What kind of and how clones are refactored? : A case study of three OSS projects

Eunjgon Choi
Graduate School of Information Science and Technology, Osaka University
ejchoi@ist.osaka-u.ac.jp

Norihiro Yoshida
Graduate School of Information Science, Nara Institute of Science and Technology
yoshida@is.naist.jp

Katsuro Inoue
Graduate School of Information Science and Technology, Osaka University
inoue@ist.osaka-u.ac.jp

Abstract

Although code clone (i.e. a code fragment that has similar or identical fragments) is regarded as one of the most typical bad smells, tools for identification of clone refactoring (i.e. merge code clones into a single method) are not commonly used. To promote the development of more widely-used tools for clone refactoring, we present an investigation of actual clone refactorings performed in the developments of three Open Source Software (OSS) projects. From the results, we confirmed that clone refactorings are mostly archived by two refactoring patterns, and token sequences of refactored code clones are suggested to have a difference of 50%.

Categories and Subject Descriptors D.1.5 [*Programming Techniques*]: Object-oriented Programming

General Terms Measurement, Experimentation

Keywords Code clone, Refactoring, Levenshtein distance

1. Introduction

Code clone (i.e. a code fragment that has similar or identical fragments) is regarded as a highly prioritized bad smell in source code [4]. So far, much research have been done on the support for clone refactoring (i.e. merge code clones into a single method) [5] [7] [13] [16]. For example, a lot of tools have developed for the detection of code clones [6] [8] [14], code transformation support for forming the template method [7], and the prioritization of code clones based on the difficulty of refactoring [5].

Currently, tools for clone refactoring are not commonly used rather than refactoring tools not intended for clone refactoring. For the development of more widely-used tools for clone refactoring, developers of tools for clone refactoring need to understand actual clone refactorings. For the further refactoring support, tools for clone refactoring need to support frequently-applied refactoring patterns, and then suggest fine-grained code transformations according to the characteristics of code clones. Therefore, the tool developers need to understand what kind of code clones are refactored, and which refactoring patterns are applied to them.

Figure 1. An overview of our research

Actual refactoring patterns applied to non-cloned code were investigated by Murphy-Hill et al. [18]. However, in the case of code clones, applied refactoring patterns are still unclear because identification of applied refactoring patterns to code clones has not been developed to the best of our knowledge.

In this paper, we present a preliminary investigation into clone refactoring in three Open Source Software (OSS)

WRT '12 June 01, 2012, Rapperswil, Switzerland

projects (see Figure 1). First, we propose a Levenshtein distance-based method to identify clone refactorings from version archives. Then, we compute the characteristic of each clone refactoring according to three measurement. The detailed approach is shown in Figure 2.

The main findings of the investigation are as follows:

- Most of refactorings are the applications of Extract Method (EM) and Replace Method with Method Object (RMMO) (see Figure 5).
- Sequences of refactored code clones have a difference of roughly 50% (see Figure 6).
- In the case of the applying RMMO into code clones, large differences are confirmed between the sizes of refactored code clones. On the other hand, in the cases of the other three refactoring patterns, the differences are relatively small (see figure 7).
- In the case of the applying RMMO into code clones, the code clones are in the same Java package but not in the same Java class (see Figure 8).

The rest of paper is organized as follows: Section 2 provides explanations of background of this study. Section 3 presents an investigation method to identify clone refactorings. Section 4 explains case study according to the method explained in Previous Section and then discusses its results and limitations. Section 5 describes related work on this study. Section 6 summarizes our paper with indications about future work.

2. Background

To give clear idea of this study, this section explains in detail code clones, clone refactoring and identification of refactoring.

2.1 Code Clone

A code clone is a code fragment that has similar or identical code fragment in the source code. A clone pair is a pair of code clones which are similar or identical each other. Code Clones are categorized into the following three types base on the textual similarity between the clone pair [2]:

Type 1: Identical code fragments except for variations in whitespace, layout and comments.

Type 2: Syntactically identical fragments except for variations in identifiers, literals, types, whitespace, layout and comments.

Type 3: Copied fragments with further modifications such as changed, added or removed statements, in addition to variations in identifiers, literals, types, whitespace, layout and comments.

A well-known assumption for introducing code clones to software is copying an existing code fragment and pasting it with or without modification. This activity frequently occurs for two main reasons. Firstly, unskilled programmers often create code clones due to the lack of program skills or knowledge of assigned projects. Secondly, mature programmers often create them to preserve high reliability of software or avoid a high-risk for creating new code logic.

Although, a significant amount of code clones are contained in the software systems [1] [15], code clones make software more difficult to be maintained. For example, when a defect is detected in a code fragment of code clone, all of its cloned code fragments should be inspected for the same defect. Therefore, to manage code clones is one of the crucial factors for the effective software maintenance.

2.2 Clone Refactoring

A representative method to manage code clones is merging code clones into a single method using refactoring patterns. Several refactoring patterns from Fowler's book [4] can be applied for clone refactoring.

An example of merging code clones into a single method through refactoring pattern using EM is shown in Figure 3. In Figure 3, two duplicated statements, code clone exist in two methods(*printOwing()* and *printAssets()*) before refactoring. But after refactoring, code clones are extracted as a new method(*printDetails()*) and the old statements are replaced by caller statements of the new method.

Moreover, RMMO also can be applied to code clones that use local variables which is originally applied to a longer method that uses local variables in such a way that developers cannot apply EM. As a result of applying the RMMO, code clones are extracted as a new method into its own object, so that all the local variables become fields on that object.

2.3 Identification of Refactoring

Past refactoring can be covered by analyzing commit logs and code histories, observing programmers and logging refactoring tool use. Using these methods, analyzing the code histories becomes more accurate and can faithfully capture the details of the refactoring event than others [17].

Several techniques for identifying refactoring on code histories have been suggested [19] [20]. Prete et al. proposed a template based refactoring reconstruction technique and tool named Ref-finder [11] [19]. Ref-finder takes two program versions as input data and decomposes input programs as a database of logic facts. After representing each refactoring pattern as template logic rule, Ref-finder identifies refactoring instances via logic queries. Weißgerber et al. proposed a signature-based refactoring detection technique [20]; It stores the most important data of source code in a relational data base and then, looks for added, changed or removed entities (e.g. classes, fields, methods) to get refactoring candidates. Finally, they rank these candidates using clone detection to identify real refactoring.

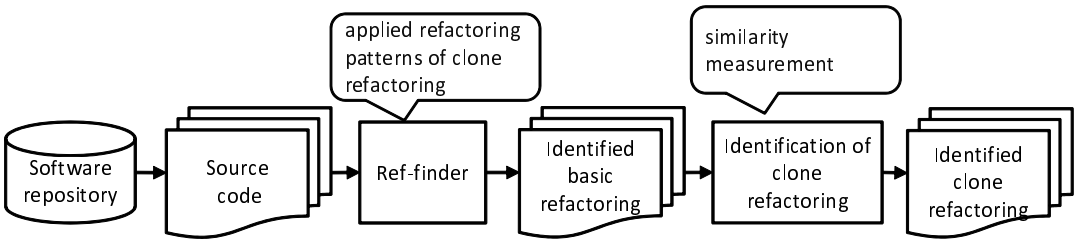

Figure 2. An overview of proposed investigation

Before Refactoring

```
void printOwing(double amount){
  printBanner();

  System.out.println("name:"+ _name);
  System.out.println("amount"+ amount);
}

void printAssets(double amount){
  printResult();

  System.out.println("name:"+ _name);
  System.out.println("amount"+ amount);
}
```

After Refactoring

```
void printOwing(double amount){
  printBanner();

  printDetails(amount);
}

void printAssets(double amount){
  printResult();

  printDetails(amount);
}

void printDetails(double amount){
  System.out.println("name:"+ _name);
  System.out.println("amount"+ amount);
}
```

Figure 3. An example of clone refactoring using Extract Method

3. Investigation Method

We investigated the characteristics of code clones where refactoring was performed based on applied refactoring patterns. The investigation is comprised of the following steps; (1)Identify refactorings on code histories as described in Section 3.1 (2)Identify code clones from the identified refactorings as described in Section 3.2 (3)Investigate the characteristics of code clones where refactoring was performed. Measurement of the refactored code clones are explained in Section 3.3

3.1 Detection of Refactorings

We analyzed past source code of software projects from their software repositories. As targeting refactoring patterns, we selected the following 7 refactoring patterns from Fowler's book; Extract Method (EM), Replace Method with Method Object (RMMO), Extract Class (EC), Parameterize Method (PM), Pull Up Method (PUM), Extract Superclass (ES), and Form Template Method (FTM)

We identified refactoring on each version of source code using Ref-finder which is explained in Section 2.3. We selected Ref-finder because its outputs were validated with high accuracy in Prete's study [19] and it can detects sixty five Fowler's refactoring patterns.

3.2 Identification of Code Clones

Many tools have been proposed for detecting code clones on source code [6] [8] [14]. However, existing code clone detection tools have their own limitations. For instance, CCFinder [8], a token-based code clone detection tool, can only detect *Type-1* and *Type-2* code clones, and can not detect *Type-3* code clones as low-similar code clones.

However, we assume that developers sometimes merge low similar code clones into a single method after much thought. Figure 4 describes an example of refactoring *Type-3* code clones. In Figure 4, a code clone is created by copying existing source code with modification and deletion. However, due to this modification and deletion, these code clones cannot be detected by CCFinder.

To overcome this limitation, we used undirected similarity (*usim*) [16] to identify *Type-3* code clones as well as *Type-1* and *Type-2* code clones in this study. *usim* is a measure originally used to identify code clones for the grow-and-prune model [16]. *usim* uses the Levenshtein distance that measures the minimal amount of changes necessary to transform one sequence of items into a second sequence of items [12]. *usim* is defined by following equation

$$usim\left(f_x, f_y\right) = \frac{\max\left(l_x, l_y\right) - \Delta f_{x,y}}{\max\left(l_x, l_y\right)} \times 100\,(\%)$$

In the upper equation, source code where refactoring was performed is represented as the normalized sequence sf_x (The normalization removes comments, line breaks and insignificant white space). The resulting edit distance

Before Refactoring

```
if (i > j) {
   i = i/2;
   i++;
}
..........

if (i < j) {
  i = i + 10;
}
```

After Refactoring

```
int compare(int i, int j){
  if (i > j) {
    i = i/2;
    i++;
  } else {
      i = i+ 1 ;
  }
  return i
}
```

Figure 4. An example of refactoring in the case of dissimilar code fragments

$\Delta f_x, y = LD(sf_x, sf_y)$ describes the number of items that have to be changed to turn method f_x into f_y. *Levenshtein distance* can be normalized to a relative value using the length of the corresponding sequence $l_x = len(sf_x)$. This equation represents the *Levenshtein distance* between two sequences that are normalized by their maximum size.

We selected pairs of code fragments in the old version which have performed refactoring into the same code fragments in the new version based the outputs of Ref-finder. Next, we determined the similarity between the refactored pairs of code fragments using the *usim* and if the value of *usim* of refactored pairs of code fragments in the old version were over 40% [1], we regarded them as a clone pair who merged into a single fragment.

3.3 Measurement of the Characteristics

To investigate characteristics of refactored clone pairs , we applied two measurement, **Sequence similarity** and **Length difference of sequences between clone pairs**. In addition, to investigate characteristics of classes who contain refactored clone pairs, we applied a measurement, **Class distance** between the classes. The details of the measurement are followings:

Sequence similarity between clone pairs
We used the *usim* to identify refactored clone pairs in Section 3.2. This time, we used the *usim* to measure the sequence similarity between clone pairs where refactorings were performed. If a *usim* value is lower between refactored clone pairs, this means that refactorings were performed between a lower-similar clone pair. Meanwhile, if *usim* values are higher between clone pairs, this means that refactorings were performed between a higher-similar clone pair. This information indicates about which refactoring pattern could be performed according to sequence similarity between clone pairs.

[1] The number 40% is the same number that used in Mende's research to identify code clones [16]

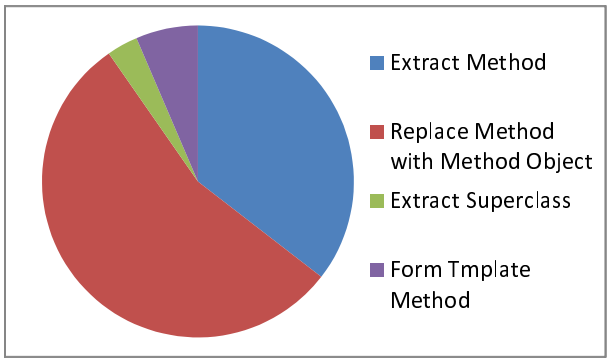

Figure 5. Statistics of identified refactorings

Length difference of sequences between clone pairs
This measurement measures the length difference of sequences between clone pairs where refactorings were performed. If length difference of sequences between a clone pair is lower, this means that refactorings were performed between similar sizes of a clone pair. Meanwhile, if the sequence length difference of sequences between a clone pair is higher, it means that refactorings were performed between a clone pairs with the different sizes. This information indicates about which refactoring pattern could be performed according to the length differences of sequences of a clone pairs.

Class distance
Class distance measures the locations between the classes who contain a clone pair where refactorings were performed. However, except RMMO, 6 refactoring patterns have constraints in the location of clone pair in the old version from 7 refactoring patterns that we selected in Section 3.1. For example, PUM can be performed only clone pairs which are distributed in subclasses who have a common superclass. Therefore, we investigated the class distances in terms of RMMO. This gives developers a clue of applying RMMO according to the location information of clone pairs.

4. Case Study

This section, we explain target systems of case study. Next, we explain the results of case study and then discuss its limitations.

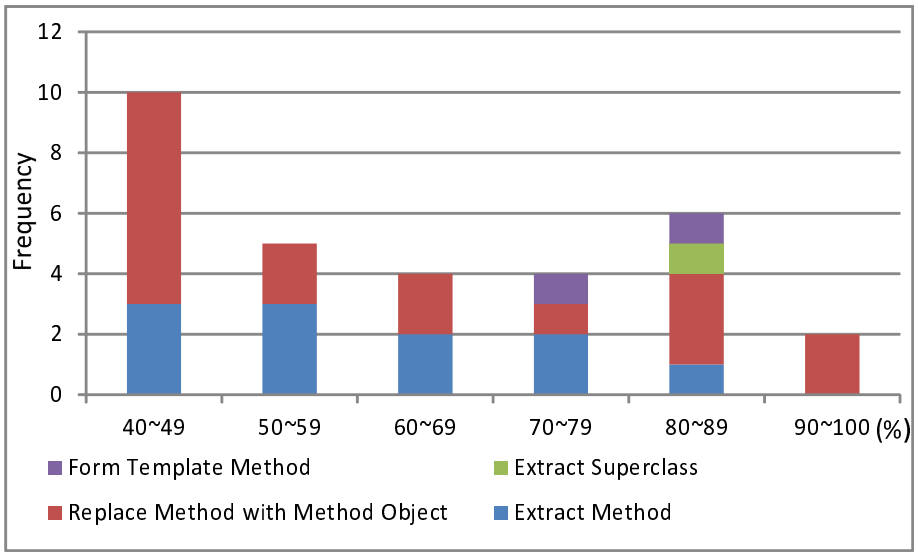

Figure 6. Values of *usim* for each refactoring pattern

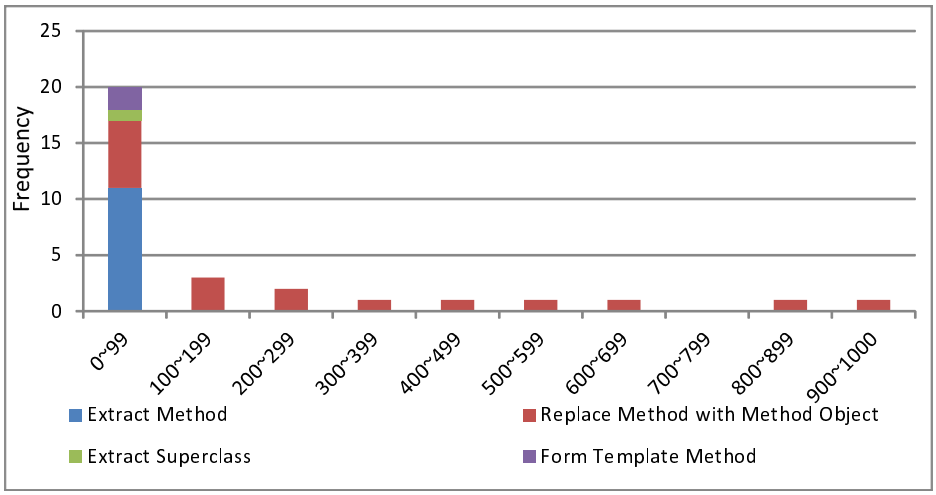

Figure 7. Sequence Length of token sequences for each refactoring pattern

Table 1. Target version pairs in OSS projects

Software Projects	#Version Pairs	Version Pairs
jEdit[2]	2	3.0-3.0.1, 3.0.2-3.1
CAROL[3]	2	302-352, 352-449
Columba[4]	6	62-63, 389-421, 421-422, 429-430, 430-480, 480-481

4.1 Target Systems

To investigate the characteristics of clone pairs where refactoring was performed, suitable software projects with reliable history information on refactoring activities had to be considered. We selected 10 version pairs from jEdit, Columba, and Carol, because their refactoring results from Ref-finder are verified in Prete's work with high recalls and precisions.

4.2 Results

From the results, 31 pairs of code clones were detected from overall subject projects.

Figure 5 shows the number of identified clone pairs in terms of applied refactoring patterns. As shown in Figure 5, four refactoring patterns (EM, RMMO, ES, and FTM) were detected and three refactoring patterns (EC, PM, and PUM) were not detected. RMMO was the most frequently

applied clone refactoring pattern between all refactoring patterns. The second frequently occurred clone refactoring pattern was EM, followed by FTM, and finally the ES pattern. The details on the results of applying three measurements explained in Section 3.3 are followings;

Sequence similarity between clone pairs

Sequence similarity between clone pairs where refactoring were performed are shown in Figure 6. We regarded as a refactored clone pair only if the values of *usim* of a pair of code fragments where refactoring were performed is over 40% as mentioned in Section 3.2. Therefore, the smallest *usim* value of a clone pair where refactoring were performed is 40%. As shown in Figure 6, the values of *usim* of EM, and RMMO were low. Meanwhile, the values of *usim* of ES, and FTM were high.

We suspected that EM, and RMMO were applied to low-similar clone pairs because these refactoring patterns make clone pairs to be moved into another class. Meanwhile, ES, and FTM were applied to high-similar clone pairs because these refactoring patterns are pull up clone pairs in the child classes into a superclass.

Length difference of sequences between clone pairs

The length difference of sequences between clone pairs where refactoring were performed are shown in Figure 7. The length difference of sequences of EM, ES, and FTM were small. Meanwhile, the length difference of RMMO was varying.

Class distance

The class distances in terms of RMMO are shown in Figure 8. Clone pairs in the same package were the most frequently occurred. Meanwhile, clone pairs in the same class or others occurred less frequently in case of RMMO pattern.

4.3 Threats to Validity

This investigation has several limitations.

First, our investigation is based on the data from Ref-finder and *usim*, the results of investigation are rely on these tools. However, the outputs of Ref-finder and *usim* are validated in the previous work, respectively. We believe that the results of investigation in this study are reliable.

Second, we regarded as a refactored clone pair if the values of *usim* between the pair of code fragment in the old version were over 40%. Because of this value we set up, some of actual refactored clone pairs might be omitted or some of wrong data might be included. In the future, we plan to investigate all of the values of *usim* of pairs of code fragments in the old version and check that appropriate value of *usim* to identify actual code clones.

Finally, our case study is conducted on three OSS projects. Therefore, our investigation may not generalize to other OSS projects. To improve generality, we need the future investigation of other OSS projects.

5. Related Work

Kim et al. have developed a clone genealogy extractor [9], and then investigated the genealogies of code clones in OSS [10]. As the result of their investigation, they confirmed that refactoring long-lived clones are difficult. They used CCFinder [8] as a code clone detection tool. It is a sort of scalable tools but detect only syntactically-equivalent code clones (i.e., only *Type-1* and *Type-2* code clones [2]). As they mentioned in their paper, the result of their investigation is affected by the output of CCFinder. In our case study, we confirmed that a lot of refactored and syntactically-different clones.

Burd and Bailey pointed out that the combination of various clone detection tools are needed to identify clones for supporting preventive maintenance [3]. Such combination is also promising for accurate identification of merged code clones from version archives. We need to confirm the usefulness of combined code clone detection tools for the identification of merged clones.

The state of the art tools for fine-grained refactoring detection [19] [21] are designed for the comparison between only two versions. For larger-scale investigation, we need to develop more scalable tool for refactoring detection from a huge number of revisions in software repositories.

6. Conclusion and Future Work

We investigated actual clone refactorings performed in three OSS projects. In this study, we identified the applied refactoring patterns for a clone refactoring. Also, we measured the characteristics of refactored clone pairs. Our suggestions according to the result of the investigation are as follows:

- Tool to support for the applying RMMO refactoring pattern to clone pairs is needed. The tool should support the refactoring of clone pairs spread into not only in the same class but also the same package. To our knowledge, such a tool has not yet be developed.
- Tool to support clone refactoring applied into clone pairs that include different token sequences is needed.
- Tool to support refactoring for clone pairs consisting of different size of token sequences is needed.

As future work, we plan to investigate all of the values of *usim* of pairs of code fragments in the old version and check that appropriate value of *usim* to identify code clones. Next, we would like to apply our investigation method to more OSS projects and industrial software for the generalization of the result of the investigation. Finally, we would like to develop tools for clone refactoring according to the result of investigation.

Acknowledgments

We thank Mr. Raula Gaikovina Kula of Nara Institute of Science and Technology for proofreading this paper. This work

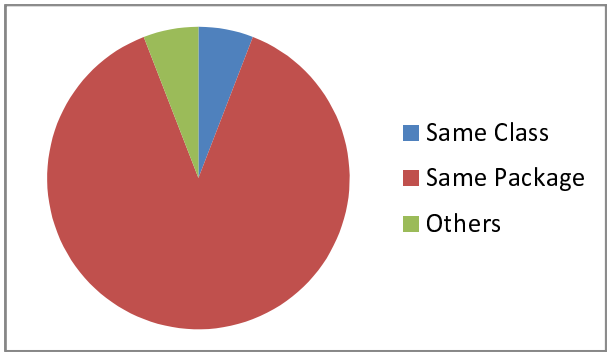

Figure 8. Class distances between clone pairs using Replace Method with Method Object

is partially supported by JSPS, Grant-in-Aid for Scientific Research (A) (21240002).

References

[1] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proc. of WCRE*, pages 86–95, July 1995.

[2] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Trans. Softw. Eng.*, 33(9):577–591, 2007.

[3] E. Burd and J. Bailey. Evaluating clone detection tools for use during preventative maintenance. In *Proc. of SCAM*, pages 36–, 2002.

[4] M. Fowler. *Refactoring: improving the design of existing code*. Addison Wesley, 1999.

[5] Y. Higo, S. Kusumoto, and K. Inoue. A metric-based approach to identifying refactoring opportunities for merging code clones in a Java software system. *Journal of Software Maintenance and Evolution: Research and Practice*, 20:435–461, 2008.

[6] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: Scalable and accurate tree-based detection of code clones. In *Proc. of ICSE*, pages 96–105, 2007.

[7] N. Juillerat and B. Hirsbrunner. Toward an implementation of the form template method refactoring. In *Proc. of SCAM*, pages 81–90, 2007.

[8] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, 2002.

[9] M. Kim and D. Notkin. Using a clone genealogy extractor for understanding and supporting evolution of code clones. In *Proc. of MSR*, pages 1–5, 2005.

[10] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy. An empirical study of code clone genealogies. In *Proc. of SIGSOFT/FSE*, pages 187–196, 2005.

[11] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit. Ref-finder: a refactoring reconstruction tool based on logic query templates. In *Proc. of SIGSOFT/FSE*, pages 371–372, 2010.

[12] V. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10:707–710, 1966.

[13] H. Li and S. Thompson. Incremental clone detection and elimination for erlang programs. In *Proc. of FASE*, pages 356–370, 2011.

[14] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Trans. Softw. Eng.*, 32(3):176–192, 2006.

[15] J. Mayrand, C. Leblanc, and E. M. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proc. of ICSM*, pages 244–253, 1996.

[16] T. Mende, R. Koschke, and F. Beckwermert. An evaluadation of code similarity identification for the grow-and-prune model. *Journal of Software Maintenance and Evolution: Research and Practice*, 21(2):143–169, 2009.

[17] E. Murphy-Hill, A. P. Black, D. Dig, and C. Parnin. Gathering refactoring data: a comparison of four methods. In *Proc. of WRT*, pages 7:1–7:5, 2008.

[18] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. *IEEE Trans. Softw. Eng.*, 38:5–18, 2012. ISSN 0098-5589.

[19] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim. Template-based reconstruction of complex refactorings. In *Proc. of ICSM*, pages 1–10, 2010.

[20] P. Weissgerber and S. Diehl. Identifying refactorings from source-code changes. In *Proc. of ASE*, pages 231–240, 2006.

[21] X. Zhenchang and S. Eleni. Refactoring detection based on umldiff change-facts queries. In *Proc. of WCRE*, pages 263–274, 2006.