Behind the Intents: An In-depth Empirical Study on Software Refactoring in Modern Code Review

Matheus Paixão
UNIFOR, Fortaleza, Brazil
matheus.paixao@unifor.br

Anderson Uchôa
PUC-Rio, Rio de Janeiro, Brazil
auchoa@inf.puc-rio.br

Ana Carla Bibiano
PUC-Rio, Rio de Janeiro, Brazil
abibiano@inf.puc-rio.br

Daniel Oliveira
PUC-Rio, Rio de Janeiro, Brazil
doliveira@inf.puc-rio.br

Alessandro Garcia
PUC-Rio, Rio de Janeiro, Brazil
afgarcia@inf.puc-rio.br

Jens Krinke
UCL, London, United Kingdom
j.krinke@ucl.ac.uk

Emilio Arvonio
UNISA, Salerno, Italy
e.arvonio@gmail.com

ABSTRACT

Code refactorings are of pivotal importance in modern code review. Developers may preserve, revisit, add or undo refactorings through changes' revisions. Their goal is to certify that the driving intent of a code change is properly achieved. Developers' intents behind refactorings may vary from pure structural improvement to facilitating feature additions and bug fixes. However, there is little understanding of the refactoring practices performed by developers during the code review process. It is also unclear whether the developers' intents influence the selection, composition, and evolution of refactorings during the review of a code change. Through mining 1,780 reviewed code changes from 6 systems pertaining to two large open-source communities, we report the first in-depth empirical study on software refactoring during code review. We inspected and classified the developers' intents behind each code change into 7 distinct categories. By analyzing data generated during the complete reviewing process, we observe: (i) how refactorings are selected, composed and evolved throughout each code change, and (ii) how developers' intents are related to these decisions. For instance, our analysis shows developers regularly apply non-trivial sequences of refactorings that crosscut multiple code elements (i.e., widely scattered in the program) to support a single feature addition. Moreover, we observed that new developers' intents commonly emerge during the code review process, influencing how developers select and compose their refactorings to achieve the new and adapted goals. Finally, we provide an enriched dataset that allows researchers to investigate the context and motivations behind refactoring operations during the code review process.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**; **Software evolution**.

KEYWORDS

Refactoring, Code Review Mining, Developers' Intents

MSR '20, October 5–6, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7517-7/20/05…$15.00
https://doi.org/10.1145/3379597.3387475

ACM Reference Format:
Matheus Paixão, Anderson Uchôa, Ana Carla Bibiano, Daniel Oliveira, Alessandro Garcia, Jens Krinke, and Emilio Arvonio. 2020. Behind the Intents: An In-depth Empirical Study on Software Refactoring in Modern Code Review. In *17th International Conference on Mining Software Repositories (MSR '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3379597.3387475

1 INTRODUCTION

Modern code review platforms such as Gerrit [2, 20] is increasingly being adopted in both industrial [44] and open-source [43] projects. Along code reviews, developers inspect and discuss the quality of each other's code changes before accepting them. Code refactoring plays a key role in modern code review [1, 19]. Refactorings are rarely ignored during code review [19]. In addition, modern code review increases the awareness of developers on the importance of code refactoring [1, 19]. Refactoring consists of applying one or more structural code transformations (i.e., refactoring operations) [18] as a means to achieve various developers' intents [23, 29, 41] and goals. The intents may vary from pure structural improvement to facilitating feature additions or bug fixes [23, 29, 30, 41].

However, there is little understanding on the refactoring practices performed by developers during the code review process. For instance, a recent study [46] have found that Extract Method [18] operations are commonly employed in code changes intended at facilitating feature additions and/or bug fixes. However, even for an apparently simple refactoring type, such as an Extract Method, it may not be easy to get it right in the first attempt [32, 49]. Moreover, developers commonly employ more than one refactoring operation to achieve a certain goal [6]. As a result, refactoring operations may be revisited, preserved, added or undone in each new revision of a code review. In addition, the developers' intents may influence how they select, compose and evolve refactoring operations during the code review process.

The understanding of how refactorings are performed along code reviews with diverse intents is of paramount importance. There is a growing body of refactoring techniques in the literature [21, 24, 26–28, 33, 34, 42]. However, it is not clear whether they are well aligned with the practice of modern code review. Existing empirical studies tend to analyze refactoring operations employed only when the developer has the explicit intent of refactoring [6, 40]. Other studies simply analyze the frequency of refactorings with different motivations [46] or intents [23, 41]. Moreover, existing literature tend to analyze refactoring changes *a posteriori* [4, 6, 10, 29, 46], which

is a method that provide no context to understand how developers compose and evolve refactoring operations as a code change evolves. Hence, to the best of our knowledge, there is no study that performs an in-depth investigation regarding the relationship between software refactoring and code review, especially when considering different developers' intents and their influence on refactoring and reviewing practices.

In this paper, we address these gaps by analyzing how developers perform code refactoring in the context of modern code review. We observe how refactoring differs across changes driven by different intents. By exploring information available along each review, we also observe how developers compose and evolve their refactorings throughout each code change. We first collected code review data of 6 real-world software systems from two large open source communities. We identified and analyzed 1,780 code reviews that employed a total of 7,259 refactoring operations identified in 13 commonly used refactoring types [29].

To achieve our study goals, we inspected and classified the developers' intents behind each code review that employed refactoring operations, reaching 7 distinct intents. We further identified and distinguished code changes with explicit and non-explicit refactoring intents. We also identified and analyzed the trends on how developers compose refactorings while reviewing a code change with explicit and non-explicit refactoring intents. Finally, we identified and analyzed five refactoring evolution patterns found during code review. Our contributions include: (i) findings on how refactorings are performed along code reviews with different intents, (ii) a discussion about the implications of these findings, and (iii) a new enriched code review dataset [39] that allows researchers to investigate the context and motivations behind refactoring operations during code review. We summarize our findings as follows:

(1) As expected, simple refactoring types, such as extracting and renaming methods, are the most common in changes intended at adding features. However, these simple operations are often not applied in isolation. They are often part of non-trivial sequences of refactoring operations, which: (i) include complex refactorings, such as moving class members, and (ii) are added or even undone along a feature-related change. These observations encourage the investigation of recommenders that better support developers along reviews intended at adding features.
(2) As expected, explicit refactoring changes often involve complex refactoring operations, such as moving or extracting classes. However, our findings contradict observations made in a recent study [19], which reports developers do not ignore explicit refactoring-related comments during review. We noticed that: (i) these changes had much less revisions than others, and (ii) the refactoring operations were rarely refined or undone along the code review. This is an intriguing finding as recent studies have shown that pure refactoring often contributes to new code smells [6, 10] or bugs [3, 17]. These observations also reinforce the importance of applying existing techniques for supporting developers when performing explicit refactoring [24, 27, 28, 34].
(3) We observed that new developers' intents commonly emerge along a review and influence how developers select and compose their refactorings. These observations show how interactive the refactoring process is along code reviews, i.e., developers tend to revisit, refine, and sometimes undo their refactorings based on their peers' feedbacks. This also demonstrates the importance of refining existing techniques for refactoring-aware code reviews (e.g., [19]), so that developers interactively receive information and recommendations that guide them on refining and selecting refactorings that contribute to their intents.

2 BACKGROUND AND RELATED WORK

Modern Code Review is a lightweight, informal, asynchronous, and commonly tool-assisted practice aimed at detecting and removing defects in parts of a software project [2]. Examples of defects include bugs, performance issues, (un)intentional violations of design and/or architectural principles or rules, and style violations [5, 50]. The modern code review process is initiated by the code owner that *modifies the original code base* and *submits a new code change* to be reviewed. Other developers on the development team will serve as reviewers for the code change. Each reviewer *inspects the code change*, looking for defects, as described above. After completing their inspection, each reviewer *provides feedback in the form of comments* to the code owner. This cycle is repeated until the reviewers reach an agreement to approve or reject the proposed code change.

Reviews usually take several iterations, consisting of reviewers providing feedback and the code owner making subsequent changes in parts of the system under review in response to the feedbacks until an agreement is reached. In our study, we use *review* to indicate the entire process of a single code review, from submitting a new code change for review to approving or rejecting the integration of the change into the codebase. In addition, we use *revision* to indicate the different iterations during the cycle of a single review.

Refactoring and Developers' Intents. Software refactoring is a common development practice that aims at improving the internal structure of a software system [18]. Previous studies have shown that developers apply refactoring not only on maintenance tasks but also in other tasks of the software development lifecycle [23, 41, 46]. Such studies indicated that feature adding and bug fixing are also intents developers have when they perform refactoring. This phenomena has became known in the literature as *floss refactoring*.

The first grasps on the developers' intents and motivations when performing refactoring emerged in the studies provided by Murphy-Hill et al. [29] and Negara et al. [30], in which the authors investigated the differences between manual and automated refactoring. These studies observed that refactoring operations are commonly performed in conjunction with other types of changes, mostly as preparation for introducing a new feature and bug fixing.

Silva et al. [46] monitored Github projects to detect refactoring operations performed in 124 systems. As these changes were incorporated into the systems, the authors performed email interviews and surveys with the changes authors' to assess their motivations behind the refactoring operations. Extracting a method was mentioned as the most common refactoring operation, where the main motivation is the preparation for new feature developments.

In previous work, we performed an investigation on the developers' intents behind software changes during code review [36, 37].

We automatically identified significant architectural changes and investigated the context in which such changes were performed. A similar investigation was carried on by Tufano et al. [54], where the authors identified the developers' intents behind commits that introduced and removed code smells. In a recent study, we analyzed refactoring sequences and observed that developers often apply the same refactoring type during a single commit [6]. However, we did not investigate how developers compose and apply refactoring sequences based on their intents.

3 MOTIVATING EXAMPLE

We adopt review 58223 [12] from the couchbase-java-client system (see Section 4.2) to motivate our empirical study and depict the phenomenon we investigate. Through the course of this particular code review, different aspects of the code change have been discussed, such as tests, licenses, style etc, which led to several modifications in the source code. However, to remain concise, we focus on the discussions and source code transformations related to the developers' intents and refactoring operations.

Figure 1 presents the refactoring operations performed in the lifecycle of review 58223 (see Section 4.2 for details on the refactorings identification procedure). This review took place from 01/04/2016 to 01/19/2016, and it is composed of 15 revisions, which are indicated in the bottom of the figure. The goal for this code change was to introduce a new set of classes to read specific parts of a JSON file without the need to process the whole document, where this internal API could be re-used by different parts of the codebase.

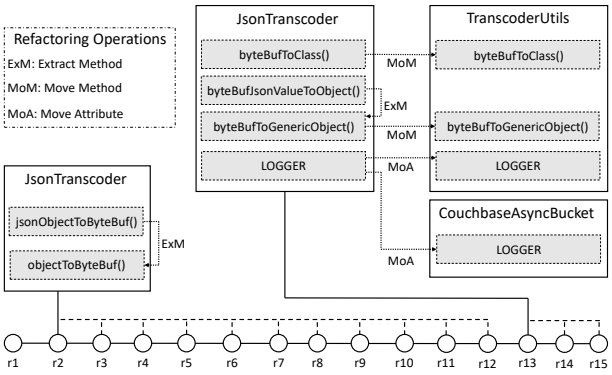

Figure 1: Refactoring operations performed during the lifecycle of review 58223 from couchbase-java-client.

An excerpt of the review's description reads as *"[The new] Subdocument API allows to mutate and read specific fragments of JSON inside an existing document without having to transfer the whole document (...)"*. One can clearly infer from the description that the code owner had the intent of adding a new feature to the codebase. Hence, the first revision consists mostly of new code being added to implement the new JSON reading functionality.

In the second revision, we observe the first refactoring operation performed in this review. Before this review started, the class JsonTranscoder had a method called jsonObjectToByteBuf() that was used to parse a JSON object into bytes to be further processed. Since the developer's goal was to create a generic set of utility classes to read JSON files, the original code from jsonObjectToByteBuf() was extracted into a new and more generic method called objectToByteBuf(). Notice that this refactoring operation was driven by the developer's goal of adding a new feature, which, at the time this revision was submitted, did not fully exist yet. Hence, existing refactoring recommenders that focus solely on structural improvement [26, 48, 51] would fail to support the developer in this refactoring activity.

In the next 10 revisions (from r3 to r12), the code owner modified parts of the code other than class JsonTranscoder. Thus, the refactoring operation discussed above remained part of the code change until revision 12. In the meantime, reviewers of this code change provided feedback regarding the codebase structure as a result of the new feature being added. One of the reviewers pointed out: *"What do you think about adding a DocumentFragment<T> interface and implementing it as JSON (...) ?"*. This comment indicates that the developers involved in this review recognized an opportunity to improve the system's structure while introducing a new feature. Hence, from this point forward, the developers' intent for this particular review changed from solely adding a new functionality to combining codebase improvement and feature implementation.

As a result, in revision 13, the code owner modified class JsonTranscoder again. In this new revision, the author first reverted the class to its original state in the codebase. Hence, the refactoring operation that has been first performed in revision 2 has been undone. Instead, more complex refactorings have been performed to reflect the review's new goal of improving the codebase structure. Method byteBufToClass() has been moved from class JsonTranscoder to class TranscoderUtils. Next, method byteBufJsonValueToObject() has been extracted into a generic method byteBufToGenericObject(), which has been later moved to class TranscoderUtils. Finally, attribute LOGGER has been moved from class JsonTranscoder to classes TranscoderUtils and CouchbaseAsyncBucket.

This example depicts different aspects of how the developers' intents towards a code change influence the way they employ refactoring operations. In addition, it shows how the iterative nature of the code review process may lead developers to change their intent during the course of a review. First, we showed how refactoring operations may be employed to support the addition of new features when this is the sole goal of a code change. Next, the feedback provided by other developers during the reviewing process led to changing the original goal for the code change. As a result, previously performed refactoring operations were undone and new refactorings were performed to reflect the new intents.

4 STUDY SETTINGS

4.1 Research Questions

RQ*$_1$*: What are common refactoring types employed under each specific intent? – **RQ**$_1$ aims at investigating the refactoring types employed when developers have different intents for a code change under review, e.g., feature adding and bug fixing. We measure the most and least common refactoring types employed, their distributions, and how they vary according to different intents. By answering **RQ**$_1$, we are able to reveal new observations on the refactoring types employed under different intents. This is useful for researchers when devising refactoring approaches that account for the developers' intents. Moreover, by analyzing the developers'

intents, we depict how refactorings are employed to support the author's original intent and intents that emerge during code review, which has not yet been discussed in the literature.

RQ$_2$: *How do developers compose refactoring sequences to support their intents?* – **RQ$_2$** aims at complementing the knowledge obtained in the previous question by investigating how developers compose refactoring sequences, i.e., refactoring operations applied in conjunction, to support their intents. By answering **RQ$_2$**, we reveal new observations about how developers compose refactoring sequences to support their intents during code review. We also highlight the most common compositions of refactoring sequences when developers have the explicit intent of refactoring or not.

RQ$_3$: *How do code changes that employ refactoring operations evolve during code review?* – **RQ$_3$** aims at investigating how refactoring operations evolve during the process of code review. By answering this question, we are able to reveal five different refactoring evolution patterns, providing new insights on the refactoring practices employed during the reviewing process. Previous studies [4, 6, 10, 46] only assess refactoring application a *posteriori*, where one cannot observe such evolution patterns. By studying refactoring operations *while* they are applied in code review, we are able to move forward the empirical knowledge on refactoring practices.

4.2 Study Steps and Procedures

Step 1: Select software systems that adopt modern code review. We selected systems provided by the Code Review Open Platform (CROP) [35], an open-source dataset that links code review data with their respective code changes. CROP currently provides data for 11 systems, accounting for a total of 50,959 code reviews and 144,906 revisions extracted from two large open source communities: Eclipse and Couchbase. All systems in CROP employ Gerrit [20] as their code review tool. Hence, by using CROP, we have access to a rich dataset of source code changes that goes beyond other platforms, such as Github. The CROP data include not only the source code change in itself, but also all the feedback and comments during review, the change's description evolution, links to the issue tracking system and so on. We selected only Java systems included in the CROP dataset due to limitations of the RefMiner tool [53] (See Step 2). Table 1 provides details about each selected system, where the Eclipse and Couchbase systems are presented in the upper and bottom halfs of the table, respectively. We also detail the number of merged reviews and revisions in each system followed by the time-span of our investigation. Finally, we report the median, maximum and minimum values of kLOC.

Table 1: Software systems investigated in this study

Systems	# of Reviews	# of Revisions	Time Span	kLOC Min	kLOC Med	kLOC Max
egit	4,502	11,430	9/09 to 11/17	16.07	70.59	107.661
jgit	4,463	11,891	10/09 to 11/17	34.00	84.25	114.36
linuxtools	3,695	10,892	6/12 to 11/17	89.99	170.28	205.89
java-client	798	2,394	11/11 to 11/17	0.55	9.3	29.16
jvm-core	785	2,184	4/14 to 11/17	1.78	13.68	24.59
spymemcached	383	1,098	5/10 to 7/17	7.19	10.78	13.68

Step 2: Identify refactoring operations during code review. We used the RefMiner tool [52] to identify refactoring operations according to 13 refactoring types that are commonly employed by developers [46]. We have identified refactoring operations by considering each revision of a code change, where each revision was compared to its parent, i.e., the codebase's version before any revision (including the previous ones) was applied. For details regarding this procedure, we recommend a recent empirical study we performed dedicated to this topic [38]. In studies performed by RefMiner's authors, the tool is reported to achieve 98% of precision and 93% of recall [46, 52], which makes it the current state-of-the-art tool for automated refactoring detection. RefMiner has been constantly developed and evolved, where the latest stable release dates from May, 2018. However, its latest version has not been employed in recent studies. On the other hand, its earlier releases have been used and evaluated in studies by researchers other than the tools' authors [10, 17, 29], achieving similar scores of precision and recall as in its original proposed paper. Hence, we chose to employ an earlier version of RefMiner (version 0.2.0) [52] due to its results in studies performed by both the tools authors and other researchers. Table 2 lists the 13 refactoring types identified by RefMiner. We identified refactoring types that affect different scopes of a code element: four types that affect a class or interface; six types that affect methods; and three types that affect an attribute.

Table 2: Refactoring types investigated in this study

Scope	Refactoring Type	Description
Class or interface	Extract Interface	Extract an interface from an existing class
	Extract Superclass	Extract a superclass from an existing class
	Move Class	Move a class across packages
	Rename Class	Update name of an existing class
Method	Extract Method	Extract a new method from an existing one
	Inline Method	Move a method body to an existing method
	Move Method	Move a method across classes
	Pull Up Method	Move method from child to parent class
	Push Down Method	Move method from parent to child class
	Rename Method	Update name of an existing method
Attribute	Move Attribute	Move an attribute across classes
	Pull Up Attribute	Move attribute from child to parent class
	Push Down Attribute	Move attribute from parent to child class

We identified 1,780 code changes that employed refactoring operations for a total of 7,259 refactoring operations performed when considering all selected systems. We also observed that the percentage of reviews that perform refactoring operations is consistent throughout all analyzed systems (from 11% to 14% of reviews). Moreover, most of the reviews have two or more refactoring operations, often reaching 4 or more. Consider the code review example discussed in Section 3, for example. All refactoring operations depicted in Figure 1 were automatically identifed by RefMiner. The complete set of refactoring operations identified for all revisions in our dataset is available in our replication package [39].

Step 3: Manually inspect and classify the developers' intents behind code review discussions. We considered all 1,780 reviews that employed refactoring operations to perform a manual inspection and classification of the developers' intents based on the review's description and reviewers' feedback through comments. We performed our manual classification by adopting a state-of-the-art procedure to classify the developers' intent in a code change [36, 37]. As part of the procedure, we considered all the data involved in a code review, such as commit messages, discussions between developers, links to issue tracking systems, and source

code. In this paper, we consider the intent of a code change to be the goals and motivations of the change's author(s). Table 3 lists the developers' intents we identified in our study. We provide a description of each intent followed by an excerpt from the reviews' discussion to serve as example of our classification process.

Table 3: Developer's intents along code review

Intents	Description
Feature	Developer is adding or enhancing a feature, e.g., *"Add option to replace selected files with version in the git index."*
Refactoring	Developer is refactoring the system, e.g., *"Refactor View mapping into distinct class (...) query handling is moved into a separate class"*
Bug Fixing	Developer is fixing a bug, e.g., *"Fix failing unit tests introduced by (...)"*
Feature Removal	Developer is removing an obsolete feature, e.g., *"Retire org.eclipse.ui.examples.presentation plug-in"*
Platform Update	Developer is updating the code for a new platform/API, e.g., *"Bump to BREE 1.6 to be consistent"*
Merge Commit	Developer is merging two branches, e.g., *"Merge branch stable-0.8"*
Not Clear	There is no evidence to suggest any of the previous.

The manual classification process consisted of two authors analyzing the data of each code change and identifying the developers' intent. We employed a two-phase process: 1) two authors solely and separately inspected and classified all reviews; 2) the authors discussed all the code changes for which there was a disagreement in the classification. During the classification process, we identified reviews with mixed intents, i.e, reviews with more than one intent, such as *Feature/Refactoring*, and *Feature/Refactoring/Bug Fixing*. We emphasize there was no disagreement on any code change after the second stage of classification. Consider the code review discussed in Section 3. Based on the review's data, as described above, we identified a *Feature/Refactoring* intent for this review. The set of manually classified code reviews that employed refactoring operations are available in our replication package [39].

Step 4: Validation of code reviews that present a refactoring intent. Due to RefMiner's limitations, there might be code reviews in which the developers present a refactoring intent but no refactoring operation is identified. This would bias our study by investigating only the code reviews in which RefMiner is capable of identifying a refactoring operation, and potentially missing on other reviews with a refactoring intent. To assess this threat to our study's validity, we make use of the code reviews' classification performed by our previous work [37]. In this study, we classified code reviews according to their architectural impact. Hence, we can employ this classification as a partial ground truth for code reviews that present a refactoring intent regardless of the presence or absence of refactoring operations. Thus, for each review previously reported [37] as having a refactoring intent (332 reviews), we used RefMiner to identify the refactoring operations that might have been employed. When considering all systems studied in the previous work, we observed that 196 (59%) of the reviews with a refactoring intent employed at least one refactoring operation. Regarding the 136 (41%) remaining reviews, we performed a qualitative analysis to investigate the reasons RefMiner was not able to identify refactoring operations.

As a result, we observed that only 4% of these reviews contained false positives, i.e., reviews that contained refactoring operations that RefMiner did not identify. The data and code for 6% of the reviews was noisy and impure, where we could not qualitatively identify whether a refactoring operation was employed or not. For the other 31% of reviews, the developers claimed to be performing a refactoring where in fact other changes were performed, such as performance improvements, for example. Hence, based on these analyses, we assess that our empirical study is only negligibly affected by RefMiner's limitations. Additional details of this validation are available in our replication package [39].

5 RESULTS AND DISCUSSION

5.1 Refactoring Types per Review's Intent

Refactoring operations and intents. We address $\mathbf{RQ}_1$ by analyzing the distribution of the number of reviews that employed refactoring operations grouped by developers' intent and refactoring types. Next, we analyze the most and least common refactoring types employed, including their distributions, and how they vary according to different intents. Figure 2 illustrates the distribution of code reviews that employed refactoring operations grouped by the developer's intent.

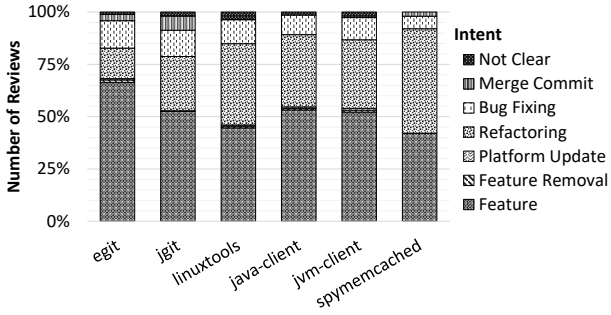

Figure 2: Distribution of reviews that employed refactoring operations grouped by developer's intents and systems

The figure shows that most of the code changes that employ refactoring operations present a *Feature* intent, accounting for 54.5% of all reviews. In 27.3% of the reviews, developers have the sole intent of *Refactoring*. *Bug Fixing* appears as the third most common intent: developers refactored the system to fix a bug in 11.9% of the reviews. Finally, the number of reviews identified with the intent of *Feature Removal*, *Platform Update* and *Merge Commit* are negligible in comparison to the most popular intents previously discussed. These results indicate that developers most commonly employ refactoring operations when they do not have an explicit intent of refactoring, i.e., the refactoring is mixed with other changes, normally for implementing a new feature or fixing a bug. These observations partly complement previous studies regarding developers' intents and motivations behind refactoring [23, 41, 46], strengthening the empirical evidence in this topic.

> **Finding 1**: Developers most commonly employ refactoring operations when they aim at introducing a new feature or enhancing an existing one.

Refactoring types and non-explicit refactoring intents. Table 4 presents the number of reviews that employed refactoring operations grouped by the developers' intent and refactoring types. Each cell represents the number of reviews that employ a certain refactoring type while having a certain intent. In Table 4, we do not consider the number of refactoring operations of the same

Table 4: Number of reviews that employ refactoring operations grouped by developers' intent and refactoring type

Refactoring Types	Feature Only	Refactoring Only	Feature/Refactoring	Bug Fixing Only	Feature/Bug Fixing	Refactoring/Bug Fixing	Feature/Refactoring/Bug Fixing
Extract Interface	16 (1.25%)	5 (0.86%)	7 (2.61%)	0 (0%)	2 (2.08%)	0 (0%)	0 (0%)
Extract Superclass	52 (4.05%)	21 (3.62%)	19 (7.09%)	3 (1.72%)	3 (3.13%)	0 (0%)	0 (0%)
Move Class	52 (4.05%)	72 (12.41%)	15 (5.60%)	5 (2.87%)	3 (3.13%)	2 (6.67%)	2 (6.67%)
Rename Class	59 (4.59%)	52 (8.97%)	14 (5.22%)	5 (2.87%)	5 (5.21%)	1 (3.33%)	3 (10%)
Extract Method	543 (42.26%)	108 (18.62%)	58 (21.64%)	84 (48.28%)	43 (44.79%)	12 (40%)	8 (26.67%)
Inline Method	55 (4.28%)	31 (5.34%)	8 (2.99%)	6 (3.45%)	5 (5.21%)	1 (3.33%)	2 (6.67%)
Move Method	122 (9.49%)	89 (15.34%)	43 (16.04%)	17 (9.77%)	9 (9.38%)	4 (13.33%)	3 (10%)
Pull Up Method	46 (3.58%)	28 (4.83%)	15 (5.60%)	3 (1.72%)	3 (3.13%)	1 (3.33%)	2 (6.67%)
Push Down Method	12 (0.93%)	7 (1.21%)	3 (1.12%)	0 (0%)	1 (1.04%)	0 (0%)	1 (3.33%)
Rename Method	215 (16.73%)	92 (15.86%)	45 (16.79%)	46 (26.44%)	15 (15.63%)	5 (16.67%)	4 (13.33%)
Move Attribute	74 (5.76%)	48 (8.28%)	29 (10.82%)	3 (1.72%)	6 (6.25%)	4 (13.33%)	3 (10%)
Pull Up Attribute	29 (2.26%)	18 (3.10%)	10 (3.73%)	2 (1.15%)	1 (1.04%)	0 (0%)	1 (3.33%)
Push Down Attribute	10 (0.78%)	9 (1.55%)	2 (0.75%)	0 (0%)	0 (0%)	0 (0%)	1 (3.33%)
Total	**1,285 (100%)**	**580 (100%)**	**268 (100%)**	**174 (100%)**	**96 (100%)**	**30 (100%)**	**30 (100%)**

type used in the same review. Instead, we simply check if a review employs a certain refactoring type or not.

Differently from previous studies (e.g., [29, 40, 46]), our methodology enables us to identify intents that emerge during code review to support the realization of the original intent. We refer to these intents as mixed intents, i.e., reviews that had at least two different intents, such as *Feature/Refactoring*, *Refactoring/Bug Fixing*, and *Feature/Refactoring/Bug Fixing*. Hence, for the remainder of this study, we discuss single and mixed intents separately. To avoid misunderstandings, we refer to the single intents as *Feature Only*, *Refactoring Only* and *Bug Fixing Only*. Finally, we differentiate intents with an explicit intent of refactoring from their non-explicit counterparts, where the first are presented in a gray background and the latter are presented in a white background in Table 4.

Table 4 indicates that a significant number of refactoring operations are concentrated in reviews with the intents of *Feature Only*, *Feature/Refactoring*, *Refactoring Only*, and *Bug Fixing Only*. Table 4 allows us to observe that for non-explicit refactoring reviews, i.e., *Feature Only*, *Bug Fixing Only*, and *Feature/Bug Fixing*, up to 48% and 26% of reviews employed Extract Method and Rename Method, respectively. Other refactoring types are less frequent than method extraction and renaming. Each of the other refactoring types occurs in less than 10% of the non-explicit refactoring reviews.

In summary, we observed that extracting and renaming methods are the most frequent refactoring operations regardless of the intent. This result provides a different perspective on the observations made by a previous study. Silva et al. [46] listed 11 reasons that motivate developers to apply Extract Method. Most of these motivations concern the improvement of the system's internal structure. However, we observed in our motivating example (see Section 3) that the motivations for extracting a method can go beyond the listed ones. In our example, the developers applied an Extract Method refactoring to support the implementation of a new feature.

> **Finding 2**: Extracting and renaming methods are the most common refactoring operations regardless of the intent.

Refactoring types and explicit refactoring reviews. For the reviews with an explicit intent of refactoring, i.e., *Feature/Refactoring* and *Refactoring Only*, a wider range of refactoring types are employed when compared to their non-explicit counterparts. For such reviews, Extract Method is still the most common operation with 21% of reviews, while Rename and Move method are the second and third most common operations with 16% of reviews. For reviews with the *Feature/Refactoring* intent, developers move classes across packages in only 5% of the reviews. Classes are more frequently moved (12%) in reviews with a *Refactoring Only* intent.

A similar observation applies for the Rename Class refactoring, which is more employed in reviews with *Refactoring Only* intent than *Feature/Refactoring*. Moreover, refactoring operations involving classes happen more often in *Refactoring Only* (25%) and *Feature/Refactoring* (21%) reviews. As for non-explicit refactoring intents, class-level refactoring operations range from 7% (*Bug Fixing Only*) to 13% (*Feature Only*).

> **Finding 3**: Developers tend to use a more even distribution of refactoring types in reviews with an explicit refactoring intent.

5.2 Refactoring Sequences per Review's Intent

In $\mathbf{RQ}_1$, we observed that developers did not apply isolated refactoring operations of different types. Thus, we hypothesize that refactoring operations might be applied in compositions. Hence, we performed an analysis to obtain an in-depth understanding on how refactoring sequences are composed under different intents.

Identification of Refactoring Sequences: We collected refactoring sequences that were applied during the review process of a code change. We considered that a refactoring sequence is composed of two or more interrelated refactoring operations applied in subsequent revisions of a code review. Two refactoring operations are interrelated when they are applied in a common set of code elements (e.g. classes). This procedure has been applied in recent empirical studies regarding refactoring sequences and compositions [6, 7, 47]. We analyzed the refactoring sequences for the following intents: *Feature Only*, *Refactoring Only*, *Feature/Refactoring* and *Bug Fixing Only*. We focus our analyses on these intents because they are the most commonly observed in our dataset (see Section 5.1). Consider review 58223, as depicted in Section 3. The refactoring sequence identified for this review is composed of 6 refactoring operations, namely: [Extract Method, Move Method, Extract Method, Move Method, Move Attribute, Move Attribute].

Table 5 presents the number of refactoring sequences, the number of refactoring operations that compose the sequences, the number of reviews, and the number of revisions for each intent. When considering the 336 reviews and 3,027 revisions that present a *Feature Only* intent, we identified 426 refactoring sequences composed of a total of 1,621 refactoring operations. Our results indicate that

3,437 (47.34%) out of 7,259 refactoring operations were applied in sequences, where these sequences are distributed in a median of 5 revisions. Besides that, developers applied refactoring sequences on 580 (25.14%) out of 2,307 reviews. Moreover, note that some reviews had more than one refactoring sequence, such as the *Refactoring Only* intent. In this particular case, we noticed that developers applied refactoring operations in different sets of code elements during these reviews, which characterize different refactoring sequences. In addition, we have observed a total of 250 (33.55%) out of 745 refactoring sequences in reviews where developers had the explicit intent of refactoring (*Refactoring Only* and *Feature/Refactoring*).

Table 5: Number of refactoring sequences per intent

Intent	Sequences	Ref. Operations	Reviews	Revisions
Feature Only	426 (57.19%)	1,621 (47.16%)	336 (57.93%)	3,027 (65,86%)
Refactoring Only	150 (20.13%)	1,047 (30.46%)	114 (19.66%)	553 (12.03%)
Feature/Refactoring	100 (13.42%)	532 (15.48%)	63 (10.86%)	657 (14.30%)
Bug Fixing Only	69 (9.26%)	237 (6.90%)	67 (11.55%)	359 (7.81%)
Total	**745 (100%)**	3437 **(100%)**	**580 (100%)**	**4,596 (100%)**

Aimed at understanding how these refactoring sequences are composed, we created four categories based on the code element's scope of each refactoring type. Table 6 presents each category followed by a description and the refactoring types that compose the category. In addition, we present the predominance of each category over the others when refactorings of different categories are employed in sequence. Finally, we detail the number of classes each category of refactoring type affects. All these attributes will be used in this section to provide insights on how developers compose refactoring types to achieve their intents.

Table 6: Categories of refactoring types

Category	Description	Refactoring Type	Predominance	Number of Classes
Hierarchical	Refactorings that affect at least two classes of a hierarchy	Pull Up Attribute Push Down Attribute Pull Up Method Push Down Method Extract Interface Extract Superclass	Motion, Intra-class, Rename	Multiple Classes
Motion	Moving a code element within the software project	Move Attribute Move Method Move Class	Intra-class, Rename	Multiple Classes
Intra-class	Refactorings that affect a specific class	Extract Method Inline Method	–	Single Class
Rename	Renaming a project element	Rename Method Rename Class	–	Single Class

Combination: The classification by combination was based on findings of previous studies, which investigated how refactoring types are combined in refactoring sequences [6–8, 47]. However, these studies do not investigate how these combinations relate to developers' intents. We classified the refactoring sequences according to the categories of their refactoring operations and the order of their combinations. Consider the refactoring sequence for the example discussed in Section 3: [Extract Method, Move Method, Extract Method, Move Method, Move Attribute, Move Attribute]. This refactoring sequence is classified into: {Intra-class, Motion, Intra-class, Motion, Motion, Motion}.

As a result of this classification, we have found 41 combinations of these categories in our data. When considering reviews with a *Refactoring Only* intent, out of the 150 refactoring sequences, 32 (21.3%) are composed of Motion only refactoring operations; 29 (19.3%) are composed of Motion and Intra-class refactorings. These results indicate that developers have applied a non-negligible number (40.6%) of refactoring sequences that affect more than one class, and they had composed non-trivial refactoring sequences with different refactoring types when having an explicit intent of refactoring. Differently, when considering the other intents, we observed 212 (49.7%) and 38 (55%) Intra-class only refactoring sequences for the *Feature Only* and *Bug Fixing Only*, respectively. These results suggest that developers more often applied refactoring sequences on single code elements when the intent was not to explicitly refactor the code.

We have found 100 refactoring sequences in reviews with a mixed *Feature/Refactoring* intent. Out of these, 26 are composed of Motion-only refactoring operations, and 11 are composed of {Motion, Hierarchical, Motion} refactoring operations. This suggests that developers regularly (36%) applied refactoring sequences on more than one class when they had more than one intent. Developers often applied a specific order of these categories, by first moving classes, next modifying classes hierarchy, and finally moving classes to support these intents. This indicates that the order in which refactorings are employed is affected by the different intents developers have towards the code change. We provide the complete categories' classification in our replication package [39].

Refactoring sequences with Extract Methods or Rename Methods. On combinations that have the categories Intra-class or Rename, we observed that 518 (69.53%) have at least one Extract Method, and 94 (12.61%) have at least one Rename Method. Previous work reported that Extract Method and Rename Method are the most common refactoring types applied when assessed in isolation [10, 46]. However, our results present that these refactoring types are often applied in conjunction with other refactoring types.

> **Finding 4**: Extract Methods are not performed in isolation, and they often occur with other refactoring types in sequences.

Predominance: We defined the predominance category based on the scope that each category affects. Categories that affect a large scope predominate over categories that affect a small scope. Thus, the order of predominance, from the largest to the smallest scope, is as follows: Hierarchical, Motion, Intra-class, and Rename. As previously shown, the review discussed in Section 3 presents a refactoring sequence of [Extract Method, Move Method, Extract Method, Move Method, Move Attribute, Move Attribute], which yields a categories' classification of {Intra-class, Motion, Intra-class, Motion, Motion, Motion}. According to Table 6, the Motion category has predominance over Intra-class regardless the refactorings order. Thus, this refactoring sequence may be classified simply as Motion.

Table 7 presents the predominance of categories for each intent. Consider the *Feature Only* intent, for example. We observe that 212 refactoring sequences are classified into predominantly Intra-class, which accounts for 49.77% of all refactoring sequences with a *Feature Only* intent. Developers often applied categories that involved a large code scope when they had the explicit intent of refactoring. We have found that 93 (62%) out 150 refactoring sequences with *Refactoring Only* intent employed code motion between classes. In addition, 10 (15%) out these employed refactoring operations on hierarchical classes, where these refactoring sequences have a

median of 12 refactoring operations. This indicates that developers often applied refactoring on more than one class in sequences where the intent is to explicitly improve the structural quality.

Refactoring sequences and *Feature Only* intent. When considering the 426 refactoring sequences in reviews with a *Feature Only* intent, we have observed that developers employed Motion and Hierarchical refactoring operations in 140 (32.86%) and 30 (7.04%) of them, respectively.

> **Finding 5**: Developers regularly applied refactoring sequences that affect multiple classes to support feature addition.

Refactoring sequences and *Bug Fixing Only* intent. In 25 (36.23%) out of 69 refactoring sequences in reviews with a *Bug Fixing Only* intent, developers employed code motion refactoring operations. This is a surprising observation, as one would expect that moving code elements between classes is not a common practice to fix bugs. Previous work have investigated the relationship between refactoring and bug fixing [4, 17]. However, these studies have investigated isolated refactorings instead of sequences. Hence, our study is the first to observe that combinations of Move Attribute, Move Method and Move Class refactorings are used for bug fixing. For instance, in review 99067 [15] from jgit, developers applied a refactoring sequence composed of [Extract Method, Move Method, Move attribute] involving classes OpenSshConfig, OpenSshConfig.Host, and OpenSshConfig.State. This refactoring sequence was applied to fix an existing bug as suggested during review: *"This avoids a few bugs in Jsch's OpenSSHConfig (...)"*.

> **Finding 6**: Developers applied a non-ignorable number of refactoring sequences that employed code motion when the intent was *Bug Fixing Only*.

Number of Classes: This classification is based on the number of classes that each category can affect in refactoring sequences. The categories that affect a large scope often affect more classes than categories that affect a small scope. Thus, the Hierarchical and Motion categories are considered categories that affect multiple classes, and the Intra-class and Rename are categories that affect a single class (see Table 6). This classification takes into account the order of the application of each category. Hence, consider the example discussed in Section 3. A refactoring sequence composed of [Extract Method, Move Method, Extract Method, Move Method, Move Attribute, Move Attribute], would yield a classification of {Single Class, Multiple Classes, Single Class, Single Class, Multiple Classes, Multiple Classes}.

Non-trivial refactoring sequences and explicit refactoring reviews. Table 8 presents the classification of refactoring sequences regarding number of classes. One may note that 60% of the refactoring sequences with a *Feature Only* intent have been classified as {Single Class}. We observed that whenever developers have the explicit intent of refactoring, the sequences usually start on multiple classes (54% and 57% for *Refactoring Only* and *Feature/Refactoring*, respectively). In contrast, when developers have other intents, most sequences effect purely single classes (60% and 59% for *Feature Only* and *Bug Fixing Only*, respectively). This number of sequences that affect multiple classes in reviews with explicit refactoring intents leads us to believe that the single-class refactorings employed in these reviews, such as Extract Method and Rename Method, are often used to support the refactorings that affect multiple classes.

> **Finding 7**: Nearly half of the refactoring sequences with an explicit intent of refactoring tend to start on multiple classes.

Interactivity, non-explicit intent and mixed-intent. Curiously, developers needed more revisions (a median of 6) when they started with categories that involved multiple classes (Hierarchical and Motion) in refactoring sequences to support the *Feature Only* intent. On the other hand, developers also needed more revisions (a median of 5) in sequences that support the *Bug Fixing Only* intent. Besides that, the refactoring sequences with the *Feature/Refactoring* intent are composed of a median of 7 revisions. Thus, the reviews which employed refactoring sequences to support a non-explicit intent of refactoring and/or more than one intent require more interactivity among reviewers. This interactivity allows for discussions on how to compose refactoring sequences to attend different intents, as presented in our motivating example (see Section 3).

> **Finding 8**: Reviews with a non-explicit refactoring and/or mixed intents tend to present high interactivity during review.

5.3 Refactoring Evolution Across Reviews

We address $\mathbf{RQ_3}$ by proposing a new classification to represent refactoring evolution patterns during code review. We are the first to investigate this phenomenon; hence, we followed a coarse-grained strategy when reporting these observations as a first-time visualization of this data. Thus, our classification serves as a baseline for future studies. This classification consists of performing a sequential observation of the refactoring operations employed throughout all revisions in the code review. Consider a code review with three revisions, for example. We sequentially compared the refactoring operations performed in the second revision to the refactoring operations performed in the first revision. Next, we compared the refactoring operations in the third revision to the ones in the second revision. This procedure enabled us to observe five possible refactoring evolution patterns.

We describe each pattern as follows: (i) **single**, when a review has only a single revision and at least one refactoring operation; (ii) **new**, when at least one refactoring was created in a subsequent revision and this refactoring has not been undone in future revisions of the same review; (iii) **undone**, when at least one refactoring operation was undone in a subsequent revision of the same review, and no new operations were identified; (iv) **both**, when both new and undone refactoring operations are identified, regardless of the order; and (v) **same**, when exactly the same set of refactoring operations is present in all revisions of a review. These patterns are mutually exclusive and comprise all code changes in our dataset. Consider our motivating example depicted in Section 3. No refactoring operations were employed in the first revision. In the second revision, an Extract Method refactoring was performed, which characterizes a new refactoring operation in this review's lifecycle. However, in revision 13, this Extracted Method was undone, and other refactoring operations were employed instead. Hence, this review is considered to have a **both** refactoring evolution pattern.

Table 7: Predominance of Categories. We report the number of sequences, revisions and median of revisions for each category.

Predominance	Feature Only			Refactoring Only			Feature/Refactoring			Bug Fixing Only		
	Sequences (%)	Revisions (%)	Med	Sequences (%)	Revisions (%)	Med	Sequences (%)	Revisions (%)	Med	Sequences (%)	Revisions (%)	Med
Hierarchical	30 (7.04%)	281 (9.28%)	7.5	15 (10%)	49 (8.86%)	3	16 (16%)	141 (21.46%)	7.5	3 (4.35%)	16 (4.46%)	2
Intra-class	212 (49.77%)	1,387 (45.82%)	5	33 (22%)	125 (22.60%)	3	23 (23%)	139 (21.16%)	5	38 (55.07%)	239 (66.57%)	5
Intra-class, Rename	44 (10.33%)	169 (5.58%)	4	6 (4%)	28 (5.06%)	3.5	11 (11%)	71 (10.80%)	5	3 (4.35%)	7 (1.95%)	2
Motion	140 (32.86%)	1,090 (36.01%)	5	93 (62%)	335 (60.58%)	3	45 (45%)	270 (41.10%)	5	25 (36.23%)	97 (27.02%)	4
Rename	0 (0%)	100 (3.31%)	5.5	3 (2%)	16 (2.90%)	3	5 (5%)	36 (5.48%)	8	0 (0%)	0 (0%)	0
Total	**426 (100%)**	**3,027 (100%)**	**5**	**150 (100%)**	**553 (100%)**	**3**	**100 (100%)**	**657 (100%)**	**5**	**69 (100%)**	**359 (100%)**	**2**

Table 8: No. of classes along sequences. We report the number of sequences, revisions and median of revisions for each category.

Complexity	Feature Only			Refactoring Only			Feature/Refactoring			Bug Fixing Only		
	Sequences (%)	Revisions (%)	Med	Sequences (%)	Revisions (%)	Med	Sequences (%)	Revisions (%)	Med	Sequences (%)	Revisions (%)	Med
Multiple Classes, Single Class	55 (12.91%)	419 (13.84%)	6	38 (25.33%)	129 (23.33%)	2	19 (19%)	109 (16.60%)	6	6 (8.70%)	15 (4.18%)	2
Multiple Classes	63 (14.79%)	553 (18.27%)	6	43 (28.67%)	156 (28.21%)	3	38 (38%)	274 (41.70%)	7	15 (21.74%)	64 (17.83%)	3
Single Class, Multiple Classes	52 (12.21%)	399 (13.18%)	4	27 (18%)	99 (17.90%)	3	4 (4%)	28 (4.26%)	4.5	7 (10.14%)	34 (9.47%)	5
Single Class	256 (60.09%)	1,656 (54.71%)	5	42 (28%)	169 (30.56%)	3	39 (39%)	246 (37.44%)	5	41 (59.42%)	246 (68.52%)	5
Total	**426 (100%)**	**3,027 (100%)**	**5.5**	**150 (100%)**	**553 (100%)**	**3**	**100 (100%)**	**657 (100%)**	**5.5**	**69 (100%)**	**359 (100%)**	**4**

Intents and refactoring evolution patterns. Figure 3 shows the distribution of reviews grouped by the intents and refactoring evolution patterns. We observed that the evolution of refactoring operations in reviews with the *Feature Only* and *Feature/Refactoring* intents tend to follow more complex evolution patterns than reviews with the explicit intent of *Refactoring Only*. While nearly 50% of reviews with the *Feature/Refactoring* intent have refactoring operations **added**, **undone** or **both** along the revisions, only around 25% of the reviews with the *Feature Only* intent have refactoring operations **added**, **undone** or **both** along the revisions. Differently, reviews with the explicit intent of *Refactoring Only* tend to largely remain the **same** throughout reviewing, with only 23% of the reviews presenting any change in the refactoring operations.

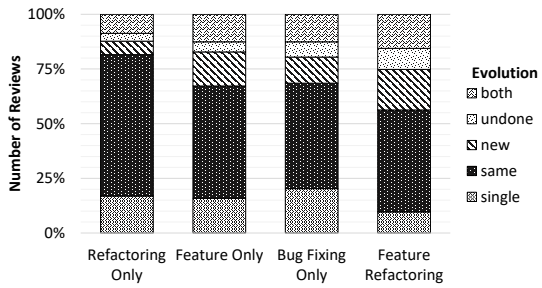

Figure 3: Distribution of evolution patterns per intent

These results indicate that reviews with a refactoring intent tend to follow a 'one and done' behavior, where most of them are integrated as soon as they are proposed. On the other hand, feature-related reviews tend to be iterative, where adaptations of the code change are commonly observed. We also observed that reviews with a *Bug Fixing Only* intent tend to behave similarly to feature-related ones. However, reviews with a single revision appear to be more frequent on *Bug Fixing Only* reviews in comparison to *Feature Only* and *Feature Refactoring* reviews.

Finding 9: Adding a feature tend to present the most diverse patterns of refactoring evolution during code review.

6 STUDY IMPLICATIONS

Our findings provide three key implications to be further discussed and that leads to implications for both researchers and tool builders.

Understanding the influence of intents on refactorings. Many existing techniques for recommending refactorings are designed to support developers in performing pure structural improvement with at least three goals: (i) pure refactoring of a specific module or the entire program [28], (ii) removing code smells [11, 21, 26, 27, 33, 34], or (iii) re-architecting a system [14, 24, 31, 42]. However, our study found that refactorings are less commonly applied in changes with the sole purpose of either optimizing or improving the low-level structure of a class (or one or more methods), its architecture or even a specific module. Developers most often employ refactorings together with other changes to support other intents, such as feature additions and bug fixes (Section 5.1). Moreover, our findings also reveal that the change's intent exerts an influence on: (i) the types and scopes of a refactoring (Section 5.1), (ii) the composition (Section 5.2), and (iii) the evolution (Section 5.3) of refactoring operations along a change. Previous studies did not investigate the role of intents on how refactorings are performed, composed and evolved during the code review process.

There is little support for refactoring with specific intents. Our aforementioned findings suggest there is room for proposing recommenders that better support developers in performing refactoring when the code change involves more than one intent. For instance, refactoring recommenders that specifically support feature additions or bug fixes should prioritize refactorings that improve the structure of the methods or classes that are likely to be impacted by the new features and/or bug fixes. However, existing multi-objective refactoring approaches are often defined in terms of objective functions that capture code or design structure metrics [28, 34], which are not fit for most of the scenarios we observed in our study regarding different intents.

Developers may also need proper guidance in these cases as the refactoring evolution patterns of such changes tend to be more complex than those found in *Refactoring Only* changes (Section 5.3). Recent advancements on feature mining [25, 45], change impact analysis [22] and bug location [13] could be explored to streamline refactorings' recommendations (and their prioritizations) that are

more likely to be relevant to floss refactoring tasks at hand. For instance, these techniques can be used to: (i) analyze the textual description of an upcoming feature and bug fix issues and/or ongoing discussions along a review, and (ii) determine program locations that are likely to be affected by upcoming changes.

Enabling automatic support to refactoring along change's revisions. Our findings suggest that developers need more interactive support for revisiting, refining and sometimes undoing refactoring decisions during the change's review. First, new intents often emerge along changes (Section 5.1) that employ refactorings. Second, many refactoring sequences have more than three revisions (Section 5.2). These observations show that the decision-making process on composing refactorings is indeed not a cut-and-dried task. Reviewers go through cycles to agree or disagree on a certain refactoring sequence. They may also need to observe the impact of a refactoring suggested in a revision first in order to make the next refactoring decisions before finally approving the change. Qualitative studies that explain the influential factors governing this process need to be performed in the future. Hence, tool builders need to assist developers in understanding how the code structure improvement achieved with refactorings is facilitating other changes, such as feature additions. Thus, a tool to support refactoring along changes could also: (i) identify the developer's intent through code changes and reviewers' comments interactively [16], (ii) highlight how refactorings in a change [19] are related to such intents, and (iii) recommend the next refactorings to support the application of their intent.

7 THREATS TO THE VALIDITY

Construct and Internal Validity. The variety of refactoring types analyzed in our study might not be representative. To mitigate this threat, we selected refactoring types that have been widely investigated by previous studies [4, 6, 10, 29, 30, 46]. Another construct threat concerns the order of the refactoring operations that compose a refactoring sequence. To alleviate this, our heuristic for refactoring sequence identification consider the order of each revision to ensure the refactorings are interrelated. Moreover, we qualitatively validated each refactoring sequence to guarantee that all refactoring operations were applied on a common set of code elements. We employed RefMiner [52] to detect refactoring operations performed during code review, whose accuracy has been reported to be high [9, 10]. In total, we detected 13 refactoring types even though some refactoring catalogs report more than 70 refactoring types. Thus, it is possible that some of the reviews we considered not to perform any refactoring actually employed operations that are not supported by RefMiner. To mitigate this threat, we manually validated whether for reviews when the developer has an explicit intent of refactoring they employ any refactoring operation based on a partial ground truth [36]. As a result, we considered RefMiner acceptable for our empirical investigation.

Conclusion and External Validity. Regarding the quantitative data analysis, we tabulated and validated all extracted data in pairs. The analysis followed well-known guidelines of descriptive data analysis [55]. Regarding the qualitative analysis of the developers' intents, we employed a two-phase manual classification procedure based on state-of-the-art empirical studies. In the first phase, all code changes were classified by two authors. In the second phase, for all code changes in disagreement, both authors discussed to reach a unified classification. Our study focuses on investigating the refactoring activities of Java projects only. Nevertheless, we highlight that Java is one of the most popular programming languages in both industry and academia. Additionally, we investigated six real-world systems from two large open source communities.

8 CONCLUSION AND FUTURE WORK

This paper aimed at better understanding the context and motivations in which developers perform refactorings in modern code review. First, we have employed code review data of two large open source communities. Second, we automatically identified occasions in which developers employed 13 refactoring types during code review. Third, we followed this automated procedure with a manual analysis of all code reviews that employed refactoring operations. The manual analysis consisted of identifying the developers' intents behind each change, such as new feature, bug fixing, and refactoring. Fourth, we identified the most common refactoring sequences employed during code review and analyzed how developers compose these sequences under different intents. Finally, we investigated how refactoring operations evolve during code review.

We observed that refactoring operations are most often used in code reviews that implement new features, accounting for 63% of the code changes we studied. Only in 31% of the code reviews that employed refactoring operations the developers had the explicit intent of refactoring. Such observations indicate that developers more often mix refactoring operations with other changes instead of submitting a code review that only performs refactoring. We also observed that developers compose sequences with more than one refactoring type to support feature additions. Moreover, we observed that developers often compose specific refactoring sequences to support multiple intents. Finally, we noticed that about 75% of refactoring operations remain unchanged during the review process for all studied systems. We observed a similar result when grouping reviews by different intents. However, feature-related changes tend to present the highest rate of refactoring evolution while refactoring-related changes presented the lowest rate.

The aforementioned findings serve as recommendations for encouraging the research community and tool builders to come up with a new generation of refactoring tools that best fit the developers' intents during refactoring application. Through our results, we shed light that (Section 6): (i) existing approaches should better support the application of different refactoring types when developers apply code changes with mixed intents; and (ii) existing refactoring tools should be more interactive to support developers on composing and evolving refactoring sequences in a step-wise manner. Finally, qualitative studies should be performed in the future to explain other influential factors governing the decision-making process in refactoring-aware code reviews.

ACKNOWLEDGEMENTS

This study was financed by the CNPq (Brazil) under grants 141285/2019-2, 141054/2019-0, 434969/2018-4, 427787/2018-1, 409536/2017-2, 312149/2016-6. This study was also financed by FAPERJ (22520-7/2016) and CAPES (Brazil), grant 175956 and Finance Code 001.

REFERENCES

[1] Everton LG Alves, Myoungkyu Song, and Miryung Kim. 2014. RefDistiller: a refactoring aware code review tool for inspecting manual refactoring edits. In *22nd FSE*. 751–754.

[2] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In *35th ICSE*. 712–721.

[3] Gabriele Bavota, Bernardino De Carluccio, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Orazio Strollo. 2012. When does a refactoring induce bugs? an empirical study. In *12th SCAM*. 104–113.

[4] Gabriele Bavota, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Fabio Palomba. 2015. An experimental investigation on the innate relationship between quality and refactoring. *J. Syst. Softw. (JSS)* 107 (2015), 1–14.

[5] Moritz Beller, Alberto Bacchelli, Andy Zaidman, and Elmar Juergens. 2014. Modern code reviews in open-source projects: Which problems do they fix?. In *11th MSR*. 202–211.

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

[7] Ana Carla Bibiano, Vinicius Soares, Daniel Coutinho, Eduardo Fernandes, João Correia, Kleber Santos, Anderson Oliveira, Alessandro Garcia, Rohit Gheyi, Baldoino Fonseca, Márcio Ribeiro, Caio Barbosa, and Daniel Oliveira. 2020. How Does Incomplete Composite Refactoring Affect Internal Quality Attributes?. In *28th IEEE/ACM International Conference on Program Comprehension (ICPC)*.

[8] Aline Brito, Andre Hora, and Marco Tulio Valente. 2019. Refactoring Graphs: Assessing Refactoring over Time. In *26nd SANER*. 504–507.

[9] Diego Cedrim, Leonardo da Silva Sousa, Alessandro F. Garcia, and Rohit Gheyi. 2016. Does refactoring improve software structural quality? A longitudinal study of 25 projects. In *Proceedings of the 30th Brazilian Symposium on Software Engineering, SBES 2016, Maringá, Brazil, September 19 - 23, 2016*, Eduardo Santana de Almeida (Ed.). ACM, 73–82. DOI: https://doi.org/10.1145/2973839.2973848

[10] Diego Cedrim, Alessandro Garcia, Melina Mongiovi, Rohit Gheyi, Leonardo Sousa, Rafael de Mello, Baldoino Fonseca, Márcio Ribeiro, and Alexander Chávez. 2017. Understanding the impact of refactoring on smells: A longitudinal study of 23 software projects. In *11th FSE*. 465–475.

[11] Alexander Chávez, Isabella Ferreira, Eduardo Fernandes, Diego Cedrim, and Alessandro Garcia. 2017. How does refactoring affect internal quality attributes?: A multi-project study. In *Proceedings of the 31st Brazilian Symposium on Software Engineering, SBES 2017, Fortaleza, CE, Brazil, September 20-22, 2017*, José Carlos Maldonado, Fabiano Cutigi Ferrari, Uirá Kulesza, and Tayana Uchôa Conte (Eds.). ACM, 74–83. DOI: https://doi.org/10.1145/3131151.3131171

[12] Couchbase. 2020. Review 58223 from the couchbase-java-client system. (2020). Available at: http://review.couchbase.org/#/c/58223/.

[13] Daniel Alencar Da Costa, Shane McIntosh, Weiyi Shang, Uirá Kulesza, Roberta Coelho, and Ahmed E Hassan. 2016. A framework for evaluating the results of the szz approach for identifying bug-introducing changes. *IEEE Trans. Softw. Eng. (TSE)* 43, 7 (2016), 641–657.

[14] Leonardo da Silva Sousa, Willian Oizumi, Anderson Oliveira, Alessandro Garcia, Diego Cedrim, and Carlos Lucena. 2020. When Are Smells Indicators of Architectural Refactoring Opportunities? A Study of 50 Software Projects.. In *Proceedings of the 28th IEEE International Conference on Program Comprehension (ICPC 2020), co-located with ICSE 2020, Seoul, South Korea, October 2020*. ACM, 1–12.

[15] Eclipse. 2019. Example of review 99067 from the jgit software project. (2019). Available at: https://git.eclipse.org/r/#/c/99067/.

[16] Eduardo Fernandes, Anderson Uchôa, Ana Carla Bibiano, and Alessandro Garcia. 2019. On the alternatives for composing batch refactoring. In *3rd IWOR*. 9–12.

[17] Isabella Ferreira, Eduardo Fernandes, Diego Cedrim, Anderson Uchôa, Ana Carla Bibiano, Alessandro Garcia, João Lucas Correia, Filipe Santos, Gabriel Nunes, Caio Barbosa, and others. 2018. The buggy side of code refactoring: Understanding the relationship between refactorings and bugs. In *40th ICSE: Poster Track*. 406–407.

[18] Martin Fowler, Kent Beck, John Brant, and Opdykem Willian. 1999. *Refactoring: Improving the Design of Existing Code*. 431 pages.

[19] Xi Ge, Saurabh Sarkar, Jim Witschey, and Emerson Murphy-Hill. 2017. Refactoring-aware code review. In *VL/HCC*. 71–79.

[20] Gerrit. 2006. Gerrit Code Review Platform. https://www.gerritcodereview.com. (2006). Accessed in: July 2019.

[21] Mark Harman and Laurence Tratt. 2007. Pareto optimal search based refactoring at the design level. In *9th GECCO*. 1106–1113.

[22] Kim Herzig and Andreas Zeller. 2013. The impact of tangled code changes. In *10th MSR*. 121–130.

[23] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2014. An empirical study of refactoring: Challenges and benefits at Microsoft. *IEEE Trans. Softw. Eng. (TSE)* 40, 7 (2014), 633–649.

[24] Yun Lin, Xin Peng, Yuanfang Cai, Danny Dig, Diwen Zheng, and Wenyun Zhao. 2016. Interactive and guided architectural refactoring with search-based recommendation. In *24th FSE*. 535–546.

[25] Jabier Martinez, Tewfik Ziadi, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. 2016. Name suggestions during feature identification: the variclouds approach. In *20th SPLC*. 119–123.

[26] Panita Meananeatra. 2012. Identifying refactoring sequences for improving software maintainability. In *27th ASE*. 406–409.

[27] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, Kalyanmoy Deb, and Mel Ó Cinnéide. 2014. Recommendation system for software refactoring using innovization and interactive dynamic optimization. In *29th ASE*. 331–336.

[28] Rodrigo Morales, Aminata Sabane, Pooya Musavi, Foutse Khomh, Francisco Chicano, and Giuliano Antoniol. 2016. Finding the best compromise between design quality and testing effort during refactoring. In *23rd SANER*, Vol. 1. 24–35.

[29] Emerson Murphy-Hill, Chris Parnin, and Andrew P Black. 2012. How we refactor, and how we know it. *IEEE Trans. Softw. Eng. (TSE)* 38, 1 (2012), 5–18.

[30] Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E Johnson, and Danny Dig. 2013. A comparative study of manual and automated refactorings. In *27th ECOOP*. 552–576.

[31] Willian Nalepa Oizumi, Leonardo da Silva Sousa, Anderson Oliveira, Luiz Carvalho, Alessandro Garcia, Thelma Elita Colanzi, and Roberto Felicio Oliveira. 2019. On the Density and Diversity of Degradation Symptoms in Refactored Classes: A Multi-case Study. In *30th IEEE International Symposium on Software Reliability Engineering, ISSRE 2019, Berlin, Germany, October 28-31, 2019*, Katinka Wolter, Ina Schieferdecker, Barbara Gallina, Michel Cukier, Roberto Natella, Naghmeh Ivaki, and Nuno Laranjeiro (Eds.). IEEE, 346–357. DOI: https://doi.org/10.1109/ISSRE.2019.00042

[32] Johnatan Oliveira, Rohit Gheyi, Melina Mongiovi, Gustavo Soares, Márcio Ribeiro, and Alessandro Garcia. 2019. Revisiting the refactoring mechanics. *Inf. Softw. Technol.* 110 (2019), 136–138. DOI: https://doi.org/10.1016/j.infsof.2019.03.002

[33] Ali Ouni, Marouane Kessentini, Mel Ó Cinnéide, Houari Sahraoui, Kalyanmoy Deb, and Katsuro Inoue. 2017. MORE: A multi-objective refactoring recommendation approach to introducing design patterns and fixing code smells. *J. Softw.: Evol. Process* 29, 5 (2017), e1843.

[34] Ali Ouni, Marouane Kessentini, Houari Sahraoui, Katsuro Inoue, and Kalyanmoy Deb. 2016. Multi-criteria code refactoring using search-based software engineering: An industrial case study. *ACM Trans. Softw. Eng. Methodol (TOSEM)* 25, 3 (2016), 23.

[35] Matheus Paixao, Jens Krinke, Donggyun Han, and Mark Harman. 2018. CROP: Linking code reviews to source code changes. In *15th MSR*. 46–49.

[36] Matheus Paixao, Jens Krinke, DongGyun Han, Chaiyong Ragkhitwetsagul, and Mark Harman. 2017. Are developers aware of the architectural impact of their changes?. In *32nd ASE*. 95–105.

[37] Matheus Paixao, Jens Krinke, DongGyun Han, Chaiyong Ragkhitwetsagul, and Mark Harman. 2019. The Impact of Code Review on Architectural Changes. *IEEE Transactions on Software Engineering* (2019).

[38] Matheus Paixao and Paulo Henrique Maia. 2019. Rebasing in Code Review Considered Harmful: A Large-Scale Empirical Investigation. In *19th International Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 45–55.

[39] Matheus Paixao, Anderson Uchôa, Ana Carla Bibiano, Daniel Oliveira, Alessandro Garcia, Jens Krinke, and Emilio Arvonio. 2020. Replication package for the paper: "Behind the Intents: An In-depth Empirical Study on Software Refactoring in Modern Code Review". (2020). https://zenodo.org/record/3710975 Accessed: 2020-03-16.

[40] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. 2018. A large-scale empirical study on the lifecycle of code smell co-occurrences. *Inform. Softw. Tech. (IST)* 99 (2018), 1–10.

[41] Fabio Palomba, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. 2017. An Exploratory Study on the Relationship between Changes and Refactoring. In *25th ICPC*. 176–185.

[42] Paula Rachow. 2019. Refactoring Decision Support for Developers and Architects Based on Architectural Impact. In *ICSA-C*. 262–266.

[43] Peter C Rigby. 2012. Open source peer review–lessons and recommendations for closed source. *IEEE Software* (2012).

[44] Peter C Rigby and Christian Bird. 2013. Convergent contemporary software peer review practices. In *9th FSE*. 202–212.

[45] A-D Seriai, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, and others. 2013. Mining features from the object-oriented source code of software variants by combining lexical and structural similarity. In *14th IRI*. 586–593.

[46] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. 2016. Why we refactor? confessions of GitHub contributors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016*. ACM Press, New York, New York, USA, 858–870.

[47] Leonardo Sousa, Diego Cedrim, Alessandro Garcia, Willian Oizumi, Ana Carla Bibiano, Daniel Tenorio, Miryung Kim, and Anderson Oliveira. 2020. Characterizing and Identifying Composite Refactorings: Concepts, Heuristics and Patterns. In *17th International Conference on Mining Software Repositories (ICSE)*.

[48] Gábor Szőke, Csaba Nagy, Lajos Fülöp, Rudolf Ferenc, and Tibor Gyimóthy. 2015. FaultBuster: An automatic code smell refactoring toolset. In *15th SCAM*. 253–258.

[49] Daniel Tenorio, Ana Carla Bibiano, and Alessandro Garcia. 2019. On the customization of batch refactoring. In *Proceedings of the 3rd International Workshop on Refactoring, IWOR@ICSE 2019, Montreal, QC, Canada, May 28, 2019*, Nikolaos Tsantalis, Yuanfang Cai, and Serge Demeyer (Eds.). IEEE / ACM, 13–16. DOI: https://doi.org/10.1109/IWoR.2019.00010

[50] Adrian Trifu and Radu Marinescu. 2005. Diagnosing design problems in object oriented systems. In *12th WCRE*. 10–pp.

[51] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. 2018. Ten years of JDeodorant: Lessons learned from the hunt for smells. In *25th SANER*. 4–14.

[52] Nikolaos Tsantalis, Victor Guana, Eleni Stroulia, and Abram Hindle. 2013. A multidimensional empirical study on refactoring activity. In *23rd CASCON*. IBM Corp., 132–146.

[53] Nikolaos Tsantalis, Matin Mansouri, Laleh M Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and efficient refactoring detection in commit history. In *40th ICSE*. 483–494.

[54] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. 2017. When and why your code starts to smell bad (and whether the smells go away). *IEEE Trans. Softw. Eng. (TSE)* 43, 11 (2017), 1063–1088.

[55] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering*. Springer Science & Business Media.