IEICE TRANS. INF. & SYST., VOL.E97–D, NO.5 MAY 2014

PAPER

An Investigation into the Characteristics of Merged Code Clones during Software Evolution

Eunjong CHOI[†], ***Nonmember***, **Norihiro YOSHIDA**[††], ***and*** **Katsuro INOUE**[†], ***Members***

SUMMARY Although code clones (i.e. code fragments that have similar or identical code fragments in the source code) are regarded as a factor that increases the complexity of software maintenance, tools for supporting clone refactoring (i.e. merging a set of code clones into a single method or function) are not commonly used. To promote the development of refactoring tools that can be more widely utilized, we present an investigation of clone refactoring carried out in the development of open source software systems. In the investigation, we identified the most frequently used refactoring patterns and discovered how merged code clone token sequences and differences in token sequence lengths vary for each refactoring pattern.

key words: *code clone, refactoring, open source software*

1. Introduction

A code clone is a code fragment that has identical or similar code fragment(s) to it in the source code [1]. It is regarded as a factor that can increase the complexity of software maintenance. For example, when a developer detects a code fragment with a defect, the developer has to inspect all of the fragment's code clones for the same defect. However, developers tend to write code clones unintentionally, even in situations where they can be easily avoided [2].

In recent decades, many tools have been developed to detect code clones [3]–[5]. Lately, the clone research community has gradually shifted its focus from detection to management [2], [6]. Clone refactoring is one of the most vital features of code clones management. It merges a set of code clones into a single function or method. Several tools for clone refactoring have been developed. For example, Eclipse plug-in that supports automatic clone refactoring based on the modified Eclipse refactoring engine [7], and a tool that provides metrics that indicate how code clones can be merged [8]. However, such tools are not commonly used compared to refactoring tools (e.g., Eclipse's refactoring features) not intended for supporting clone refactoring.

Murphy-Hill et al. investigated instances of refactoring in the development of open source software systems [9]. Their study provided valuable insight that could be used to develop more widely used refactoring tools. However, these insights proved insufficient in developing tools for clone refactoring specifically, because merging clones is considerably more complicated than other types of refactoring (e.g., simple code extraction and method renaming/moving) [10].

In this paper, we investigated instances of clone refactoring in open source software systems to uncover clues that could contribute to the development of more widely used tools for clone refactoring. We began by detecting instances of refactoring from consecutive program versions of open source software systems using a refactoring detection tool named Ref-Finder [11]. From the detected instances of refactoring, we further selected instances of the seven refactoring patterns (e.g., Extract Method and Replace Method with Method Object) suggested by Fowler [10], which could be used to merge sets of code clones into the same method. Next, to mitigate the false positive problem, we manually analyzed the outputs of the tool. Then, we measured the similarity of token sequences in the identified instances of refactoring in order to identify instances of clone refactoring. Finally, we analyzed the statistics of the instances of clone refactoring from 63 releases of three open source software systems. The contributions of this paper are summarized as follows:

- **Presenting an approach to investigate how clones were performed refactoring** In order to investigate instances of clone refactoring in three open source software systems, this study presents an approach using code clone identification technique named undirected similarity (usim) and refactoring detection tool named Ref-Finder.
- **Discovering the most used refactoring patterns in clone refactoring and the characteristics of merged clones** This study discovered that Extract Method (EM) and Replace Method with Method Object (RMMO) patterns were the most used when developers perform clone refactoring. Moreover, it found out that large token differences between merged code clones in cases where RMMO and EM patterns were used on pairs of code clones.
- **Suggestions for clone refactoring tools** This paper gives several suggestions for developing tools to support code refactoring based on the results of investigation.

The remainder of this article is organized into the following sections. Section 2 provides an overview of the background for this study. Next, Sect. 3 details our design for investigating the characteristics of merged code clones. Section 4 analyzes the results of our investigation into three open source software systems, discusses sugges-

Manuscript received September 17, 2013.
Manuscript revised December 27, 2013.
[†]The authors are with Osaka University, Suita-shi, 565–0871 Japan.
[††]The author is with Nara Institute of Science and Technology, Ikoma-shi, 630–0192 Japan.
DOI: 10.1587/transinf.E97.D.1244

tions for tools to support clone refactoring based on the results, and then discuses threats to validity. Section 5 reviews related work, and finally Sect. 7 concludes with possible future work.

2. Background

To provide the necessary background for this study, this section explains refactoring, code clone, and clone refactoring in detail.

2.1 Refactoring

Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior [10]. Detecting instances of refactoring in software systems helps developers understand the intentions of code changes made by other developers [12], and can aid researchers in their investigation on how refactoring affects code quality [13].

Several tools for detecting instances of refactoring have been suggested in [11], [14], [15]. For example, Prete et al. proposed a template-based refactoring detection technique and developed a tool named Ref-Finder [11]. Ref-Finder represents a revision pair as logic facts, and then infers instances of refactoring by matching a logic rule for each refactoring pattern with the logic facts. Weißgerber et al. also proposed a technique for detecting instances of refactoring [14]. This technique detects instances of refactoring by analyzing added, changed or removed entities (classes, fields, methods) and then ranking refactoring candidates based on the token similarities between entities before and after refactoring. Hayashi et al. proposed a search-based refactoring detection technique that detects instances of refactoring by finding a sequence of refactoring operations from the initial state (old version) to the final state (new version) using a graph search [15].

2.2 Code Clone

Code clones are code fragments that have similar or identical code fragments in the source code. They are categorized into the following four types based on the textual (Type-1, Type-2, and Type-3) and semantic (Type-4) similarities between the pairs of code clones [1]:

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.

Type-4: Two or more code fragments that perform the same computation but implemented through different syntactic variants.

Dissimilarity, and abstraction in the definition of code clones increase from Type-1 to Type-4. There are numerous tools that utilize various techniques designed to detect these types of code clones [4], [16]. For example, DECKARD detects Type-1, Type-2, and Type-3 code clones using characteristic vectors [4], and MECC detects all code clones types based on a path-sensitive semantic-based static analyzer [16]. However, many code clones, especially those belonging to Type-3, and Type-4, remain undetected by these tools because they are too dissimilar.

2.3 Clone Refactoring

Clone refactoring can be defined as the merging of a set of various code clone types into a single function or method using refactoring patterns [10]. It represents a major aspect of code clone management.

Figure 1 shows examples of clone refactoring based on Fowler's refactoring book [10]. Figure 1 (a) illustrates an example of clone refactoring using the *Extract Method (EM)*, which can be used to merge code clones with similar expressions in the same class. In this figure, version k includes two duplicated statements (shown in bold) existing in two separate methods (printOwing and printAssets). After performing clone refactoring based on *EM* pattern (version $k + 1$), these code clones are extracted from the two methods to create a new method (*printDetails*) and the old statements are replaced by caller statements to the new method.

The *Replace Method with Method Object (RMMO)* can also be used to merge code clones that use local variables by extracting code clones into a new method that is its own object, and all the local variables become fields on that object. This pattern is originally used to a longer method that uses local variables in such a way that developers cannot use *EM* pattern. Figure 1 (b) illustrates an example of clone refactoring using *RMMO* pattern. Before clone refactoring (version k), two cloned methods (normalPrice and salePrice), shown in bold, use local variables. After clone refactoring (version $k + 1$), these code clones are extracted to a new method of a new class (PriceCalculator), and all the local variables are moved into fields of the PriceCalculator class.

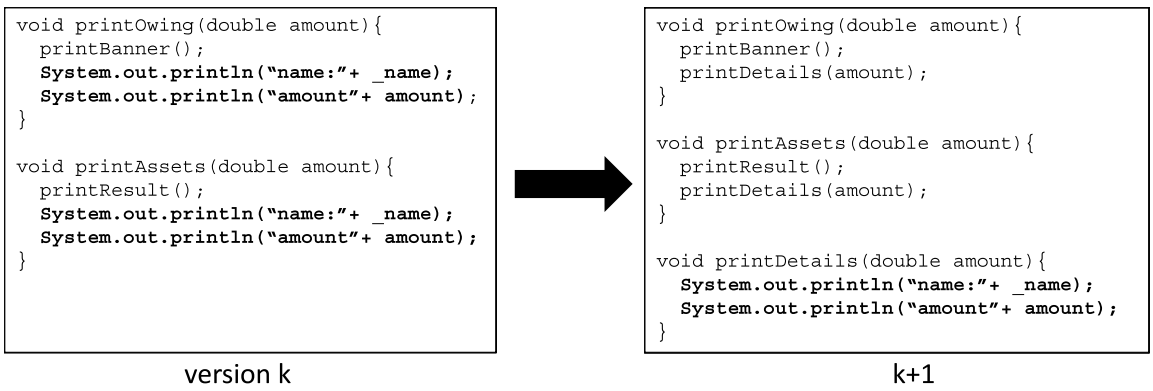
3. Investigation Design

This section details our research approach for investigating the characteristics of merged code clones. Section 3.1 explains the motivations behind our Research Questions (RQs) and Sect. 3.2 describes each step of the investigation.

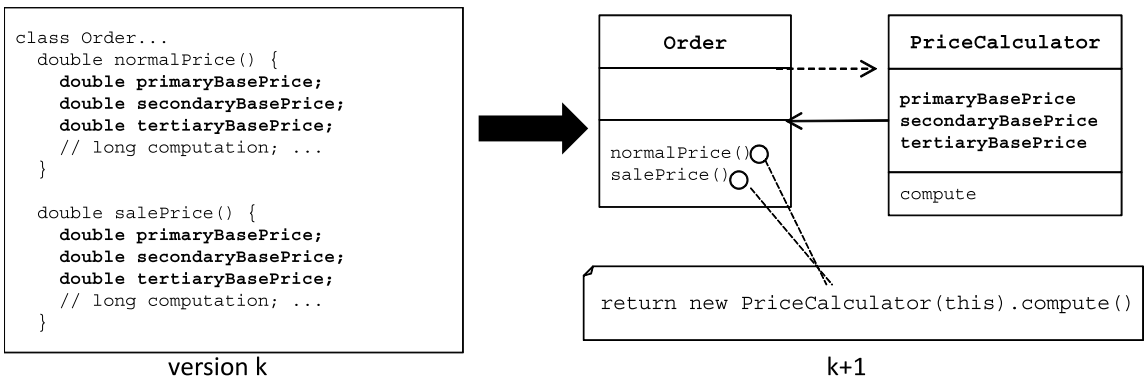
3.1 Motivations of Research Questions

The RQs in this study were devised to identify important clues regarding the development of more widely used tools for clone refactoring. Our RQs were as follows:

Which refactoring patterns are the most frequently



(a) Example of clone refactoring using *EM* pattern



(b) Example of clone refactoring using *RMMO* pattern

Fig. 1 Examples of clone refactoring.

used in clone refactoring? (RQ1) Among refactoring patterns that can be used for clone refactoring, tools for clone refactoring should preferentially support the most frequently used refactoring patterns. Therefore, RQ1 aims to identify which refactoring patterns are the most common. We believe that by answering this RQ, the information could help develop clone refactoring tools that support frequently used refactoring patterns.

How similar are the token sequences between pairs of merged code clones? (RQ2) and **How different are the lengths of token sequences between pairs of merged code clones? (RQ3)** Whether code clones are merged into the same method highly depends on their similarities. Code clones that are very similar to each other are more easily merged into the same method. However, even if they share few similarities, developers are often able to merge them with some effort based on refactoring pattern. For instance, after identical code fragments are merged into the same method, different parts are extracted as an each method using the *Form Template Method* pattern. These RQs aim to uncover how merged code clones differ with respect to the token content (RQ2), and token lengths (RQ3) based on their refactoring pattern. We believe that the answers to these RQs could help develop clone refactoring tools to better detect candidates for pairs of code clones based on their similarities and differences.

How far are pairs of code clones located before clone refactoring? (RQ4) Tools for clone refactoring should be capable of suggesting candidates for clone refactoring. However, it is difficult to select the appropriate candidates because code clones are spread out over various locations (e.g., the same class, different packages). Therefore, RQ4 aims at to find out how far code clones were located before they were refactored. It is our belief that this would further improve a clone refactoring tool's ability to locate pairs of code clone candidates.

3.2 Steps of Our Approach

To our knowledge, no techniques or tools for detecting instances of clone refactoring have ever been proposed. Therefore, we used a refactoring detection tool to first detect instances of refactoring, and then identify instances of clone refactoring from the results using a code clone identification technique. Figure 2 provides an overview of our investigation into the characteristics of merged code clones. It is comprised of the following three steps:

Step 1. Detect instances of refactoring between two consecutive program versions.

Step 2. Identify instances of code refactoring from the set of instances of detected refactoring.

Step 3. Measure the characteristics of merged code fragments in old version of software and categorize the data based on the refactoring pattern.

The following sections explain the detailed of each step.

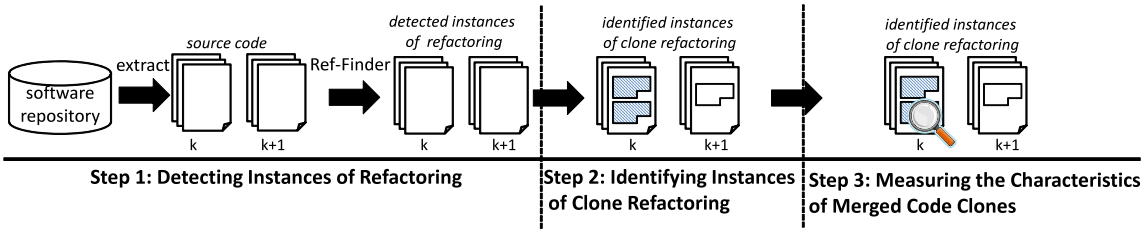


Fig. 2 Overview of the investigation.

Table 1 Statistics of subject systems.

Software	Versions	#Versions	Period
Apache Ant	1.2–1.8.2	17	Jan. 2000–Dec. 2010
ArgoUML	0.12–0.34	13	Oct. 2002–Dec. 2011
Xerces-J	1.0.4–2.9.1	33	Nov. 1999–Nov. 2010

3.2.1 Step 1 : Detecting Instances of Refactoring

In this step, we detected instances of refactoring in open source software systems. To accomplish this, each system's source code was extracted from its respective software repository. Next, Ref-Finder [11] was applied to two consecutive program versions (e.g., version k, $k+1$) to detect instances of refactoring. Ref-Finder takes two consecutive program versions as input data and reports instances of refactoring. It can detect 65 of Fowler's refactoring patterns.

Then, we selected seven refactoring patterns that could be used for clone refactoring specifically. These patterns included the *Extract Class (EC)*, *Parameterize Method (PM)*, *Pull Up Method (PUM)*, *Extract Superclass (ES)*, and *Form Template Method (FTM)*. In addition to these patterns, *EM* and *RMMO* patterns explained in Sect. 2.3 were also included. Finally, we manually validated the outputs of the Ref-Finder since they contained many false positives [17]. To accurately validate the outputs, we referred to existing validated output data of Ref-Finder that was used in a previous study by Bavota et al. [13]. In the study, two master course students at the University of Salerno performed manual validation.

For the purposes of our investigation, we selected the same release versions as in the previously validated data. This included 63 release versions of three Java open source software systems: Apache Ant[†], ArgoUML[††], and Xerces-J[†††]. Table 1 provides statistical data on each of the software systems.

3.2.2 Step 2 : Identifying Instances of Clone Refactoring

In this step, *undirected similarity (usim)* [18] was used to identify instances of clone refactoring from the instances of refactoring detected in Step 1. To elaborate, each refactored pair original was defined as an instance of clone refactoring, only if it satisfied the following three conditions:

Condition 1 : Each pair of code fragments was refactored into the same new method in the new software version. This means that each pair of code fragments in the old version was merged into the same new method in the new version.

Condition 2 : The computed *usim* value of each pair of code fragments in the old version was more than 65%. Many token-based clone detection tools have been proposed [3], [5] because they detect code clones with high accuracy [19]. However, existing tools fail to identify many code clones when they contain large dissimilarities, as are often found in Type-3, Type-4 code clones. For example, CCFinder can only detect Type-1 and Type-2 code clones [3]. Consequently, existing token-based clone detection tools are incapable of accurately detecting certain instances of clone refactoring because developers sometimes perform Type-3, Type-4 clone refactoring.

In order to identify all code clone types, this study used *usim* to identify code clones. Originally used to identify code clones in order to find candidates for clone refactoring in the evolution of software, *usim* uses *Levenshtein distance*, which measures the minimal amount of changes necessary to transform one sequence of items into a second sequence of items [20]. For instance, the *Levenshtein distance* between *survey* and *surgery* is 2, and the one between *color* and *colour* is 1 [21].

The *usim* is defined in Eq. (1) [18]. Each instance of refactoring in its original version is represented as a normalized sequence $sf_x = norm(fx)$, where the normalization function *norm* removes comments, line breaks, and insignificant white spaces. The resulting distance $\Delta f_{x,y} = LD(sf_x, sf_y)$ then describes the number of tokens that must be changed to turn the code fragment f_x into f_y. The *Levenshtein distance* can be normalized to a relative value using the length of the token sequence $l_x = len(sf_x)$:

$$usim\left(f_x, f_y\right) = \frac{\max\left(l_x, l_y\right) - \Delta f_{x,y}}{\max\left(l_x, l_y\right)} \times 100\,(\%) \qquad (1)$$

To confirm this condition, we first concatenated the old version of each instance of refactoring into a single token sequence. During this process, comments and white spaces were removed from token sequences. Then, we normalized the concatenated token sequences by replacing variables and identifiers with a special token. Finally, we computed the *usim* values of each pair of the token sequences that were

[†]http://ant.apache.org/
[††]http://argouml.tigris.org/
[†††]http://xerces.apache.org/xerces-j/

merged into the same new method in a later version of the software. We defined the pairs as code clones if the *usim* values between their token sequences were more than 65%. This threshold was used based on Mende's study [18] because authors discovered that the best compromise of recall and precision can be obtained at 65% *usim* value.

Condition 3 : The token length of each refactored pair was greater than 10 in the old version. In Mende's study [18], the best compromise of recall and precision were obtained at 65% usim values with a minimal token length parameter of 10 tokens. Therefore, we also excluded instances of refactoring where the code fragments in the old version had a token length of fewer than 10 tokens.

3.2.3 Measuring the Characteristics of Merged Code Clones

After identifying instances of clone refactoring, the next step was to measure their characteristics in order to answer the RQs raised in Sect. 3.1.

To answer **RQ1**, we analyzed the number of sets of merged code clones between refactoring patterns. In the analysis, we categorized pairs of code clones based on whether they were merged into the same newly-created method using the refactoring patterns.

To address **RQ2**, we measured the token similarities between pairs of merged code clones using the *usim*, which was also used to identify code clones in Sect. 3.2.2. Among a set of merged code clones, the *usim* values can sometimes differ from each other, because pairs of code clones were categorized into the same set based on the merged new method in the new version. To analyze the distribution of the *usim* values accurately, we measured U_{mi} (a set containing the minimum usim values of merged code clones), U_{av} (a set containing the average usim values of merged code clones), and U_{mx} (a set containing the maximum usim values of merged code clones) between refactoring patterns.

Suppose that $S_1, S_2, \cdots, S_n$ (where $1 \leq i \leq n$) represents sets of merged code clones refactored by the same refactoring pattern, u_{mi_i} represents the minimum *usim* value of S_i, u_{av_i} represents the average *usim* value of S_i, and u_{mx_i} represents the maximum *usim* value of S_i, then $U_{mi} = \{u_{mi_1}, u_{mi_2}, \cdots, u_{mi_n}\}$, $U_{av} = \{u_{av_1}, u_{av_2}, \cdots, u_{av_n}\}$, and $U_{mx} = \{u_{mx_1}, u_{mx_2}, \cdots, u_{mx_n}\}$.

To answer **RQ3**, we investigated how the length of token sequences differed between pairs of merged code clones. First, we defined the length of token sequences differences between a pair of merged code clones c_1 and c_2 as $LD = |lt_1 - lt_2|$, where lt_1 represents the length of token sequences of c_1 and lt_2 represents the length of token sequences of c_2. Secondly, we measured L_{mi} (a set containing the minimum LD values of merged code clones), L_{av} (a set containing the average LD values of merged code clones), and L_{mx} (a set containing the maximum LD values of merged code clones) between refactoring patterns because among a set of merged code clones, the LD values sometimes vary.

Suppose that $S_1, S_2, \cdots, S_n$ (where $1 \leq i \leq n$) represents sets of merged code clones refactored by the same refactoring pattern, l_{mi_i} represents the minimum LD value of S_i, l_{av_i} represents the average LD value of S_i, and l_{mx_i} represents the maximum LD value of S_i, then $L_{mi} = \{l_{mi_1}, l_{mi_2}, \cdots, l_{mi_n}\}$, $L_{av} = \{l_{av_1}, l_{av_2}, \cdots, l_{av_n}\}$, and $L_{mx} = \{l_{mx_1}, l_{mx_2}, \cdots, l_{mx_n}\}$.

In response to **RQ4**, we defined the term *class distance* as an indicator of the location between pairs of merged code clones in the old version of the code. Locations can be categorized into one of the following categories: Same Class, Same Package, and Different Packages.

We only investigated the *class distances* for code clone refactored by *RMMO* pattern because the other six refactoring patterns already contained constraints regarding the location of pairs of the code clones. For example, *PUM* pattern can only be used to pairs of code clones in subclasses that have a common superclass. In order to answer RQ4, we analyzed the *class distances* between pairs of merged code clones refactored by *RMMO* pattern.

4. Results and Suggestions

This section describes the results of our investigation and discusses our suggestions for developing tools for clone refactoring based on these results[†]. This section also identifies threats to validity of our results.

4.1 Investigation Results

This section details the investigation results and provides answers to the RQs based on these results.

Which refactoring patterns are the most frequently used in clone refactoring? (RQ1)

To answer **RQ1**, Table 2 shows the number of sets of merged code clones, and the numbers of pairs of merged code clones (in parenthesis) organized by each refactoring pattern.

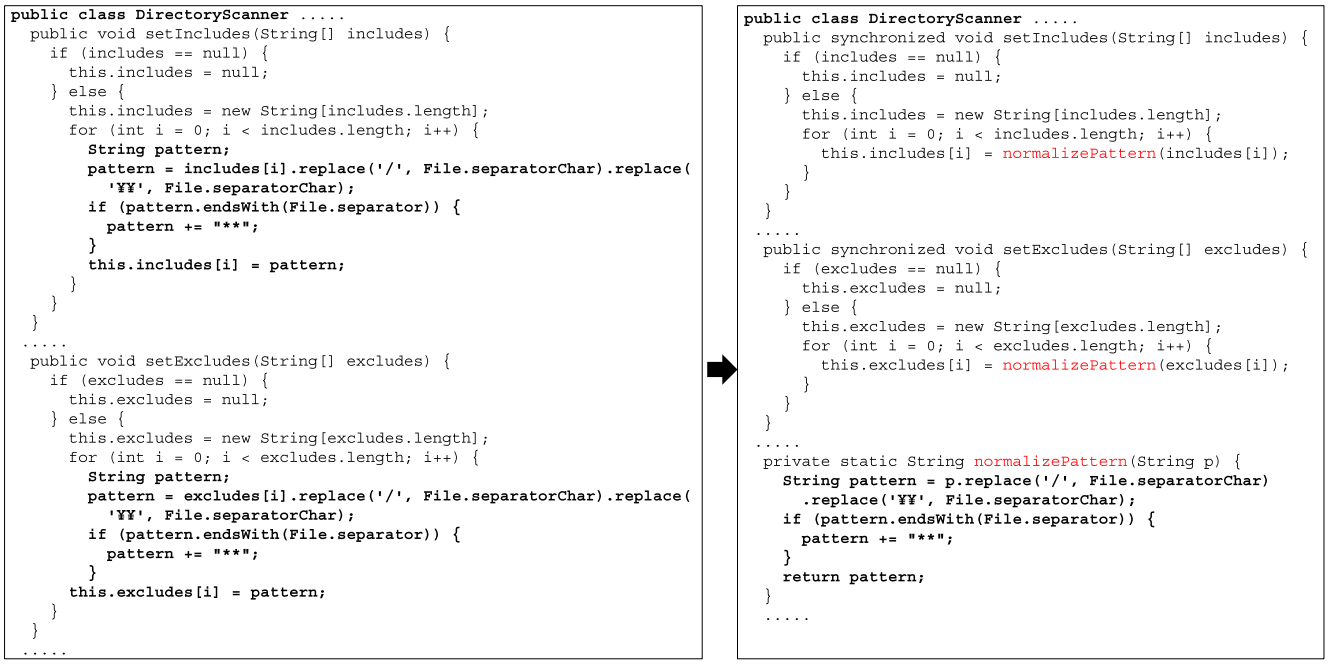
The table reveals that a total of 35 sets of merged code clones were identified from the three software systems under investigation. Surprisingly, only four types of clone refactoring (*EM*, *ES*, *FTM*, and *RMMO* patterns) were found, while there were no detected instances of *EC*, *PM*, and *PUM* patterns.

Figure 3 depicts examples of found pairs of merged code clones in the subject systems. Figure 3 (a) shows an example of clone refactoring using *EM* pattern in Apache Ant between release 1.6.2 and 1.6.3. In this figure, release

Table 2 The number of sets of merged code clones, and the number of pairs of merged code clones in parentheses from overall subject systems.

Refactoring pattern	EM	ES	FTM	RMMO
# Instances	11 (12)	1 (15)	1 (6)	22 (455)

[†]Our analyzed data is available at http://sel.ist.osaka-u.ac.jp/~ejchoi/refactoredclones



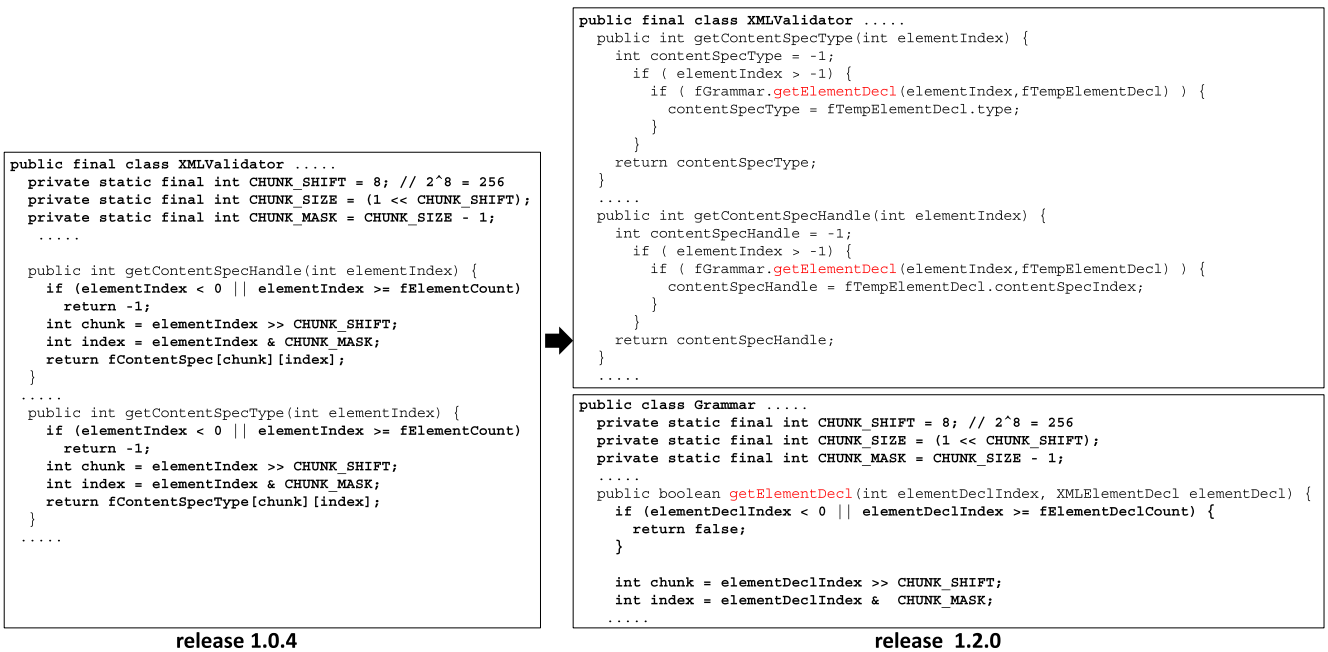
```
public class DirectoryScanner .....
  public void setIncludes(String[] includes) {
    if (includes == null) {
      this.includes = null;
    } else {
      this.includes = new String[includes.length];
      for (int i = 0; i < includes.length; i++) {
        String pattern;
        pattern = includes[i].replace('/', File.separatorChar).replace(
          '¥¥', File.separatorChar);
        if (pattern.endsWith(File.separator)) {
          pattern += "**";
        }
        this.includes[i] = pattern;
      }
    }
  }
 .....
  public void setExcludes(String[] excludes) {
    if (excludes == null) {
      this.excludes = null;
    } else {
      this.excludes = new String[excludes.length];
      for (int i = 0; i < excludes.length; i++) {
        String pattern;
        pattern = excludes[i].replace('/', File.separatorChar).replace(
          '¥¥', File.separatorChar);
        if (pattern.endsWith(File.separator)) {
          pattern += "**";
        }
        this.excludes[i] = pattern;
    }
  }
 .....
```

release 1.6.2

```
public class DirectoryScanner .....
  public synchronized void setIncludes(String[] includes) {
    if (includes == null) {
      this.includes = null;
    } else {
      this.includes = new String[includes.length];
      for (int i = 0; i < includes.length; i++) {
        this.includes[i] = normalizePattern(includes[i]);
      }
    }
  }
 .....
  public synchronized void setExcludes(String[] excludes) {
    if (excludes == null) {
      this.excludes = null;
    } else {
      this.excludes = new String[excludes.length];
      for (int i = 0; i < excludes.length; i++) {
        this.excludes[i] = normalizePattern(excludes[i]);
      }
    }
  }
 .....
  private static String normalizePattern(String p) {
    String pattern = p.replace('/', File.separatorChar)
      .replace('¥¥', File.separatorChar);
    if (pattern.endsWith(File.separator)) {
      pattern += "**";
    }
    return pattern;
  }
  .....
```

release 1.6.3

(a) Example of clone refactoring using *EM* pattern in Apache Ant: Code clones (shown in bold) exist in two separate methods (setUncludes and setExclues) in a class named org.apache.tools.ant.DirectoryScanner are extracted as a new method named normalizePattern (shown in red) in the same class between release 1.6.2 and 1.6.3.



```
public final class XMLValidator .....
  private static final int CHUNK_SHIFT = 8; // 2^8 = 256
  private static final int CHUNK_SIZE = (1 << CHUNK_SHIFT);
  private static final int CHUNK_MASK = CHUNK_SIZE - 1;
   .....

  public int getContentSpecHandle(int elementIndex) {
    if (elementIndex < 0 || elementIndex >= fElementCount)
      return -1;
    int chunk = elementIndex >> CHUNK_SHIFT;
    int index = elementIndex & CHUNK_MASK;
    return fContentSpec[chunk][index];
  }
 .....
  public int getContentSpecType(int elementIndex) {
    if (elementIndex < 0 || elementIndex >= fElementCount)
      return -1;
    int chunk = elementIndex >> CHUNK_SHIFT;
    int index = elementIndex & CHUNK_MASK;
    return fContentSpecType[chunk][index];
  }
 .....
```

release 1.0.4

```
public final class XMLValidator .....
  public int getContentSpecType(int elementIndex) {
    int contentSpecType = -1;
      if ( elementIndex > -1) {
        if ( fGrammar.getElementDecl(elementIndex,fTempElementDecl) ) {
          contentSpecType = fTempElementDecl.type;
        }
      }
    return contentSpecType;
  }
  .....
  public int getContentSpecHandle(int elementIndex) {
    int contentSpecHandle = -1;
      if ( elementIndex > -1) {
        if ( fGrammar.getElementDecl(elementIndex,fTempElementDecl) ) {
          contentSpecHandle = fTempElementDecl.contentSpecIndex;
        }
      }
    return contentSpecHandle;
  }
  .....
```

```
public class Grammar .....
  private static final int CHUNK_SHIFT = 8; // 2^8 = 256
  private static final int CHUNK_SIZE = (1 << CHUNK_SHIFT);
  private static final int CHUNK_MASK = CHUNK_SIZE - 1;
  .....
  public boolean getElementDecl(int elementDeclIndex, XMLElementDecl elementDecl) {
    if (elementDeclIndex < 0 || elementDeclIndex >= fElementDeclCount) {
      return false;
    }

    int chunk = elementDeclIndex >> CHUNK_SHIFT;
    int index = elementDeclIndex &  CHUNK_MASK;
   .....
```

release 1.2.0

(b) Example of clone refactoring using *RMMO* pattern in Xerces-j: Code clones (shown in bold) exist in two separate methods (getContentSpecType and getContentSpecHandle) in a org.apache.xerces.validators.common.XMLValidator class. They are extracted as a new method named getElementDecl (shown in red) in a new class named org.apache.xerces.validators.common.Grammar between release 1.0.4 and 1.2.0. Moreover, used variables (shown in bold) in the code clones were also moved into the Grammar class.

Fig. 3 Example of clone refactoring in open source software systems: We changed layouts of these examples to save space.

1.6.2 includes pairs of code clones (shown in bold) existing in two separate methods (setIncludes and setExcludes) in a class named org.apache.tools.ant.DirectoryScanner. In release 1.6.3, these code clones are extracted from the two methods to create a new method named normalizePattern, which shown in red, and the old statements are replaced by caller statements to the new method. Figure 3 (b) shows an example of clone refactoring using *RMMO* pattern in Xerces-j between release 1.0.4 and 1.2.0. In release 1.0.4, two cloned methods (getContentSpecHandle

and getContentSpecType), shown in bold, use local variables. In release 1.0.6, these code clones are extracted to a new method named getElementDecl, shown in red, of a new class (org.apache.xerces.validators.common.Grammar), and all the local variables are also moved into fields of the Grammar class.

Taking a closer look at the table, 11 sets, and 12 pairs of merged code clones across eight releases from three software systems were identified as having been refactored by *EM* pattern. In most cases, we found that *EM* pattern was only used to merge one pair of code clones, meaning that it was frequently used to merge small sets of code clones.

In examining instances of *ES* and *FTM* patterns, we uncovered that they were only found in one release of ArgoUML for each pattern. *ES* and *FTM* pattern was used to merge 15 and 6 pairs of code clones, respectively. To summarize, *ES* and *FTM* patterns were rarely used for clone refactoring, but were used to merge a large number of code clones.

In contrast, 22 sets, and 455 pairs of merged code clones were refactored using *RMMO* pattern across 10 releases in two software systems, ArgoUML, and Xerces-J. In particular, *RMMO* pattern was most commonly used in release 0.26 of ArgoUML. In it, 34 pairs of coded methods named initWizard, which were distributed across 11 classes, were merged into a single getToDoItem method in the org.argouml.cognitive.critics.Wizard class. Furthermore, 142 pairs of coded methods named doIt, getChoices, and getSelected located in 17 classes were also merged into a single getTarget method in a class named org.argouml.uml.ui.AbstractActionAddModelElement2. An additional 245 pairs of coded methods in 23 classes named stillValid were merged into a method named isActive in the org.argouml.cognitive.Critic class. We observed that *RMMO* pattern was used to merge sets of code clones of various sizes.

***RMMO* was the most frequently used refactoring pattern observed, followed by *EM* pattern. Conversely, *ES* and *FTM* patterns were used the least.**

How similar are the token sequences between pairs of merged code clones? (RQ2)

The results of **RQ2** can be seen in the box-plots of Fig. 4. We observed that U_{av} had the same distribution as U_{mi}, and U_{mx} for *EM*, *ES*, and *FTM* patterns. This was caused by the fact that sets of merged code clones were mainly comprised of a pair of code clones (Especially with *EM* pattern), or only one set of merged code clones was identified (Seen with *ES* and *FTM* patterns). Figure 4 (a) shows the distribution of U_{av} for *EM*, *ES*, *FTM*, and *RMMO* patterns. The distributions of U_{mi}, U_{av}, and U_{mx} are different for *RMMO* pattern, as shown in Fig. 4 (b). In these figures, the vertical axis represents the *usim* value, which start from 65%, because this is the minimum value we used to define a pair of merged code clones (See Sect. 3.2.2).

Figure 4 (a) shows that the token similarities of pairs of merged code clones refactored by *EM* and *RMMO* patterns were relatively low compared to that of *ES* and *FTM* patterns. We believe that this was caused by the fact that *ES* and *FTM* patterns merge pairs of code clones from subclasses into the same superclass. On the other hand, *EM* and *RMMO* patterns merge pairs of code clones into the same new method within the same class or another class.

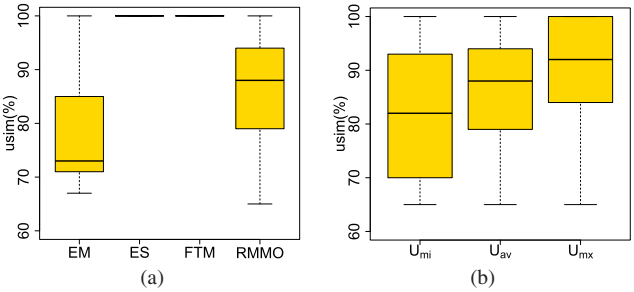


Fig. 4 Box plots of U_{av} for EM, ES, FTM, and RMMO patterns (a), and of U_{mi}, U_{av}, and U_{mx} for RMMO pattern (b).

We discovered that *EM* and *RMMO* patterns were used to merge pairs of code clones of various token similarities. However, these two patterns were mainly used to merge pairs of relatively dissimilar code clones. *EM* pattern (median 73%) was used to merge pairs of code clones that were less similar than the pairs refactored by *RMMO* pattern (median 88%). This implies that *EM* pattern was used to merge pairs of code clones with fewer similarities. Largely dissimilar pairs of code clones refactored by *EM* pattern were consistently observed across all three software systems. Compared to *EM* pattern, pairs of code clones merged using *RMMO* pattern shared more similarities with one another. (Median of the U_{mi}, U_{av}, and U_{mx} was approximately 88%). Similar results were also obtained in ArgoUML and Xerces-J.

***EM* and *RMMO* patterns were mainly used to merge pairs of code clones of various token similarities. Conversely, *ES* and *FTM* patterns were used mainly to merge highly similar pairs of code clones.**

How different are the lengths of token sequences between pairs of merged code clones? (RQ3)

The results of **RQ3** were also analyzed via box-plots. We observed that the distribution of L_{av} was the same as those of L_{mi}, and L_{mx} for *EM*, *ES*, and *FTM* patterns. As mentioned above, this was largely due to the fact that sets of merged code clones primarily comprised a pair of code clones (*EM* pattern), or only one set of merged code clones was identified (*ES* and *FTM* patterns). Figure 5 (a) shows the distribution of L_{av} for *EM*, *ES*, *FTM*, and *RMMO* patterns. Further, the distribution of L_{mi}, L_{av}, and L_{mx} for *RMMO* pattern, which can be seen in Fig. 5 (b), also differed. In the figures, the vertical axis represents differences in token lengths between pairs of merged code clones.

We discovered that differences in token lengths between pairs of merged code clones varied more for *EM* and *RMMO* patterns than for *ES* and *FTM* patterns. Even though the differences in token lengths between merged code clones

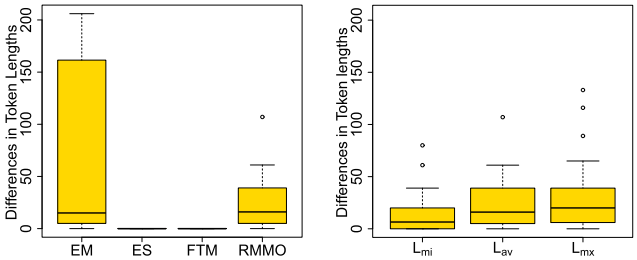


Fig. 5 Bot plots of L_{av} for EM, ES, FTM, and RMMO patterns (a), and L_{mi}, L_{av}, and L_{mx} for RMMO pattern (b).

Table 3 The number of pairs of code clones and percentage share categorized by class distance.

Class distance	# of pairs of code clones	Percentage (%)
Same Class	13	3
Same Package	324	71
Different Packages	118	28

refactored by *EM* pattern varied, this pattern was mainly used to merge pairs with relatively small differences in token lengths (median 15). Only small differences in token lengths between pairs of merged code clones were found in Apache Ant and ArgoUML. Conversely, the differences in token lengths varied relatively widely in Xerces-J. *RMMO* pattern was also mainly used to merge pairs of code clones with similar differences in token lengths (median = 16). Similar results were obtained from ArgoUML and Xerces-J.

> ***RMMO* and *EM* patterns were used to merge pairs of code clones with tokens of varying lengths. In contrast, there was no difference in length in the token sequences of pairs of code clones performed refactoring by *ES* and *FTM* patterns.**

How far are pairs of code clones located before clone refactoring? (RQ4)

In response to **RQ4**, Table 3 shows the three different class distance categories identified in instances of *RMMO* pattern, along with the number of pairs of code clones (in the *# of pairs of code clones* Column) and the percentage value (in the *Percentage* Column) of that category. Table 3 shows that pairs of code clones within the same Java package were the most prevalent, followed by pairs of code clones in different packages and in the same class

> **Pairs of code clones in the same Java package were the most prevalent, followed by pairs in different packages and in the same class.**

4.2 Suggestions for Clone Refactoring Tools

This section details our suggestions for developing clone refactoring tools based on the answers to our RQs. The suggestions are as follows:

- It is vital for tools to support *RMMO* and *EM* patterns, as evidenced in the answer to RQ1.
- To support *RMMO* pattern, tools should suggest the following code clones as candidates for clone refactoring.
  - Pairs of code clones of various token similarities, as shown in the response to RQ2
  - Pairs of code clones with various differences in token size, as shown in the response to RQ3
  - Pairs of code clones that are distributed in the same Java package, as shown in the response to RQ4
- To support *EM* pattern, tools should suggest pairs of code clones with various token similarities as candidates for clone refactoring, as shown in the result of RQ2. Moreover, code clone candidates with different token lengths should also be suggested on the basis of the results of RQ3.

Our findings provide evidence that how (RQ1) and which code clones (RQ2, RQ3, and RQ4) were refactored by *RMMO* and *EM* patterns. These findings can be utilized when tools suggest candidates for *RMMO* or *EM* pattern. Figure 6 shows an overview of a tool that we suggest for supporting *EM* pattern based the findings. As shown in this figure, when a developer extracts a code clone as a new method, the tool detects developer's behavior to perform clone refactoring in the background and then suggests candidates for clone refactoring to a developer. In the suggestion, the tool suggests candidates according to the results of RQ2 and RQ3, code clones with various token and/or different token lengths. On the other hand, for supporting *RMMO* pattern, when a developer extracts a code clone as a new method in the newly created class, the tool detects developer's behavior to perform clone refactoring in the background and then suggests candidates for clone refactoring. In the suggestion, the tool suggests code clones with various token similarities and/or different token lengths in the same Java package according to the results of RQ2, RQ3, and RQ4.

As future challenge, a system for ranking code clones is needed for the efficient suggestion of them. The further investigation of OSS version archives should be conducted to discover the characteristic of code clones should be highly prioritized. Also, a study should be done on *how does the suggested tool actively detect a developer's behavior to perform clone refactoring*. This study is necessary because the suggested tool in the previous paragraph needs to detect developer's behavior to perform clone refactoring. In the case of non-clone refactoring, active detection of refactoring has been already realized by Foster et al. [22]. Their tool named WitchDoctor observes developer's programming activities in a background process in order to detect the beginning of refactoring on the fly. Once the beginning of refactoring is detected, it suggests code transformations to complete it. By extending WitchDoctor, we plan to realize active detection of clone refactoring in order to develop the tool suggested in the previous paragraph. Moreover, after developing the

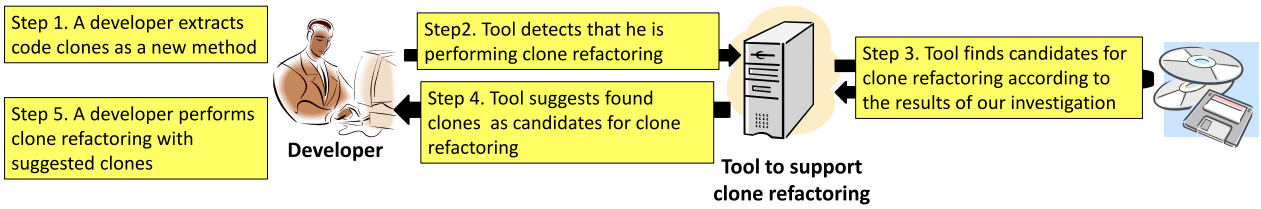


Fig. 6 Overview of a tool that we suggest for supporting EM pattern based the findings.

suggested tool, validation should be done on whether the suggested tool accurately detects clone refactoring.

4.3 Threats to Validity

This investigation had three limitations.

The first limitation was that the investigation results might have been too dependent on the output of the Ref-Finder and the *usim* since our investigation was based on the data from these two process. However, the output of both was validated in [11], [18] respectively. Moreover, we manually validated the results of the Ref-Finder based on Bavota's study [13] to improve the accuracy of our investigation on clone refactoring. Therefore, we believe that the results of investigation in this study are reliable.

The second limitation was that because we used 10 tokens as minimal token length parameter and 65% *usim* value to identify instances of clone refactoring, we might miss real instances of clone refactoring. However, we believe that results of investigation are reliable because these parameters were validated in Mende's study with best compromise of recall and precision [18]. We also believe that missed small-scale instances are trivial to software maintenances.

The final limitation was that because our case study was conducted on three open source software systems. Investigating different systems could have led to different results. However, we believe that our investigation results can be generalized an applied to other open source software systems because they spanned 63 release versions from three separate systems.

5. Related Work

As we mentioned in Sect. 1, Murphy-Hill et al. also investigated instances of refactoring in open source software systems [9]. However, they focused on more typical refactoring patterns (e.g., Extract method and Rename method) as opposed to clone refactoring. Furthermore, their investigation was based on manual, and automatic observations of developer behavior using Eclipse Usage Collector, Mylyn, and commit logs in version control systems. On the other hand, our investigation was based on semi-automatic observations of code evolution.

Numerous techniques have been developed to detect instances of refactoring in version archives [11], [23]. In this study, we used Ref-Finder to detect instances of non-clone refactoring (e.g., rename method, extract method) before manually filtering for instances of clone refactoring. Previous studies on refactoring detection have mainly focused on the prevalence of different types of refactoring, and the precision/recall of detection. We instead focused our investigation on how developers performed clone refactoring using one of the state-of-the-art refactoring detection tool named Ref-Finder.

Our earlier workshop paper [24] presented a preliminary investigation into instances of clone refactoring in the version archives of open source software systems. For this journal paper, we manually verified the output of Ref-Finder to improve the accuracy of our investigation. We accomplished this by using the datasets produced by Ref-Finder that were validated by two master course students at the University of Salerno, in Bavota's study [13][†]. We also extended the descriptions of our introduction and related works sections.

6. Conclusion and Future Work

In this paper, we presented an investigation into instances of clone refactoring identified in three open source software systems to provide insights into the development of clone refactoring tools that could be more widely used in industry. In the investigation, we detected instances of refactoring from consecutive program versions of software systems using Ref-finder. Next, we identified instance of clone refactoring using undirected similarity. To improve the accuracy of our investigation, we manually validated the instances of clone refactoring and the statistics of pairs of merged code clones to answer our RQs.

From the investigation results, we found that it would be vital for clone refactoring tools to support *RMMO* and *EM* patterns. Such tools should also suggest pairs of code clones with varying tokens in the same Java package as candidates for *RMMO* pattern. In addition, suggested pairs of code clones candidates should be differences in token lengths. To support *EM* pattern, pairs of code clones with varying levels of similarities should be suggested as candidates for clone refactoring.

For future work, we plan on investigating additional open source software systems and industrial software systems. We also plan to carry out the further studies on *how does the suggested tool actively detect a developer's behavior to perform clone refactoring* and on the development of

[†]The validated datasets are available at http://www.distat.unimol.it/reports/refactoring-defect/.

a system for ranking code clones as discussed in Sect. 4.2, in order to realize efficient support of clone refactoring. We would also like to develop tools for clone refactoring in accordance with these results.

Acknowledgments

This work was supported by JSPS KAKENHI 2S220003.

References

[1] C.K. Roy, J.R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," Sci. Comput. Program., vol.74, no.7, pp.470–495, 2009.
[2] Y. Yamanaka, E. Choi, N. Yoshida, K. Inoue, and T. Sano, "Applying clone change notification system into an industrial development process," Proc. ICPC, pp.199–206, 2013.
[3] 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., vol.28, no.7, pp.654–670, 2002.
[4] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and accurate tree-based detection of code clones," Proc. ICSE, pp.96–105, 2007.
[5] 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., vol.32, no.3, pp.176–192, 2006.
[6] N. Göde and J. Harder, "Oops! ... I changed it again," Proc. IWSC, pp.14–20, 2011.
[7] R. Tairas and J. Gray, "Increasing clone maintenance support by unifying clone detection and refactoring activities," Inf. Softw. Technol., vol.54, no.12, pp.1297–1307, 2012.
[8] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, "Aries: Refactoring support environment based on code clone analysis," Proc. IASTED SEA, pp.222–229, 2004.
[9] E. Murphy-Hill, C. Parnin, and A.P. Black, "How we refactor and how we know it," IEEE Trans. Softw. Eng., vol.38, no.1, pp.5–18, 2012.
[10] M. Fowler, Refactoring: Improving the design of existing code, Addison Wesley, 1999.
[11] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim, "Template-based reconstruction of complex refactorings," Proc. ICSM, 2010.
[12] J. Henkel, "CatchUp!: Capturing and replaying refactorings to support API evolution," Proc. ICSE, pp.274–283, 2005.
[13] G. Bavota, B.D. Carluccio, A.D. Lucia, M.D. Penta, R. Oliveto, and O. Strollo, "When does a refactoring induce bugs? An empirical study," Proc. SCAM, pp.104–113, 2012.
[14] P. Weißgerber and S. Diehl, "Identifying refactorings from source-code changes," Proc. ASE, pp.231–240, 2006.
[15] S. Hayashi, Y. Tsuda, and M. Saeki, "Search-based refactoring detection from source code revisions," IEICE Trans. Inf. & Syst., vol.E93-D, no.4, pp.754–762, April 2010.
[16] H. Kim, Y. Jung, S. Kim, and K. Yi, "MeCC: Memory comparison-based clone detector," Proc. ICSE, pp.301–310, 2011.
[17] G. Soares, R. Gheyi, E. Murphy-Hill, and B. Johnson, "Comparing approaches to analyze refactoring activity on software repositories," J. Syst. Softw., vol.86, no.4, pp.1006–1022, 2013.
[18] T. Mende, R. Koschke, and F. Beckwermert, "An evaluation of code similarity identification for the grow-and-prune model," J. Softw. Maint. Evol., vol.21, no.2, pp.143–169, 2009.
[19] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," IEEE Trans. Softw. Eng., vol.33, no.9, pp.577–591, 2007.
[20] V. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," Soviet Physics Doklady, vol.10, no.8, pp.707–710, 1966.
[21] R. Baeza-Yates and B. Ribeiro-Neto, Modern Information Retrieval: The Concepts and Technology behind Search, 2nd ed., Addison Wesley, 2011.
[22] S.R. Foster, W.G. Griswold, and S. Lerner, "WitchDoctor: IDE support for real-time auto-completion of refactorings," Proc. ICSE, pp.222–232, 2012.
[23] Z. Xing and E. Stroulia, "Refactoring detection based on umldiff change-facts queries," Proc. WCRE, pp.263–274, 2006.
[24] E. Choi, N. Yoshida, and K. Inoue, "What kind of and how clones are refactored?: A case study of three OSS projects," Proc. WRT, pp.1–7, 2012.

Eunjong Choi received her Master from Osaka University in 2011. She is a Ph.D. candidate at Osaka University since 2012. Her research interests include software maintenance, code clone analysis, refactoring, and defect detection. She is a member of the ACM.

Norihiro Yoshida received his B.E. from the Kyushu Institute of Technology in 2004 and his Master and Ph.D. from Osaka University in 2006 and 2009, respectively. He is an assistant professor at the Nara Institute of Science and Technology since 2010. His research interests include program analysis and software development environment. He is a member of the IEEE, the IEEE Computer Society, and the ACM.

Katsuro Inoue received the B.E., M.E., and D.E. degrees in information and computer sciences from Osaka University, Japan, in 1979, 1981, and 1984, respectively. He was an assistant professor at the University of Hawaii at Manoa from 1984–1986. He was a research associate at Osaka University from 1984–1989, an assistant professor from 1989–1995, and a professor beginning in 1995. His interests are in various topics of software engineering such as software process modeling, program analysis, and software development environment. He is a member of the IEEE, the IEEE Computer Society, and the ACM.