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

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 *Mbassador.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 `Mbassador.java` to a new file `FilteredAsynchronousSubscription.java`, and then she/he removed that class from the source file `Mbassador.java` in the second commit. Overall, she/he moved a class from `Mbassador.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.

The remainder of this paper is organized as follows. The next section explains our study design. Then, we present a preliminary evaluation of 19 open-source projects and the answers to the three research questions in Section 3. Finally, in Section 4, we conclude and state our plans for future work.

2 STUDY DESIGN

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 and can be squashed into coarse-grained ones are extracted from the commit history. In the detection and comparison phase, refactoring

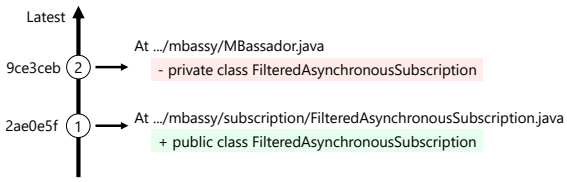

Figure 1: Two commits in *mbassador*.

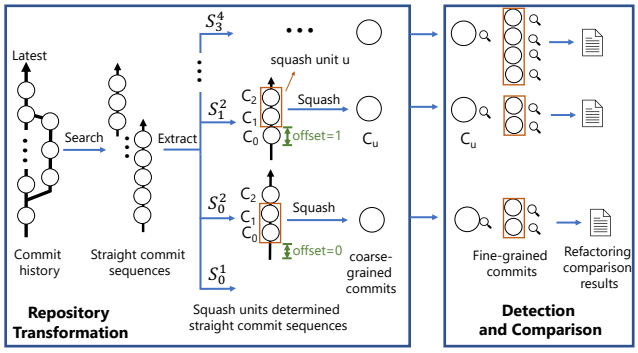

Figure 2: Overview of the study procedure

detection is conducted on both fine-grained and coarse-grained commits, and their results are compared.

2.1 Repository Transformation

In this phase, firstly, the Git-based commit history, as a set of fine-grained commits H ($\subseteq C$), is extracted from the given repository, where C is the universal set of commits. By searching the commit history, we can extract straight commit sequences. Each sequence consists of fine-grained commits that excludes *merge commits*, which have more than one parent, and *branch sources*, which have multiple children. Merge commits are excluded to avoid duplicate detection of refactoring in the later phase, and branch sources are excluded for simplicity when extracting squash units.

A *squash unit* u ($\subseteq C$) is a set of multiple adjacent fine-grained commits that are squashed into a single coarse-grained commit. Here, if a commit is the parent or child of another commit, these two commits are considered adjacent. The adjacent commits are shown as circles next to each other in Figure 2. Different strategies labelled S_o^l (for appropriate values of o and l) are used to extract squash units from straight commit sequences. Here, the *granularity level* l (≥ 1) specifies the size of the squash units, and straight commit sequences are divided into multiple squash units of the specified size. Because each unit is squashed into one coarse-grained commit, this level expresses the coarse granularity of the coarse-grained commits to be generated. The granularity level $l = 1$ exactly produces original fine-grained commits. The *offset* $0 \leq o \leq l-1$ is the number of commits to be skipped from the beginning of the given straight commit sequence when extracting the squash units to adjust which commits will be merged. For example, the commit c_1 in Figure 2 is squashed together with c_0 when strategy S_0^2 is used, whereas it is squashed together with c_2 when strategy S_1^2 is used. For each squash unit u, $sq(u)$ is used to squash all the commits in u into a single coarse-grained commit, which we name c_u.

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 $ref(c)$ ($\subseteq R$) 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 $c_u = sq(u)$, we judged refactoring $r \in ref(c_u)$ 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

$$CGR(u) = \{\, r \in ref(sq(u)) \mid r.type \notin types(u) \,\},$$
$$types(u) = \{\, r.type \mid \exists r \in ref(c) \wedge c \in u \,\}. \quad (1)$$

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

$$isEffective(u) = CGR(u) \neq \emptyset. \quad (2)$$

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

$$U_l(H) = \bigcup_{0 \leq o \leq l-1} unit_{S_o^l}(H). \quad (3)$$

where $unit_{S_o^l}(H)$ denotes the squash units extracted from H according to strategy S_o^l.

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.

- ***RQ***$_1$: *How frequently do CGRs appear because of granularity change?*
- ***RQ***$_2$: *Which types of refactorings tend to be coarse-grained?*
- ***RQ***$_3$: *What are the reasons for the occurrence of CGRs?*

A quantitative analysis is provided for RQ_1 and RQ_2. We manually examine the experiment results to present a qualitative explanation for RQ_3.

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

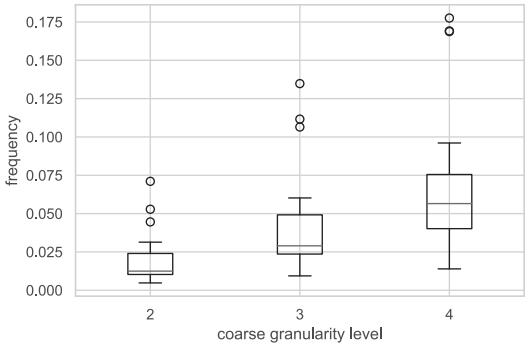

Figure 3: Frequency of CGRs.

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 RQ_1: 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:

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

We calculate *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.

> 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.

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
jeromq	*Move Class*	0.19
seyren	*Merge Package*	1.00
retrolambda	*Push Down Method*	1.21
baasbox	*Replace Variable With Attribute*	0.29
sshj	*Remove Parameter*	0.34
xabber-android	*Move Method*	0.30
android-async-http	*Remove Parameter Modifier*	1.40
giraph	*Remove Variable Modifier*	0.91
spring-data-rest	*Move Attribute*	0.19
blueflood	*Parameterize Variable*	0.08
HikariCP	*Move Attribute*	1.82
redisson	*Push Down Method*	0.12
goclipse	*Move Package*	0.05
atomix	*Move And Rename Class*	0.33
morphia	*Move Attribute*	0.71
PocketHub	*Move And Rename Class*	0.12

3.4 RQ_2: 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:

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

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 is copied to the destination and is potentially followed by other changes, e.g., renaming, inline, or no change. Second, the original object is removed. These two steps may be included in separate commits. Another possible reason is that *Move*-related refactoring can be combined with other refactoring, such as *Rename* or *Inline*.

> Considering the average ratio over the whole dataset, the top three types are *Move And Rename Class*, *Move Method*, *Move And Inline Method*. We conclude that *Move*-related refactoring types are most likely to be coarse-grained.

3.5 RQ_3: What are the reasons for the occurrence of CGRs?

3.5.1 Study Design. The *git diff* command is used to extract code changes from the fine-grained and coarse-grained commits. After extraction, we manually compare and analyze the changes and refactorings detected.

3.5.2 Results and Discussion. The reasons for the occurrence of CGRs are categorized into two types according to their composition: *Generation* and *Combination*.

Generation. This type of CGR is generated from non-refactoring changes. The example shown in Figure 1 belongs to this type; the *Move Method* refactoring is generated by two non-refactoring changes: 1) copy the class implementation to a new file 2) remove the origin class. Another example is in repository *javapoet*. In the parent commit *6a3595c*, the attribute `body` is defined, and the method call `methodWriter.write()` is removed. In child commit *4ff9adf*, the developer adds method call `body.write()`. In the coarse-grained commit, the above code changes are detected as *Rename Variable with Attribute*; the variable `methodWriter` is renamed to attribute `body`.

Combination. In contrast with *Generation*, this type is the combined result of multiple refactorings detected in finer-grained commits. Figure 4 shows an example of this type. For clarity, only part of the package hierarchy of the repository is shown in the figure. In the parent commit *ce2a9e9*, the developer moves class `ProperyMailSender` under package `services` to package `core.util`, which is detected as refactoring *Move Class*. In child commit *989bf50*, she/he split package `core.value` into `core.util.email` and another one, and then she/he move class `PropertyMailSender` to the package `core.util.email`, which are detected as *Split Package* and *Move Class*. In terms of result, she/he applied *Merge Package* to merge part of the package `core.value` and the entire package `services` into a new package `core.util.email`.

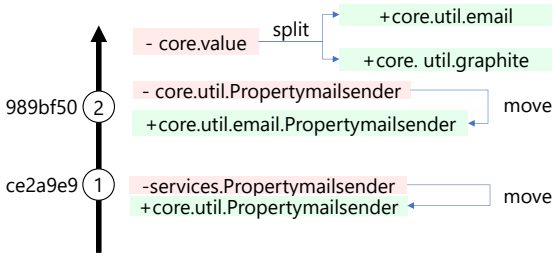

Figure 4: Example of coarse-grained *Merge Package*.

Generation type CGRs will influence judgments of whether a module is refactored or not. We note that this type may also occur because of developers' awareness of refactoring; developers do not realize that the conducted code changes belong to refactoring operations. Supporting tools to guess developers' manual edits and recognize refactoring activities [12, 13] may assist them in development. Because the *Combination* type may influence type-based refactoring studies, such as investigations on frequently-performed refactoring types, researchers may reconsider their results by covering coarse-grained types.

> 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 Shinpei 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 Willian Oizumi. 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] Shinpei Hayashi. 2018. git-stein. https://github.com/sh5i/git-stein.

[15] Miryung Kim, Matthew Gee, Alex Loh, and Napol Rachatasumrit. 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 Rachatasumrit, 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 Version Histories. In *Proceedings of the 14th IEEE/ACM International Conference on Mining Software Repositories (MSR 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.