Tracking Method-Level Clones and a Case Study

Kyohei Uemura*, Akira Mori†, Eunjong Choi*, and Hajimu Iida*
*Nara Institute of Science and Technology, Japan
{uemura.kyohei.ub9@is, choi@is, iida@itc}.naist.jp
†National Institute of Advanced Industrial Science and Technology, Japan
a-mori@aist.go.jp

***Abstract*—Analyzing histories of code clones is important for understanding how they affect software development and developers. For this, many studies have been devoted to the approach of tracking code clones. However, to the best of our knowledge, no existing studies have attempted to track code clones in long-term and fine-grained change histories.**

In this paper, we report on the analysis of histories of method-level code clones hosted by a fine-grained version control system called historage, which allowed us to track source code entities across commits.

We have tracked and analyzed method-level code clones in 10 open source software projects and found out that (1) in many projects, method-level code clones are removed regardless of whether they were changed or how frequently they were changed, and (2) a group of method-level code clones created at the same time tend to survive longer than those created individually. We believe that these findings will provide useful insights for future research on code clones such as determining the priority of code clone management.

***Index Terms*—code clone, code clone detection, code clone genealogy, mining software repository**

I. Introduction

Studying histories of code clones is important for understanding how they affect software development and developers in terms of software project management. Since the elements of code clone history include how code clones are created and whether they have been changed or removed, various studies for tracking histories of code clones have been reported. However, existing approaches for tracking code clones have only targeted a partial period of the whole development, changes between specific release versions [1]–[3], or the history of individual clone sets [4]. To the best of our knowledge, the research on extracting history of an unspecified number of code clones over a long development period across fine-grained changes such as commits has not been conducted. We believe that the reason behind this is the high computation complexity required for tracking histories of code clones. Also, investigating long-term history increases the amount of data and leads to the diversity of the analysis target. Therefore, in this research, we propose a lightweight method for tracking code clones in long-term and fine-grained change histories.

To track code clones, it is necessary to identify and link cloned fragments across versions. In existing approaches, the name of the source code entity containing the cloned fragment or the position information of the cloned fragment such as the line number have been used for linking cloned fragments across versions. In our approach, we use historage, a special Git repository containing pre-analyzed syntactic information [5], for detecting and tracking type 1 and type 2 code clones.

Our main motive is to show that investigating histories of code clones and knowing how code clones are created, changed, and deleted is important for code clone management. For this, we focus, in this paper, on the method-level code clones as the primary target of the analysis. The reason is twofold. Firstly, the method-level code clones occupy a major portion (8%) of code clones as Kim et al. reported [6]. Secondly, the method is the most noticeable syntactic entity when considering code clone management such as refactoring support for removing code clones [7], [8] and simultaneous editing with tools to record copy and paste operations [9], [10].

We conducted a small case study using the proposed approach. In the case study, we tracked histories of code clones in 10 open source software projects and analyzed creation, changes and the survival time of clone sets. The results showed that the number of clone sets that experienced changes is rather small, and that the patterns of creation affect the survival time, for method-level code clone. We believe that this finding is useful for determining the priority of code clones in providing a better tool support for the developers.

The rest of the paper is organized as follows. Section 2 explains the related work and Section 3 describes the proposed approach. After explaining the setting and the result of the experiment in Section 4, we discuss the result of the experiment, proposed approach and future work in Section 5. Finally, Section 6 concludes with future work.

II. Related Work

This section describes the related work. Firstly, we describe existing approaches for analyzing code clone histories. Secondly, we show some studies for managing code clones. Finally, we introduce historage [5], a particular Git repository including analyzed syntax information.

A. Histories of Code Clones

Kim et al. described the change patterns of code clones as clone genealogy and proposed an approach for analyzing them [1]. Their approach detects code clones in each version, and then relates code clones across adjacent versions, to trace changes of code clones. To relate code clones, they use the text similarity and the positions of code clones.

Saha et al. used information of the method to link code clones between two versions [2]. They rely on the fact that the method is the smallest source code entity that has a relatively stable name. The information of the method containing the detected code clones is utilized for tracking code clones across versions.

Duala-Ekoko and Robillard propose a method for identifying code clones called the clone region description (CRD) [3]. The CRD consists of class names, method names, block types and conditional expressions. Their approach links code fragments using the CRD. However, there is a possibility that the CRD conflicts between multiple clone sets. If CRDs of multiple cloned fragment are conflicting, their approach identifies each cloned fragment by heuristics based on metrics calculated from the code in the block.

Inoue et al. proposed an approach of tracking histories of code fragments that are similar to the code fragment given as a query [4]. Although this approach can extract the history of a code clone in long-term and fine-grained changes, it is not intended to track and analyze the histories of an unspecified number of code clones from the repository.

In order to track histories of code clones, it is necessary to detect code clones for each version and link them across versions. The task of linking code clones across versions is similar to that of detecting code clones. However, in addition, it is necessary to verify that similar code fragments discovered across versions are indeed the same code clone by checking the position information. In many cases, the position information of a code clone is managed by the number of lines from the start of the file. Even if the position information of the cloned fragment is managed by the number of lines, the position of the cloned fragment may change across versions because of the changes occurring around it. Therefore, collation by the position information is not a simple task, and a linking code clones across versions is time-consuming. Also, in the approach using information of method and block, the syntactic analysis of the detected code clone is necessary, and which takes time. For this reason, it has been difficult to trace history over the entire software development across fine-grained changes. As a result, many existing studies investigate only part of development history or target only specific release versions.

B. Management Code Clones

Choi et al. proposed an approach for identifying code clones to which refactoring is applicable [7]. Silva et al. proposed a system for prioritizing the target of extract-method refactoring using method dependency metrics and recommending it to developers [8]. It is noted that these approaches rely on source code metrics. On the other hand, there exist studies on bug prediction claiming the effectiveness of process metrics [11], [12]. Analysis of fine-grained code clone histories can be used for calculating process metrics related to code clones. Such process metrics may be useful for prioritization of the refactoring targets.

Hou et al. and Jacob et al. proposed tools that record copy-paste operations on IDE [9], [10]. These tools are intended to help managing cloned fragments such as simultaneous editing of code fragments stemming from copy-paste operations. To predict which code clones are likely to be deleted in the future is also useful for such support on IDE. For example, the capability of suggesting to suppress unnecessary code clones when the copy-paste operation is detected is worth consideration.

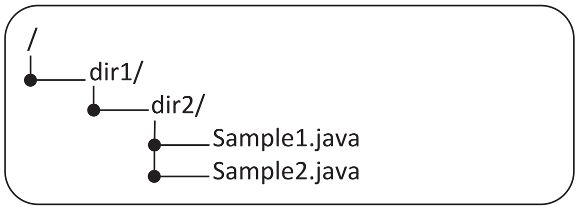
C. Historage: Git Repository with Syntactic Information

Hata et al. proposed a Git repository augmented by pre-analyzed syntactic information, called historage [5]. A historage repository is created by converting a normal Git repository. The historage system can efficiently perform syntactic analysis of the source code in the history by minimizing the range of analysis based on the history management architecture of Git. On top of a Git repository which records the file-level information of the source code, historage records the method-level information. Fig. 1 shows a simple example of an original Git repository and the converted historage repository built on top of that. As the figure shows, in historage, the directory corresponding to each original file is created in the highest hierarchy. The name of the directory expresses the directory hierarchy of the original file by separating it with an underscore (_). Under the directory, the information of classes and methods is stored in the corresponding file in the directory tree. The body of the method is described in the **body** file under the directory of each method name. Because of this, in historage, the git-diff command can report which lines have been changed for each changed methods. It differs from the regular Git repository that only reports the changed lines in the file. Hata et al. have proposed fine-grained bug prediction by using historage [13]. Also, Fujiwara et al. have proposed a refactoring detection method based on historage [14].

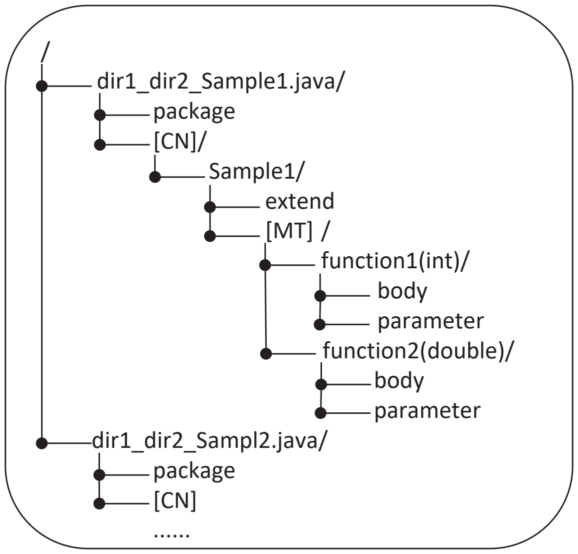
III. Tracking Code Clone Histories using Historage

As described in Section II-A, the existing methods for tracking histories of code clones first detect a code clone in each commit and then determine the relationship of detected code clones in each version. This requires a large amount of computation, and it is difficult to track clones across individual minor versions such as commit units. In this paper, we focus on method-level code clones and propose a lightweight approach for tracing histories of code clones using historage. The key of the proposed approach is that it limits the granularity of the detected code clone to the method-level and identifies the cloned fragment using the method name to reduce the computational complexity for linking across the versions. Furthermore, using pre-analyzed syntactic information in historage contributes to the reduction of computational complexity.

In the proposed approach, we have extended historage with the capability of detecting Type 2 code clones. An example of a directory tree in the extended historage is shown in Fig. 2. In the extended historage, under the method name directory is a



(a) Original directory tree.



(b) Converted directory tree.

Fig. 1. Example of historage.

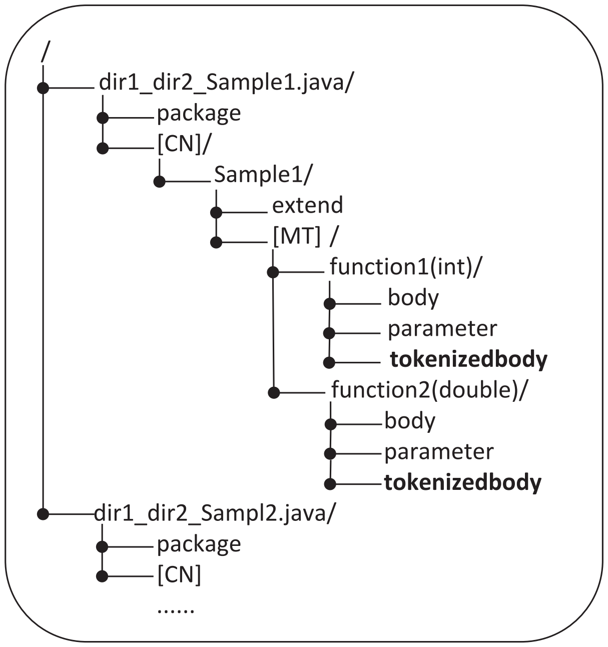


Fig. 2. An example of directory tree of the extended historage.

```
{
  array.sort();
  System.out.println(array[0]);
}
```

(a) Body file.

```
{
identifier
.
Functioncall
(
)
;
Classname
.
Classname
.
Functioncall
(
identifier
[
number
]
)
;
}
```

(b) Tokenizedbody file.

Fig. 3. Example of body file and tokenizedbody file.

tokenizedbody file, which contains the tokenized source code. Fig. 3 shows an example of **body** file and **tokenizedbody** file. In the **tokenizedbody** file, as shown by Fig. 3b, tokens such as variable names, function names, and so on are normalized and described one by one for each line.

We describe a method of detecting code clones and linking them across versions using hiftorage.

A. Code Clone Detection

Algorithm 1 shows a procedure of detecting code clones in the proposed approach. The code clone detection is based on a hash value of the blob object managed by Git. Git manages all files with an object called blob. A blob is an object managed with a hash value of the contents of a file without considering the path. Blobs are shared by each commit, and unless there is a change, the same file is managed with the same blob. If there are multiple files with the same contents in one commit, those files are managed by the same blob. Thus, if multiple tokenizedbody files refer to the same blob within a commit, it means that the tokens are matched, so they are Type 2 code clones. Naturally, the hash values of blobs are calculated when Git is converted to historage. Therefore, a procedure of code clone detection only requires checking the hash value of blobs referenced by tokenizedbodys and can be processed very efficiently.

The detected clone sets have a name for identification. The name is the hash value of the cloned fragments. In many cases, in the detection result of the code clone detection tool, the position information of the cloned fragment is represented by the file path and the position on the file such as line number. On the other hand, in the case of the proposed approach, it is represented by the file path on hisotrage, that includes the file path of the original source code, the class name, and the method name. In the proposed approach, the history of code clones is tracked by completing the above procedure for all two consecutive commit pairs.

B. Tracking Histories of Code Clones

In the proposed approach, a tracking histories of code clones means that creating a graph of a clone set by giving a reference of the related clone set existing in the previous and next commits to the clone set existing in each commit. The related clone set mentioned here refers to a clone set including the same cloned fragment existing between successive commits.

Algorithm 2 shows a procedure of tracking histories of code clones. In proposed code clone detection approach, the detected clone set has a unique name. The name is a hash value calculated from tokens of cloned fragment in clone set. If nothing is modified in cloned fragment, the name of clone set is not different between previous and next commit. Therefore, if nothing modification in cloned fragments, it can merely link clone sets by only referring names of clone sets in the previous and next commit. On the other hand, if there are some modifications in cloned fragments, the names of clone sets are changed in previous and next commit. Therefore, if there are some modifications in cloned fragments, it is necessary to detect corresponding clone sets across previous and next commit using information of cloned fragments. In the proposed approach, information for detecting corresponding clone sets is the names of cloned fragment. In the proposed method, the name of cloned fragment is method name. If the name of cloned fragment in a certain clone set in the previous commit is found in a list of name of cloned fragment in a certain clone set in the next commit, those clone set corresponds and can be linked. In the proposed approach, the history of code clones is tracked by completing the above procedure for all two consecutive commit pairs.

The history information of code clones output by the proposed approach includes the following information.

- a clone set name (hash value of fragment)
- a list of cloned fragment (method) name
- a list of related clone set in a previous commit
- a list of related clone set in a next commit

Also, since each cloned fragment corresponds to one file managed by Git, changes of the cloned fragment can be easily obtained. From these pieces of information, we can track the commit that the clone set was created, changed, and deleted. In addition, since the proposed approach can track differences between commits, we can also analyze using commit message of Git and external information such as bug management system, like existing repository mining research.

IV. Investigation

We conducted a small case study to demonstrate code clone histories are analyzed in the proposed approach. We will first describe the subjects and conditions of the investigation and then show the results.

A. Setting

We chose open source software projects written in Java shown in TABLE I as the subjects of investigation. We chose products provided by a web service called Kataribe. We will explain this later. These products are popular and have domain diversity. We tracked code clones in all commits of these projects. In this study, we have investigated the creation pattern, the presence or absence of changes, and the survival time of code clones in these projects.

We set the minimum token length, which is an option of our code clone detection, to 50 and measured the survival time of code clones.

Algorithm 1 Detecting code clone in each commit

```
Input: $Files$:a set of files in a commit
Output: $Clones$:a set of detected clone sets
1: $Clones := \emptyset$;
2: foreach $File \in Files$ do
3:     $Clones[File.gitId].push(File)$;
4: end for
5: foreach $FileList \in Clones$ do
6:     if $FileList$.len = 1 then
7:         $FileList.pop()$;
8:     end if
9: end for
```

Algorithm 2 Linking clone sets between consecutive commit pairs

```
Input: $Clones_p$:a set of clones in a previous commit,
    $Clones_n$: a set of clones in a next commit
Output: $Relations$:a set of information of relationship of
    clone set in previous commit and clone set in next commit
1: $Relations = \emptyset$
2: foreach $Clone \in Clones_p$ do
3:     if $Clone.gitId\ in\ Clones_n$ then
4:         $Relations[Clone.gitId].push(Clone.gitId)$
5:     else
6:         foreach $File \in Clone.Files$ do
7:             foreach $nextClone \in Clones_n$ do
8:                 if $File\ in\ nextClones.File$ then
9:                     $Relations[Clone.gitId].push($
10:                      $nextClone.gitId)$
11:                end if
12:            end for
13:        end for
14:    end if
15: end for
```

TABLE I
Subject of Investigation

Project	#commit
cassandra	4410
commons-lang	5069
druid	3856
gson	1286
jEdit	7967
jUnit	1229
LeafPic	632
okhttp	1776
picasso	649
poi	965

B. Results

The results of code clone detection are shown in TABLE II. In this table, **#deleted clone** denotes the number of code clones removed during development and **#clone not deleted** denotes the number of code clones in the latest revision. Although the amounts of code clones are different, the method-level code clones do exist in every project. The trend of the ratio of deleted and not deleted code clones differs for each project. This means that the policies for managing code clones such as when to merge cloned fragments are different for the projects. For example, cassandra, commons-lang and jEdit have a large number of code clones while many of them have been deleted. From this, it can be inferred that policy of these projects is to remove code clones proactively. On the other hand, druid has a large number of code clones, most of which have not been deleted. From this, it can be inferred that there is no policy in these project to remove code clones proactively.

The survival time of the code clones is shown in TABLE III. The survival time is counted by the number of commits. In many projects, the survival time of the deleted clone sets is shorter than that of the non-deleted clone sets. On the other hand, there are projects where the survival time is shorter for the clone sets that are not deleted such as jEdit. In such a project, it is presumed that removal of code clones is frequently performed.

Let us focus on the changes on code clones. Note that changes leading to the code clone removal are excluded. TABLE IV shows the number of change experiment clone sets. In most projects, we found that many of the clone sets have not been modified, regardless of whether they would be deleted or not. In poi, almost all clone sets experience changes, but in fact, they were changes that do not affect tokens of the code fragment.

To address the difference attributable to the ways the code clones are created, we classify clone sets according to the information of how code clones are created to compare the survival time of the clone sets:

Creating at once:
clone sets created by adding all cloned fragments at the same commit.

Emerging over time:
clone sets not created by adding all cloned fragments at the same commit.

TABLE V shows average survival time of the clone sets according to the creation types in each project. In many projects, the code clones created at once survive a longer time than others.

TABLE II
NUMBER OF DETECTED CLONE SETS

Project	#deleted clone	#clones not deleted
cassandra	406	168
commons-lang	1337	305
gson	72	67
druid	304	642
jEdit	391	75
jUnit	104	41
LeafPic	67	8
okhttp	118	134
picasso	41	22
poi	33	131

TABLE III
AVERAGE SURVIVAL TIME (NUMBER OF COMMITS) OF CLONE SETS

Project	survival time of deleted clones	survival time of clones not deleted
cassandra	432.02	2039.8
commons-lang	229.87	2907.08
gson	144.6	610.04
druid	511.8	1594.6
jEdit	977.68	561.84
jUnit	69.11	557.73
LeafPic	45	116.38
okhttp	214.58	859.63
picasso	81.98	116.38
poi	180.21	711.73

TABLE IV
NUMBER OF CHANGE EXPERIMENT CLONE SETS

Project	#changes in deleted clones	#changes in clones not deleted
cassandra	15	13
commons-lang	7	1
gson	4	0
druid	9	6
jEdit	3	0
jUnit	1	0
LeafPic	4	0
okhttp	1	0
picasso	1	1
poi	31	131

TABLE V
AVERAGE SURVIVAL TIME (NUMBER OF COMMITS) OF CLONE SETS BY CREATION TYPE

Project	survival time of creating at once clones	survival time of emerging over time clones
cassandra	201	411
commons-lang	1735.5	0
gson	1614	538
druid	583.5	206
jEdit	621	135
jUnit	22	1
leafPic	23	15.5
okhttp	638	119
picasso	130	112.5
poi	963	416.5

V. DISCUSSION

In this section, we discuss the results of the case study and explain future research topics.

A. Investigation Result

It has been reported that, in general, a system with the larger number of changes runs higher risk of bugs [11]. In addition, a large number of changes in code clones cause increase necessary amount of edits and result in adverse effects on the development efficiency. On the other hand, unchanged code clones cause little risk in development. Therefore, we might think that we should choose a code clone that has experienced changes when remove code clones by way of refactoring. However, this hypothesis is denied by the results of the case study. TABLE IV indicates that a rather few code clones experienced changes. On the other hand, TABLE II shows that many code clones have been deleted. These results suggest that the clone sets are removed regardless of whether the cloned fragments have been changed or not in many projects. Further investigation is necessary to clarify the actual reasons why the code clone has been deleted. For example, analyzing other changes in the commit where the code clone was deleted, or tracking changes after a cloned fragment ceases to be a clone are some possible additional investigation. Such investigations can be conducted by using information obtained by the proposed approach based on historage.

In addition, TABLE V shows that code clones have different lifetime depending on the patterns of how they were created. Considering the above results together, it can be said that whether a code clone will be deleted in the future is determined at the time of its creation without being influenced by its changes in the development process. We speculate the following possibilities as the reason for this result. Many of the clone sets, including multiple cloned fragments, which are introduced at the same time may have been the consequence of careful consideration of software design. Conversely, cloned fragments created by copying existing methods may be considered as the temporary code. In order to verify this hypotheses, detailed investigation of software design, such as information of class which including cloned fragment and how cloned fragments are used, is necessary. This can also be conducted with the information obtained from the proposed approach based on historage.

B. Validity of Proposed Approach

The proposed approach for code clone detection assigns a hash value calculated from the normalized tokens to each clone set, and the method name to each cloned fragment, as a unique identifier. The proposed approach for tracking histories of code clones uses these identifiers for linking clone sets between across versions efficiently. Approaches for linking clone sets using the information of the method and the block have been proposed so far. However, these approaches require analyzing the methods containing the cloned fragments. In addition, in the case of block-level code clones, the additional task of calculating metrics is necessary. Since the proposed approach uses historage that manages code fragments on the method unit, the identifier is obtained at the same time as the code clones are detected.

There is a possibility that the code clone detection result of the proposed approach is inaccurate. However, since the proposed approach detects code fragments which have the same hash value of the normalized token string, we believe that the false negatives of Type 1 and 2 code clones is low. There is also a possibility that code fragments have little in common are detected as a clone set. However, since the code fragments of 50 tokens or less are excluded from detection subjects in this experimental study, we believe that there are not many cases where the token strings coincidentally coincide.

C. Future work and Applicability of the Proposed Approach

In this paper, we proposed an approach for tracking histories of code clones and reported on a case study based on it. However, we need more large-scale samples to confirm the effectiveness of the proposed approach. A more detailed study based on the proposed approach is a future work. For example, it is necessary to conduct a survey to support considerations in Section V-A.

The investigation results showed that a very few clone sets experience changes. On the other hand, among the clone sets that experienced changes, there were a few that changed frequently. It is also a future task to investigate what kinds of clone sets are frequently changed, how they are changed, and why the number of changes is large.

Comparison between code fragments which are cloned fragments and code fragments which are not cloned fragments is important for analyzing the characteristics of the cloned method. The use of historage is particularly useful for this kind of tasks.

VI. Conclusion

In this paper, we have proposed an integrated approach of detecting code clones and tracking their histories using historage. Historage offers an augmented form of the Git repository that includes pre-analyzed syntactic information. The proposed approach focuses on method-level code clones and uses a mechanism of Git for detecting and tracking them. Because of this, compared to the conventional approaches, the proposed approach tracks histories of code clones in larger volume much more efficiently.

Furthermore, to confirm the effectiveness of the proposed approach, we have conducted a case study on 10 open source projects. As the result, we have demonstrated that the removal of code clones is not affected by the changes they go through in the development, but by the ways they were created. We believe that this finding is useful for designing better code clone management tools. In this investigation, we have only focused on the presence of changes and the pattern of creation of code clones as the factors that might affect the decision to remove the code clones in the later stage of the development. For example, some tool has been proposed that suggest refactoring subjects with priorities [7], [8]. In such a tool, using the information on whether the subject code clones were created at different times may improve usability. As for future research topics, we are considering tracking method changes after the clone set has been removed in order to compare the cloned methods with the methods with no clones. In another work, we are developing a Web service called Kataribe that provides historage of open source software tools and the information obtained by them [15], [16]. The Web service covers projects described in C#, Java, Python, Ruby. We will provide histories of code clones tracked by the proposed approach on this service. Although we focused on Java in this paper, the proposed approach can detect code clones in the projects implemented in other programming languages if the language can be converted to historage.

REFERENCES

[1] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. An empirical study of code clone genealogies. *ACM SIGSOFT Software Engineering Notes*, 30(5):187–196, September 2005.

[2] Ripon K. Saha, Chanchal K. Roy, and Kevin A. Schneider. An automatic framework for extracting and classifying near-miss clone genealogies. In *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 293–302, Sept. 2011.

[3] Ekwa Duala-Ekoko and Martin P. Robillard. Tracking code clones in evolving software. In *Proceedings of the 29th Internatilnal Conference on Software Engineering*, pages 158–167, 2007.

[4] Katsuro Inoue, Yusuke Sasaki, Pei Xia, and Yuki Manabe. Where does this code come from and where does it go? integrated code history tracker for open source systems. In *Proceedings of 34th the International Conference on Software Engineering (ICSE)*, pages 331–341, June 2012.

[5] Hideaki Hata, Osamu Mizuno, and Tohru Kikuno. 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 (IWPSE-EVOL)*, Sept. 2011.

[6] Miryung Kim, Lawrence Bergman, Tessa Lau, and David Notkin. An ethnographic study of copy and paste programming practices in oopl. In *Proceedings ofInternational Symposium on Empirical Software Engineering (ISESE)*, pages 83–92, Aug. 2004.

[7] Eunjong Choi, Norihiro Yoshida, Takashi Ishio, Katsuro Inoue, and Tateki Sano. Extracting code clones for refactoring using combinations of clone metrics. In *Proceedings of the 5th International Workshop on Software Clones (IWSC)*, pages 7–13, May. 2011.

[8] Danilo Silva, Ricardo Terra, and Marco Tulio Valente. Recommending automated extract method refactorings. In *Proceedings of the 22nd International Conference on Program Comprehension (ICPC)*, pages 146–156, June. 2014.

[9] Daqing Hou, Ferosh Jacob, and Patricia Jablonski. Exploring the design space of proactive tool support for copy-and-paste programming. In *Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research (CASCON)*, pages 188–202, Nov. 2009.

[10] Ferosh Jacob, Daqing Hou, and Patricia Jablonski. Actively comparing clones inside the code editor. In *Proceedings of the 4th International Workshop on Software Clones (IWSC)*, pages 9–16, May. 2010.

[11] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th International Conference on Software Engineering*, 28th International Conference on Software Engineering (ICSE), pages 181–190, 2008.

[12] Haidar Osman, Mohammad Ghafari, Oscar Nierstrasz, and Mircea Lungu. An extensive analysis of efficient bug prediction configurations. In *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering(PROMISE)*, Nov. 2017.

[13] Hideaki Hata, Osamu Mizuno, and Tohru Kikuno. Bug prediction based on fine-grained module histories. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 200–210, June 2012.

[14] Kenji Fujiwara, Norihiro Yoshida, and Hajimu Iida. An approach for fine-grained detection of refactoring instances using repository with syntactic information. *IPSJ Journal*, 56(12):2346–2357, Dec. 2015. in Japanese.

[15] Kyohei Uemura, Yusuke Saito, Shin Fujiwara, Daiki Tanaka, Kenji Fujiwara, Hajimu Iida, and Kenichi Matsumoto. A hosting service of multi-language historage repositories. In *Proceedings of the IEEE/ACIS 15th International Conference on Computer and Information Science (ICIS)*, pages 1–6, June 2016.

[16] Tomohiro Ichinose, Kyohei Uemura, Daiki Tanaka, Hideaki Hata, H. Iida, and K. Matsumoto. Rocat on kataribe: Code visualization for communities. In *Proceedings of the 4th International Conference on Applied Computing and Information Technology/3rd International Conference on Computational Science/Intelligence and Applied Informatics/1st International Conference on Big Data, Cloud Computing, Data Science Engineering (ACIT-CSII-BCD)*, pages 158–163, Dec. 2016.