineering. In: 5th International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 275–284 (2011)
19. da Costa, D.A., McIntosh, S., Shang, W., Kulesza, U., Coelho, R., 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)
20. Di Penta, M., Bavota, G., 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), pp. 556–567 (2020)
21. Dig, D., Comertoglu, C., Marinov, D., Johnson, R.: Automated detection of refactorings in evolving components. In: 20th European Conference on Object-Oriented Programming (ECOOP), pp. 404–428 (2006)
22. Dig, D., Johnson, R.: How do APIs evolve? a story of refactoring. In: 22nd International Conference on Software Maintenance (ICSM), pp. 83–107 (2005)
23. Fernandes, E.: 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)
24. Fernandes, E., Uchôa, A., Bibiano, A.C., Garcia, A.: On the alternatives for composing batch refactoring. In: 3rd International Workshop on Refactoring (IWOR), pp. 9–12 (2019)
25. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999)
26. Ge, X., Sarkar, S., Murphy-Hill, E.: Towards refactoring-aware code review. In: 7th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE), pp. 99–102. ACM (2014)
27. Ge, X., Sarkar, S., Witschey, J., Murphy-Hill, E.: Refactoring-aware code review. In: Symposium on Visual Languages and Human-Centric Computing (VL/HCC), pp. 71–79 (2017)
28. Gómez, V.U., Ducasse, S., D'Hondt, T.: Visually supporting source code changes integration: the Torch dashboard. In: 17th Working Conference on Reverse Engineering (WCRE) (2010)
29. Gómez, V.U., Ducasse, S., D'Hondt, T.: Visually characterizing source code changes. Science of Computer Programming **98**(P3), 376–393 (2015)
30. Grund, F., Chowdhury, S., Bradley, N., Hall, B., Holmes, R.: CodeShovel: Constructing method-level source code histories. In: 43rd International Conference on Software Engineering: Companion Proceedings (ICSE), pp. 1510–1522 (2021)
31. Hattori, L., Lanza, M.: Mining the history of synchronous changes to refine code ownership. In: 6th International Working Conference on Mining Software Repositories (MSR), pp. 141–150 (2009)
32. Hayashi, S., Thangthumachit, S., Saeki, M.: Rediffs: Refactoring-aware difference viewer for Java. In: 20th Working Conference on Reverse Engineering (WCRE), pp. 487–488 (2013)
33. Higo, Y., Hayashi, S., Kusumoto, S.: On tracking Java methods with git mechanisms. Journal of Systems and Software (JSS) **165** (2020)
34. Hinkle, D., Wiersma, W., Jurs, S.: Applied Statistics for the Behavioral Sciences. Houghton Mifflin (2003)
35. Hora, A., Robbes, R.: Characteristics of Method Extractions in Java: A Large Scale Empirical Study. Empirical Software Engineering **25**, 1798–1833 (2020)

36. Hora, A., Silva, D., Robbes, R., Valente, M.T.: Assessing the threat of untracked changes in software evolution. In: 40th International Conference on Software Engineering (ICSE), pp. 1102–1113 (2018)
37. Iammarino, M., Zampetti, F., Aversano, L., 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), pp. 186–190 (2019)
38. Jiang, Y., Liu, H., Niu, N., Zhang, L., Hu, Y.: Extracting concise bug-fixing patches from human-written patches in version control systems. In: 43rd International Conference on Software Engineering (ICSE), pp. 1–13 (2021)
39. Jiau, H.C., Mar, L.W., Chen, J.C.: OBEY: Optimal batched refactoring plan execution for class responsibility redistribution. Transactions on Software Engineering **39**(9), 1245–1263 (2013)
40. Kim, J., Batory, D., Dig, D., Azanza, M.: Improving refactoring speed by 10x. In: 38th International Conference on Software Engineering (ICSE), pp. 1145–1156 (2016)
41. Kim, M., Gee, M., Loh, A., Rachatasumrit, N.: 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)
42. Kim, M., Zimmermann, T., Nagappan, N.: 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)
43. Kim, M., Zimmermann, T., Nagappan, N.: An empirical study of refactoring challenge and benefits at Microsoft. Transactions on Software Engineering **40**(7), 633–649 (2014)
44. Kim, S., Zimmermann, T., Pan, K., Whitehead, E.J.J.: Automatic identification of bug-introducing changes. In: 21st International Conference on Automated Software Engineering (ASE), pp. 81–90 (2006)
45. Leung, C.: Technical notes on extending gSpan to directed graphs. Tech. rep., Singapore Management University (2010)
46. Lin, B., Nagy, C., Bavota, G., Lanza, M.: On the impact of refactoring operations on code naturalness. In: 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 594–598 (2019)
47. Lin, Y., Peng, X., Cai, Y., Dig, D., Zheng, D., Zhao, W.: Interactive and guided architectural refactoring with search-based recommendation. In: 24th International Symposium on Foundations of Software Engineering (FSE), pp. 535–546 (2016)
48. Mahmoudi, M., Nadi, S., 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), pp. 151–162 (2019)
49. Mazinanian, D., Ketkar, A., Tsantalis, N., Dig, D.: Understanding the use of lambda expressions in Java. Programming Languages **1**(85), 85:1–85:31 (2017)
50. Meananeatra, P.: Identifying refactoring sequences for improving software maintainability. In: 27th International Conference on Automated Software Engineering (ASE), pp. 406–409 (2012)
51. Meneely, A., Williams, O.: Interactive churn metrics: socio-technical variants of code churn. Software Engineering Notes **37**(6) (2012)
52. Murphy-Hill, E., Parnin, C., Black, A.P.: How we refactor, and how we know it. In: 31st International Conference on Software Engineering (ICSE), pp. 287–297 (2009)
53. Negara, S., Chen, N., Vakilian, M., Johnson, R.E., Dig, D.: A comparative study of manual and automated refactorings. In: 27th European Conference on Object-Oriented Programming (ECOOP), pp. 552–576 (2013)
54. Neto, E.C., 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), pp. 380–390 (2018)
55. Paixao, M., Uchôa, A., Bibiano, A.C., Oliveira, D., Garcia, A., Krinke, J., Arvonio, E.: Behind the intents: An in-depth empirical study on software refactoring in modern code review. In: 17th International Conference on Mining Software Repositories (MSR), pp. 125–136 (2020)
56. Palomba, F., Zaidman, A., Oliveto, R., Lucia, A.D.: An exploratory study on the relationship between changes and refactoring. In: 25th International Conference on Program Comprehension (ICPC), pp. 176–185 (2017)

57. Pantiuchina, J., Zampetti, F., Scalabrino, S., Piantadosi, V., Oliveto, R., Bavota, G., Penta, M.D.: Why developers refactor source code: A mining-based study. ACM Transactions on Software Engineering and Methodology **37**(4), 1–32 (2020)
58. Peruma, A., Mkaouer, M., Decker, M., Newman, C.: An empirical investigation of how and why developers rename identifiers. In: 2nd International Workshop on Refactoring (IWoR), pp. 26–33 (2018)
59. Rahman, F., Devanbu, P.: Ownership, experience and defects: a fine-grained study of authorship. In: 33rd International Conference on Software Engineering (ICSE), pp. 491–500 (2011)
60. Rahman, F., Posnett, D., Hindle, A., Barr, E., Devanbu, P.: BugCache for inspections: hit or miss? In: 19th International Symposium on the Foundations of Software Engineering (FSE), pp. 322–331 (2011)
61. Ray, B., Hellendoorn, V., Godhane, S., Tu, Z., Bacchelli, A., Devanbu, P.: On the naturalness of buggy code. In: 38th International Conference on Software Engineering (ICSE), pp. 428–439 (2016)
62. Shen, B., Zhang, W., Zhao, H., Liang, G., Jin, Z., Wang, Q.: IntelliMerge: A refactoring-aware software merging technique. Programming Languages **3**(170), 170:1–170:28 (2019)
63. Silva, D., da Silva, J.P., Santos, G., Terra, R., Valente, M.T.: RefDiff 2.0: A multi-language refactoring detection tool. IEEE Transactions on Software Engineering **1**(1), 1–17 (2021)
64. Silva, D., Tsantalis, N., Valente, M.T.: Why we refactor? Confessions of GitHub contributors. In: 24th International Symposium on the Foundations of Software Engineering (FSE), pp. 858–870 (2016)
65. Silva, D., Valente, M.T.: RefDiff: Detecting refactorings in version histories. In: 14th International Conference on Mining Software Repositories (MSR), pp. 269–279 (2017)
66. Silva, H., 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)
67. Sousa, L., Cedrim, D., Garcia, A., Oizumi, W., Bibiano, A.C., Oliveira, D., Kim, M., Oliveira, A.: Characterizing and identifying composite refactorings: Concepts, heuristics and patterns. In: 17th International Conference on Mining Software Repositories (MSR), pp. 186–197 (2020)
68. Spadini, D., Aniche, M., Bacchelli, A.: PyDriller: Python framework for mining software repositories. In: 26th Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE), pp. 908–911 (2018)
69. Spinellis, D.: A repository of Unix history and evolution. Empirical Software Engineering **22**(3), 1372–1404 (2017)
70. Szóke, G., Nagy, C., Ferenc, R., Gyimóthy, T.: Designing and developing automated refactoring transformations: An experience report. In: 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), pp. 693–697 (2016)
71. Tenorio, D., Bibiano, A.C., Garcia, A.: On the customization of batch refactoring. In: 3rd International Workshop on Refactoring (IWOR), pp. 13–16 (2019)
72. Terra, R., Valente, M.T., Miranda, S., , Sales, V.: JMove: A novel heuristic and tool to detect move method refactoring opportunities. Journal of Systems and Software **138**, 19–36 (2018)
73. Tsantalis, N., Guana, V., Stroulia, E., Hindle, A.: A multidimensional empirical study on refactoring activity. In: 23th Conference of the Center for Advanced Studies on Collaborative Research (CASCON), pp. 132–146 (2013)
74. Tsantalis, N., Ketkar, A., Dig, D.: RefactoringMiner 2.0. IEEE Transactions on Software Engineering (2020)
75. Tsantalis, N., Mansouri, M., Eshkevari, L.M., Mazinanian, D., Dig, D.: Accurate and efficient refactoring detection in commit history. In: 40th International Conference on Software Engineering (ICSE), pp. 483–494 (2018)
76. Vassallo, C., Grano, G., Palomba, F., Gall, H., Bacchelli, A.: A large-scale empirical exploration on refactoring activities in open source software projects. Science of Computer Programming **180**, 1–15 (2019)
77. Wang, Y.: What motivate software engineers to refactor source code? evidences from professional developers. In: International Conference on Software Maintenance (ICSM), pp. 413–416 (2009)

78. Xifeng Yan, Jiawei Han: gSpan: graph-based substructure pattern mining. In: 2nd International Conference on Data Mining (ICDM), pp. 721–724 (2002)
79. Zimmermann, T., Kim, S., Zeller, A., Whitehead Jr., E.J.: Mining version archives for co-changed lines. In: 3rd International Workshop on Mining Software Repositories (MSR), pp. 72–75 (2006)

About the authors

Aline Brito is a PhD candidate in the Computer Science Department at the Federal University of Minas Gerais (UFMG), where she also received a Master's Degree in Computer Science. She received a Bachelor's Degree in Computer Engineering from the Pontifical Catholic University of Minas Gerais (PUC Minas). Brito also was a software developer for five years. Her research interests include software quality analysis, software maintenance and evolution, and software repository mining. Contact her at `alinebrito@dcc.ufmg.br`; `alinebrito.com`.

Andre Hora is a professor in the Computer Science Department at the Federal University of Minas Gerais (UFMG). His research interests include software evolution, software repository mining, and empirical software engineering. Hora received a PhD in Computer Science from the University of Lille. He was a Postdoctoral researcher at the ASERG/UFMG group during two years and a software developer at Inria/Lille during one year. Contact him at `andrehora@dcc.ufmg.br`; `www.dcc.ufmg.br/~andrehora`.

Marco Tulio Valente is an associate professor in the Computer Science Department at the Federal University of Minas Gerais (UFMG), where he also heads the Applied Software Engineering Research Group (ASERG). His research interests include software architecture and modularity, software maintenance and evolution, and software quality analysis. Valente received a PhD in Computer Science from the Federal University of Minas Gerais. He is a Researcher I-D of the Brazilian National Research Council (CNPq) and holds a Researcher from Minas Gerais State scholarship, from FAPEMIG. Contact him at `mtov@dcc.ufmg.br`; `www.dcc.ufmg.br/~mtov`.