Inter-Project Functional Clone Detection toward Building Libraries - An Empirical Study on 13,000 Projects -

Tomoya Ishihara, Keisuke Hotta, Yoshiki Higo, Hiroshi Igaki, Shinji Kusumoto
Graduate School of Information Science and Technology, Osaka University, Japan
1-5, Yamadaoka, Suita, Osaka, 565-0871, Japan
{*t-ishihr, k-hotta, higo, igaki, kusumoto*}*@ist.osaka-u.ac.jp*

***Abstract*—Libraries created from commonly used functionalities offer a variety of benefits to developers. To locate such widely used functionalities, clone detection on a large corpus of source code could be useful. However, existing clone detection techniques did not address the creation of libraries. Therefore, existing clone detectors are sometimes unbefitting to detect candidates to be included in libraries. This paper proposes a method-based clone detection technique focusing on building libraries. This method-level granularity is appropriate for building libraries because a method composes a functionally coherent unit, and so it can be easily pulled up into libraries. Also, such a granularity realizes a scalable detection on huge data sets. Our experimental results on a huge data set (360 million lines of code, 13,000 projects) showed that the proposed technique could detect functional clones which might be beneficial on the creation of libraries within a short time frame.**

***Keywords*-Clone detection, Huge data set, Library building**

I. Introdution

Software libraries are useful for development. They could not only reduce development costs but also increase reliability of software. This is because developers need not implement new functionalities if the functionalities are already included in libraries.

In order for developers to gain benefits from libraries, the libraries should be equipped sufficiently with functionalities needed by many of them. This indicates requirements for creating or refining libraries. It is necessary to detect functionalities that are commonly used across multiple software projects to meet the requirements. To detect library candidates, which means functionalities to be included in libraries, code clone detection could be beneficial. However, detecting clones on a large corpus of source code is a problematic task due to its scalability.

Several research efforts have tackled large-scale clone detection on huge data sets or across multiple software projects [1], [2], [3], [4], [5], [6], [7], [8]. These techniques can be categorized into the following two categories.

File-based Detection: Some research detects clones by comparing source files [3], [4]. These techniques regard identical (or similar) files as clones. Such a coarse grain realizes fast detection on large code bases. However, they might miss clones that can be detected with fine-grained (such as token- or line-based) detection.

Scalable Fine-Grained Detection: Some research realizes scalable detection on large data sets with some artifices [1], [2], [5], [6], [7], [8]. They can detect many clones that file-based techniques cannot detect, however, these techniques are inferior to file-based techniques in speed,

These techniques open ways to analyze clones across multiple software projects. However, they are not always sufficient for the purpose of building libraries. One of the issues is that their granularity is not adequate for library creation. For file-based techniques, they can only detect source files as clones if they are identical or similar in file-level. This means that if only a part of source files is a library candidate, file-based techniques miss it. Fine-grained techniques will not miss such candidates. However, clones detected with fine-grained techniques may not be easily extracted as libraries because they can report a part of functionalities as clones. In addition, fine-grained techniques detect too many clones from a massive amount of source code. Identifying library candidates from such a large amount of clones may require many efforts.

This paper proposes a method-based clone detection technique to encourage the creation of software libraries. The method-level detection resolves the above issues: we can locate library candidates even if their owner files are not wholly duplicated, and we can easily extract clones because a method composes a functionally coherent unit. Also, this approach can realize a scalable detection on a huge corpus of source code.

In this paper, we conducted a pilot study to reveal the following two research questions for confirming the effectiveness of the proposed technique.

RQ1: Can the proposed technique complete its detection in a practical time frame on a huge data set?

RQ2: Can the proposed technique detect method-level clones that are useful for building libraries?

Hereafter, we use some terms. A **file/method clone** means a relationship of duplications among two or more files/methods, and a **file/method clone set** is a set of files/methods that have file/method clone relationships to each other in the set. In addition, it uses more two terms,

cloned file and **cloned method**, which are similar to the above terms. However, they are different from each other. A cloned file/method means a file/method that has at least one file/method clone relationship with other files/methods.

II. Related Work

A. File Clone Detection

Sasaki et al. developed a file clone detection tool, FCFinder, and applied it to software systems included in FreeBSD Ports Collection [4]. FCFinder generates a hash value from every source file after removing white spaces, tabs, and comments. As a result, they found that 68% of all the source files were instances of cloned files. Also, they reported that the size of a source file has no impact on whether it becomes an instance of cloned files or not.

Ossher et al. proposed a file clone detection technique [3]. Their method is a combination of three types of detection methods, exact, FQN, and fingerprint [3].

Exact: Compare hash values calculated with strings created from every source file treated as a single string.

FQN: Compare fully qualified names (FQN) of the public classes in every source file.

Fingerprint: Compare method names and field names in source files.

Their empirical study on about 13,000 software systems showed that over 10% source files are identified as cloned files. Also, they reported that using the same libraries and reusing the previous systems for developing new one are typical situations that file clones occur.

The file-based approach realizes high scalability of detection on very large corpora of code. However, file-level granularity is not suited to the creation of libraries. This is because file-based techniques cannot detect functional clones if their owner files are not wholly duplicated. Therefore, file-based detection miss many of library candidates.

B. Scalable Fine-Grained Clone Detection

Hummel et al. proposed a clone detection technique using indexes to achieve incremental and scalable detection [6]. Their technique calculates an index for each statement chunk and compares the indexes. They succeeded to detect clones in 36 minutes from 73 million lines of code with 100 machines.

Cordy and Roy implemented a tool named DebCheck that detects clones between input source files and a Debian source distribution [7]. It requires 10 hours for a preparation of detection on 3 million of C functions. However, the preparation is required at only once, and it can detect clones between input source files and prepared functions within a few minutes after the preparation.

Koschke proposed a scalable clone detection technique focused on detecting lisence violations with suffix trees [8]. His basic idea to reduce time required for detection is to generate a suffix tree for either the subject system or the source code corpus that is a set of other systems. He confirmed that the approach is faster than existing index-based techniques.

These techniques can detect fine-grained clones in a practical time frame on large data sets. This means that they can catch duplications that are missed with file-based techniques. However, these techniques are also not suited for building libraries. Unfortunately, the advantage that they can detect fine-grained clones adversely affects library creation due to the following reasons.

- The amount of detected clones with fine-grained techniques becomes huge. Thus, it requires many efforts to identify library candidates from all the detected clones.
- Clones detected with fine-grained techniques contain only a part of functionalities very often. Therefore, it sometimes requires many efforts to extract such a partially functional clone as libraries.

III. Method-Clone Detection

Herein, we describe how efficiently the proposed technique identifies method clones from huge data sets. The process consists of five steps. In the remainder of this section, we explain every step in detail. The process contains some specializations for Java; however, it is not difficult to apply it to other programming languages by changing the specializations.

STEP1 *Extract methods:* We build ASTs (Abstract Syntax Trees) for every source file, then extract their subtrees corresponding to methods. Why we build ASTs is to perform STEP2.

STEP2 *Normalize methods:* This step removes white spaces, tabs, blank lines, modifiers, annotations, and comments from every method. In addition, variables and each literal is replaced with a special token. The purpose of this step is to absorb trivial differences between methods.

STEP3 *Filter out methods:* There are many simple methods that are obviously not suited to the purpose of library creation. Getter and setter methods are typical instances of them. Many of such simple methods are detected as method clones; however, we do not need such method clones.

Consequently, in this step, the proposed technique excludes simple methods from the target of clone detection. More concretely, it removes methods that have no block statement. Herein, block statements mean do, for, if, synchronized, switch, try, and white in the case of Java.

STEP4 *Generate hash values:* ASTs of target methods are transformed to textual representations, and then a hash value is generated from every textual representation. The hash values are stored into a database for ease of access in the final step.

STEP5 *Make hash groups:* Methods are grouped based on their hash values. A method group consisting of a single method means that the method is not duplicated with any other methods. In other words, it is not an instance of cloned methods. Meanwhile, a method group consists of two or more methods means that they are duplicated to one another. This step extracts only the latter method groups.

IV. A Pilot Study

A. Experimental target

In this research, we selected "*UCI source code data sets*" (in short, *UCI dataset*) [3], [9] as the experimental target. It is a huge set of open source software projects written in Java, and it was used in the study conducted by Ossher et al. [3]. Table I shows an overview of the target data set. As shown in Table I, the target dataset has over 13,000 projects and 360 million lines of code.

B. Experimental setup

We used a single workstation to conduct this experiment. It has 4 logical CPUs, 8 GBytes memory, and a SSD (solid state drive). The proposed method was developed as a software tool with Java. All the steps except STEP5, which are described in Section III, were performed in a parallel way. The tool uses a SQL database to store the information of source files, methods, and detected duplications. The target data set and the database was located in SSD.

In order to compare the proposed technique with the file-based approach, we implemented a file-based detection tool based on the literature [4] by ourselves.

C. Metric for Identifying Library Candidates

It is not realistic to investigate all the method clones detected from a huge data set because of the massive amount of clones. To promote the efficiency to identify suitable library candidates, we use a metric defined as follows.

NOP(m): the **N**umber **O**f **P**rojects that have one or more cloned methods forming method clone set *m*.

NOP represents *how many projects each method clone set is distributed in*. The use of this metric relies on the following Assumption.

Assumption: The more projects a method clone set is distributed in, the more it is suitable for a library candidate.

The authors believe that this assumption is valid according to the following bases.

- If a method clone set is shared by many projects, it should have high reliability because it is used and tested by many developers.
- If a method clone set is shared by many projects, a library created from it will be used by many developers.

Table I
Overview of target data set

# of .java files	2,072,490
# of software projects	13,193
# of methods	5,953,165
total LOC of .java files	361,663,992
size	30.6 GBytes

The calculation of this metric is processed as an additional computation in STEP5.

D. An Overview of Detected Method Clones

We found that 2,937,047 methods become instances of cloned methods, and the number of method clone sets is 814,391. 490,206 out of the 814,391 method clone sets, which is about 60% of all the method clone sets, are across multiple software projects.

Table II shows the number of detected cloned methods. This table tells us that the number of cloned methods included in cloned files is 1,772,488, and the number of cloned methods not in cloned files is 1,164,559. The former is also detected by file-based detection meanwhile the latter can be detected only by the method-based detection. In other words, the method-based technique retrieved 1,164,559 functional clones that the file-based technique missed.

Table III shows the number of source files that have file clone relationships or method clone ones to at least one other source file. The file-based approach reported 791,589 files as cloned files. On the other hand, the method-based approach reported additional 288,200 source files as including at least one method-level duplication. Such source files were 14% of all the source files.

E. Answer to RQ1

RQ1: *Can the proposed technique complete its detection in a practical time frame on a huge data set?*

The answer is **Yes**.

Table IV shows the detection time of method clones in this pilot study. As shown in the table, the total time required to complete the detection is only 3.50 hours. That is to say, the proposed technique could detect method clones from 360 million lines of code within 4 hours. Of course, some artifices such as putting a SQL database on a SSD could impinge on the detection speed. Although those artifices could affect positively, we think, the 4 hours are enough practical to detect method clones on such a huge data set.

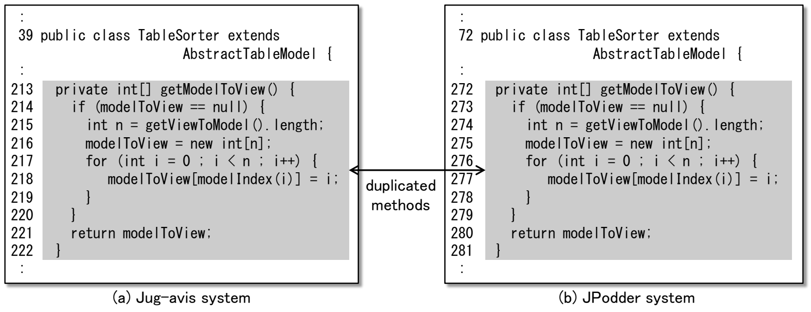
The file-based detection of Sasaki et al. required about 17 hours to finish a file clone detection [4]. The size of data used in their research is 11.2 GBytes. Both the target and the

Table II
Number of cloned methods

	across projects	within a system	total
in cloned files	1,407,338	365,150	1,772,488
not in cloned files	658,500	506,059	1,164,559

Table III
Number of source files being cloned files or including method clones

	across projects	within a system	total
cloned file	592,964	198,625	791,589
having cloned methods	147,532	140,668	288,200



```java
public class TableSorter extends
                    AbstractTableModel {

  private int[] getModelToView() {
    if (modelToView == null) {
      int n = getViewToModel().length;
      modelToView = new int[n];
      for (int i = 0 ; i < n ; i++) {
         modelToView[modelIndex(i)] = i;
      }
    }
    return modelToView;
  }
```

```java
public class TableSorter extends
                    AbstractTableModel {

  private int[] getModelToView() {
    if (modelToView == null) {
      int n = getViewToModel().length;
      modelToView = new int[n];
      for (int i = 0 ; i < n ; i++) {
         modelToView[modelIndex(i)] = i;
      }
    }
    return modelToView;
  }
```

Figure 1. An Instance of Method Clone Sets that is Extracted as a Real Library

experimental environments are different between our pilot study and Sasaki et al.'s experiment, and so it is impossible to compare the speed of detection directly. However, our pilot study revealed that the method clone detection is not so slower than a file clone detection. In Ossher et al.'s paper, there is no description about the detection time, so that we cannot compare detection speed between the proposed method and their method.

F. Answer to RQ2

RQ2: *Can the proposed technique detect method-level clones that are useful for building libraries?*

The answer is **Yes**.

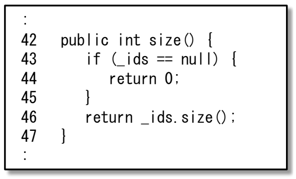
In order to answer this RQ, we browsed the source code of method clone sets whose $NOP(m)$ values were in the top 100, and judged whether they should be extracted as libraries or not. The reason why we browsed a part of method clone sets is that it is impossible to investigate all the 814,391 method clone sets. The top 100 values of $NOP(m)$ fell within the range of 70 and 277.

As a result of our manual inspection, we judged 56 method clone sets out of 100 as suitable for library candidates. Figure 1 shows an instance of such method clone sets. The value of $NOP(m)$ for this clone set is 114. These cloned methods sort rows of JTable instances. The functionalities to sort rows of tables was added to the standard library of Java when version 1.6 was released. Before the version, users of JTable needed to implement the sort functionalities by themselves if they needed it. This is a real example that functionalities implemented in many software projects were pulled up to libraries.

All of the 44 method clone sets judged as not suitable were *stereotype functions*. Herein, a stereotype function indicates a simple and versatile function such as size() or close(). Although the proposed technique attempts to filter out such simple methods in STEP3, unfortunately, some of stereotype functions passed the filtering. Figure 2 shows an example of stereotype functions. The undesirable behavior of the filtering is mainly caused by presences of error checkings. The proposed technique filters out methods that have no block statement, and so stereotype functions having error checkings tend to pass the filtering because most of them include if blocks.

Table IV
DETECTION TIME

Processing	Step1	Step2	Step3	Step4	Step5	Total
Time	2.79h				0.71h	3.50h



```java
  public int size() {
     if (_ids == null) {
        return 0;
     }
     return _ids.size();
  }
```

Figure 2. An Example of Stereotype Functions

V. Threats to validity

Assumption: In the pilot study, we hypothesized that a method clone set is suitable as a library candidate if it is shared by many software projects. Although we believe that this assumption is valid, there may be other criteria to identify suitable library candidates. For instance a combination of $NOP(m)$ and complexity metrics, may work well. The basis of this is that a complex functionality requires many efforts and much attention for its implementation, and so a library created from it will drastically reduce developers' efforts and increase reliability. In addition, the information on how often a method occurs in a single project may be an evidence that the method should be in a library.

Manual Inspection: In order to evaluate the effectiveness of the method-based approach for building libraries, we inspected 100 method clone sets manually and made subjective judgements whether they are suitable for the creation of libraries or not. However, criteria of such judgements are different from person to person. Thus, we will get another result if we ask other people to made such judgements.

Hash collision: The proposed technique uses the MD5 hashing algorithm to compare methods in source code efficiently. However, if a hash collision occurs, non-duplicated methods are accidentally regarded as cloned. In order to check how often hash collisions occur, we performed an automated textual comparison for all the method pairs that have the same hash values. As a result, we found that hash collisions did not occur at all.

Code normalization: The proposed technique replaces each of variables and literals with a special token, which means that the normalization ignores the types of them. If we use more intelligent normalizations, for example, replacing variables and literals with their type names, the amount of detected duplications should be decreased. On the other hand, if we do not normalize source code, trivial differences of methods prevent them from being detected. Selecting an appropriate set of code normalization is a considerably difficult problem.

Method filtering: The proposed technique eliminates methods that have no block statement not for detecting simple methods as clones. However, this elimination may delete methods that should be detected as clones.

Multiple versions: The *UCI dataset* includes multiple versions of some software projects. If we use it as it is, many clones will be detected between different versions of the same software. However, obviously, such clones are not desirable for promoting the creation of libraries. Thus, we removed all the versions except the latest one before conducting the pilot study. The data cleansing might overlook some of redundant versions. However, we found no clones between different versions of a software system through our manual inspections.

Data set: In this experiment, we targeted only a single data set. If we perform the same experiment on different data sets, the rate of duplications will be different. However, the experimental result, we believe, should be general for Java source code because of the massive amount of source code in the experimental target.

VI. Conclusion

This paper proposed a method-based clone detection technique to encourage the creation of libraries. We conducted a pilot study on a huge data set written in Java, which includes about 360 million lines of code among about 13,000 software projects.

Our pilot study proved that the proposed technique could complete its clone detection in 4 hours from the huge data set. Also, we manually inspected 100 detected clones and judged whether they are suitable as library candidates or not. As a result, we confirmed that the proposed technique could detect suitable library candidates including real instances that are pulled up to libraries. In addition, we found that about 40% of method clones were not detected by a file-based approach. This fact shows that the method-based approach could retrieve many functional clones that the file-based approach missed.

As future work, we are going to conduct more experiments to confirm the effectiveness of the proposed technique with human subjects. In addition, the proposed technique should have a wide range of applications such as detecting license violations. We plan to look for other usages of it and customizations for each usage. It may open ways for other usages to detect method clones having some gaps, and so we plan to extend the proposed technique with the LSH algorithm [10], [11].

Acknowledgment

This work was supported by MEXT/JSPS KAKENHI 21240002, 23650014, 24650011, 24680002, and 24700030.

References

[1] Y. Higo, K. Tanaka, and S. Kusumoto, "Toward Identifying Inter-project Clone Sets for Building Useful Libraries," in *Proc. of the 4th International Workshop on Software Clones*, May 2010, pp. 87–88.

[2] S. Livieri, Y. Higo, M. Matsushita, and K. Inoue, "Very-Large Scale Code Clone Analysis and Visualization of Open Source Program Using Distributed CCFinder: D-CCFinder," in *Proc. of the 29th International Conference on Software Engineering*, May 2007, pp. 106–115.

[3] J. Ossher, H. Sajnani, and C. V. Lopes, "File Cloning in Open Source Java Projects: The Good, the Bad, and the Ugly," in *Proc. of the 27th International Conference on Software Maintenance*, Sep. 2011, pp. 283–292.

[4] Y. Sasaki, T. Yamamoto, Y. Hayase, and K. Inoue, "Finding File Clones in FreeBSD Ports Collection," in *Proc. of the 7th Working Conference on Mining Software Repositories*, May 2010, pp. 102–105.

[5] W. Shang, B. Adams, and A. E. Hassan, "An Experience Report on Scaling Tools for Mining Software Repositories Using MapReduce," in *Proc. of the 25th International Conference on Automated Software Engineering*, Sep. 2010, pp. 275–284.

[6] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt, "Index-Based Code Clone Detection: Incremental, Distributed, Scalable," in *Proc. of the 26th IEEE International Conference on Software Maintenance*, Sep. 2010, pp. 1–9.

[7] J. R. Cordy and C. K. Roy, "DebCheck: Efficient Checking for Open Source Code Clones in Software Systems," in *Proc. of the 19th International Conference on Program Comprehension*, June 2011, pp. 217–218.

[8] R. Koschke, "Large-Scale Inter-System Clone Detection Using Suffix Trees," in *Proc. of the 16th European Conference on Software Maintenance and Reengineering*, Mar. 2012, pp. 309–318.

[9] C. Lopes, S. Bajracharya, J.Ossher, and P.Baldi, "UCI Source Code Data Sets," http://www.ics.uci.edu/~lopes/datasets/.

[10] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, "Locality-Sensitive Hashing Scheme Based on p-Stable Distributions," in *Proc. of the 20th Symposium on Computational Geometry*, June 2004, pp. 253–262.

[11] S. Uddin, C. K. Roy, K. Schneider, and A. Hindle, "On the Effectiveness of Simhashing in Clone Detection on Large Scale Software System," in *Proc. of the 18th Working Conference on Reverse Engineering*, Oct. 2011, pp. 13–22.