Identifying Clone Removal Opportunities Based on Co-evolution Analysis

Yoshiki Higo, Shinji Kusumoto
Graduate School of Information Science and Technology, Osaka University, JAPAN
{higo,kusumoto}@ist.osaka-u.ac.jp

ABSTRACT

Previous research efforts have proposed various techniques for supporting code clone removal. They identify removal candidates based on the states of the source code in the latest version. However, those techniques suggest many code clones that are not suited for removal in addition to appropriate candidates. That is because the presence of code clones do not necessarily motivate developers to remove them if they are stable and do not require simultaneous modifications. In this paper, we propose a new technique that identifies removal candidates based on past records of code modifications. By using the proposed technique, we can identify code clones that actually required simultaneous modifications in the past.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*

General Terms

Design, Measurement

Keywords

Code clone, Co-evolution analysis, Refactoring

1. INTRODUCTION

The presence of code clones (in short, *clones*) is regarded as one of the factors that makes it more difficult to keep source code consistent [11]. If a clone is modified, we must consider to apply the same modifications to its correspondences simultaneously. Higo and Kusumoto revealed that delay propagation often happen in clones [5]. Clone is well-known as targets of refactoring [4].

On the other hand, several research efforts reported that removing clones are not necessarily good for source code. For example, Kim et al. found the following [10].

- Some clones exist only for a short period. They evolve differently after they have become unduplicated.
- If clones exist for a long period, it should be difficult to merge them as a single module because of limitations of programming languages.

Kapser and Godfrey reported that clones can be a reasonable design decision based on an empirical study of open source systems [8]. They derived several patterns of clones, and they discussed the pros and cons of clones based on the patterns. Bettenburg et al. reported that the presence of clones does not have much a negative impact on software quality [2]. They investigated inconsistent changes to clones at release level on open source systems. In the empirical study, only 1.26% to 3.23% of inconsistent changes introduced software errors into the target systems.

Consequently, it is obvious that not all clones need to be removed. However, some clones need to be considered as candidates for simultaneous modifications or for refactoring to keep source code consistent easily. In this paper, we propose a new technique to identify clones that should be removed. The proposed technique considers past modifications on clones.

If every clone in a clone group was modified in the same way even if the modifications were not simultaneous, the proposed technique suggest the clone group as a refactoring candidate. Even if there are clones that are long or spread into many files, they are not recommended as refactoring candidates unless they were modified in the same way. The authors think that the same modifications in the past can be a sufficient motivation for removing clones.

The main contributions of this paper are as follows.

- The proposed technique has a new feature that considers past modifications for identifying clone removal candidates. The source code analysis (repository mining) for the identification is rapid. We can obtain candidates within a short time frame even if there are several thousand or more revisions in the repository of the target software.
- We confirmed that most of the identified candidates actually had been removed in subsequent revisions, or they seemed to be reasonable candidates based on our manual investigation on their source code.

The remainder of this paper is organized as follows: Section 2 introduces a motivating example of this research; Section 3 explains the proposed technique; Section 4 shows an experimental result on open source software; in Section 5, we introduce some research related to this work; lastly, Section 6 concludes this paper.

2. MOTIVATING EXAMPLE

Figure 1 shows an actual modification on open source software. In this modification, the 116th and 120th lines were changed. The

IWPSE'13, August 19-20, 2013, Saint Petersburg, Russia
Copyright 2013 ACM 978-1-4503-2311-6/13/08...$15.00
http://dx.doi.org/10.1145/2501543.2501552

```
115  if (org.argouml.model.ModelFacade.isAInstance(target)) {
116     MInstance inst = (MInstance) target;                    DELETED
117 //  ((MInstance) target).setClassifier((MClassifier) element);
118
119     // delete all classifiers
120     Collection col = inst.getClassifiers();                 DELETED
121     if (col != null) {
122  Iterator iter = col.iterator();
```

(a) Before modification

```
115   if (org.argouml.model.ModelFacade.isAInstance(target)) {
116     Object inst = /*(MInstance)*/ target;                   ADDED
117 //  ((MInstance) target).setClassifier((MClassifier) element);
118
119     // delete all classifiers
120     Collection col = ModelFacade.getClassifiers(inst);      ADDED
121     if (col != null) {
122  Iterator iter = col.iterator();
```

(b) After modification

Figure 1: Actual modification on OSS. The same modifications were performed on three files simultaneously

same modifications were performed on three source files simultaneously. This is evidence that clones require the same modifications, which makes it more difficult to keep source code consistent. If developer had forgotten to modify one of the clones, an unintended inconsistency would have occurred. Such inconsistencies sometimes become bugs [5].

Authors think that refactoring is a reasonable design choice for clones that actually required the same modifications in the past. If such clones are merged as a single module like a method or class, unintended inconsistencies never occur in the code even if the code requires modifications repeatedly in the future.

Consequently, in this paper, we propose a new technique identifying clones that were modified in the same way in the past. Of course, we can obtain clones for removal with conventional techniques. However, there is a big difference between the proposed technique and conventional ones.

- Conventional techniques identify clones in the latest version as refactoring candidates.
- The proposed technique identifies clones that required the same modifications in the past as refactoring candidates.

That is, the proposed technique considers modification records. The fact that a given group of clones required the same modifications in the past will motivate developers to remove it.

A simple way to obtain both the code before and after modification is using the Unix diff command. However, the diff command is not sufficient because it does not consider structures of the source code. For example, in the case of Figure 1, two lines of code were modified: however, they are not consecutive ones. If we apply the diff to the code, diff outputs two different modifications. On the other hand, the proposed technique outputs a single modification in this case because it ignores the presence of comment lines.

3. PROPOSED TECHNIQUE

3.1 Modification Pattern

In this paper, we use two terms **Modification Instance** (in short, **MI**) and **Modification Pattern** (in short, **MP**). An MI is an instance of source code modification and an MP is a pattern that a code fragment was changed to another code fragment. MPs hold both code fragments before and after the modifications. In this research, code fragments are represented by sequences of tokens.

MPs also have the following metric values.

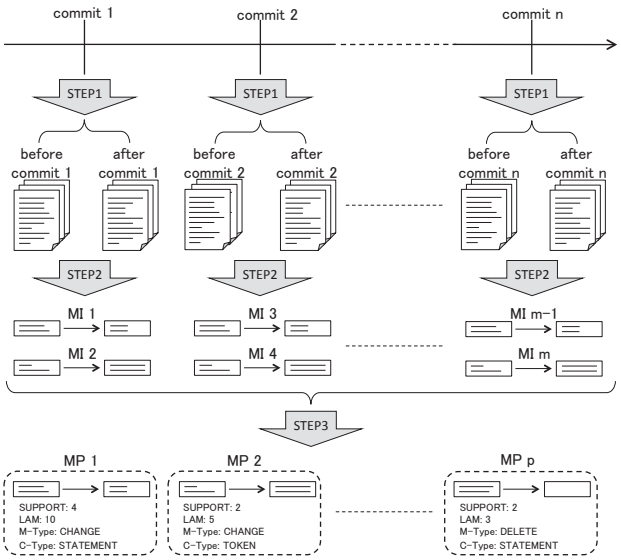

Figure 2: Overview of the proposed technique

SUPPORT the number of occurrences of a given MP.

LAM (Length After Modification) the number of statements in code fragments after the modification. Please refer to the description of STEP2B in Subsection 3.2 for the definition of statement in the proposed technique.

M-TYPE (Modification-TYPE) If the length of both the before and after modification of a given MP are not 0, its type becomes **CHANGE**. If the length before modification is 0, its type becomes **ADD**. If the length after modification is 0, its type becomes **DELETE**.

C-Type (Change-TYPE) if a given MP is a token-level modification, it becomes **TOKEN**. If not, it becomes **STATEMENT**.

Those metrics are used for extracting removal candidates from MPs, not for providing insights on the refactoring that could remove clones.

3.2 Deriving Modification Patterns

Here, we explain how we derive MPs based on co-evolution analysis. The input and output of the proposed technique are as follows:

INPUT software repository of a target software system, and

OUTPUT clones as refactoring candidates.

Deriving MPs consists of the following steps.

STEP1 a list of source files modified is identified for every revision. Then, the contents before and after the modification for each source file are retrieved.

STEP2 MIs are extracted by comparing the contents before and after the modification.

STEP3 MPs are derived from the set of MIs.

In the remainder of this section, we explain every step in detail. In this explanation, we assume that R is an input repository.

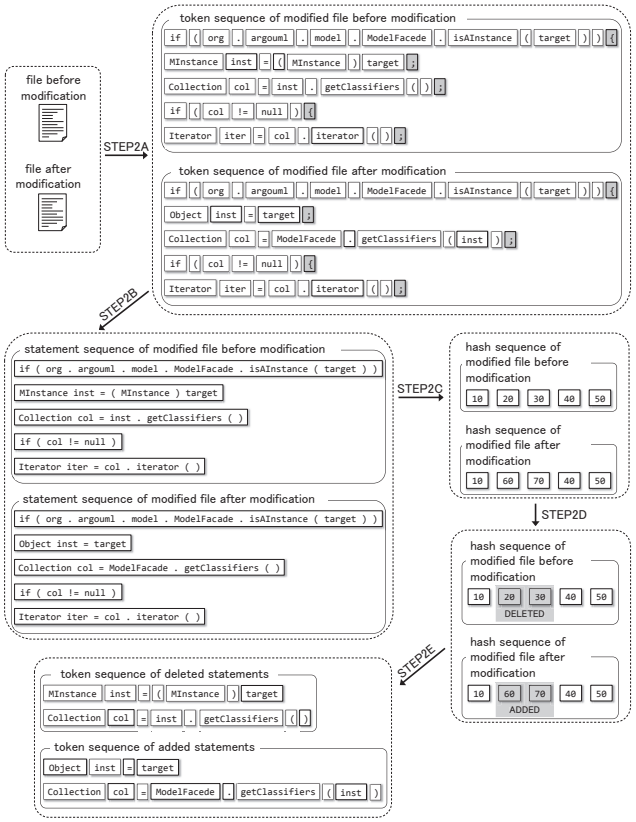

Figure 3: Details of STEP2

STEP1: Retrieving Source Code

We assume that $C(R)$ is the whole set of commits included in R. If the number of elements in $C(R)$ is n, $C(R)$ can be represented with the following formula.

$$C(R) = \{c_1, c_2, \cdots, c_n\} \quad (1)$$

We assume that $S(c_k)$ is a set of source files modified in commit c_k. If s files are modified in c_k, $S(c_k)$ can be represented with the following formula.

$$S(c_k) = \{f_1, f_2, \cdots, f_s\} \quad (2)$$

Note that $S(c_k)$ does not include source files added or deleted in c_k. it includes only changed files. The output of STEP1 is $S(c_k)(\forall c_k \in C(R))$.

STEP2: Extracting Modification Instances

This step extracts MIs by comparing source code before and after modification, which are included in $S(c_k)$. This step consists of the following sub-steps. Figure 3 shows how each sub-step works.

STEP2A token sequences before and after modification are generated by performing lexical analysis.

STEP2B statements are identified in the token sequences. Herein, we define that a statement is a token sequence among semicolon (“;”), open bracket (“{”), and close bracket (“}”). The three kinds of tokens are not included in statements.

Table 1: Repository information for ArgoUML

oldest revision (date)	1 (1998-01-27)
latest revision (date)	19,910 (2013-02-08)
LOC of end revision	370,152

STEP2C a hash value is generated from every statement. We obtain a hash sequence from each of statement sequences before and after modification, respectively.

STEP2D modified statements are identified by applying the Longest Common Subsequence algorithm [1] to the two hash sequences.

STEP2E identified subsequences of hash values are inversely transformed to sequences of tokens, which are the output of STEP2.

Each MI extracted in STEP2 has the following information:

- token sequences before and after the modification,
- date of the modification, and
- name of the source file.

Herein, we assume that $M(c_k)$ is a set of MIs extracted from $S(c_k)$. The output of STEP2 are $M(c_k)(\forall c_k \in C(R))$.

STEP3: Deriving Modification Patterns

In STEP3, all the MIs extracted in STEP2 are compared based on their token sequences before and after modification. If both the token sequences before and after modification of an MI are equal to token sequences of another MI. The MIs are merged as a single MP. Even if an MI is not merged with any other MIs, it is regarded as a single MP whose SUPPORT value is 1.

4. EXPERIMENT

A software tool has been developed based on the proposed technique[1]. Currently, the tool handles only source code written in Java and Subversion’s repositories. However, it is not difficult to extend it for other programming languages and other repositories. The tool uses method *java.lang.String.hashCode()* to generate hash values for statements in the source code.

We conducted a small experiment on an open source project, ArgoUML. The purpose of this experiment is to derive insights on the usefulness of the proposed technique.

Table 1 summarizes the repository for ArgoUML. ArgoUML has been developed and maintained for 15 years, and its repository includes about 20,000 revisions. The latest revision consists of about 370,000 lines of code. If we detect clones from the latest revision for identifying refactoring candidates, many refactoring candidates are detected. Thus, it is difficult to find which clones should be removed in priority to the others.

We derived 64,124 MPs from the target repository with the tool. The derivation took around 31 minutes. Then, we filtered out them by using the following conditions:

CONDITION1 lower threshold of SUPPORT is 3,

CONDITION2 lower threshold of LAM is 5,

CONDITION3 M-Type is only CHANGE, and

CONDITION4 C-Typeis only STATEMENT.

Why we used CONDITION2 is that size of clones probably affects the motivation for removing them. We think that large clones are more likely to be removed than small ones. In order to obtain MPs that existing code had been changed to another code, we used CONDITION3. CONDITION4 was used for identifying MPs that had required large modifications.

[1]https://github.com/YoshikiHigo/MPAnalyzer

```
41 public class ActionStateDiagram extends ActionAddDiagram {
     ...
82   public boolean isValidNamespace(MNamespace ns) {
83     if (ns instanceof MClassifier) return true;          DELETED
84       return false;
85   }
     ...
```

(a) Before modification (revision 3,760)

```
54 public class ActionStateDiagram extends ActionAddDiagram {
     ...
108  public boolean isValidNamespace(Object handle) {
109    if (!ModelFacade.isANamespace(handle)) {
110      cat.error("No namespace as argument");
111      cat.error(handle);
112      throw new IllegalArgumentException(
113      "The argument " + handle + "is not a namespace.");
114    }
115    MNamespace ns = (MNamespace)handle;
116    if (ns instanceof MClassifier)
117      return true;                                       ADDED
118    return false;
119  }
     ...
```

(b) After modification (revision 3,761)

Figure 4: Example of identified refactoring candidates that were actually refactored by pulling up clones to the common base class in the latest revision.

We extracted 13 MPs with the conditions[2], then we manually investigated them by browsing their source code before and after the modifications, and of the latest revision. Table 2 summarizes the investigation result.

Category **A** is a set of MPs that has already been removed in the latest revision. There were six MPs classified into **A**. Four of them were clones that were pulled up to the common base classes (**A1**), and the remaining two were extracted as new methods (**A2**).

Figure 4 shows an MP classified into **A1**. In the modification, code labeled DELETED was deleted and code labeled ADDED was added in revision 3,761. The same modifications were performed on 4 methods, each of which was defined in different classes but they have a common base class. The 4 methods have the same signature, and they have been pulled up to the common base class in the latest revision.

Figure 5 shows an MP classified into **A2**. In this modification, procedure for generating an object that variable persister points to was added. The type of the generated object depends on the type of the object that variable projectMember points to. In the latest revision, the added if-else statements were replaced with a constructor call. The SUPPORT value of this MP is 3, and all the three code were modified in the same way in the latest revision.

Category **B** is a set of MPs that were regarded as not appropriate as removal candidates. Two of them were code that had been generated by tools like compiler-compiler. The modifications were not made by human but the grammar for the generated code was changed. If we were the developers of the target software, we would know which source files had been automatically generated by tools. Consequently, in actual usages of this tool, it will not be difficult to filter out such generated code.

We found that one MP deemed to be a typical modification of Java language (see Figure 6). In the modification, an iterative procedure with Enumeration was changed to a procedure with a for-statement. In Java language, there are several ways to implement iterative procedures such as Iterator, enhanced for-statement, conventional for-statement, Enumeration, and so on. Changing code implemented using one of them to another is often performed. Such change strongly depends on Java language, and depend neither on software itself nor on its domain. Consequently, the authors did not think that the code was appropriate as a removal candidate.

[2]If we had used looser conditions, we would have extracted more MPs. The reason why we used relatively strict conditions is to extract a small number of MPs for manual investigation.

```
45 public class XmiFilePersister extends AbstractFilePersister {
     ...
77   public void doSave(Project project, File file)
78       throws SaveException {
       ...
107    if (projectMember.getType().equalsIgnoreCase("xmi")) {
108      if (LOG.isInfoEnabled()) {
109        LOG.info("Saving member of type: "
110              + ((ProjectMember) project.getMembers()
111                    .get(i)).getType());
112      }
113      projectMember.save(writer, null);                  DELETED
144    }
       ...
```

(a) Before modification (revision 7,436)

```
47 public class XmiFilePersister extends AbstractFilePersister {
     ...
79   public void doSave(Project project, File file)
80       throws SaveException {
       ...
109    if (projectMember.getType().equalsIgnoreCase("xmi")) {
110      if (LOG.isInfoEnabled()) {
111        LOG.info("Saving member of type: "
112              + ((ProjectMember) project.getMembers()
113                    .get(i)).getType());
114      }
115      MemberFilePersister persister = null;
116      if (projectMember instanceof ProjectMemberDiagram) {
117        persister = new DiagramMemberFilePersister();
118      } else if (projectMember instanceof ProjectMemberTodoList) {
119        persister = new TodoListMemberFilePersister();
120      } else if (projectMember instanceof ProjectMemberModel) {
121        persister = new ModelMemberFilePersister();
122      }
123      persister.save(projectMember, writer, null);       ADDED
124    }
       ...
```

(b) After modification (revision 7,437)

```
208    if (projectMember.getType().equalsIgnoreCase(getExtension())) {
209      if (LOG.isLoggable(Level.INFO)) {
210        LOG.log(Level.INFO, "Saving member of type: {0}",
211                projectMember.getType());
212      }
213      MemberFilePersister persister = new ModelMemberFilePersister();
214      persister.save(projectMember, stream);
215    }                                                    DELEGATED
```

(c) Latest code (revision 19,910)

Figure 5: Example of identified refactoring candidates that were actually refactored by extracting clones as a new constructor in the latest revision.

We found that, in the latest revision, source files including an MP had been deleted. We regarded that the remaining 3 MPs were appropriate as removal candidates. They exist in the latest revision, and it is not difficult to remove them with relatively simple operations such as *Pull Up Method* or *Extract Method*.

5. RELATED WORK

5.1 Distilling Changed Code

Kim et al. proposed a technique to summarize code changes and

Table 2: Investigation Result

	category	# of candidates
A:	already removed	6
	A1: pull up method	4
	A2: extract method	2
B:	not appropriate	4
	B1: generated code	2
	B2: typical change	1
	B3: not exist in latest revision	1
C:	appropriate	3

```
222 /** Reply the bounding box for this FigEdge. */
223 public Rectangle getBounds() {
224   Rectangle res = _fig.getBounds();
225   Enumeration enum = _pathItems.elements();
226   while (enum.hasMoreElements()) {
227     Fig f = ((PathItem) enum.nextElement()).getFig();
228     res = res.union(f.getBounds());
229   }
230   return res;
231 }
```

DELETED

(a) Before modification (revision 145)

```
240 /** Reply the bounding box for this FigEdge. */
241 public Rectangle getBounds() {
242   Rectangle res = _fig.getBounds();
243   int size = _pathItems.size();
244   for (int i = 0; i < size; i++) {
245     Fig f = ((PathItem) _pathItems.elementAt(i)).getFig();
246     res.add(f.getBounds());
247   }
248   return res;
249 }
```

ADDED

(b) After modification (revision 146)

Figure 6: The identified modification that an iteration with Enumeration **was changed to one with** for**-statement**

developed a tool, LSDiff based on their technique [9]. The technique permits us to know easily what kinds of changes were performed on APIs or program elements such as method invocations. Their technique abstracts code changes because its purpose is supporting the understanding of code changes. On the other hand, our technique is intended to identify refactoring candidates. It does not abstract code changes: however it extracts code changes from token sequences generated from the source code not from the source code itself in order to identify appropriate refactoring candidates.

Fluri et al. proposed a change distilling technique [3]. Their technique compares two versions of abstract syntax trees, computes tree-edit operations, and maps each tree-edit to atomic AST-level change types. AST-level comparison can distill code changes that line-level comparison cannot (e.g., reordering parameters of methods). In this research, we adopted token-level comparison to identify removal candidates because of its rapidity. However, AST-level comparison will also be useful to identify refactoring candidates.

5.2 Identifying Refactoring Candidates

Higo et al. proposed a technique to identify refactoring candidates based on clone analysis [6]. In their technique, candidates are wholly-duplicated program units such as classes, methods and blocks. Duplicated units are characterized with some metrics in order to infer how they can be refactored. For example, a metric used in their technique indicates each set of duplicated units has the a common base class or not. If so, it may not be difficult to remove them by pulling up them to the common base class.

Hotta et al. proposed using program dependency graph (in short, PDG) for identifying refactoring candidates [7]. Using PDG enables the identification of refactoring candidates even if a program unit is not wholly-duplicated with other units.

The two existing techniques identify refactoring candidates based on code analysis of the latest version. Clones are suggested to users as refactoring candidates regardless of how they evolved. On the other hand, the proposed technique identifies candidates that were actually modified in the same way during their evolution.

6. CONCLUSION

In this paper, we proposed a technique to identify refactoring opportunities based on co-evolution analysis. If clones were modified in the same way in the past, they are suggested as refactoring candidates in the proposed technique. Using past modification records is different from conventional identification techniques. We have conducted a small experiment on an open source project, and the proposed technique identified 13 candidates. We carefully investigated all the candidates and confirmed that only 4 out of them were not appropriate for refactoring. In the future, we are going to introduce more semantic analyses to extract code changes. Semantic analysis will requires more time to extract code changes: however, we will obtain more useful changes for refactoring.

7. ACKNOWLEDGEMENTS

This work was supported by MEXT/JSPS KAKENHI 25220003, 24650011, and 24680002.

8. REFERENCES

[1] L. Bergroth, H. Hakonen, and T. Raita. A survey of longest common subsequence algorithms. In *Proceedings of the 7th International Symposium on String Processing Information Retrieval*, pages 39–48, 2000.

[2] N. Bettenburg, W. Shang, W. Ibrahim, B. Adams, Y. Zou, and A. E. Hassan. An empirical study on inconsistent changes to code clones at release level. In *Proceedings of the 16th Working Conference on Reverse Engineering*, pages 85–94, 2009.

[3] B. Fluri, M. Wuersch, M. Pinzger, and H. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, 2007.

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

[5] Y. Higo and S. Kusumoto. How often do unintended inconsistencies happened? –deriving modification patterns and detecting overlooked code fragments–. In *Proceedings of the 28th International Conference on Software Maintenance*, pages 222–231, 2012.

[6] 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(6):435–461, 2008.

[7] K. Hotta, Y. Higo, and S. Kusumoto. Identifying, tailoring, and suggesting form template method refactoring opportunities with program dependence graph. In *Proceedings of the 16th European Conference on Software Maintenance and Reengineering*, pages 53–62, 2012.

[8] C. J. Kapser and M. W. Godfrey. "Cloning considered harmful" considered harmful: patterns of cloning in software. *Empirical Softw. Engg.*, 13(6):645–692, 2008.

[9] M. Kim, D. Notkin, D. Grossman, and G. Wilson Jr. Identifying and summarizing systematic code changes via rule inference. *IEEE Transactions on Software Engineering*, 39:45–62, 2013.

[10] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th International Symposium on Foundations of Software Engineering*, pages 187–196, 2005.

[11] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools : A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009.