Improving Process of Source Code Modification Focusing on Repeated Code

Ayaka Imazato[1], Yui Sasaki[1], Yoshiki Higo[1], and Shinji Kusumoto[1]

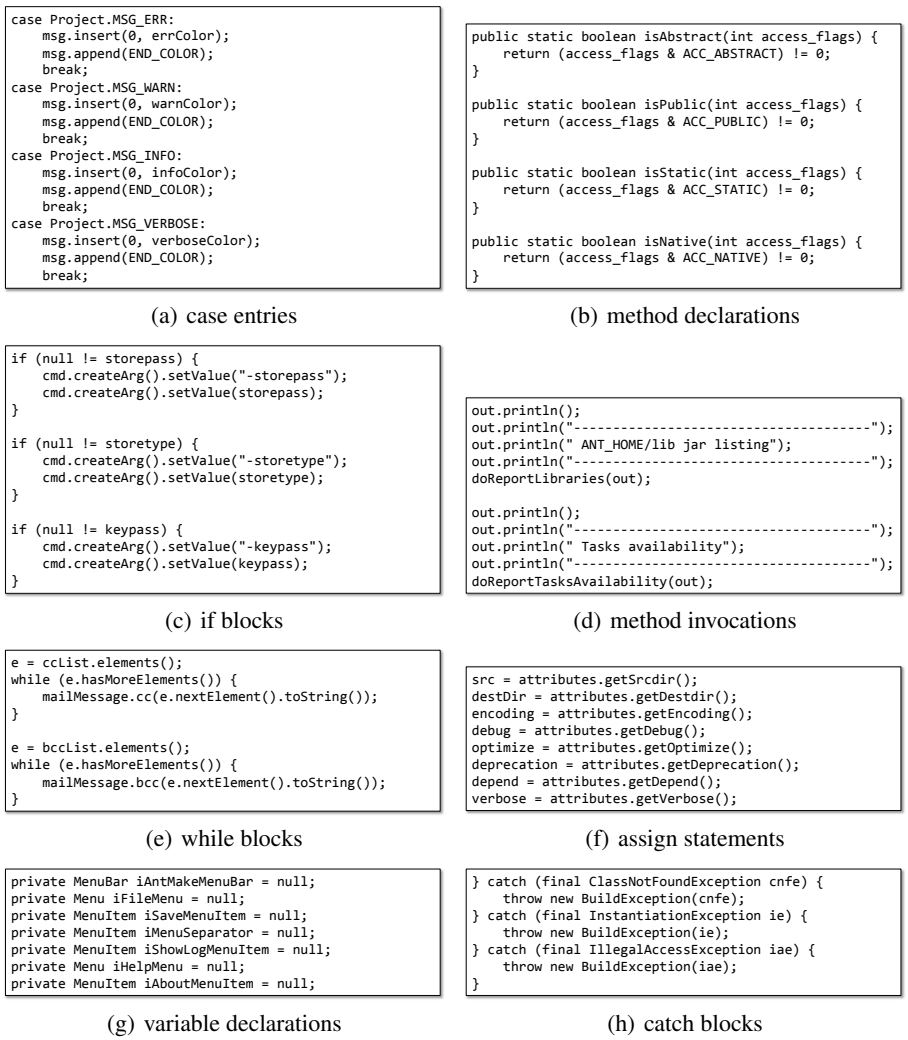
Graduate School of Information Science and Technology, Osaka University,
1-5, Yamadaoka, Suita, Osaka, Japan,
{i-ayaka, s-yui, higo, kusumoto}@ist.osaka-u.ac.jp

Abstract. There are various kinds of repeated code such as consecutive if-else statements or case entries in program source code. Such repeated code sometimes require simultaneous modifications on all of its elements. Applying the same modifications to many places on source code is a burdensome work and introduces new bugs if some places to be modified are overlooked. For these reasons, it is necessary to support modifications on repeated code. Appropriate supports for repeated code can improve process of source code modification. In this paper, as a first step for supporting modifications on repeated code, we investigate how repeated code are modified during software evolution. As a result, we revealed that, 73-89% of repeated code were modified at least once in their life and 31-58% of modifications on repeated code were simultaneous ones for all of their elements.

1 Introduction

Recent studies have revealed that a significant fraction (between 7% and 23%) of program source code has become code clones [2, 14]. A code clone is a code fragment in source code that is similar to or identical to other code fragments [15]. Code clones are introduced into source code because of various reasons such as copy-and-paste programming [10]. An advantage of copy-and-paste programming is that: we can implement necessary functions quite rapidly. However, if the copied code includes a latent bug, copy-and-paste programming unintentionally scatters the bug into its pasted places [1, 9, 11]. Moreover, code clones often require simultaneous modifications. If we overlook some code clones to be modified simultaneously, new bugs are introduced to the overlooked code fragments [7, 12]

Authors are thinking that the same problems have been occurring in repeated code. Repeated code means a list of the same instructions such as consecutive case entries or if-else statements. If an element of repeated code requires a modification, we may need to modify the other elements of the repeated code in the same way simultaneously; besides, if the number of repetition is large, manual modifications on every element of the repeated code is a burdensome and error-prone operation. Some research efforts investigated program source code and found that there are many repetitions in it [6, 13, 16]. Figure 1 shows actual repeated code in Java source code found in the investigation of literature [6]. As shown in this figure, various instructions in source code can become repeated code.



```
case Project.MSG_ERR:
    msg.insert(0, errColor);
    msg.append(END_COLOR);
    break;
case Project.MSG_WARN:
    msg.insert(0, warnColor);
    msg.append(END_COLOR);
    break;
case Project.MSG_INFO:
    msg.insert(0, infoColor);
    msg.append(END_COLOR);
    break;
case Project.MSG_VERBOSE:
    msg.insert(0, verboseColor);
    msg.append(END_COLOR);
    break;
```

(a) case entries

```
public static boolean isAbstract(int access_flags) {
    return (access_flags & ACC_ABSTRACT) != 0;
}

public static boolean isPublic(int access_flags) {
    return (access_flags & ACC_PUBLIC) != 0;
}

public static boolean isStatic(int access_flags) {
    return (access_flags & ACC_STATIC) != 0;
}

public static boolean isNative(int access_flags) {
    return (access_flags & ACC_NATIVE) != 0;
}
```

(b) method declarations

```
if (null != storepass) {
    cmd.createArg().setValue("-storepass");
    cmd.createArg().setValue(storepass);
}

if (null != storetype) {
    cmd.createArg().setValue("-storetype");
    cmd.createArg().setValue(storetype);
}

if (null != keypass) {
    cmd.createArg().setValue("-keypass");
    cmd.createArg().setValue(keypass);
}
```

(c) if blocks

```
out.println();
out.println("---------------------------------------");
out.println(" ANT_HOME/lib jar listing");
out.println("---------------------------------------");
doReportLibraries(out);

out.println();
out.println("---------------------------------------");
out.println(" Tasks availability");
out.println("---------------------------------------");
doReportTasksAvailability(out);
```

(d) method invocations

```
e = ccList.elements();
while (e.hasMoreElements()) {
    mailMessage.cc(e.nextElement().toString());
}

e = bccList.elements();
while (e.hasMoreElements()) {
    mailMessage.bcc(e.nextElement().toString());
}
```

(e) while blocks

```
src = attributes.getSrcdir();
destDir = attributes.getDestdir();
encoding = attributes.getEncoding();
debug = attributes.getDebug();
optimize = attributes.getOptimize();
deprecation = attributes.getDeprecation();
depend = attributes.getDepend();
verbose = attributes.getVerbose();
```

(f) assign statements

```
private MenuBar iAntMakeMenuBar = null;
private Menu iFileMenu = null;
private MenuItem iSaveMenuItem = null;
private MenuItem iMenuSeparator = null;
private MenuItem iShowLogMenuItem = null;
private Menu iHelpMenu = null;
private MenuItem iAboutMenuItem = null;
```

(g) variable declarations

```
} catch (final ClassNotFoundException cnfe) {
    throw new BuildException(cnfe);
} catch (final InstantiationException ie) {
    throw new BuildException(ie);
} catch (final IllegalAccessException iae) {
    throw new BuildException(iae);
}
```

(h) catch blocks

Fig. 1. Repeated code in Java source code (, which were identified in the investigation of literature [6])

It is generally said that switch statements, where repeated code often occur, are not recommended instruction in object-oriented design [5]. There are some research efforts that have proposed ways to transforms switch statements and consecutive if-else statement into multiple classes using polymorphism [4].

Consequently, paying special attention to repeated code can improve source code modification process. For example, firstly identifying repeated code in source code in an automatic way; then, if an element of repeated code is modified, the same modifica-

```
    …
947   excludes = new String[ 0 ];
948 }
949
950 filesIncluded = new Vector();
951 filesNotIncluded = new Vector();
952 filesExcluded = new Vector();
953 dirsIncluded = new Vector();
954 dirsNotIncluded = new Vector();
955 dirsExcluded = new Vector();
956
957 if( isIncluded( "" ) )
958 {
    …
```

(a) Before modification (revision 270,290)

```
    …
947   excludes = new String[ 0 ];
948 }
949
950 filesIncluded = new ArrayList();
951 filesNotIncluded = new ArrayList();
952 filesExcluded = new ArrayList();
953 dirsIncluded = new ArrayList();
954 dirsNotIncluded = new ArrayList();
955 dirsExcluded = new ArrayList();
956
957 if( isIncluded( "" ) )
958 {
    …
```

(b) After modification (revision 270,291)

Fig. 2. Actual modification on repeated code in file DirectoryScanner.java of Software Ant. Consecutive object generations were changed to ArrayList fromVector.

tions are (semi-)automatically applied to the other elements of the repeated code. those kinds of supports would be helpful for programmer. In this research, as a first step of modification support for repeated code, we investigate how repeated code are modified and evolved. Finding and analyzing their modification/evolution patterns will make it possible to propose useful ways of modification supports for repeated code.

As a result of the investigations we conducted on open source software, we obtained the following knowledge, which are main contributions of this paper:

- elements forming repeated code were too small to be detected by existing code clone detection tools;
- 73-89% of repeated code were modified at least once;
- 31-58% modifications on repeated code were applied to all the elements of repeated code simultaneously;
- any instruction type of repeated code was modified. Especially, try block, while block and variable declarations were more likely to be modified than the others; and,
- the lesser repetitions repeated code had, the higher the ratio of simultaneous modifications on all the elements of them was.

The remainder of this paper is organized as follows: Section 2 shows actual modifications on repeated code, which motivated us to conduct this research; Section 3 explains how we investigated modifications applied to repeated code; Section 4 shows the investigation result on three open source software systems; then, Section 6 describes some threats to validities on the investigation; finally, Section 7 concludes this paper.

2 Motivation

Figures 2 and 3 show actual modifications on repeated code in Ant. In Figure 2, six assignment statements creating Vector objects were changed to ones creating ArrayList

```
    …
135  private File[] getAnt1Files() {
136    List files = new ArrayList();
137    addJavaFiles(files, TASKDEFS_ROOT);
138    addJavaFiles(files, new File(TASKDEFS_ROOT, "compilers"));
139    addJavaFiles(files, new File(TASKDEFS_ROOT, "condition"));
140    addJavaFiles(files, DEPEND_ROOT);
141    addJavaFiles(files, new File(DEPEND_ROOT, "constantpool"));
142    addJavaFiles(files, TYPES_ROOT);
143    addJavaFiles(files, FILTERS_ROOT);
144    addJavaFiles(files, UTIL_ROOT);
145    addJavaFiles(files, new File(UTIL_ROOT, "depend"));
146    addJavaFiles(files, ZIP_ROOT);
147    addJavaFiles(files, new File(UTIL_ROOT, "facade"));
148    addJavaFiles(files, INPUT_ROOT);
149
150    files.add(new File(PACKAGE_ROOT, "BuildException.java"));
       …
```

(a) Before modification (revision 272,635)

```
    …
135  private File[] getAnt1Files() {
136    List files = new ArrayList();
137    addJavaFiles(files, TASKDEFS_ROOT, false);
138    addJavaFiles(files, new File(TASKDEFS_ROOT, "compilers"), true);
139    addJavaFiles(files, new File(TASKDEFS_ROOT, "condition"), true);
140    addJavaFiles(files, DEPEND_ROOT, true);
141    addJavaFiles(files, TYPES_ROOT, true);
142    addJavaFiles(files, FILTERS_ROOT, false);
143    addJavaFiles(files, UTIL_ROOT, false);
144    addJavaFiles(files, new File(UTIL_ROOT, "depend"), false);
145    addJavaFiles(files, new File(UTIL_ROOT, "facade"), true);
146    addJavaFiles(files, ZIP_ROOT, true);
147    addJavaFiles(files, INPUT_ROOT, true);
148
149    files.add(new File(PACKAGE_ROOT, "BuildException.java"));
       …
```

(b) After modification (revision 272,636)

Fig. 3. Actual modification on repeated code in file Builder.java of Software Ant. The number of parameters of consecutively invoked methods were increased.

objects simultaneously. In Figure 3, a parameter was added to every method invocation in repeated code.

As shown in these examples, all the elements forming a repeated code are modified on the same way simultaneously. The authors are thinking that there are two problems in such modifications:

- applying modifications to multiple (even many) places is a time-consuming and burdensome task;
- they introduce new bugs if some places to be modified are overlooked.

Consequently, modification supports on repeated code are necessary. For example, the following support may be useful: if we modify an element in a repeated code, (semi-)automatic modifications are performed on the other elements in the repeated code. In this paper, as a first step of modification support on repeated code, we investigate how repeated code are modified during software evolution.

3 Investigating Modifications on Repeated Code

Herein, we introduce a method for investigating how modifications were applied to repeated code during software evolution. The input and the output of the method are as follows.

INPUT repository of the target software.
OUTPUT data related to repeated code, for example the followings are distilled:
- instruction types included in repeated code;
- number of repetitions in repeated code;
- number of modifications applied to repeated code.

The investigation method consists of the following steps:

STEP1 identifying revisions where source files were modified;
STEP2 distilling data related to repeated code modified between every consecutive two revisions, each of which was identified in STEP1;
STEP3 making evolutional data from the results of STEP2.

In the reminder of this section, Subsections 3.1, 3.2, and 3.3 explain each step of the investigation method, respectively. Then, Subsection 3.4 describes software tool that we have developed based on the investigation method.

3.1 STEP1: identifying revisions where source files are modified

In STEP1, the method identifies revisions where one or more source files were modified. Source code repositories contain not only source files but also other files such as manual or copyright files, so that there are revisions that no source files were modified in software repositories. The purpose of STEP1 is eliminating revisions where no source files were modified because we focus on only modifications on source files.

Herein, we assume that:

- R is a target repository;
- there are n revisions where at least one source file was modified in R;
- index i represents the order of revisions included in R, that is, r_i means that its revision is the i-th oldest in R.

By using the above assumptions, repository R can be defined as:

$$R = \{r_1, r_2, \cdots, r_{n-1}, r_n\} \tag{1}$$

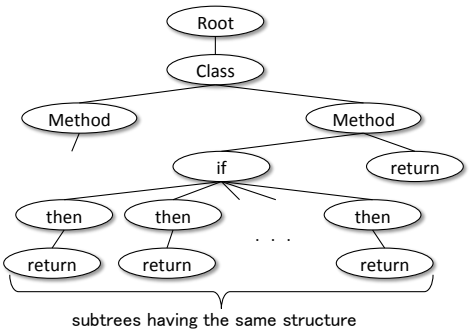
3.2 STEP2: distilling data related to repeated code modified between every consecutive two revisions

Differences between two consecutive revisions r_i and r_{i+1} $(1 \leq i \wedge i < n)$ are analyzed for finding whether repeated code were modified or not.

If repeated code were modified, the following information is distilled:

```
…
  public static int getColumnIndex(…) {
    if (…)
      return NDX_TI_ID_TITOLO;
    else if (…)
      return NDX_TI_ID_COMPAGNIA;
    else if (…)
      return NDX_TI_ID_NODO;
    else if (…)
      return NDX_TI_ID_CAUSALE;
    …
    else if (…)
      return NDX_FI_DESC_FILIALE;
    return -1;
  }
…
```

(a) source code



(b) abstract syntax tree

Fig. 4. An Example of constructing AST and finding repeated structures in it

```
 1: A
 2: B
 3: line will be changed 1
 4: line will be changed 2
 5: C
 6: D
 7: line will be deleted 1
 8: line will be deleted 2
 9: E
10: F
11: G
12: H
```

(a) before modification

```
 1: A
 2: B
 3: line changed 1
 4: line changed 2
 5: C
 6: D
 7: E
 8: F
 9: G
10: line added 1
11: line added 2
12: H
```

(b) after modification

```
3,4c3,4
<  line will be changed 1
<  line will be changed 2
---
>  line changed 1
>  line changed 2
7,8d6
<  line will be deleted 1
<  line will be deleted 2
11a10,11
>  line added 1
>  line added 2
```

(c) diff output

Fig. 5. A simple example of comparing two revisions of a source file with `diff` (changed region is represented with identifier 'c' like 3,4c3,4, deleted region is represented with identifier 'd' like 7,8d6, and added region is represented with identifier 'a' like 11a10,11. The numbers before and after the identifiers show the corresponding lines)

- instruction types forming the modified repeated code;
- numbers of repetitions of the modified repeated code;
- token numbers of elements of the modified repeated code;
- whether the modified repeated code sustained repeated structure or not.

We find whether repeated code were modified or not with the following steps.

STEP2A: Identifying repeated code in revisions r_i and r_{i+1} by using AST generated from the revisions. AST sibling nodes are sorted in the order of the appearance on the source code. If there are consecutive similar structures in the sibling nodes, their code are regarded as repeated structure.

- In the case that the sibling nodes are leaves, conditions for satisfying the similarity are (1) they are the same type nodes in AST and (2) they are textually similar to each other. For the 2nd condition, we use *Levenshtein* distance.

- In the case that the sibling nodes are branches, the whole subtrees under the branches have the similar structures. That is, structure similarity is checked recursively.

Repeated structures in AST are regarded as repeated code in source code.

Figure 4 shows an example of constructing AST and identifying repeated structures from it. In this case, subtrees under the if node in Figure 4(b) are the same structure, so consecutive if-else statements in the source code are regarded as repeated code. After identifying repeated code, their location information (line number) is distilled.

AST used herein is not a usual one. We applied some heuristics to AST for easily identifying repeated structures. If readers have an interested in the detail of the specialized AST and repeated code identification, see literature [16].

STEP2B: In order to identify where were modified in the source files between revisions r_i and r_{i+1}, we use UNIX `diff` command. Figure 5 shows an example of `diff` output. As shown in Figure 5, it is easy to identify line number modified between two revisions. All we have to do is just parsing the output of `diff` so that the start line and end line of all the modifications are identified.

STEP2C: By comparing the result of STEP2A and STEP2B, we find whether repeated code in revision r_i were modified or not. If modified, the above information is distilled.

3.3 STEP3: making evolutional data by using the results of STEP2

In this step, we track repeated code through all the target revisions by using diff information between every consecutive two revisions (the result of STEP2). Tracking repeated code allows us to obtain the following data:

- when a given repeated code appeared and disappeared;
- the number of modifications applied to a given repeated code.

We used the method proposed in literature [3] for tracking repeated code.

Finally, evolutional data related to repeated code is output textually. In this step, any visualization of the data is not performed. If necessary, user can create some graphs or perform statistical tests for understanding data by themselves.

3.4 Implementation

We have developed a software tool based on the investigation method. In the tool, we are using

- JDT (Java Development Tool) for Java source code analysis, and
- SVNKit for handling Subversion repositories.

That is, currently the tool is just a prototype, and it can be applied to only Java software managed with Subversion. Output of STEP3 is in CSV format, which is intended for analyzing data with Excel or R.

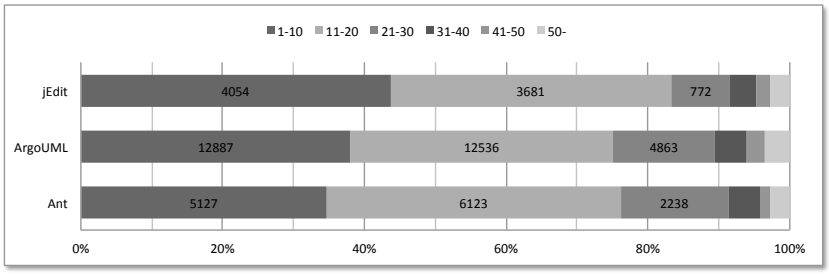


Fig. 6. Distribution of Element Size (number of tokens) on the End Revision

4 Investivation on Open Source Software

In order to reveal how repeated code is modified during software evolution, we investigated three open source software systems. We chose Ant, ArgoUML, and jEdit as our targets because they are well-known and widely-used systems. Table 1 shows the detail information of the systems.

In this investigation, we reveal the followings:

RQ1 how large are elements of repeated code?
RQ2 how often are repeated code modified during software evolution?
RQ3 what is the rate of simultaneous modifications on repeated code?

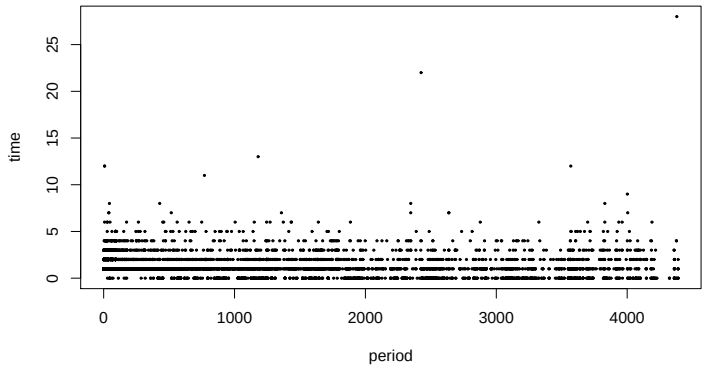
In the reminder of this section, we describe the experimental results and answer the RQs.

4.1 Answer to RQ1 (How large are elements of repeated code?)

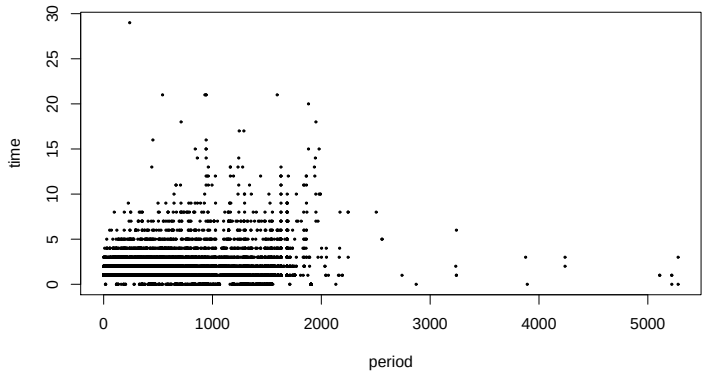
Figure 6 shows distributions of element sizes (the number tokens) on the end revision. We can see that small size dominates a large part: 1-10 are between 35-44%; 11-20 are between 37-41%; 21-30 are between 8-15%. Totally, 1-30 elements dominate 89-92% for all the elements. Code clone detection tools take a threshold of minimal size of code clones to be detected. In many cases, "30 tokens" is used for the threshold [8]. Of course, we can use smaller thresholds in code clone detection. However, if we use

Table 1. Overview of Target Software

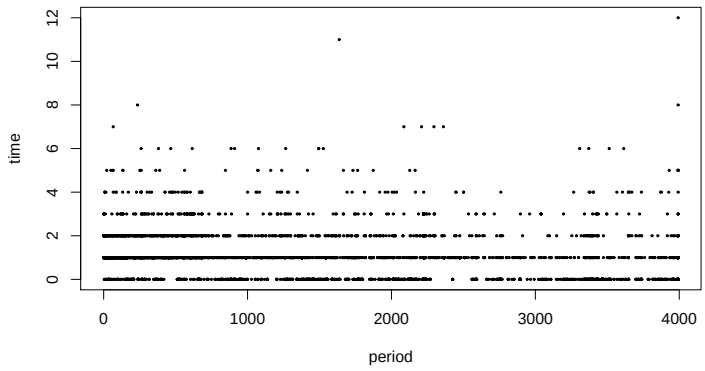
Software	Start revision (date)	End revision (date)	# of target revisions	LOC of end revision
Ant	267,549 (2000-1-13)	1,233,420 (2012-1-20)	12,621	255,061
ArgoUML	2 (1998-1-27)	19,893 (2012-7-10)	17,731	369,583
jEdit	3,791 (2001-9-2)	21,981 (2012-8-7)	5,292	183,006



(a) Ant



(b) ArgoUML



(c) jEdit

Fig. 7. Relationships between survival period and the number of modifications. Y-axis is the number of modifications, and X-axis is survival period.

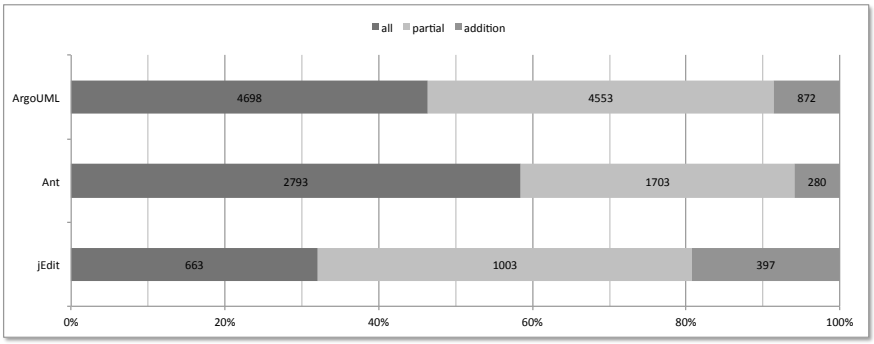


Fig. 8. Ratio of the three types of modifications on each target system

smaller thresholds, tools would detect a large amount of code clones , which include many false positives. Extracting necessary code clones from a large result is not an easy task. That is, code clone detection techniques are not suited for identifying repeated code.

Consequently, in order to identify repeated code, it is necessary to use a technique that are tailored to detect repeated code. In this paper, we proposed a method using similarities of AST subtrees for identifying repeated code. The method is scalable, so that we could finish repository analysis of the target software within 30 hours, 65 hours, and 18 hours from 12,621, 17,731, and 5,292 revisions of source code, respectively.

4.2 Answer to RQ2 (How often are repeated code modified during software evolution?)

Figure 7 shows relationships between survival period and the number of modifications: Y-axis is the number of modifications and X-axis is survival period. A dot locating on 1 or above of Y-axis means its repeated code was modified at least once. We can see the following from this figure:

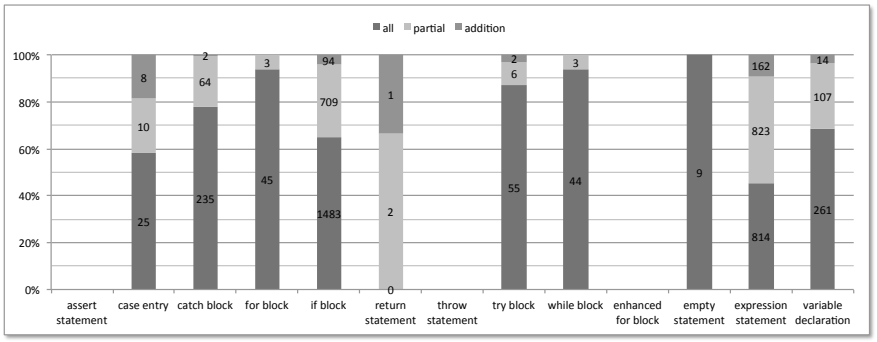
- 84%, 89%, and 73% repeated code were modified at least once,
- there was no correlation between survival period and the number of modifications.

The numbers of modifications on repeated code were 4,776, 10,123, and 2,063, respectively. By dividing them with the number of target revisions, we obtained 0.438, 0.395, and 0.356. That is, repeated code were modified every two or three revisions.

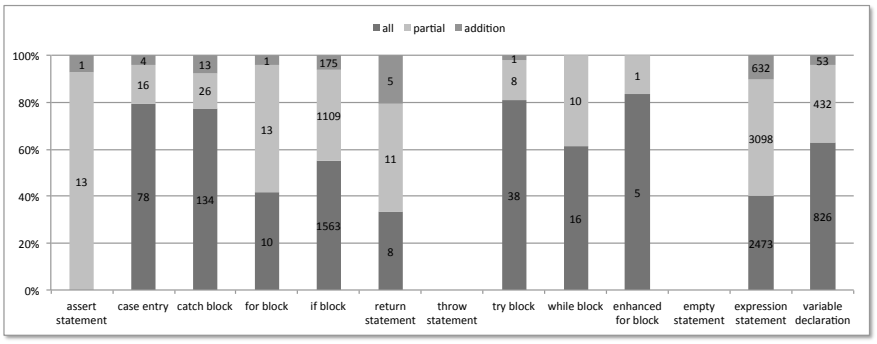
4.3 Answer to RQ3 (What is the rate of simultaneous modifications on repeated code?)

We analyzed modifications on repeated code and classified them as follows:

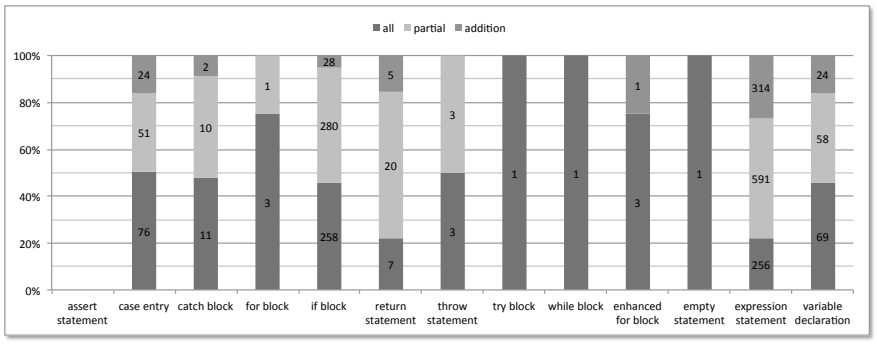
all all the elements in a repeated code were modified simultenously;



(a) Ant



(b) ArgoUML

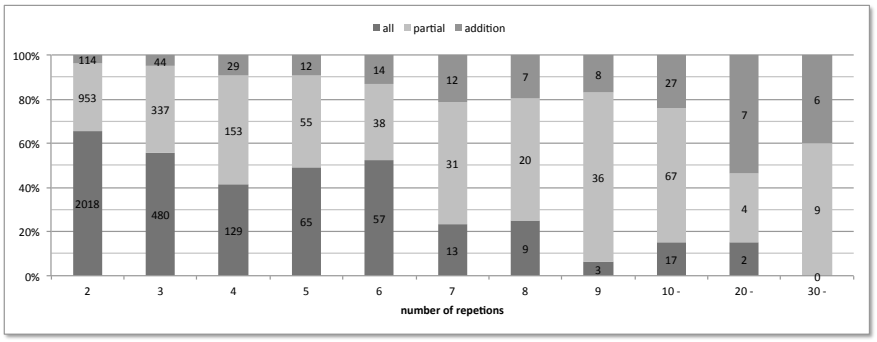


(c) jEdit

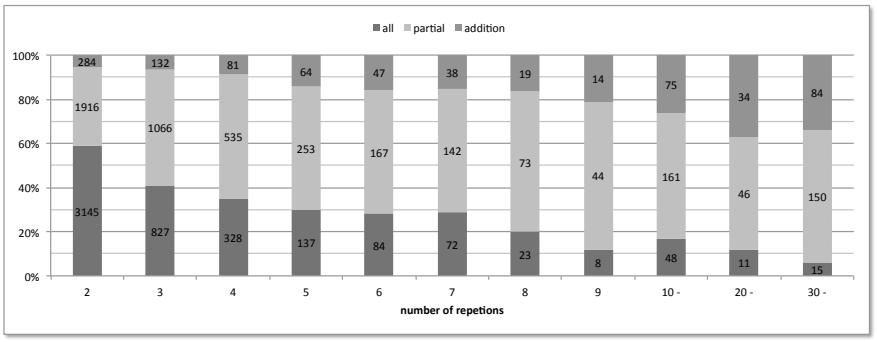
Fig. 9. Ratio of the three types of modifications by focusing on instruction types in repeated code

partial only a part of elements was modified;

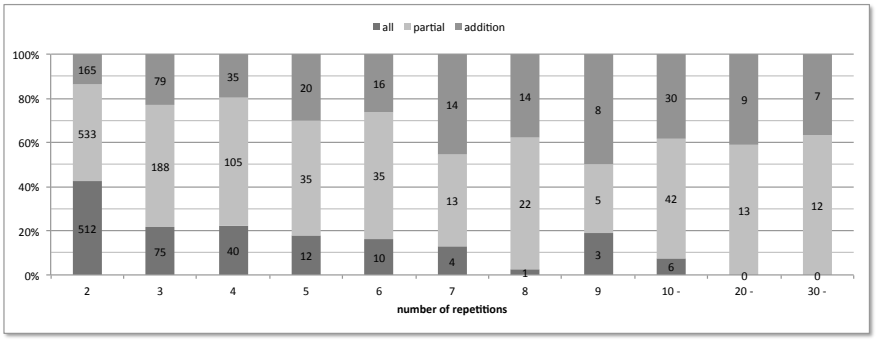
addition existing elements of a repeated code were not modified but new elements were added to the repeated code.



(a) Ant



(b) ArgoUML



(c) jEdit

Fig. 10. Ratio of the three types of modifications by focusing on the number of repetitions of repeated code

Figure 8 shows ratio of the three types of modifications. There were many *all* modifications on all the target systems. The numbers were 663, 2,793, and 4,698, respectively. They dominated 31%, 58%, and 46% for the modifications on repeated code.

Furthermore, we investigated the ratio of the three types of modifications by focusing on the followings characteristics of repeated code:

- instruction types in repeated code;
- the number of repetitions of repeated code.

Figure 9 shows the former result. Areas with no bars mean there was no modification on the instruction types. We can see that some types have a higher ratio of *all* modifications. For example, try block, while block, and variable declarations are near to or more than 50%.

Figure 10 shows the latter result. The followings are common phenomena in all the target systems.

- The lesser number of repetitions is, the higher ratio of *all* modifications is.
- The higher number of repetitions is, the higher ratio of *addition* modifications is. Repeated code including many repetition are more likely to get new repetitions than repeated code with a few repetitions.

5 Useful Support on Repeated Code

In this research, we found that 73-89% of repeated code was modified at least once in their life. Thus, modification supports on repeated code is necessary to reduce cost of source code modification.

We found that if a repeated code has lesser repetitions, all of its elements are more likely to be modified simultaneously. Thus, we are thinking that interactive modification completions are useful for repeated code. For example, if an element of a repeated code is modified, a plugin in IDE recommends the same modification for each of the other elements of the repeated code interactively. All programmers have to do is to answer “yes” or “no” for every recommendation. If he/she answers “yes”, the element is modified automatically as recommended. If “no”, it is not modified. If the number of repetitions are large, a bunch of interactive replacements is also a burdensome task. However, for small number of repetitions, such interactive modification supports will be great helpful.

Also, we found that repeated code had gained more elements as they evolved. Consequently, following support will be useful: if programmers pull the trigger, a plugin of IDE generated a template of repeated element based on the structure of existing elements of repeated code and it was automatically inserted to the bottom of the repeated code. All they have to do is to fulfilling holes of the template. In most cases, only variable names or method invocations are inserted to holes.

6 Threats to Validity

6.1 Number of target systems

In this investigation, the number of target systems was only three. In order to generalize the investigation result, we have to conduct experiments on more software systems.

Currently, we can investigate only Java software managed with Subversion due to the implementation limitations. In the future, we are going to extend the tool for other programming languages such as C/C++ and other version control systems such as git for investigating various software systems.

6.2 Not regarding modification types

In this investigation, we did not take care of modification types. For example, in the case of variable declaration statement, there may be a modification that inserts a single white space between its operand and its operator. Such modification does not have a direct impact on program behavior. Consequently, if we extracted and used only the modifications that are bug fixes or function additions, the investigation result would be different from the investigation result of this paper.

6.3 Disappearing repeated code

In this investigation, we regarded that a repeated code had disappeared if it satisfied either of the conditions:

- the repeated code is completely removed from the source code;
- the number of its repetition became one.

In the latter case, an element of repeated code remains in the source code after modifications. Hence, the latter case should not be regarded as disappearance of repeated code. If we conducted the investigation with the setting, the investigation result would be changed.

7 Conclusion

In this paper, we investigated how repeated code had been modified during software evolution as a first step for improving modification process on repeated code.

We selected three famous open source software systems, Ant, ArgoUML, and jEdit as experimental targets. As a result, we obtained the following knowledge.

- Element size of repeated code was too small to be detected with code clone detection tools.
- 73-89% of repeated code were modified at least once.
- 31-58% of modifications were simultaneous modifications for all the elements of them.
- Any instruction type of repeated code was modified. Especially, try block, while block and variable declarations were more likely to be modified than the others.
- The lesser repetitions repeated code had, the higher the ratio of simultaneous modifications on all the elements of them was.

Acknowledgment

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

References

1. Aversano, L., Cerulo, L., Di Penta, M.: How clones are maintained: An empirical study. In: Proceedings of the 11th European Conference on Software Maintenance and Reengineering. pp. 81–90. CSMR '07, IEEE Computer Society, Washington, DC, USA (2007)
2. Baker, B.S.: On finding duplication and near-duplication in large software systems. In: Proceedings of the Second Working Conference on Reverse Engineering. pp. 86–. WCRE '95, IEEE Computer Society, Washington, DC, USA (1995)
3. Canfora, G., Cerulo, L., Penta, M.D.: Identifying changed source code lines from version repositories. In: Proceedings of the Fourth International Workshop on Mining Software Repositories. pp. 14–. MSR '07, IEEE Computer Society, Washington, DC, USA (2007)
4. Ducasse, S., Demeyer, S., Nierstrasz, O.: Transform conditionals to polymorphism. In: Proceedings EUROPLOP'00 (5th European Conference on Pattern Languages of Programming and Computing, 1999. pp. 219–252. UVK Universitätsverlag Konstanz GmbH, Konstanz, Germany (jul 2000)
5. Fowler, M.: Refactoring: improving the design of existing code. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1999)
6. Higo, Y., Kamiya, T., Kusumoto, S., Inoue, K.: Method and implementation for investigating code clones in a software system. Inf. Softw. Technol. 49(9-10), 985–998 (Sep 2007)
7. Higo, Y., Kusumoto, S.: How often do unintended incosistencies happend? –deriving modification patterns and detecting overlooked code fragments–. In: Proceedings of the 2012 28th IEEE International Conference on Software Maintenance. pp. 222–231. ICSM '12, IEEE Computer Society, Washington, DC, USA (2012)
8. Kamiya, T., Kusumoto, S., Inoue, K.: Ccfinder: A multilinguistic token-based code clone detection system fo r large scale source code. IEEE Transactions on Software Engineering 28, 654–670 (2002)
9. Kapser, C.J., Godfrey, M.W.: "cloning considered harmful" considered harmful: patterns of cloning in software. Empirical Softw. Engg. 13(6), 645–692 (Dec 2008)
10. Kim, M., Bergman, L., Lau, T., Notkin, D.: An ethnographic study of copy and paste programming practices in oopl. In: Proceedings of the 2004 International Symposium on Empirical Software Engineering. pp. 83–92. ISESE '04, IEEE Computer Society, Washington, DC, USA (2004)
11. Kim, M., Sazawal, V., Notkin, D., Murphy, G.: An empirical study of code clone genealogies. In: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering. pp. 187–196. ESEC/FSE-13, ACM, New York, NY, USA (2005)
12. Li, Z., Lu, S., Myagmar, S., Zhou, Y.: Cp-miner: Finding copy-paste and related bugs in large-scale software code. IEEE Trans. Softw. Eng. 32(3), 176–192 (Mar 2006)
13. Murakami, H., Hotta, K., Higo, Y., Igaki, H., Kusumoto, S.: Folding repeated instructions for improving token-based code clone detection. In: Proceedings of the 2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation. pp. 64–73. SCAM '12, IEEE Computer Society, Washington, DC, USA (2012)
14. Roy, C.K., Cordy, J.R.: An empirical study of function clones in open source software. In: Proceedings of the 2008 15th Working Conference on Reverse Engineering. pp. 81–90. WCRE '08, IEEE Computer Society, Washington, DC, USA (2008)
15. Roy, C.K., Cordy, J.R., Koschke, R.: Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. Sci. Comput. Program. 74(7), 470–495 (May 2009)
16. Sasaki, Y., Ishihara, T., Hotta, K., Hata, H., Higo, Y., Igaki, H., Kusumoto, S.: Preprocessing of metrics measurement based on simplifying program structures. In: International Workshop on Software Analysis, Testing and Applications. pp. 120–127 (12 2012)