Impact of Change Granularity in Refactoring Detection

Lei Chen and Shinpei Hayashi
{chenlei,hayashi}@se.c.titech.ac.jp
Tokyo Institute of Technology
Meguro-ku, Tokyo, Japan

ABSTRACT

Detecting refactorings in commit history is essential to improve the comprehension of code changes in code reviews and to provide valuable information for empirical studies on software evolution. Several techniques have been proposed to detect refactorings accurately at the granularity level of a single commit. However, refactorings may be performed over multiple commits because of code complexity or other real development problems, which is why attempting to detect refactorings at single-commit granularity is insufficient. We observe that some refactorings can be detected only at coarser granularity, that is, changes spread across multiple commits. Herein, this type of refactoring is referred to as coarse-grained refactoring (CGR). We compared the refactorings detected on different granularities of commits from 19 open-source repositories. The results show that CGRs are common, and their frequency increases as the granularity becomes coarser. In addition, we found that Move-related refactorings tended to be the most frequent CGRs. We also analyzed the causes of CGR and suggested that CGRs will be valuable in refactoring research.

CCS CONCEPTS

• Software and its engineering → Software evolution.

KEYWORDS

Refactoring detection, Squashed commit, Git

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACM Reference Format:
Lei Chen and Shinpei Hayashi. 2022. Impact of Change Granularity in Refactoring Detection. In 30th International Conference on Program Comprehension (ICPC ’22), May 16–17, 2022, Virtual Event, USA. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3524610.3528386

1 INTRODUCTION

Mining refactorings in commit history is essential to help programmers comprehend code changes and code reviews [16], and this can provide valuable information for empirical studies on software evolution[7, 17]. For example, Chávez et al. [8] and Fernandes et al. [11] detected and analyzed refactorings to investigate the refactoring performance in improving internal quality attributes.

Refactoring detectors [10, 15, 18, 21, 23, 24] detect refactorings by comparing two source code snapshots. Although traditional approaches aim to detect refactorings over releases [18, 24], recent detectors such as RefDiff [19, 21] and RefactoringMiner [22, 23] use a commit as a change unit to detect refactorings, which means that two snapshots before and after a single commit are compared. These methods have achieved high accuracy in detecting refactoring in commits.

However, refactorings that are performed over multiple commits may not be detected. The sample history shown in Figure 1 consists of two commits extracted from the mbassador repository [1], where commit 2ae0e5f is the parent of commit 9ce3ceb. The intention of the developer, as expressed by these two commits, is to decompose the source file Massador.java, which contains multiple top-level classes, into multiple source files to ensure that each file contains only one top-level class. In the first commit, the developer copied the implementation of class FilteredAsynchronousSubscription in Massador.java to a new file FilteredAsynchronousSubscription.java, and then she/he removed that class from the source file Massador.java in the second commit. Overall, she/he moved a class from Massador.java to a new source file. A detection based on either of the single commits shown in Figure 1 cannot reveal this kind of refactoring because each commit contains only part of the code changes for detecting Move Class refactoring. However, this refactoring can be detected if we consider a coarse-grained commit generated by merging the changes from the two commits.

The existence of refactorings detected only in the granularity of coarse-grained commits suggests that detectors based on single commits may have missed some refactorings. We conducted an empirical study on 19 open-source Git-based Java repositories to investigate the impact of change granularity in refactoring detection. To change the granularity of commits, we squashed multiple fine-grained commits into one to form a coarse-grained commit. The number of fine-grained commits squashed into one coarse-grained commit is referred to as coarse granularity. Refactoring detection is conducted on both fine-grained and coarse-grained commits using the state-of-the-art tool RefactoringMiner [22, 23]. If a refactoring type is detected in the coarse-grained commit but not in the fine-grained commits, which formed the coarse one, this refactoring is defined as a coarse-grained refactoring (CGR).

Our results indicate that CGRs are common, and their frequency increases as the granularity becomes coarser. The type of refactoring that is most likely to be coarse-grained varies in each repository; however, in general, the Move-related refactoring type tends to be CGR.

In summary, our study makes the following contributions:

• We propose the definition of CGR.
• We evaluate features of CGRs to understand its effect on refactoring detection.
• We analyze the reason for the occurrence of CGR.
2 STUDY DESIGN

The overview of our study procedure is shown in Figure 2. Our pro-

2.1 Repository Transformation

The overview of our study procedure is shown in Figure 2. Our procedure can be divided into two phases: repository transformation and detection and comparison. In the repository transformation phase, squash units that contain multiple fine-grained commits can be squashed into coarse-grained ones are extracted from the commit history. In the detection and comparison phase, refactoring detection is conducted on both fine-grained and coarse-grained commits, and their results are compared.

2.2 Detection and Comparison

Refactoring detection is conducted on each commit in all extracted squash units and on coarse-grained commits, and the results are compared for each pair of commits. From commit c, a set of refactorings ref(c) (⊆ R) are detected, where R is the universal set of refactorings. The detection result for one commit contains: 1) the refactoring type, 2) a description of how this refactoring is conducted, and 3) the location where this refactoring is applied in the source code. Because the location and description of a refactoring may change owing to squashing, we conservatively compared only the type of detected refactorings. Refactorings detected with invalid locations were excluded. For a squash unit u and its coarse-grained commit cu = sq(u), we judged refactoring r ∈ ref(cu) as coarse-grained if and only if no refactoring of its type r.type was found in the detected refactorings from each fine-grained commit in u. More specifically, the set of CGRs of u can be explained as

\[ \text{CGR}(u) = \{ r \in \text{ref}(sq(u)) \mid r.\text{type} \notin \text{types}(u) \}, \]

\[ \text{types}(u) = \{ r.\text{type} \mid \exists r \in \text{ref}(c) \land c \in u \}. \]

A squash unit u is regarded as an effective squash when at least one CGR is detected from it:

\[ \text{isEffective}(u) = \text{CGR}(u) \neq \emptyset. \]

When the coarse granularity is set to l, the set of squash units for the repository H is

\[ U_l(H) = \bigcup_{0 \leq o \leq l-1} \text{unit}_o^{l-1}(H). \]

where \( \text{unit}_o^{l-1}(H) \) denotes the squash units extracted from H according to strategy \( S_l^o \).

3 PRELIMINARY EVALUATION

3.1 Research Questions

Our objective in this study is to investigate features of CGRs. We answer the following research questions (RQs) to better achieve this goal.

- **RQ1**: How frequently do CGRs appear because of granularity change?
- **RQ2**: Which types of refactorings tend to be coarse-grained?
- **RQ3**: What are the reasons for the occurrence of CGRs?
A quantitative analysis is provided for RQ1 and RQ2. We manually examine the experiment results to present a qualitative explanation for RQ3.

### 3.2 Experimental Setup

We used the Git repository rewriting tool git-stein [14] to change the granularity and the latest version of RefactoringMiner (ver. 2.2) to detect refactoring in 19 open source Git-based Java repositories.

#### 3.2.1 Data Collection

The repositories that we selected are from a dataset collected by Silva et al. [20], containing 185 GitHub-hosted Java projects. Refactorings exist in these projects, some of which have been identified by RefactoringMiner, studied, and confirmed by researchers. On account of computation time, we chose 19 repositories whose number of commits is no more than 7,000 from the dataset. To be specific, the number of commits ranges from 342 (mbassador) to 6,955 (redisson [6]).

### 3.3 RQ1: How frequently do CGRs appear because of granularity change?

#### 3.3.1 Study Design

The techniques introduced in Section 2 are applied to the selected repositories to extract squash units, change the granularity of commits, and compare the refactoring detection results to find CGRs.

The frequency of CGRs in the commit history $H$ can be expressed as the ratio of the number of squash units that can generate at least one CGR:

$$\text{Frequency}(H, l) = \frac{|\{u \in U_l(H) \mid \text{isEffective}(u)\}|}{|U_l(H)|}. \quad (4)$$

We calculate $\text{Frequency}$ for our dataset when the coarse granularity is set to 2, 3, and 4, respectively.

#### 3.3.2 Results and Discussion

Figure 3 shows box plots of the CGR frequency at different levels of coarse granularity in the 19 repositories. The minimum values of all three box plots are greater than zero, indicating that CGRs were detected in all the repositories at all levels of coarse granularity.

We can conclude that the CGR is a common phenomenon in refactoring detection. The highest frequency was observed in the repository goclipse [4], which was 0.071, 0.135, and 0.178 when the coarse granularity was set to 2, 3, and 4, respectively. The box plots show that the more the coarse granularity increases, the more the frequency increases in all repositories. The minimum increase in the frequency when the coarse granularity was changed from 2 to 3 was in the repository baasbox [3], which increased by 14.1%, whereas the maximum increase was 331.9% in javapoet [5]. The average increase for all repositories was 129.4%. When the coarse granularity increases from 3 to 4, a minimum increase of 24.4% appears in seyren [2], a maximum increase of 147.6% appears in mbassador, and the average increase is 65.6%. The average frequencies for all the repositories were 2.0%, 4.3%, and 6.9% when the coarse granularities were 2, 3, and 4, respectively. The observed tendency of frequency to increase as the coarse granularity increases can be explained as follows. The CGR detected in the commits with finer granularity may also exist in those with coarser granularity. In addition, a new CGR may be detected in coarser-grained commits because more code changes are transferred into these commits through the granularity change.

However, we also observed that not all CGRs detected in commits of finer granularity could be detected in a coarser-grained one. Code changes in other commits may hinder the currently detected CGR when those commits are squashed with the current coarse-grained commit.

A CGR is a common phenomenon in all repositories. The average frequencies of CGR for all repositories were 2.0%, 4.3%, and 6.9% when the coarse granularities were 2, 3, and 4, respectively. CGRs are more frequent when coarse granularity increases.

### 3.4 RQ2: Which types of refactorings tend to be coarse-grained?

#### 3.4.1 Study Design

To investigate this RQ, we calculate the appearance ratio of a specific CGR type at all the three granularity levels. The ratio expresses the average number of CGRs in one effective squash. For a certain refactoring type $t$ in commit history $H$, the ratio can be expressed as follows:

$$\text{Ratio}(t) = \frac{\sum_{2 \leq l \leq 4} |\{r \mid \exists u \in U_l(H) \land r \in \text{CGR}(u) \land r.type = t\}|}{\sum_{2 \leq l \leq 4} |\{u \in U_l(H) \mid \text{isEffective}(u)\}|}. \quad (5)$$

We calculate the ratio of each type of CGR in our dataset.

#### 3.4.2 Results and Discussion

The CGR type with the highest ratio for each repository is listed in Table 1. Among the 19 repositories, we found that Change Class Access Modifier occurs at the highest ratio (2.00) in mbassador, and Move Attribute in HikariCP reaches 1.82.

We find that the CGR type with the highest ratio varies with repositories. In our dataset, we also find that Move-related refactoring types, e.g., Move Class and Move Attribute, appear most frequently for eight repositories. By calculating the average ratio over our dataset for all types of refactorings, we observed that the top three highest-ratio refactoring types were Move And Rename Class (0.46%), Move Method (0.34%), and Move And Inline Method (2.9%).

As a result, we can conclude that Move-related refactoring types are most likely to be coarse-grained. A possible explanation for this is that in Move-related refactoring, Move on the refactored object is not performed directly but is performed in two steps. First, an object
Table 1: Highest ratio CGR type

| repository   | refactoring type               | ratio |
|--------------|--------------------------------|-------|
| jfinal       | Change Method Access Modifier  | 0.49  |
| mbassador    | Change Class Access Modifier   | 2.00  |
| javapoet     | Replace Variable With Attribute| 0.80  |
| jersey       | Move Class                     | 0.19  |
| seyren       | Merge Package                  | 1.00  |
| retrolambda  | Push Down Method               | 1.21  |
| baasbox      | Replace Variable WithAttribute | 0.29  |
| sshj         | Remove Parameter               | 0.34  |
| xabber-android| Move Method                    | 0.30  |
| android-asyn-http| Remove Parameter Modifier     | 1.40  |
| giraph       | Remove Variable Modifier       | 0.91  |
| spring-data-rest| Parameterize Variable          | 0.08  |
| HikariCP     | Move Attribute                 | 1.82  |
| redisson     | Push Down Method               | 0.12  |
| goclipe      | Move Package                   | 0.05  |
| atomix       | Move And Rename Class          | 0.33  |
| morphia      | Move Attribute                 | 0.71  |
| PocketHub    | Move And Rename Class          | 0.12  |

We found reasons for two categories. Generation refers to new refactorings generated by non-detected fine-grained ones. Combination is a high-level refactoring combined with detected fine-grained ones.

4 CONCLUSION AND FUTURE WORK

In this study, we investigated the impact of refactoring detection on different granularities of commits in 19 open source Git-based Java repositories. We observed that it is common for a CGR to occur, and its frequency increases as the granularity becomes coarser. Move-related refactoring types tend to be coarse-grained. We analyzed the causes of CGR and categorized them into two types according to their composition: Generation and Combination. The studied list of CGR is attached as a supplemental material [9]. We suggest that refactoring detectors should cover CGRs. For future work, we plan to extend the current experiment by comparing different refactoring detection tools on a larger dataset.

ACKNOWLEDGMENTS

This work was partly supported by the JSPS Grants-in-Aid for Scientific Research JP18K11238, JP21K18302, JP21KK0179, and JP21H04877.
REFERENCES

[1] 2012. mbassador. https://github.com/bennidi/mbassador.
[2] 2012. seyren. https://github.com/scobal/seyren.
[3] 2013. baasbox. https://github.com/baasbox/baasbox.
[4] 2013. GoClipse. https://github.com/GoClipse/goclipse.
[5] 2013. javapoet. https://github.com/square/javapoet.
[6] 2014. redisson. https://github.com/redisson/redisson.
[7] 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 Proceedings of the 12th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2012). 104–113.
[8] 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). 74–83.
[9] Lei Chen and Shinya Hayashi. 2022. Appendix of “Impact of Change Granularity in Refactoring Detection”. https://doi.org/10.5281/zenodo.6399250.
[10] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. 2006. Automated detection of refactorings in evolving components. In Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP 2006). 404–428.
[11] Eduardo Fernandes, Alexander Chávez, Alessandro Garcia, Isabella Ferreira, Diego Cedrim, Leonardo Sousa, and William Ozumi. 2020. Refactoring effect on internal quality attributes: What haven’t they told you yet? Information and Software Technology 126 (2020), 106347.
[12] Stephen R. Foster, William G. Griswold, and Sorin Lerner. 2012. WitchDoctor: IDE support for real-time auto-completion of refactorings. In Proceedings of the 34th International Conference on Software Engineering (ICSE 2012). 222–232.
[13] Xi Ge and Emerson Murphy-Hill. 2014. Manual Refactoring Changes with Automated Refactoring Validation. In Proceedings of the 36th International Conference on Software Engineering (ICSE 2014). 1095–1105.
[14] Shinya Hayashi. 2018. git-stein. https://github.com/sh5i/git-stein.
[15] Miryung Kim, Matthew Gee, Alex Loh, and Napol Rachatusumrit. 2010. RefFinder: A Refactoring Reconstruction Tool Based on Logic Query Templates. In Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2010). 371–372.
[16] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2012. A field study of refactoring challenges and benefits. In Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE 2012). 1–11.
[17] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2014. An empirical study of refactoring challenges and benefits at microsoft. IEEE Transactions on Software Engineering 40, 7 (2014), 633–649.
[18] Kyle Prete, Napol Rachatusumrit, Nikita Sudan, and Miryung Kim. 2010. Template-based Reconstruction of Complex Refactorings. In Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM 2010). 1–10.
[19] Danilo Silva, João Paulo da Silva, Gustavo Santos, Ricardo Terra, and Marco Tulio Valente. 2021. RefDiff 2.0: A Multi-Language Refactoring Detection Tool. IEEE Transactions on Software Engineering 47, 12 (2021), 2786–2802.
[20] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. 2016. Why we refactor? Confessions of GitHub contributors. In Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016). 858–870.
[21] Danilo Silva and Marco Tulio Valente. 2017. RefDiff: Detecting refactorings in Vesion Histories. In Proceedings of the 14th IEEE/ACM International Conference on Mining Software Repositories (NSR 2017). 269–279.
[22] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. 2020. RefactoringMiner 2.0. IEEE Transactions on Software Engineering 48, 3 (2020), 930–950.
[23] Nikolaos Tsantalis, Matin Mansouri, Laleh M. Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and Efficient Refactoring Detection in Commit History. In Proceedings of the 40th International Conference on Software Engineering (ICSE 2018). 483–494.
[24] Peter Weißgerber and Stephan Diehl. 2006. Identifying Refactorings from Source-Code Changes. In Proceedings of the 21st International Conference on Automated Software Engineering (ASE 2006). 231–240.