The Journal of Systems and Software 165 (2020) 110571

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

On tracking Java methods with Git mechanisms

Yoshiki Higo[a,*], Shinpei Hayashi[b], Shinji Kusumoto[a]

[a] Graduate School of Information Science and Technology, Osaka University, Yamadaoka 1–5, Suita, Osaka 565–0871, Japan
[b] School of Computing, Tokyo Institute of Technology, Ookayama 2–12–1–W8–71, Ookayama, Meguro-ku, Tokyo 152–8550, Japan

ARTICLE INFO

Article history:
Received 11 August 2019
Revised 27 January 2020
Accepted 9 March 2020
Available online 13 March 2020

Keywords:
Mining software repositories
Source code analysis
Tracking Java methods

ABSTRACT

Method-level historical information is useful in various research on mining software repositories such as fault-prone module detection or evolutionary coupling identification. An existing technique named Historage converts a Git repository of a Java project to a finer-grained one. In a finer-grained repository, each Java method exists as a single file. Treating Java methods as files has an advantage, which is that Java methods can be tracked with Git mechanisms. The biggest benefit of tracking methods with Git mechanisms is that it can easily connect with any other tools and techniques build on Git infrastructure. However, Historage's tracking has an issue of accuracy, especially on small methods. More concretely, in the case that a small method is renamed or moved to another class, Historage has a limited capability to track the method. In this paper, we propose a new technique, FinerGit, to improve the trackability of Java methods with Git mechanisms. We implement FinerGit as a system and apply it to 182 open source software projects, which include 1,768K methods in total. The experimental results show that our tool has a higher capability of tracking methods in the case that methods are renamed or moved to other classes.

1. Introduction

One feature of version control systems is the ability to know file-level change information. Thus, it is easy to identify which files were changed in given commits or counting changes for files in a given repository. However, many approaches in mining software repositories (in short, MSR) require information on finer-grained units such as Java methods or C functions. If we want to count changes for Java methods, we need to parse source files to identify method positions and then we need to match method positions with changed code positions to identify which methods were changed. To conduct finer-grained analyses, developers have to implement code/scripts. Besides, incorrect analysis results will be obtained if the implemented code/scripts include bugs.

Hata et al. proposed a technique, Historage, which enables Java methods to be tracked with Git mechanisms (Hata et al., 2011a). Historage takes a Git repository of a Java project as its input, and it outputs another Git repository in which each method gets extracted as a file. Treating Java methods as files realizes that developers/practitioners can obtain method-level historical information only by executing Git commands such as `git-log`.

Fig. 1 shows a simple model of Git and Historage repositories. In the Git repository, file `Person.java` is managed. We can see that `Person.java` was changed in two commits `c100` and `c101`. Information for the changes on `Person.java` can be retrieved by executing `git-log`. However, if we want to know which methods were changed in the two commits, we have to parse `Person.java` to obtain the positions of the methods and then we have to match method positions with the positions of the changed code in the two commits. On the other hand, in the Historage repository, each method exists as a file. Thus, just executing `git-log` is sufficient to know in which commits the two methods were changed. The command identifies that `getLength()` in `Person.java` was changed in commit `c100` and `setLength(int)` was changed in `c101`.

However, Historage has a limited capability of tracking methods in the case that methods are renamed or moved to other classes. We explain the issue with Fig. 2, which shows refactorings on file `Person.java` in Fig. 1. The refactorings include the following four changes.

Rename Class: `Person` → `Engineer`
Rename Field: `length` → `height`
Rename Method (Getter): `getLength` → `getHeight`

* Corresponding author.
E-mail addresses: higo@ist.osaka-u.ac.jp (Y. Higo), hayashi@c.titech.ac.jp (S. Hayashi), kusumoto@ist.osaka-u.ac.jp (S. Kusumoto).

https://doi.org/10.1016/j.jss.2020.110571

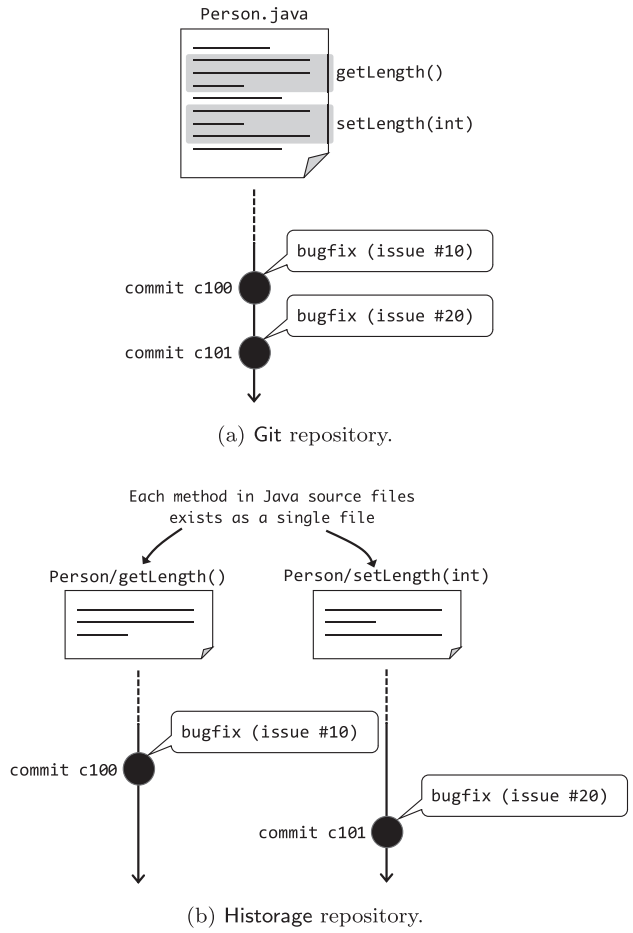


Fig. 1. Differences between Git and Historage repositories.

RENAME METHOD (Setter): `setLength` → `setHeight`

In the case of the changes in Fig. 2(a), the Git rename detection function can identify that file `Person.java` was renamed to `Engineer.java` because the two files sufficiently share the identical lines. On the other hand, in the Historage repository, files of Java methods get much smaller than their original file as shown in Fig. 2(b). Thus, the ratio of the changed lines against all the lines gets higher, which makes the Git function not work well.

Hata et al. addressed that changing the threshold for the Git rename function is a way to realize a better method tracking (Hata et al., 2011a). They recommend using 30% instead of 60%, which is a default value of Git. However, we consider that only using a lower threshold may produce incorrect tracking results. For example, if we use 30% instead of 60%, the Git rename function can identify that `Engineer/getHeight()` is a renamed file of `Person/getLength()`. However, at the same time, `Person/getLength()` can be tracked wrongly from `Engineer/setHeight(int)` because their similarity is 1/3, which is higher than 30%.

Tracking method accurately is essential. If not, MSR approaches using historical data gets affected. Hora et al. reported that between 10 and 21% of changes at the method level in 15 large Java systems were untracked in the context of refactoring detection (Hora et al., 2018). They also found that 37% of the top-25% most changed entities (classes and methods) have at least one untracked change in their histories. By assessing two MSR approaches, they detected that their results could be improved when untracked changes were resolved.

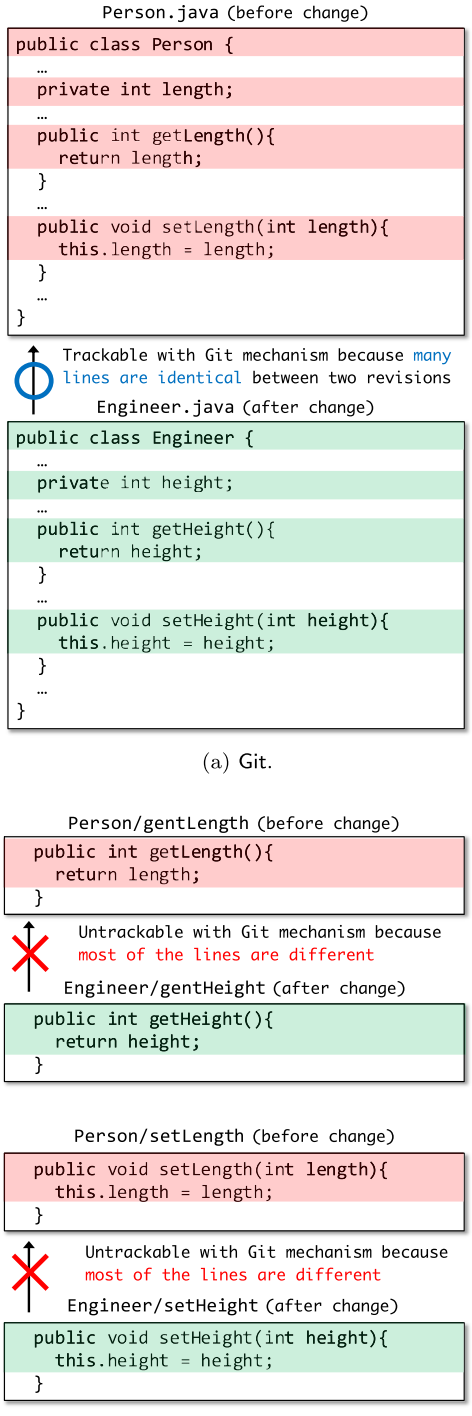


Fig. 2. Trackability differences between Git and Historage repositories.

In this paper, we propose a new technique named FinerGit to improve the trackability of Java methods. Several research areas benefit from FinerGit. FinerGit is useful for studies in the context of assessing bug introducing changes (Kim et al., 2006; 2008; Śliwerski et al., 2005) or detecting code authorship (Meng et al., 2013; Rahman and Devanbu, 2011). More broadly, any study that compares two versions of methods can be benefited, for example, API evolution detection (Kim et al., 2011; Soares et al., 2010), code warning prioritization (Balachandran, 2013; Kim and Ernst, 2007), and many other.

The main contributions of this paper are the followings.

- We raise an issue on method trackability in Historage.
- We propose a new technique, FinerGit, to increase method trackability with Git mechanisms.
- We provide a software tool based on FinerGit. The tool is open to the public on GitHub.[1] The tool is sufficiently fast even for huge repositories, as shown in the evaluation.
- We show the experimental results on the tracking results of 182 open source software (OSS) projects. These experiments have two aspects. First, they clarify the advantage of FinerGit with an existing technique, Historage. Second, they are the first attempt of large-scale empirical studies for the tracking results of method-level repositories.

The remainder of this paper is organized as follows: in Section 2, we explain our research goal and our key idea to achieve the goal; in Section 3, we propose our new technique named FinerGit on the top of the key idea; Section 4 describes an implementation of FinerGit; then, we report the evaluation results with the implementation in Section 5; we also describe threats to validity on the experiments in Section 7; related work is introduced in Section 8; lastly, we conclude this paper in Section 9.

2. Basic approach

At present, there are various techniques of tracking source code entities (Dig et al., 2006; Godfrey and Zou, 2005; Kim et al., 2005; Wu et al., 2010). Those techniques utilize many types of information such as text similarities, data dependencies, and call dependencies. On the other hand, in this research, we utilize only line-based text similarity to track Java methods. The reason is that our research goal is realizing accurate method tracking with Git mechanisms.

The biggest benefit of tracking methods with Git mechanisms is that it can easily connect with any other tools and techniques built on Git infrastructure. For example, the following analyses can be easily performed by using the basic commands provided by Git.

- We can know how many times each method was changed in the past by `git-log`.
- We can know how many developers changed a specified method in the past by collecting author names of the commits in which the method was changed.

Git performs file comparisons by using hash values. If the size of a line is equal to or shorter than 64 bytes, a hash value is calculated from the entire line. If the size of a line is longer than 64 bytes, the line is chunked by 64 bytes, and a hash value is calculated from each chunk. Thus, even if just a single token in a given line (which is shorter than 64 bytes) has been changed, Git regards that the entire line has been changed.

Method-level tracking with Git mechanisms can be realized by treating each method as a single file (a *method file* hereafter). Based on this idea, Hata et al. developed technique named Historage (Hata et al., 2011b). However, as explained with Fig. 2, simple extraction as files are inadequate for small methods. In this research, we propose a file format that each line includes only a single token. By using this format, each hash is calculated from a single token. In Fig. 2(b), Git regards that the two red lines of methods `getLength` and `setLength` were changed, though only the method name and the field name were changed in methods. As a result, the ratio of unchanged lines becomes 1/3, which is less than 60% of Git's default value so that the method is not tracked with Git mechanisms.

[1] https://github.com/kusumotolab/FinerGit.

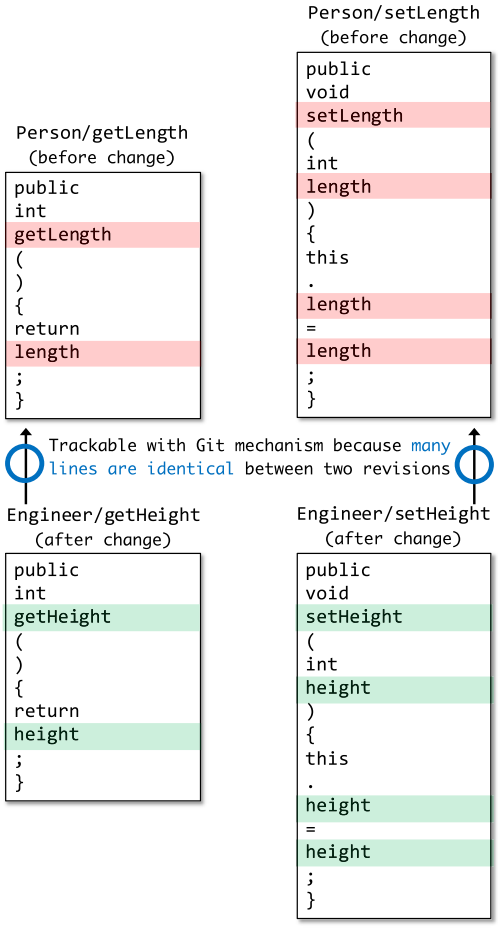


Fig. 3. Tracking files with our technique.

We state two restrictions for the techniques to improve method tracking with Git mechanisms as follows.

- Since the file tracking mechanism in Git is based on line-based text similarity, the characteristics of methods to be used in comparison must be represented as a sequence of text lines. Based on this restriction, complex comparison techniques of file contents such as *tf*/*idf* are not applicable.
- Since the contents of method files are visible and are utilized by developers, they should follow a representation of source code in an understandable way by users. Users may apply `git-diff` command to a method file to see how a method was modified, and the obtained difference should represent the difference of method contents in this case. Based on this restriction, converting method contents to a sequence of computed numeric values used only for a comparison purpose is not suitable.

Fig. 3 shows how the changes in Fig. 2(b) are treated in FinerGit. The file changing mechanism in this technique satisfies the above restrictions. The ratio of unchanged lines becomes 8/10 for `getLength` and 11/15 for `setLength`. Both values are higher than 60%, so that both methods are tracked with Git mechanisms.

3. Proposed technique

Herein, we explain our proposed technique named FinerGit to realize a better method tracking with Git mechanisms. FinerGit is designed on the top of the basic approach explained in Section 2. FinerGit consists of (1) naming convention and (2) two heuristics.

3.1. Naming convention

In FinerGit, a file name for a Java method includes the following information:

- A class name including the method,
- Access modifiers of the method,
- A return type of the method,
- A name of the method, and
- A list of parameter types of the method.

For example, the file name for method `setLength` in Fig. 2 becomes as follows.

`Person#public_void_setLength(int).mjava`

Extension `.mjava` means that this is a method file and the file includes source code of a Java method. Including the above information in the file name reflects code changes around a given method as follows.

- If the name of the class including the given method is changed, the file name of the given method gets changed, but its contents are not changed.
- If another method in the class including the given method is changed, neither file name nor contents of the given method are changed.
- If the signature of the given method is changed, the file name of the given method gets changed and its contents are also slightly changed since the contents include the tokens of the method signature.
- If the contents of the given method are changed, the file name of the given method does not get changed while its contents get changed.

We can track methods with Git mechanisms in any of the above cases if either of them occurs alone. However, if a signature of a method is changed and its contents are also changed broadly, it is difficult to track the method.

3.2. Introducing heuristics

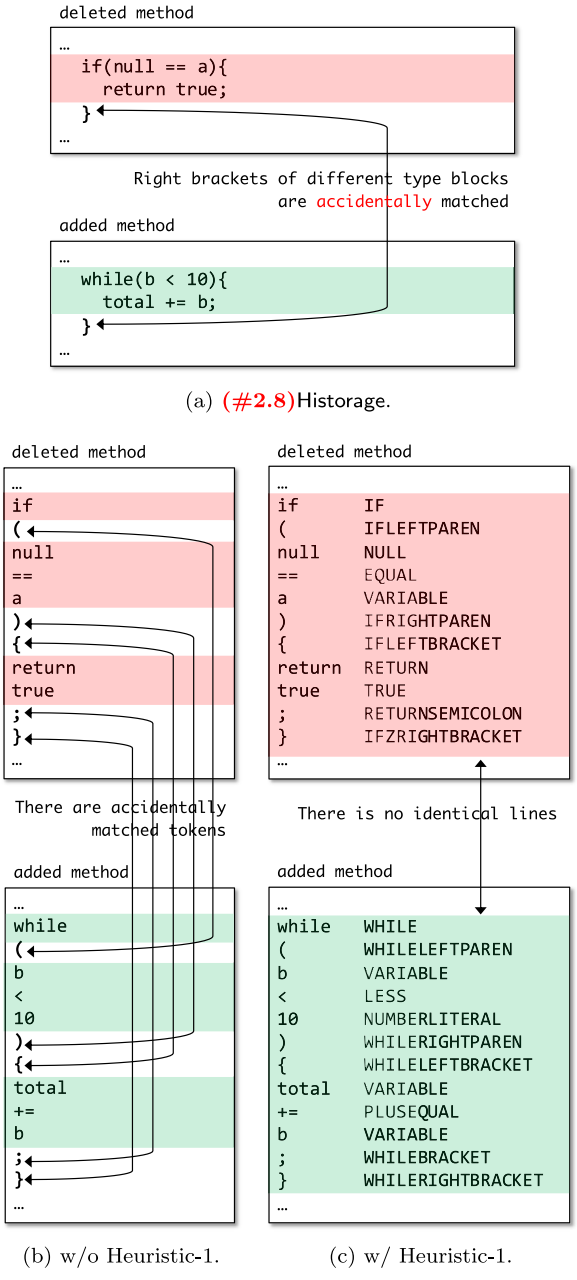
It is not difficult to imagine that Git tracks wrong methods with FinerGit because each line has only a single token and such lines will coincidentally match with many other lines. Thus, we introduce two heuristics to reduce such coincidental matches of unrelated lines.

Heuristic-1: Classifying brackets, parentheses, and semicolons of termination characters in detail.

Heuristic-2: Removing tokens existing in all methods from the targets of similarity calculation.

3.2.1. Heuristic-1

Some termination characters such as brackets, parentheses, and semicolons are omnipresent in Java source code. Such termination characters are used as a part of various program elements. For example, brackets ("{" and "}") are used to initialize arrays in addition to code blocks such as if-statements and for-statements. Thus, if just a bracket is placed on a line, brackets of different roles are coincidentally matched with each other. Such accidental matchings make the similarity between deleted and added methods inappropriately higher. To prevent such accidental matchings, we classify termination characters in detail. More concretely, we add a token explanation to each line. Token explanations prevent accidental matchings of different-role characters from being matched. In this heuristic, semicolons, brackets, and parentheses are classified into 18, 21, and 20 categories, respectively.

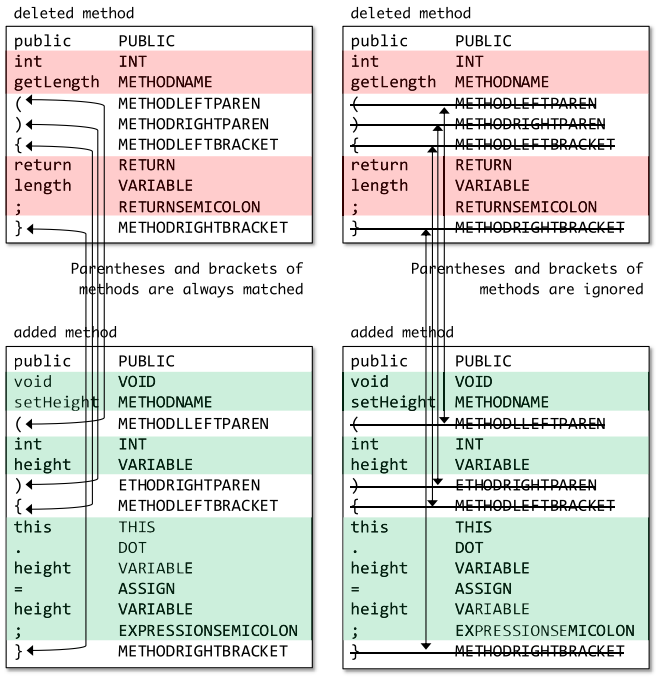


(a) (#2.8)Historage.

(b) w/o Heuristic-1. (c) w/ Heuristic-1.

Fig. 4. Tracking files w/o and w/ Heuristic-1.

Fig. 4 shows how Heuristic-1 affects method tracking. Fig. 4(a) is a method file that Historage outputs. The deleted method includes an if-statement for checking whether variable `a` is `null` or not. The added method includes a while-statement for adding variable `b` to variable `total` repeatedly. Those are different methods, which means a lower similarity between them is better. In the case of Historage, the last line of the if-statement coincidentally matches with the last line of the while-statement so that the similarity between them becomes 1/3 (= 33%). In the case of FinerGit without Heuristic-1, the parentheses and the brackets of the if-statement coincidentally matches with ones of the while-statement. Moreover, the semicolon of the return-statement coincidentally matches with the one of the expression-statement. As a result, the similarity between them becomes 5/12 (= 42%). If we introduce Heuristic-1 to this example, the parentheses, the brackets, and the semicolons get unmatched. Thus, the similarity between them becomes 0/12 (= 0%).



(a) w/o Heuristic-2. (b) w/ Heuristic-2.

Fig. 5. Tracking files w/ and w/o Heuristic-2.

3.2.2. Heuristic-2

The parentheses for parameters and the brackets for method bodies are omnipresent in compilable Java methods. The fact means that at least the four tokens always match between any Java methods. Thus, the similarity between non-related methods gets inappropriately higher. If methods include many tokens, the impact of the four tokens is negligible. However, if methods are small such as getters and setters, the impact of the four tokens become serious. Consequently, we decided not to put the four tokens into files for methods. By removing the four tokens, we prevent the similarity of two non-related methods from getting higher inappropriately.

Fig. 5 shows how Heuristic-2 affects tracking. This example shows a similarity calculation between `getLength` (before refactoring) and `setHeight` (after refactoring) in Fig. 2. A lower similarity between the two methods is better because they are different methods. In the case that we calculate a similarity without Heuristic-2, the similarity becomes 5/10 (= 50%). However, in the case that we adopt Heuristic-2, the similarity becomes 1/6 (= 17%) because the four tokens are ignored.

4. Implementation

We have implemented a tool based on FinerGit. Our tool is open to the public in GitHub, and anyone can use it freely. Our tool takes a Git repository of a Java project, and it outputs another Git repository where each Java method gets extracted as a file. In FinerGit repositories, method files have extension `.mjava`. By executing `git-log` command with option `--follow` for `.mjava` files, we can get their histories.

The name of a method file includes the information of the signature of the method and the class name including the method so that the file name occasionally becomes very long. Very long file names are not compatible with widely-used operating systems. For example, in the case of Windows 10, the absolute path of a file must not exceed 260 characters. If a file name violates the restriction, its file cannot be accessed with Windows' file manager and some other problems occur. In the case of Linux and MacOS, a file name (not a file path) must not exceed 255 characters. For practical use in such widely-used operating systems, if a file name becomes longer than the restriction of operating systems, our tool cuts the file name in the middle and then it appends a hash value that is calculated from the entire file name. This manipulation can shorten the file name while keeping its identity.

There are three types of comments in Java source code: line comments, block comments, and Javadoc comments. Line and block comments are removed from `.mjava` files while Javadoc comments are included in `.mjava` files as they are in `.java` files. This means that a Javadoc comment exists in the header part of `.mjava` file if its original method has it.

Our tool also has a function to extract each field in Java source code as a single file. Files for fields have extension `.fjava`. A field declaration includes multiple tokens such as field name, field type, modifiers, initializations, and annotations. Thus, fields can be tracked as well as methods by placing a single token on a line. A file name for a Java field include the following information:

- A class including the field,
- Access modifiers of the field,
- A type of the field, and
- A name of the field.

For example, the file name for field `length` in Fig. 2 becomes as follows.

`Person#private_int_length.fjava`

Including the above information in the file name reflects code changes around a given field as follows.

- If the name of class including the given field is changed, the file name of the given method gets changes, but its contents are not changed.
- If another method or field in the class including the given field is changed, neither file name nor contents of the given method are changed.
- If the access modifiers, type, or name of the field is changed, the file name of the given field gets changed and its contents are also changed.
- If the annotations and/or initializations of the field are changed, the file name of the given field does not get changed while its contents get changed.

In Historage repository, a file path of a method includes its signature information. Historage makes a directory for each Java class. Methods included in a class are placed in its corresponding directory. On the other hand, our technique places files of Java methods in the same directory of their original Java files. A reason why FinerGit does not make new directories for Java classes is that the conversion time of Historage is long and making a large number of directories in the conversion process is a factor of taking a long time. Both FinerGit and Historage make a large number of files because each Java method is extracted as a single file, but our technique does not make new directories for Java classes. In the both FinerGit and Historage, file name collisions for extracted files do not occur as long as their source code is compilable.

5. Evaluation

We evaluated FinerGit by comparing it with Historage (Hata et al., 2011a). We did not use the published version of Historage implementation[2] but we added Historage's functionality to our tool. By using the same implementation for FinerGit and

[2] https://github.com/niyaton/kenja.

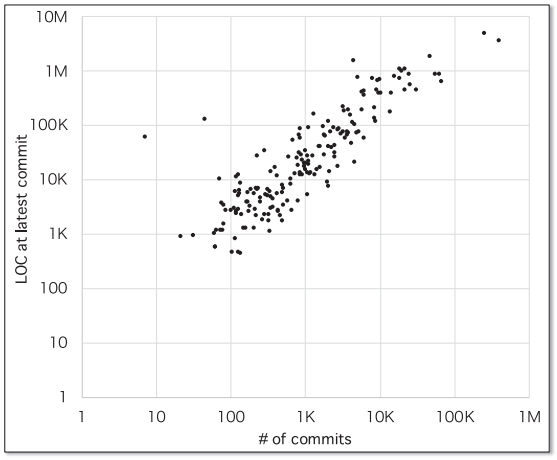


Fig. 6. Project size.

Historage, we can avoid different tracking results due to the differences in implementation details. For example, original Historage makes directories for each Java class while our Historage implementation outputs files of Java methods in the same directory as their original files. The file name convention of our Historage implementation is the same as FinerGit. Thus, in this way, we can evaluate how much method trackability with Git mechanisms gets improved by FinerGit.

We selected 182 Java projects in GitHub as our evaluation targets. In the process of our target selection, we used Borges dataset (Hudson Borges, 2016). This dataset includes 2279 popular projects in GitHub. Firstly, we extracted 202 projects that are labeled as "Java projects". Borges et al. classified the projects in the dataset into six categories: *Application software, System software, Web libraries and frameworks, Non-web libraries and frameworks, Software tools*, and *Documentation*. Secondly, we extracted 185 projects that are other than *Documentation* projects because they are repositories with documentation, tutorials, source code examples, etc. (e.g., java-design-patterns[3]). *Documentation* projects are outside of the scope of this evaluation. Then, we cloned the 185 repositories to our local storage on March 4th 2019. Unfortunately, we found that three of the 185 projects did not include `.java` file. The three projects (google/iosched, afollestad/material-dialogs, and googlesamples/android-topeka) are Kotlin projects. Finally, we removed the three projects from the 185 projects.

Fig. 6 shows the distributions of the number of commits and LOC of the target projects. The two largest repositories in the targets are platform_frameworks_base[4] and intellij-community.[5] The two repositories include approximate 380K and 240K commits, and their latest revisions consist of about 3.7M and 5.0M LOC, respectively.

We generated FinerGit repositories and Historage ones from the 182 target projects. Herein, FinerGit repositories have the file format of including a single token per line with the two heuristics while Historage repositories have the same line format as the original repositories.

We have evaluated FinerGit from the five viewpoints:

- Tracking accuracy,
- Heuristics impacts,
- Project-level tracking results,
- Method-size-level tracking results, and
- Execution time.

Hereafter in this section, we report the results in detail.

5.1. Tracking accuracy

It is not realistic to manually check whether FinerGit generates correct tracking results for each method in the target projects. Thus, we make an oracle for a method for each target project with the following procedure.

1. A method was randomly selected from each target project. In total, 182 methods were selected.
2. Each of the methods in FinerGit repositories was tracked with the following command.
   `> git log --follow -U15 -M20% -C20% -p -- path/to/method.mjava`
   With the above command, a specified file is tracked even if the file was renamed. If there is a file that has a 20% or more similarity, Git regards that file renaming or copying occurred.
3. The tracking results were examined, and oracles of renaming and copying history were made by two of the authors independently. Each author spent several hours on this task. The two authors made different oracles for 34 out of the 182 methods.
4. The two authors discussed the 34 methods so that they obtain consensus for them. After a two-hour discussion, they got consensus oracles for the 34 methods.

With the above procedure, we obtained consensus oracles of tracking results for the 182 methods. Finally, we obtained the resulting oracle set consisting of 426 renaming/copying changes for the 182 methods in total.

Next, we track the methods in FinerGit's repositories and Historage's ones with different thresholds. We used the following command to count how many times Git found renaming and copying with a specified threshold.

```
> git log --follow --oneline -Mt -Ct -p
-- path/to/method.mjava
| grep -e ''^rename from\|^copy from''
| wc -l
```

In the above command, t is the threshold that Git regards given two files have a renaming or copying relationship. We tracked the target methods with 13 different thresholds (i.e., 20%, 25%, 30%, ..., 80%). If tracking results for a method include a higher number of renaming/copying than its oracle, we regard renaming/copying in the over-tracking part as false positives. If tracking results for a method include a lower number of renaming/copying than its oracle, we regard renaming/copying that are not detected as false negatives. We calculated precision, recall, and F-measure for each threshold by summing up the number of false positives and false negatives of all the methods.

Fig. 7 shows how precision, recall, and F-measure changes according to given thresholds. The graphs of Historage and FinerGit have the following features.

- Precision of Historage is very high. Historage has 93.01% of precision even in the case of threshold 20%.
- Recall of Historage is low. Historage has only 57.04% of recall in the case of threshold 20%.
- FinerGit has high precision in the case of high thresholds, but precision gets rapidly decreased for lower thresholds.
- FinerGit has higher recall than Historage for all the thresholds. The recall differences between FinerGit and Historage get bigger for lower thresholds.

[3] https://github.com/iluwatar/java-design-patterns.
[4] https://github.com/aosp-mirror/platform_frameworks_base.
[5] https://github.com/JetBrains/intellij-community.

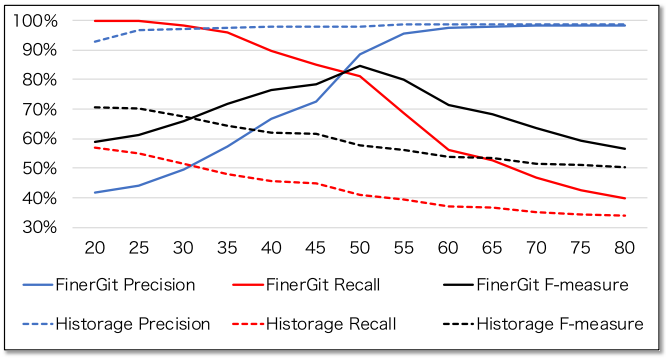


Fig. 7. Precision, recall, and F-measure values.

Table 1
Maximum F-measure and maximum recall.

Repository type		Max F-measure (thr.)	Max Recall (thr.)
H1 OFF,	H2 OFF	82.63% (40%)	58.45% (55%)
H1 ON,	H2 OFF	83.77% (55%)	56.81% (65%)
H1 OFF,	H2 ON	83.26% (35%)	60.09% (50%)
H1 ON,	H2 ON	84.52% (50%)	68.78% (55%)

Historage has a low possibility to track wrong methods while it often misses renaming and copying. On the other hand, in FinerGit repositories, precision gets decreased for lower thresholds while recall improves much. The highest F-measure on FinerGit is 84.52% on threshold 50% while the highest F-measure on Historage is 70.72% and 70.23% on thresholds 20% and 25%, respectively.

5.2. Heuristics impacts

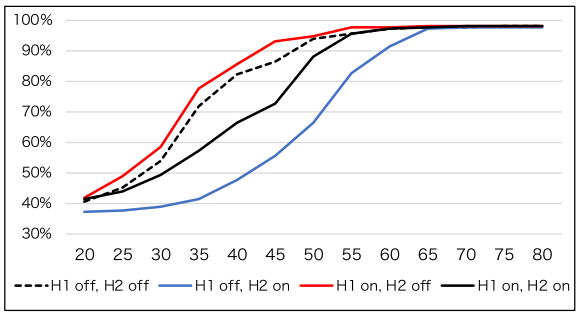
To reveal how each heuristic impacts on method tracking, we measured precision, recall, and F-measure and we also counted found renames for the following four types of fine-grained repositories. The target methods are the same as Subsection 5.1. Herein, rename count means the sum of found renames for all the target methods in a type of repositories.

H1 OFF, H2 OFF: neither heuristics are applied to.
H1 ON, H2 OFF: only Heuristic-1 is applied to.
H1 OFF, H2 ON: only Heuristic-2 is applied to.
H1 ON, H2 ON: both heuristics are applied to. This is the same repository as what we used in Subsection 5.1.

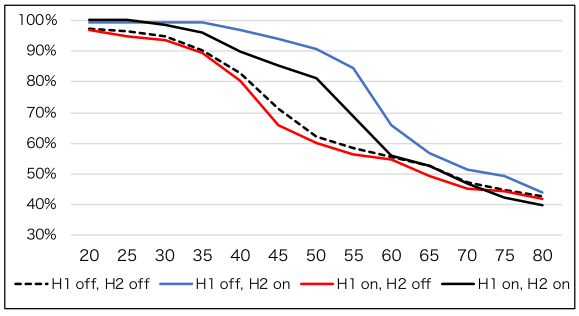
Fig. 8 shows the results. Applying only Heuristic-1 makes it possible to find more renaming so that precision gets decreased while recall gets increased. On the other hand, applying only Heuristic-2 slightly shorten method tracking. As a result, precision gets increased while recall gets decreased. The reasons why applying Heuristic-1 and Heuristic-2 have opposite impacts on method tracking are as follows.

- Applying Heuristic-1 reduces similarities between methods. How much the similarities are decreased depends on the contents on methods. Thus, a different method can be tracked at a commit compared to the case that Heuristic-1 is not applied to.
- Applying Heuristic-2 reduces similarities between all methods. Unlike Heuristic-1, Heuristic-2 does not make a different method tracked. Thus, Heuristic-2 just shortens method tracking.

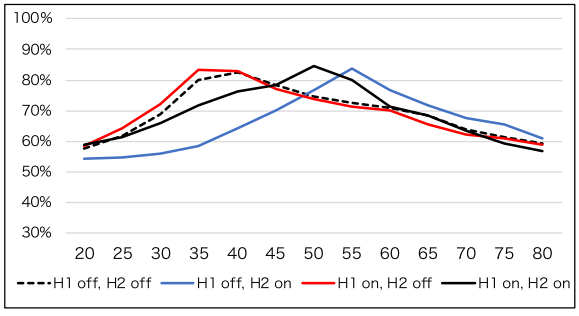
Table 1 shows the maximum F-measure for each type of finer-grained repositories. In this table, the maximum F-measure is the greatest F-measure in all data. All types have almost the same



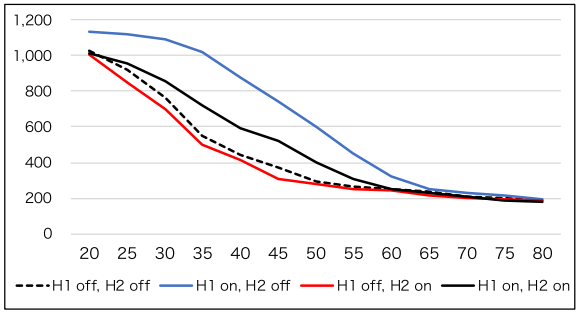
(a) Precision.



(b) Recall.



(c) F-measure.



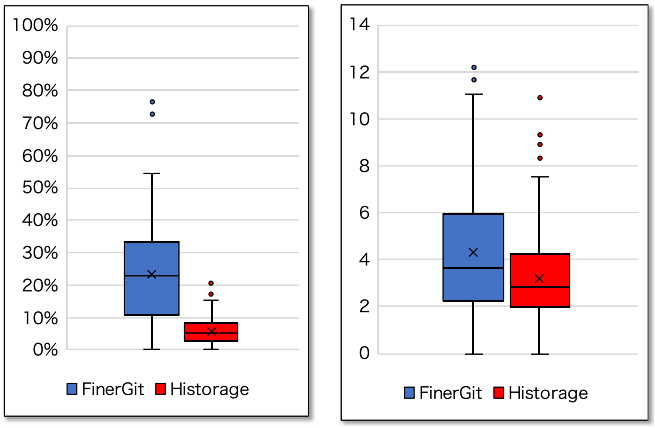
(d) Rename count.

Fig. 8. Precision, recall, F-measure, and rename count when heuristics 1 and 2 are on and/or off.

maximum values. This table also shows the maximum recall when we track methods with over 95% precision. These results show that more method renames are found with keeping 95% precision by applying both heuristics.

5.3. Project-Level tracking results

In this evaluation, we measured the ratio of methods whose tracking results are different between the two tools for each



(a) Ratio of different tracking results. (b) Average change counts.

Fig. 9. Project-level comparisons. (a) shows the ratio of methods whose tracking results are different between FinerGit and Historage for each project. (b) shows the average of change counts for all the methods for each project.

project. We compare how much the number of detected renames is different from FinerGit and Historage under the same precision. As shown in the previous subsection, the two tools have different precision values for different thresholds. To realize a fair comparison, we decided to select different thresholds for FinerGit and Historage that satisfy the following condition: method tracking results with the thresholds have the same precision values and the precision values are as high as possible. Thus, we used threshold 55% for FinerGit and 25% for Historage. The precision of FinerGit on threshold 55% is 95.73%, and Historage on threshold 25% is 96.60%. Those precision values are almost the same and high enough.

Fig. 9 shows the comparison results. In Fig. 9(a), the blue boxplot shows the ratio of methods for which FinerGit found more renames than Historage per project and the red boxplot shows the opposite one. FinerGit found more renames for 22.71% methods on average while the ratio of methods that Historage found more renames than FinerGit is only 5.26%. In Fig. 9(b), the blue boxplot shows the average number of changes identified by FinerGit for all methods of each project. The red one shows the average number of changes identified by Historage. The median values of those boxplots are 3.67 and 2.86, respectively. These results mean that FinerGit can find more renames for all the methods on average.

Next, we show that the tracking improvement by FinerGit is effective via the following two ways:

- considering the fact that some methods were never changed after their initial creation, and
- conducting statistical testing for the tracking results.

5.3.1. Considering never-changed methods

In software development, some methods are never changed after their initial creation. If the 182 target projects include many never-changed methods, it is quite natural that the comparison results between FinerGit and Historage are not so different from each other. Thus, we investigate how many never-changed methods are included in the projects. It is not realistic to manually collect real never-changed methods. In this experiment, we decided to regard methods that both FinerGit and Historage were not able to detect any changes as never-changed methods.

Fig. 10 shows the relationship between the ratio of never-changed methods and the ratio of methods for which FinerGit found more renames than Historage. The 25 percentile, the median, and the 75 percentile of never-changed methods are 6.88%, 15.27%, and 26.50%, respectively. The figure indicates that the more never-changed methods there are, the fewer methods FinerGit found more renames for. Fig. 11 shows the same figures as Fig. 9(a) only for the projects that include 50% or more never-changed methods. As shown in Fig. 11(a), the differences between FinerGit and Historage are small because the majority of their methods is never-changed. Fig. 11(b) shows the differences after we removed never-changed methods from the projects. We can see that the differences between the two tools get much larger. MSR approaches are naturally applied to methods that have change histories. Never-changed methods are exempt from MSR approaches.

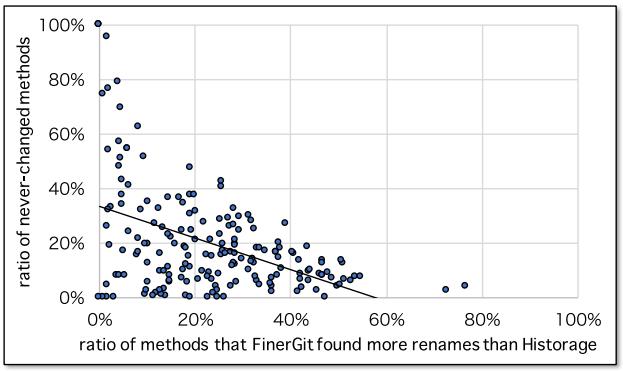
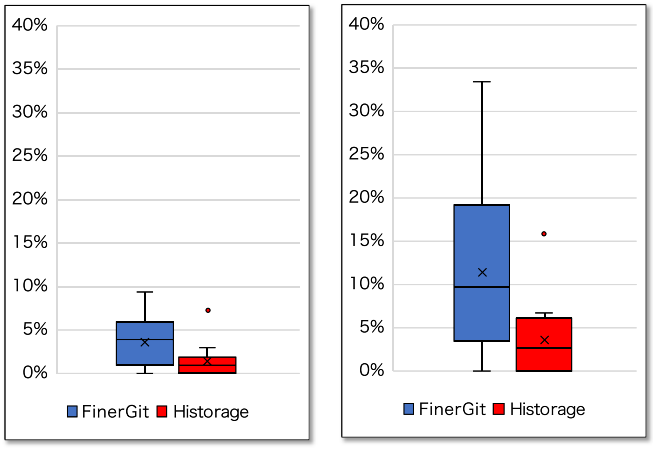


Fig. 10. Relationships between the ratio of methods for which FinerGit found more renames than Historage and the ratio of never-changed methods.



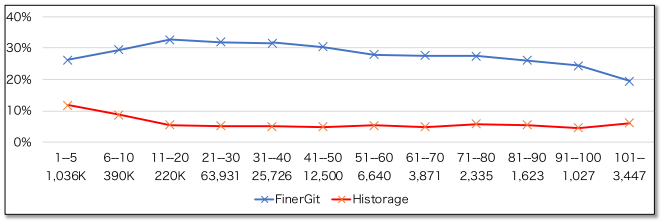
(a) w/ never-changed methods. (b) w/o never-changed methods.

Fig. 11. The ratio of methods whose tracking results are different between FinerGit and Historage for projects where 50% or more methods are never-changed ones.

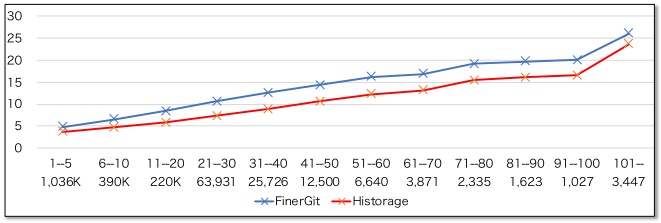
We also investigated how many methods only FinerGit or Historage found at least a change for. The former number is 97,629 and the latter one is 35,553. They are 5.52% and 2.01% of all methods, respectively. Finding changes for more methods means that various MSR approaches requiring past changes can be applied more broadly.

5.3.2. Conducting statistical testing

We applied Paired Wilcoxson's signed ranked test to the comparison results between FinerGit and Historage shown in Fig. 9. The test showed that the comparison results include significant differences regarding both aspects of the ratio (p-value $<$ 0.001) and average change counts (p-value $<$ 0.001). We also applied Cliff's Delta to the comparison results to see the effect size. The resulting values were computed as 0.712 for the ratio and 0.221



(a) Ratio of methods for which FinerGit or Historage found more renames than the other tool.



(b) Average renames that were found by FinerGit or Historage.

Fig. 12. Comparison based on method size.

for the average change counts, which revealed a *large* and a *small* effect size of the improvement achieved by using FinerGit, respectively. Consequently, we can say that FinerGit significantly improves tracking Java methods compared to Historage.

5.4. Method-size-level tracking results

We also conducted comparisons based on method size. In this comparison, we made several method groups based on their size. Then, we compared the tracking results for each group. Fig. 12 shows the comparison results. We can see that there are 1,036K methods whose LOC is in the range between 1 and 5. Herein, the LOC was computed using the original format, not the single-token-per-line one. FinerGit generated longer tracking results for 26.21% of the 1,036K methods. Our research motivation was improving the trackability for small methods, but surprisingly FinerGit improved the trackability for methods of any size.

This figure also shows the average rename counts that were found by FinerGit and Historage. We can see that FinerGit found more renames for methods of any size than Historage. Interestingly, more renames tend to be found for larger methods by both tools.

Consequently, we conclude that the method tracking capability of FinerGit is higher than Historage.

5.5. Execution time

We measured the time that FinerGit reconstructed the repositories of the target projects on MacBook Pro.[6] Fig. 13 shows the measurement results. This figure shows that FinerGit is scalable enough for large repositories. In the longest case, FinerGit took 4209 seconds to reconstruct the repository of intellij-community, which includes more than 240K commits. Of course, this execution time can be shorter if a higher specification computer is used.[7]

Fig. 13 includes the regression line for all the data. The regression line shows that FinerGit takes around 100 s to process each 10K commits for large repositories.

[6] CPU: 2.7GHz quad-core Intel Core i7, memory size: 16 GBytes.

[7] We also measured execution time with our workstation whose CPU is 3.6GHz octet-core Intel Core i9 and memory size is 32 GBytes. The execution time was approximately 22% of MacBook Pro's one.

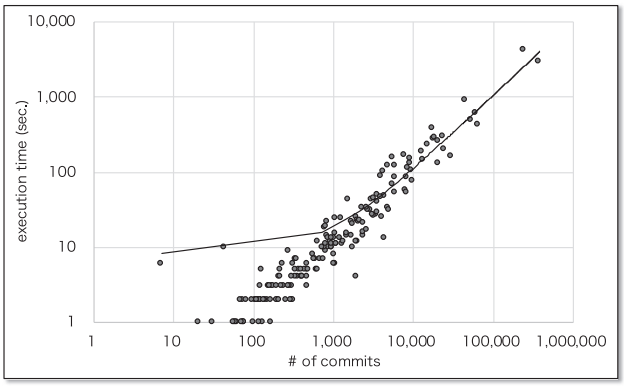


Fig. 13. Execution time of FinerGit.

6. Comparisons with other techniques

We also compared FinerGit with two other techniques, AURA and RefactoringMiner (RMiner). The first comparison target is AURA, which is a technique that takes two versions of Java source code and generates mappings of methods between them (Wu et al., 2010). AURA performs call dependency and text similarity analyses to generate mappings. The second comparison target is RMiner, which is a technique that detects refactorings from commit history (Tsantalis et al., 2018). RMiner's refactoring detection is based on an AST-based statement matching algorithm. RMiner defines different rules for different refactoring patterns. RMiner checks if matching results of two ASTs before and after changes in a given commit follow any of the rules.

We conducted this comparison on the development history of JHotDraw between releases 5.2 and 5.3. This development history is one of the evaluation targets in AURA's literature (Wu et al., 2010). Releases 5.2 and 5.3 include 1519 and 1981 methods, respectively. There are 19 commits between releases 5.2 and 5.3.

6.1. AURA

We made FinerGit's repository and tracked the 1981 methods with 20% threshold with the command shown in Subsection 5.1. The tracking results of 185 methods included renaming and the total number of renaming was 241. Two of the authors independently examined the tracking results to make oracles. Each author spent several hours on this task. The two authors make different oracles for 18 out of the 185 methods. The authors had a discussion on the 18 methods to obtain consensus for them. After a one-hour discussion, they got consensus oracles for the 18 methods. Our consensus oracle includes 161 renamings on 124 methods.

Next, we tracked the 1981 methods with 50% threshold, which is the best F-measure threshold in the evaluation in Subsection 5.1. As a result, we obtained 161 renamings on 124 methods. By comparing the tracking results of 50% threshold with the consensus oracle, We calculated two kinds of precision and recall: one was calculated based on renaming instances; the other was calculated based on methods whose tracking results included at least one renaming in the consensus oracle.

- From the viewpoint of renaming instances, precision and recall were 91.30% and 83.52%, respectively.
- From the viewpoint of methods including renames, precision and recall were 86.29% and 83.59%, respectively.

Table 2
Refactorings detected by RMiner.

Refactoring pattern	# of detected instances
Change Parameter Type	56
Change Return Type	10
Move Method	3
Rename Method	44
Rename Parameter	45
Total	158

Table 3
Precision and recall of RMiner in literature (Tsantalis et al., 2018).

Refactoring pattern	Precision	Recall
Move Method	95.17%	76.36%
Rename Method	97.78%	83.28%

According to AURA's literature (Wu et al., 2010), AURA generated mappings for 97 rules[8] and its precision was 92.38%. By comparing those results, we conclude that FinerGit generated mappings for more methods with slightly-lower precision.

AURA utilizes text similarity and call dependency to generate mappings while FinerGit utilizes only text similarity. On the other hand, AURA takes only two versions of source code to generate mappings while FinerGit utilizes all commits to track methods. Those are the reason why the precision values of the two tools were not so different.

6.2. RefactoringMiner

We performed RMiner[9] on the commit history of JHotDraw between release 5.2 and 5.3. RMiner has a capability of detecting 38 types of refactoring patterns and the following five refactoring patterns correspond to renamings that FinerGit detects: Change Parameter Type, Change Return Type, Move Method, Rename Method, and Rename Parameter. RMiner detected 158 refactoring instances of the five patterns. The detail numbers of refactorings detected by RMiner are shown in Table 2. We compared the 158 refactorings with the 161 renamings detected by FinerGit with 50% threshold. The number of common instances was 65, which was 41.14% of RMiner's refactorings and 40.37% of FinerGit's renamings.

The FinerGit evaluation in Subsection 6.1 shows that FinerGit's tracking accuracy on JHotDraw is high (precision and recall are 91.30% and 83.52%, respectively in 50% threshold). Table 3 shows precision and recall of RMiner for each refactoring pattern in literature (Tsantalis et al., 2018).[10] According to this table, precision and recall of RMiner are also high. However, the common instances between FinerGit and RMiner do not occupy a large portion of all instances detected by either of the techniques. We manually investigated renames and refactorings that had been detected only either of the techniques and found that the results faithfully reflected their different inheritances. There were two major cases of renames that were detected only by FinerGit.

- New parameters were added to methods or return types of methods were changed according to the changes in method's bodies. Those changes were not refactorings but functional enhancements.
- Access modifiers (`public`, `protected`, and `private`) were added/removed/changed. Such changes were refactorings; however they were not supported by RMiner.

On the other hand, refactorings that were detected only by RMiner had changed a large part of method's bodies. Thus, line similarities of method's bodies between such refactorings become low, which leaded to fail to be detected as a renaming by FinerGit.

Herein, we compared FinerGit with RMiner; however their purposes are different from each other. The FinerGit's purpose is tracking Java methods with high accuracy. No matter what kinds of changes are made, FinerGit is able to track methods if a line similarity of the method's bodies between a change is higher than a given threshold. On the other hand, the purpose of RMiner is detecting refactorings in a commit history. No matter how unsimilar between method's bodies are between a refactoring, RMiner is able to detect the refactoring if the refactoring is supported by RMiner.

7. Threats to validity

In the experiment, we used 182 Java projects, and we investigated on tracking results on 1,768K methods in total. Those numbers of projects and methods are large enough so that we expect that the same results are obtained if we conduct another experiment on different Java projects.

To measure precision, recall, and F-measure of method tracking by FinerGit and Historage, we manually constructed oracle for 182 methods. Firstly, two of the authors made oracle for all the 182 methods independently, and then they discussed for which they made different oracle. This process of making oracle is designed to avoid making mistakes and to reduce subjective view on constructing oracle as much as possible.

One more thing about oracle is that, essentially, oracle should be made independently from tracking results of FinerGit and Historage. However, constructing oracle with a fully-manual work is extraordinarily difficult even for a small number of methods. Consequently, in the experiment, we firstly obtained high-recall tracking results with an enough low threshold, and then, we checked how many false positives were included in the tracking results. We consider that this construction process does not ensure 100%-correct oracle but high enough for comparing different techniques. In other word, we made oracle of reasonable quality with a realistic time cost.

In the manual investigation, we checked surrounding 15 lines (as shown in Subsection 5.1) of changes in commits to judge whether method tracking by FinerGit was correct or not. The number 15 came from our experiences with FinerGit because we had checked tracking results of FinerGit before conducting the experiment in this paper.

In the experiment, we discussed the comparison results by focusing on whether FinerGit had found more renaming and copying for Java methods than Historage. However, we also need to see the fact that there were some cases that short tracking results by FinerGit were better than long tracking results by Historage. Such cases mean that FinerGit was able to avoid tracking methods incorrectly. We investigated some of such cases, and then we found that the reason why Historage found a higher number of renames is due to the existences of coincidentally matched lines as shown in Fig. 4(a).

8. Related work

The research that is most related to this paper is of course Historage (Hata et al., 2011a). Historage is useful in research

[8] A rule is a mapping group of multiple methods.

[9] RMiner is available at https://github.com/tsantalis/RefactoringMiner. We used the latest version of the tool at 17th November, 2019. The commit ID is `4bb0e11550b781b61ce1c382a58ea182a2f46944`.

[10] Change Parameter Type, Change Return Type, and Rename Parameter were not investigated in the literature because those refactoring patterns have been recently supported by RMiner.

on mining software repositories because researchers can obtain Java method histories without implementing code/scripts by themselves. Historage has been used in many research before now.

- Hata et al. researched predicting fault-prone Java methods by using method histories obtained with Historage (Hata et al., 2012). Their experimental results showed that the method-level prediction outperformed package-level and file-level predictions from the viewpoint of efforts for finding bugs.
- Hata et al. also used Historage to infer restructuring operations on the logical structure of Java source code (Hata et al., 2011b).
- Fujiwara et al. developed a hosting service of Historage repositories, Kataribe[11] (Fujiwara et al., 2014). Kataribe enables researchers/practitioners to browse method histories on the web, and they can clone Historage repositories in Kataribe into their local storages if they want to conduct further analyses.
- Tantithamthavorn et al. investigated the impact of granularity levels (class-level and function-level) on a feature location technique (Tantithamthavorn et al., 2014). The results indicated that function-level feature location technique outperforms class-level feature location technique. Moreover, function-level feature location technique also required seven times less effort than class-level feature location technique to localize the first relevant source code entity.
- Kashiwabara et al. proposed a technique to recommend appropriate verbs for a method name of a given method so that developers can use various verbs consistently (Kashiwabara et al., 2015). Their technique recommends candidate verbs by using association rules extracted from existing methods. They extracted renamed methods from repositories of target projects using Historage.
- Oliveira et al. presented an approach to analyze the conceptual cohesion of the source code associated with co-changed clusters of fine-grained entities (Oliveira et al., 2015). They obtained change histories of Java methods with Historage. By using the change histories, they identified a set of methods that were frequently changed together.
- Yamamori et al. proposed to use two types of logical couplings of Java methods for recommending code changes (Yamamori et al., 2017). The first type is logical couplings that are extracted from code repositories. They used Historage and Kataribe to obtain logical couplings of Java methods. The second type is logical couplings that are extracted from interaction data. They used a dataset that had been collected by Mylyn (Kersten and Murphy, 2005). Their experimental results showed that there was a significant improvement in the efficiency of the change recommendation process.
- Yuzuki et al. conducted an empirical study to investigate how often change conflicts happen in large projects and how they are resolved (Yuzuki et al., 2015). In their empirical study, they used Historage to conduct method-level analysis. As a result, they found that 44% of conflicts were caused by changing concurrently the same positions of methods, 48% is by deleting methods, and 8% is by renaming methods. They also found that 99% of the conflicts were resolved by adopting one method directly.
- Suzuki et al. investigated relationships between method names and their implementation features (Suzuki et al., 2017). They showed that focusing on the gap between method names and their implementation features is useful to predict fault-prone methods. They used Historage to collect change histories of Java methods in the investigation.

All the above research can be conducted with FinerGit instead of Historage. Moreover, the experimental results may change if FinerGit is used because there is a significant difference in the tracking results between FinerGit and Historage.

We are not the first research group that has used single-token-per-line format for Git repositories. To the best of our knowledge, the study by German et al. was the first attempt to follow this approach (German et al., 2019). They proposed to rearrange source files with single-token-per-line for enabling fine-grained `git-blame`. By using their technique, we can see the person who changed last for each token of the source code. They showed that blame-by-token reports the correct commit that adds a given source code token between 94.5% and 99.2% of the times, while the traditional approach of blame-by-line reports the correct commit that adds a given token between 74.8% and 90.9%. German developed a system cregit[12] based on their proposed technique. cregit has being used in Linux development community.[13] cregit does not extract Java methods as files, which is a difference between cregit and FinerGit.

Heuristic-1, which is described in Subsection 3.2, is refining symbols in source code. On the one hand, symbol refinements are often performed in the process of code clone detection techniques. In the context of clone detection, some symbols are replaced with special ones prior to the matching process. For example, in CCFinder (Kamiya et al., 2002) and NICAD (Roy and Cordy, 2008), which are representative code clone detection techniques, all variables and literals are replaced with a specific wildcard symbol. The purpose of replacements is to detect syntactically-similar code as code clones as much as possible. Such replacements can realize that the matching process ignores differences in variables or literals. On the other hand, in the context of FinerGit, we do not want to ignore differences in variables or literals. If we ignore such differences, the similarity between non-related methods can rise accidentally, which leads FinerGit to make wrong method tracking. The purpose of our Heuristic-1 is to calculate lower similarity values between non-related methods.

There are many research studies of program element matching other than Historage (Godfrey and Zou, 2005). Lozano et al. and Saha et al. implemented method tracking techniques since they need to track method-level clones in their experiments (Angela Lozano, 2008; Saha et al., 2011). Their method-level tracking techniques are line-based comparisons and their comparisons compute numerical similarity values by comparing lines as texts. Thus, in the case that only a small part of a line is changed, the similarity between a before-change line and its after-change line should be high while a simple line-based comparison like `diff` regards that a before-change line is completely different from its after-change line. However, their comparisons are still line-based ones, which include some flaws compared to token-based ones.

- In the cases that the first token of the line is moved to the previous line or the last token of the line is moved to the next line (e.g., left bracket ("{") is moved to the next line due to format change), their line-based techniques regard that multiple lines have been changed while our technique regards that no lines have been changed.
- The same changes have different impacts on lines of different length. For example, variable `abc` is changed to `def` in a 10-character line, the similarity becomes 7/10 while the same change occur in a 40-character line, the similarity becomes 37/40.

[11] http://sdlab.naist.jp/kataribe/.

[12] https://github.com/cregit/.

[13] https://cregit.linuxsources.org/.

Godfrey and Zou detected merging and splitting source code entities such as files and functions. They extended origin analysis (Tu and Godfrey, 2002) to track source code entities. They utilize various information for entities such as entity names, caller/callee relationship, and code metrics values. Wu et al. proposed a technique to identify change rules for one-replaced-by-many and many-replaced-by-one methods (Wu et al., 2010). Their approach is a hybrid one, which means that it uses two kinds of data: caller/callee relationship and text similarity. Kim et al. proposed a technique to track functions even if their names get changed (Kim et al., 2005). Their technique computes function similarities between given two methods. They introduced eight similarity factors such as complexity metrics and clone existences to determine if a function is renamed from another function. Dig et al. proposed a technique to detect refactorings performed during component evolution (Dig et al., 2006). Their technique can track methods even if refactorings change their names. Their detection algorithm uses a combination of a fast syntactic analysis to detect refactoring candidates and a more expensive semantic analysis to refine the results. There are many other approaches for identifying refactorings, and many of them support refactorings that changes method names/signatures such as Rename Method and Parameterize Method pattern (Kim et al., 2010; Milea et al., 2014; Prete et al., 2010; Silva and Valente, 2017; Tsantalis et al., 2018; Weissgerber and Diehl, 2006; Xing and Stroulia, 2005). The advantage of the proposed technique against the above approach should be the ease to use because it utilizes Git mechanisms to track methods. A researcher/practitioner who wants method evolution data does not have to learn how to use new tools.

9. Conclusion

In this paper, we firstly introduce Historage, which converts a Git repository to a finer-grained one. In the finer-grained repository, each Java method exists as a single file. Thus, we can track Java method with Git commands such as `git-log`. However, tracking small methods with Git mechanisms does not work well because small methods do not have good chemistry with the Git rename detection function. Thus, we proposed a new technique that puts only a single token of Java methods per line. In other words, in our technique, each line includes only a single token. We also derived two heuristics to reduce incorrect tracking.

We implemented a software tool based on the proposed technique. We applied our tool and Historage to 182 repositories of Java OSS projects to compare the two tools. The 182 repositories include 1,768K methods in total, which are the targets our comparisons. We found that FinerGit scored 84.52% as maximum F-measure while Historage scored 70.23%. We also confirmed that the proposed technique worked well for methods of any size in spite that our research motivation was to realize better tracking for small methods. Furthermore, we showed that our tool took only short time to construct finer-grained repositories even for large repositories.

In the future, we are going to replicate some experiments of existing research with FinerGit to check whether the better tracking of our tool changes experimental results or not.

Declaration of Competing Interests

There are no interests to declare.

CRediT authorship contribution statement

Yoshiki Higo: Conceptualization, Methodology, Software, Validation, Investigation, Writing - original draft, Writing - review & editing, Project administration. **Shinpei Hayashi:** Software, Validation, Investigation, Data curation, Writing - review & editing. **Shinji Kusumoto:** Investigation, Data curation, Writing - original draft, Writing - review & editing.

Acknowledgements

This work was supported by JSPS KAKENHI grant number JP17H01725 and JP18K11238.

References

Angela Lozano, M.W., 2008. Assessing the effect of clones on changeability. In: Proceedings of the 24th IEEE International Conference on Software Maintenance, pp. 227–236.
Balachandran, V., 2013. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In: Proceedings of the 35th International Conference on Software Engineering, pp. 931–940.
Dig, D., Comertoglu, C., Marinov, D., Johnson, R., 2006. Automated detection of refactorings in evolving components. In: Proceedings of the 20th European Conference on Object-Oriented Programming, pp. 404–428.
Fujiwara, K., Hata, H., Makihara, E., Fujihara, Y., Nakayama, N., Iida, H., Matsumoto, K., 2014. Kataribe: a hosting service of Historage repositories. In: Proceedings of the 11th Working Conference on Mining Software Repositories, pp. 380–383.
German, D.M., Adams, B., Stewart, K., 2019. cregit: token-level blame information in git version control repositories. Empir. Softw. Eng. 24 (4), 2725–2763.
Godfrey, M.W., Zou, L., 2005. Using origin analysis to detect merging and splitting of source code entities. IEEE Trans. Softw. Eng. 31 (2), 166–181.
Hata, H., Mizuno, O., Kikuno, T., 2011. Historage: fine-grained version control system for Java. In: Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th Annual ERCIM Workshop on Software Evolution, pp. 96–100.
Hata, H., Mizuno, O., Kikuno, T., 2011. Inferring restructuring operations on logical structure of java source code. In: Proceedings of 3rd International Workshop on Empirical Software Engineering in Practice, pp. 17–22.
Hata, H., Mizuno, O., Kikuno, T., 2012. Bug prediction based on fine-grained module histories. In: Proceedings of the 34th International Conference on Software Engineering, pp. 200–210.
Hora, A., Silva, D., Tulio, M., Robbes, R., 2018. Assessing the threat of untracked changes in software evolution. In: Proceedings of the 40th International Conference on Software Engineering, pp. 1102–1113.
Hudson Borges, M.T.V., Hora, A., 2016. Understanding the factors that impact the popularity of GitHub repositories. In: Proceedings of the 32nd International Conference on Software Maintenance and Evolution, pp. 1–11.
Kamiya, T., Kusumoto, S., Inoue, K., 2002. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. IEEE Trans. Softw. Eng. 28 (7), 654–670.
Kashiwabara, Y., Ishio, T., Hata, H., Inoue, K., 2015. Method verb recommendation using association rule mining in a set of existing projects. IEICE Trans. Inf. Syst. E98–D (3), 627–636.
Kersten, M., Murphy, G.C., 2005. Mylar: a degree-of-interest model for IDEs. In: Proceedings of the 4th International Conference on Aspect-oriented Software Development, pp. 159–168.
Kim, M., Cai, D., Kim, S., 2011. An empirical investigation into the role of API-level refactorings during software evolution. In: Proceedings of the 33rd International Conference on Software Engineering, pp. 151–160.
Kim, M., Gee, M., Loh, A., Rachatasumrit, N., 2010. Ref-Finder: a refactoring reconstruction tool based on logic query templates. In: Proceedings of the 18th International Symposium on Foundations of Software Engineering, pp. 371–372.
Kim, S., Ernst, M.D., 2007. Which warnings should I fix first? In: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, pp. 45–54.
Kim, S., Pan, K., Whitehead Jr., E.J., 2005. When functions change their names: automatic detection of origin relationships. In: Proceedings of the 12th Working Conference on Reverse Engineering, pp. 143–152.
Kim, S., Whitehead Jr., E.J., Zhang, Y., 2008. Classifying software changes: clean or buggy? IEEE Trans. Softw. Eng. 34 (2), 181–196.
Kim, S., Zimmermann, T., Pan, K., Whitehead, E.J.J., 2006. Automatic identification of bug-introducing changes. In: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering, pp. 81–90.
Meng, X., Miller, B.P., Williams, W.R., Bernat, A.R., 2013. Mining software repositories for accurate authorship. In: Proceedings of the 29th IEEE International Conference on Software Maintenance, pp. 250–259.
Milea, N.A., Jiang, L., Khoo, S.-C., 2014. Vector abstraction and concretization for scalable detection of refactorings. In: Proceedings of the 22nd International Symposium on Foundations of Software Engineering, pp. 86–97.
Oliveira, M.C.D., Almeida, R.B.D., Ramos, G.N., Ribeiro, M., 2015. On the conceptual cohesion of co-change clusters. In: Proceedings of the 29th Brazilian Symposium on Software Engineering, pp. 120–129.

Prete, K., Rachatasumrit, N., Sudan, N., Kim, M., 2010. Template-based reconstruction of complex refactorings. In: Proceedings of the 26th International Conference on Software Maintenance, pp. 1–10.
Rahman, F., Devanbu, P., 2011. Ownership, experience and defects: a fine-grained study of authorship. In: Proceedings of the 33rd International Conference on Software Engineering, pp. 491–500.
Roy, C.K., Cordy, J.R., 2008. NICAD: accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In: Proceedings of the 16th IEEE International Conference on Program Comprehension, pp. 172–181.
Saha, R.K., Roy, C.K., Schneider, K.A., 2011. An automatic framework for extracting and classifying near-miss clone genealogies. In: Proceedings of the 27th IEEE International Conference on Software Maintenance, pp. 293–302.
Silva, D., Valente, M.T., 2017. RefDiff: Detecting refactorings in version histories. In: Proceedings of the 14th International Conference on Mining Software Repositories, pp. 269–279.
Śliwerski, J., Zimmermann, T., Zeller, A., 2005. When do changes induce fixes? In: Proceedings of the 2nd International Workshop on Mining Software Repositories, pp. 1–5.
Soares, G., Gheyi, R., Serey, D., Massoni, T., 2010. Making program refactoring safer. IEEE Software 27 (4), 52–57.
Suzuki, S., Aman, H., Kawahara, M., 2017. Empirical study of fault-prone method's name and implementation: analysis on three prefixes—get, set and be. In: Proceedings of 2nd International Conference on Big Data, Cloud Computing, Data Science & Engineering, pp. 266–271.
Tantithamthavorn, C., Ihara, A., Hata, H., Matsumoto, K., 2014. Impact analysis of granularity levels on feature location technique. In: Proceedings of 1st Asia Pacific Requirements Engineering Symposium, pp. 135–149.
Tsantalis, N., Mansouri, M., Eshkevari, L.M., Mazinanian, D., Dig, D., 2018. Accurate and efficient refactoring detection in commit history. In: Proceedings of the 40th International Conference on Software Engineering, pp. 483–494.
Tu, Q., Godfrey, M., 2002. An integrated approach for studying software architectural evolution. In: Proceedings of 10th International Workshop on Program Comprehension, pp. 127–136.
Weissgerber, P., Diehl, S., 2006. Identifying refactorings from source-code changes. In: Proceedings of the 21st International Conference on Automated Software Engineering, pp. 231–240.
Wu, W., Guéhéneuc, Y.-G., Antoniol, G., Kim, M., 2010. AURA: A hybrid approach to identify framework evolution. In: Proceedings of the 32nd International Conference on Software Engineering, pp. 325–334.
Xing, Z., Stroulia, E., 2005. UMLDiff: An algorithm for object-oriented design differencing. In: Proceedings of the 20th International Conference on Automated Software Engineering, pp. 54–65.
Yamamori, A., Hagward, A.M., Kobayashi, T., 2017. Can developers' interaction data improve change recommendation? In: Proceedings of 41st Annual Computer Software and Applications Conference, pp. 128–137.
Yuzuki, R., Hata, H., Matsumoto, K., 2015. How we resolve conflict: An empirical study of method-level conflict resolution. In: Proceedings of 1st International Workshop on Software Analytics, pp. 21–24.

Yoshiki Higo received his master's degree and Ph.D degree in information science and technology from Osaka University in 2004 and 2006, respectively. At present he is an associate professor at Osaka University. His research interests include mining software repositories, program analysis, and automated program repair. He is a member of IEEE, IPSJ, IEICE, and JSSST.

Shinpei Hayashi is an associate professor of School of Computing at Tokyo Institute of Technology. His research interests include software maintenance and evolution, software development environments, and mining software repositories. He received a DE degree in computer science from Tokyo Institute of Technology in 2008. He is a member of IEEE and ACM.

Shinji Kusumoto received the BE, ME, and DE degrees in information and computer sciences from Osaka University in 1988, 1990, and 1993, respectively. He is currently a professor in the Graduate School of Information Science and Technology at Osaka University. His research interests include software metrics and software quality assurance technique. He is a member of IEEE, IEICE, and JFPUG.