Hindawi Publishing Corporation
Advances in Software Engineering
Volume 2012, Article ID 938296, 22 pages
doi:10.1155/2012/938296

Research Article

An Empirical Study on the Impact of Duplicate Code

Keisuke Hotta, Yui Sasaki, Yukiko Sano, Yoshiki Higo, and Shinji Kusumoto

Graduate School of Information Science and Technology, Osaka University, Osaka 565-0871, Japan

Correspondence should be addressed to Keisuke Hotta, k-hotta@ist.osaka-u.ac.jp

Received 4 January 2012; Accepted 5 March 2012

Academic Editor: Osamu Mizuno

It is said that the presence of duplicate code is one of the factors that make software maintenance more difficult. Many research efforts have been performed on detecting, removing, or managing duplicate code on this basis. However, some researchers doubt this basis in recent years and have conducted empirical studies to investigate the influence of the presence of duplicate code. In this study, we conduct an empirical study to investigate this matter from a different standpoint from previous studies. In this study, we define a new indicator "modification frequency" to measure the impact of duplicate code and compare the values between duplicate code and nonduplicate code. The features of this study are as follows the indicator used in this study is based on modification places instead of the ratio of modified lines; we use multiple duplicate code detection tools to reduce biases of detection tools; and we compare the result of the proposed method with other two investigation methods. The result shows that duplicate code tends to be less frequently modified than nonduplicate code, and we found some instances that the proposed method can evaluate the influence of duplicate code more accurately than the existing investigation methods.

1. Introduction

Recently, duplicate code has received much attention. Duplicate code is also called as "code clone." Duplicate code is defined as identical or similar code fragments to each other in the source code, and they are generated by various reasons such as copy-and-paste programming. It is said that the presence of duplicate code has negative impacts on software development and maintenance. For example, they increase bug occurrences: if an instance of duplicate code is changed for fixing bugs or adding new features, its correspondents have to be changed simultaneously; if the correspondents are not changed inadvertently, bugs are newly introduced to them.

Various kinds of research efforts have been performed for resolving or improving the problems caused by the presence of duplicate code. For example, there are currently a variety of techniques available to detect duplicate code [1]. In addition, there are many research efforts for merging duplicate code as a single module like function or method, or for preventing duplications from being overlooked in modification [2, 3]. However, there are precisely the opposite opinions that code cloning is a good choice for design of the source code [4].

In order to answer the question whether duplicate code is harmful or not, several efforts have proposed comparison methods between duplicate code and nonduplicate code. Each of them compares a characteristic of duplicate code and nonduplicate code instead of directly investigating their maintenance cost. This is because measuring the actual maintenance cost is quite difficult. However, there is no consensus on this matter.

In this paper, we conduct an empirical study that compares duplicate code to nonduplicate code from a different standpoint of previous research and reports the experimental result on open source software. The features of the investigation method in this paper are as follows:

(i) every line of code is investigated whether it is duplicate code or not; such a fine-grained investigation can accurately judge whether every modification conducted to duplicate code or to nonduplicate code;

(ii) maintenance cost consists of not only source code modification but also several phases prior to it; in order to more appropriately estimate maintenance cost, we define an indicator that is not based on modified lines of code but the number of modified places;

(iii) we evaluate and compare modifications of duplicate code and nonduplicate code on multiple open source software systems with multiple duplicate code detection tools, that is, because every detection tool detects different duplicate code from the same source code.

We also conducted a comparison experiment with two previous investigation methods. The purpose of this experiment is to reveal whether comparisons between duplicate code and nonduplicate code with different methods yield the same result or not. In addition, we carefully analyzed the results in the cases that the comparison results were different from each method to reveal the causes behind the differences.

The rest of this paper is organized as follows: Section 2 describes related works and our motivation of this study. Section 3 introduces the preliminaries. We situate our research questions and propose a new investigation method in Section 4. Section 5 describes the design of our experiments, then we report the results in Sections 6 and 7. Section 8 discusses threats to validity, and Section 9 presents the conclusion and future work of this study.

2. Motivation

2.1. Related Work. At present, there is a huge body of work on empirical evidence on duplicate code shown in Table 1. The pioneering report in this area is Kim et al.'s study on clone genealogies [5]. They have conducted an empirical study on two open source software systems and found 38% or 36% of groups of duplicate code were consistently changed at least one time. On the other hand, they observed that there were groups of duplicate code that existed only for a short period (5 or 10 revisions) because each instance of the groups was modified inconsistently. Their work is the first empirical evidence that a part of duplicate code increases the cost of source code modification.

However, Kapser and Godfrey have different opinions regarding duplicate code. They reported that duplicate code can be a reasonable design decision based on the empirical study on two large-scale open source systems [4]. They built several patterns of duplicate code in the target systems, and they discussed the pros and cons of duplicate code using the patterns. Bettenburg et al. also reported that duplicate code does not have much a negative impact on software quality [6]. They investigated inconsistent changes to duplicate code at release level on two open software systems, and they found that only 1.26% to 3.23% of inconsistent changes introduced software errors into the target systems.

Monden et al. investigated the relation between software quality and duplicate code on the file unit [7]. They use the number of revisions of every file as a barometer of quality: if the number of revisions of a file is great, its quality is low. Their experiment selected a large-scale legacy system, which was being operated in a public institution, as the target. The result showed that modules that included duplicate code were 40% lower quality than modules that did not include duplicate code. Moreover, they reported that the larger duplicate code a source file included, the lower quality it was.

Lozano et al. investigated whether the presence of duplicate code was harmful or not [8]. They developed a tool, CloneTracker, which traces which methods include duplicate code (in short, duplicate method) and which methods are modified in each revision. They conducted a pilot study, and found that: duplicate methods tend to be more frequently modified than nonduplicate methods; however, duplicate methods tend to be modified less simultaneously than nonduplicate methods. The fact implies that the presence of duplicate code increased cost for modification, and programmers were not aware of the duplication, so that they sometimes overlooked code fragments that had to be modified simultaneously.

Also, Lozano and Wermelinger investigated the impact of duplicate code on software maintenance [9]. Three barometers were used in the investigation. The first one is *likelihood*, which indicates the possibility that the method is modified in a revision. The second one is *impact*, which indicates the number of methods that are simultaneously modified with the method. The third one is *work*, which can be represented as a product of *likelihood* and *impact* ($work = likelihood \times impact$). They conducted a case study on 4 open source systems for comparing the three barometers of methods including and not including duplicate code. The result was that *likelihood* of methods including duplicate code was not so different from one of methods not including duplicate code; there were some instances that *impact* of methods including duplicate code were greater than one of methods not including duplicate code; if duplicate code existed in methods for a long time, their *work* tended to increase greatly.

Moreover, Lozano et al. investigated the relation between duplicate code, features of methods, and their changeability [10]. Changeability means the ease of modification. If changeability decreased, it will be a bottleneck of software maintenance. The result showed that the presence of duplicate code can decrease changeability. However, they found that changeability was more greatly affected by other properties such as length, fan-out, and complexity of methods. Consequently, they concluded that it was not necessary to consider duplicate code as a primary option.

Krinke hypothesized that if duplicate code is less stable than nonduplicate code, maintenance cost for duplicate code is greater than for nonduplicate code. He conducted a case study in order to investigate whether the hypothesis is true or not [11]. The targets are 200 revisions (a version per week) of source code of 5 large-scale open-source systems. He measured *added*, *deleted*, and *changed* LOCs on duplicate code and nonduplicate code and compared them. He reported that nonduplicate code was more *added*, *deleted*, and *changed* than duplicate code. Consequently, he concluded that the presence of duplicate code did not necessarily make it more difficult to maintain source code.

Göde and Harder replicated Krinke's experiment [12]. Krinke's original experiment detected line-based duplicate code meanwhile their experiment detected token-based duplicate code. The experimental result was the same as Krinke's one. Duplicate code is more stable than nonduplicate code in the viewpoint of added and changed. On

Table 1: Summarization of related work.

	How to investigate	Impact of duplicate code
Kim et al. [5]	Using clone linages and clone genealogies	A part of duplicate code is negative
Kapser and Godfrey [4]	Build several patterns of duplicate code and discuss about them	Nonnegative
Bettenburg et al. [6]	Investigate inconsistent changes to duplicate code at the release revel	Nonnegative
Monden et al. [7]	Calculate the number of revisions on every file	Negative
Lozano et al. [8]	Count the number of modifications on methods including duplicate code	Negative
Lozano and Wermelinger [9]	Using *work*	A part of duplicate code is negative
Lozano et al. [10]	Using changeability (the ease of modification)	Negative but not so high
Krinke [11]	Using stability (line level)	Nonnegative
Göde and Harder [12]	Using stability (token level)	Nonnegative
Krinke [13]	Using ages	Nonnegative
Rahman et al. [14]	Investigate the relationship between duplicate code and bugs	Nonnegative
Göde and Koschke [15]	Count the number of changes on clone genealogies	A part of duplicate code is negative

the other hand, from the deleted viewpoint, nonduplicate code is more stable than duplicate code.

Also, Krinke conducted an empirical study to investigate ages of duplicate code [13]. In this study, he calculated and compared average ages of duplicate lines and nonduplicate lines on 4 large-scale Java software systems. He found that the average age of duplicate code is older than nonduplicate code, which implies duplicate code is more stable than nonduplicate code.

Eick et al. investigated whether source code decays when it is operated and maintained for a long time [16]. They selected several metrics such as the amount of *added* and *deleted* code, the time required for modification, and the number of developers as indicators of code decay. The experimental result on a 15-year-operated large system showed that cost required for completing a single requirement tendS to increase.

Rahman et al. investigated the relationship between duplicate code and bugs [14]. They analyzed 4 software systems written in C language with bug information stored in Bugzilla. They use Deckard, which is an AST-based detection tool, to detect duplicate code. They reported that only a small part of the bugs located on duplicate code, and the presence of duplicate code did not dominate bug appearances.

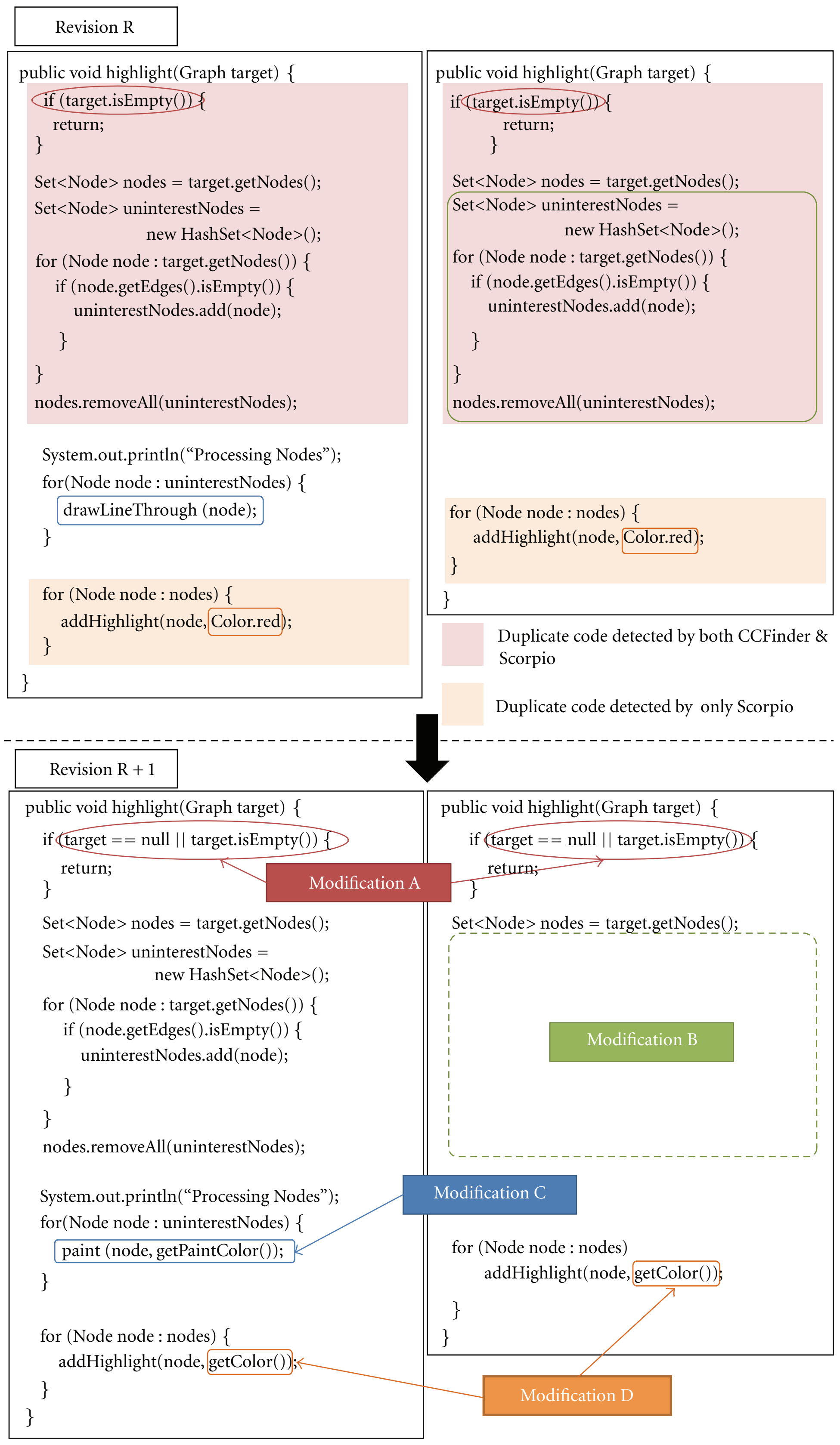
Göde modeled how type-1 code clones are generated and how they evolved [17]. Type-1 code clone is a code clone that is exactly identical to its correspondents except white spaces and tabs. He applied the model to 9 open-source software systems and investigated how code clones in them evolved. The result showed that the ratio of code duplication was decreasing as time passed; the average life time of code clones was over 1 year; in the case that code clones were modified inconsistently, there were a few instances that additional modifications were performed to restore their consistency.

Also, Göde and Koschke conducted an empirical study on clone evolution and performed a detailed tracking to detect when and how clones had been changed [15]. In their study, they traced clone evolution and counted the number of changes on each clone genealogy. They manually inspected the result in one of the target systems and categorized all the modifications on clones into consistent or inconsistent. In addition, they carefully categorized inconsistent changes into intentional or unintentional. They reported that almost all clones were never changed or only once during their lifetime, and only 3% of the modifications had high severity. Therefore, they concluded that many of clones do not cause additional change effort, and it is important to identify the clones with high threat potential to manage duplicate code effectively.

As described above, some empirical studies reported that duplicate code should have a negative impact on software evolution meanwhile the others reported the opposite result. At present, there is no consensus on the impact of the presence of duplicate code on software evolution. Consequently, this research is performed as a replication of the previous studies with solid settings.

2.2. Motivating Example. As described in Section 2.1, many research efforts have been performed on evaluating the influence of duplicate code. However, these investigation methods still have some points that they did not evaluate. We explain these points with the example shown in Figure 1. In this example, there are two similar methods and some places are modified. We classified these modifications into 4 parts, modification A, B, C, and D.

Investigated Units. In some studies, large units (e.g., files or methods) are used as their investigation units. In those investigation methods, it is assumed that duplicate code has a negative impact if files or methods having a certain amount of duplicate code are modified, which can cause a problem. The problem is the incorrectness of modifications count. For example, if modifications are performed on a method which has a certain amount of duplicate code, all the modifications are assumed as performed on the duplicate code even if they are actually performed on nonduplicate code of the method. Modification C in Figure 1 is an instance of this problem. This modification is performed on nonduplicate


Revision R
Duplicate code detected by both CCFinder & Scorpio
Duplicate code detected by only Scorpio
Revision R + 1
Modification A
Modification B
Modification C
Modification D

```
public void highlight(Graph target) {
  if (target.isEmpty()) {
    return;
  }
  Set<Node> nodes = target.getNodes();
  Set<Node> uninterestNodes =
                 new HashSet<Node>();
  for (Node node : target.getNodes()) {
    if (node.getEdges().isEmpty()) {
      uninterestNodes.add(node);
    }
  }
  nodes.removeAll(uninterestNodes);

  System.out.println("Processing Nodes");
  for(Node node : uninterestNodes) {
    drawLineThrough (node);
  }

  for (Node node : nodes) {
    addHighlight(node, Color.red);
  }
}
```

```
public void highlight(Graph target) {
  if (target.isEmpty()) {
        return;
      }

  Set<Node> nodes = target.getNodes();
  Set<Node> uninterestNodes =
                 new HashSet<Node>();
  for (Node node : target.getNodes()) {
    if (node.getEdges().isEmpty()) {
      uninterestNodes.add(node);
    }
  }
  nodes.removeAll(uninterestNodes);

  for (Node node : nodes) {
    addHighlight(node, Color.red);
  }
}
```

```
public void highlight(Graph target) {
  if (target == null || target.isEmpty()) {
    return;
  }
  Set<Node> nodes = target.getNodes();
  Set<Node> uninterestNodes =
                 new HashSet<Node>();
  for (Node node : target.getNodes()) {
    if (node.getEdges().isEmpty()) {
      uninterestNodes.add(node);
    }
  }
  nodes.removeAll(uninterestNodes);

  System.out.println("Processing Nodes");
  for(Node node : uninterestNodes) {
    paint (node, getPaintColor());
  }

  for (Node node : nodes) {
    addHighlight(node, getColor());
  }
}
```

```
public void highlight(Graph target) {
  if (target == null || target.isEmpty()) {
    return;
  }
  Set<Node> nodes = target.getNodes();

  for (Node node : nodes)
    addHighlight(node, getColor());

  }
}
```

Figure 1: Motivating example.

code; nevertheless, it is regarded that this modification is performed on duplicate code if we use method as the investigation units.

Line-Based Barometers. In some studies, line-based barometers are used to measure the influence of duplicate code on software evolution. Herein, the line-based barometer indicates a barometer calculated with the amount of added/changed/deleted lines of code. However, line-based barometer cannot distinguish the following two cases: the first case is that *consecutive 10 lines of code were modified for fixing a single bug*; the second case is that *1 line modification was performed on different 10 places of code for fixing 10 different bugs*. In real software maintenance, the latter requires much more cost than the former because we have to conduct several steps before the actual source code modification such as identifying buggy module, informing the maintainer about the bugs, and identifying buggy instruction.

In Figure 1, Modification A is 1 line modification, and performed on 2 places, meanwhile Modification B is 7 lines modification on a single place. With line-based barometers, it is regarded that Modification B has the impact 3.5 times larger than Modification A. However, this is not true because we have to identify 2 places of code for modifying A meanwhile 1 place identification is required for B.

A Single Detection Tool. In the previous studies, a single detection tool was used to detect duplicate code. However, there is neither a generic nor strict definition of duplicate code. Each detection tool has its own unique definition of duplicate code, and it detects duplicate code based on the own definition. Consequently, different duplicate code is detected by different detection tools from the same source code. Therefore, the investigation result with one detector is different from the result from another detector. In Figure 1, a detector `CCFinder` detects lines highlighted with red as duplicate code, and another detector `Scorpio` detects not only lines highlighted with red but also lines highlighted with orange before modification. Therefore, if we use `Scorpio`, Modification D is regarded as being affected with duplicate code, nevertheless, it is regarded as not being affected with duplicate code if we use `CCFinder`. Consequently, the investigation with a single detector is not sufficient to get the generic result about the impact of duplicate code.

2.2.1. Objective of This Study. In this paper, we conducted an empirical study from a different standpoint of previous research. The features of this study are as follows.

Fine-Grained Investigation Units. In this study, every line of code is investigated whether it is duplicate code or not, which enables us to judge whether every modification is conducted on duplicate code or nonduplicate code.

Place-Based Indicator. We define a new indicator based on the number of modified places, not the number of modified lines. The purpose of place-based indicator is to evaluate the impact of the presence of duplicate code with different standpoints from the previous research.

Multiply Detector. In this study, we use 4 duplicate code detection tools to reduce biases of each detection method.

3. Preliminaries

In this section, we describe preliminaries used in this paper.

3.1. Duplicate Code Detection Tools. There are currently various kinds of duplicate code detection tools. The detection tools take the source code as their input data, and they provide the position of the detected duplicate code in it. The detection tools can be categorized based on their detection techniques. Major categories should be line based, token based, metrics based, AST (Abstract Syntax Tree) based, and PDG (Program Dependence Graph) based. Each technique has merits and demerits, and there is no technique that is superior to any other techniques in every way [1, 18]. The following subsections describe 4 detection tools that are used in this research. We use two token-based detection tools, which is for investigating whether both the token-based detection tools always introduce the same result or not.

3.1.1. `CCFinder`. `CCFinder` is a token-based detection tool [19]. The major features of `CCFinder` are as follows.

(i) `CCFinder` replaces user-defined identifiers such as variable names or function names with special tokens before the matching process. Consequently, `CCFinder` can identify code fragments that use different variables as duplicate code.

(ii) Detection speed is very fast. `CCFinder` can detects duplicate code from millions lines of code within an hour.

(iii) `CCFinder` can handle multiple popular programming languages such as C/C++, Java, and COBOL.

3.1.2. `CCFinderX`. `CCFinderX` is a major version up from `CCFinder` [20]. `CCFinderX` is a token-based detection tool as well as `CCFinder`, but the detection algorithm was changed to *bucket sort* to *suffix tree*. `CCFinderX` can handle more programming languages than `CCFinder`. Moreover, it can effectively use resources of multi core CPUs for faster detection.

3.1.3. `Simian`. `Simian` is a line-based detection tool [21]. As well as `CCFinder` family, `Simian` can handle multiple programming languages. Its line-based technique realizes duplicate code detection on small memory usage and short running time. Also, `Simian` allows fine-grained settings. For example, we can configure that duplicate code is not detected from *import* statements in the case of Java language.

3.1.4. `Scorpio`. `Scorpio` is a PDG-based detection tool [22, 23]. `Scorpio` builds a special PDG for duplicate code detection, not traditional one. In traditional PDGs, there are two types of edge representing data dependence and control dependence. The special PDG used in `Scorpio` has one

```
(a) before modification
   (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

(b) after 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

(c) diff output
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
```

Algorithm 1: A simple example of comparing two source files with `diff` (changed region is represented with identifier "c" like 3,4c3,4; deleted region is represented with identifier "d" like 7,8d6, added region is represented with identifier "a" like 11a10,11). The number before and after the identifier shows the correspond lines.

more edge, execution-next link, which allows detecting more duplicate code than traditional PDG. Also, `Scorpio` adopts some heuristics for filtering out false positives. Currently, `Scorpio` can handle only Java language.

3.2. Revision. In this paper, we analyze historical data managed by version control systems for investigation. Version control systems store information about changes to documents or programs. We can specify changes by using a number, "*revision*". We can get source code in arbitrary revision, and we can also get modified files, change logs, and the name of developers who made changes in arbitrary two consecutive revisions with version control systems.

Due to the limit of implementation, we restrict the target version control system to Subversion. However, it is possible to use other version control systems such as CVS.

3.3. Target Revision. In this study, we are only interested in changes in source files. Therefore, we find out revisions that have some modifications in source files. We call such revisions as *target revisions*. We regard a revision R as the target revision, if at least one source file is modified from R to $R + 1$.

3.4. Modification Place. In this research, we use the number of places of modified code, instead lines of modified code. That is, even if multiple consecutive lines are modified, we regard it as a single modification. In order to identify the number of modifications, we use UNIX diff command. Algorithm 1 shows an example of diff output. In this example, we can find 3 modification places. One is a change in line 3 and 4, another is a deletion in line 7 and 8, and the other is an addition at line 11. As shown in the algorithm, it is very easy to identify multiple consecutive modified lines as a single modification; 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.

4. Proposed Method

This section describes our research questions and the investigation method.

4.1. Research Questions and Hypotheses. The purpose of this research is to reveal whether the presence of duplicate code really affects software evolution or not. We assume that *if duplicate code is more frequently modified than nonduplicate code, the presence of duplicate code has a negative impact on software evolution*. This is because if much duplicate code is included in source code though, it is never modified during its lifetime, the presence of duplicate code never causes inconsistent changes or additional modification efforts. Our research questions are as follows.

RQ1: Is duplicate code more frequently modified than nonduplicate code?

RQ2: Are the comparison results of stability between duplicate code and nonduplicate code different from multiple detection tools?

RQ3: Is duplicate code modified uniformly throughout its lifetime?

RQ4: Are there any differences in the comparison results on modification types?

To answer these research questions, we define an indicator, *modification frequency* (in short, MF). We measure and compare MF of duplicate code (in short, MF_d) and MF of nonduplicate code (in short, MF_n) for investigation.

4.2. Modification Frequency

4.2.1. Definition. As described above, we use MF to estimate the influence of duplicate code. MF is an indicator based on the number of modified code, not lines of modified code. This is because this research aims to investigate from a different standpoint from previous research.

We define MF_d in the formula:

$$\mathrm{MF}_d = \frac{\sum_{r\in R} \mathrm{MC}_d(r)}{|R|}, \tag{1}$$

where R is a set of target revisions, $\mathrm{MC}_d(r)$ is the number of modifications on duplicate code between revision r and $r+1$. We also define MF_n in the formula:

$$\mathrm{MF}_n = \frac{\sum_{r\in R} \mathrm{MC}_n(r)}{|R|}, \tag{2}$$

where $\mathrm{MC}_n(r)$ is the number of modifications on nonduplicate code between revision r and $r + 1$.

These values mean the average number of modifications on duplicate code or nonduplicate code per revision. However, in these definitions, MF_d and MF_n are very affected by the amount of duplicate code included the source code. For example, if the amount of duplicate code is very small, it is quite natural that the number of modifications on duplicate code is much smaller than nonduplicate code. However, if a small amount of duplicate code is included but it is quite frequently modified, we need additional maintenance efforts to judge whether its correspondents need the same modifications or not. We cannot evaluate the influence of duplicate code in these situations in these definitions.

In order to eliminate the bias of the amount of duplicate code, we normalize the formulae (1) and (2) with the ratio of duplicate code. Here, we assume that

(i) $\mathrm{LOC}_d(r)$ is the total lines of duplicate code in revision r,

(ii) $\mathrm{LOC}_n(r)$ is the total lines of nonduplicate code on r,

(iii) $\mathrm{LOC}(r)$ is the total lines of code on r, so that the following formula is satisfied:

$$\mathrm{LOC}(r) = \mathrm{LOC}_d(r) + \mathrm{LOC}_n(r). \tag{3}$$

Under these assumptions, the normalized MF_d and MF_n are defined in the following formula:

$$\begin{aligned}\text{normalized } \mathrm{MF}_d &= \frac{\sum_{r\in R} \mathrm{MC}_d(r)}{|R|} \times \frac{\sum_{r\in R} \mathrm{LOC}(r)}{\sum_{r\in R} \mathrm{LOC}_d(r)},\\ \text{normalized } \mathrm{MF}_n &= \frac{\sum_{r\in R} \mathrm{MC}_n(r)}{|R|} \times \frac{\sum_{r\in R} \mathrm{LOC}(r)}{\sum_{r\in R} \mathrm{LOC}_n(r)}.\end{aligned} \tag{4}$$

In the reminder of this paper, the normalized MF_d and MF_n are called as just MF_d and MF_n, respectively.

4.2.2. Measurement Steps. The MF_d and MF_n are measured with the following steps,

Step 1. It identifies target revisions from the repositories of target software systems. Then, all the target revisions are checked out into the local storage.

Step 2. It normalized all the source files in every target revision.

Step 3. It detects duplicate code within every target revision. Then, the detection result is analyzed in order to identify the file path, the lines of all the detected duplicate code.

Step 4. It identifies differences between two consecutive revisions. The start lines and the end lines of all the differences are stored.

Step 5. It counts the number of modifications on duplicate code and nonduplicate code.

Step 6. It calculates MF_d and MF_n.

In the reminder of this subsection, we explain each step of the measurement in detail.

Step 1. It obtains Target Revisions. In order to measure MF_d and MF_n, it is necessary to obtain the historical data of the source code. As described above, we used a version control system, Subversion, to obtain the historical data.

Firstly, we identify which files are modified, added, or deleted in each revision and find out target revisions. After identifying all the target revision from the historical data, they are checked out into the local storage.

Step 2. It normalizes Source Files. In the Step 2, every source file in all the target revisions is normalized with the following rules:

(i) deletes blank lines, code comments, and indents,

(ii) deletes lines that consist of only a single open/close brace, and the open/close brace is added to the end of the previous line.

The presence of code comments influences the measurement of MF_d and MF_n. If a code comment is located within a duplicate code, it is regarded as a part of duplicate code even if it is not a program instruction. Thus, the LOC of duplicate code is counted greater than it really is. Also, there is no common rule how code comments should be treated if they are located in the border of duplicate code and nonduplicate code, which can cause a problem that a certain detection tool regards such a code comment as duplicate code meanwhile another tool regards it as nonduplicate code.

As mentioned above, the presence of code comments makes it more difficult to identify the position of duplicate code accurately. Consequently, all the code comments are removed completely. As well as code comments, different detection tools handle blank lines, indents, lines including only a single open or close brace in different ways, which also influence the result of duplicate code detection. For this reason, blank lines and indents are removed, and lines that consist of only a single open or close brace are removed, and

Table 2: Target software systems—Experiment 1.

(a) Experiment 1.1

Name	Domain	Programming language	Number of Revisions	LOC (latest revision)
EclEmma	Testing	Java	788	15,328
FileZilla	FTP	C++	3,450	87,282
FreeCol	Game	Java	5,963	89,661
SQuirrel SQL Client	Database	Java	5,351	207,376
WinMerge	Text Processing	C++	7,082	130,283

(b) Experiment 1.2

Name	Domain	Programming language	Number of Revisions	LOC (latest revision)
ThreeCAM	3D Modeling	Java	14	3,854
DatabaseToUML	Database	Java	59	19,695
AdServerBeans	Web	Java	98	7,406
NatMonitor	Network (NAT)	Java	128	1,139
OpenYMSG	Messenger	Java	141	130,072
QMailAdmin	Mail	C	312	173,688
Tritonn	Database	C/C++	100	45,368
Newsstar	Network (NNTP)	C	165	192,716
Hamachi-GUI	GUI, Network (VPN)	C	190	65,790
GameScanner	Game	C/C++	420	1,214,570

Table 3: Overview of Investigation Methods.

Method	Krinke [11]	Lozano and Wermelinger [9]	Proposed method
Target Revisions	A revision per week	All	All
Investigation Unit	Line	Method	Place (consecutive lines)
Measure	ratio of Modified lines	*Work*	Modification frequency

the removed open or close brace is added to the end of the previous line.

Step 3. It detects Duplicate Code. In this step, duplicate code is detected from every target revision, and the detection results are stored into a database. Each detected duplicate code is identified by 3-tuple (v, f, l), where v is the revision number that a given duplicate code was detected; f is the absolute path to the source file where a given duplicate code exists; l is a set of line numbers where duplicate code exists. Note that storing only the start line and the end line of duplicate code is not feasible because a part of duplicate code is non-contiguous.

This step is very time consuming. If the history of the target software includes 1,000 revisions, duplicate code detection is performed 1,000 times. However, this step is fully automated, and no manual work is required.

Step 4. It identifies Differences between Two Consecutive Revisions. In Step 4, we find out modification places between two consecutive revisions with UNIX diff command. As described above, we can get this information by just parsing the output of diff.

Step 5. It Counts the Number of Modifications. In this step, we count the number of modifications of duplicate code and nonduplicate code with the results of the previous two steps. Here, we assume the variable for the number of modifications of duplicate code is MC_d, and the variable for nonduplicate code is MC_n. Firstly, MC_d and MC_n are initialized with 0, then they are increased as follows; if the range of specified modification is completely included in duplicate code, MC_d is incremented; if it is completely included in nonduplicate code, MC_n is incremented; if it is included in both of duplicate code and nonduplicate code, both MC_d and MC_n are incremented. All the modifications are processed with the above algorithm.

Step 6. It calculates MF_d and MF_n. Finally, MF_d and MF_n defined in the formula (4) are calculated with the result of the previous step.

5. Design of Experiment

In this paper, we conduct the following two experiments.

Experiment 1. Compare MF_d and MF_n on 15 open-source software systems.

Experiment 2. Compare the result of the proposed method with 2 previous investigation methods on 5 open-source software systems.

Table 4: Target software systems—Experiment 2.

Name	Domain	Programming language	Number of Revisions	LOC (latest revision)
OpenYMSG	Messenger	Java	194	14,111
EclEmma	Testing	Java	1,220	31,409
MASU	Source Code Analysis	Java	1,620	79,360
TVBrowser	Multimedia	Java	6,829	264,796
Ant	Build	Java	5,412	198,864

We describe these experiments in detail in the reminder of this section.

5.1. Experiment 1

5.1.1. Outline. The purpose of this experiment is to answer our research questions. This experiment consists of the following two subexperiments.

Experiment 1.1. We compare MF_d and MF_n on various size software systems with a scalable detection tool, `CCFinder`.

Experiment 1.2. We compare MF_d and MF_n on small size software systems with 4 detection tools, described in Section 3.1.

The reason why we choose only a single clone detector, `CCFinder`, on Experiment 1.1 is that the experiment took much time. For instance, we took a week to conduct the experiment on SQuirrel SQL Client.

The following items are investigated in each subexperiment.

Item A. Investigate whether duplicate code is modified more frequently than nonduplicate code. In this investigation, we calculate MF_d and MF_n on the entire period.

Item B. Investigate whether MF tendencies differ according to the time.

To answer RQ1, we use the result of Item A of Experiments 1.1 and 1.2. For RQ2, we use the result of Item A of Experiment 1.1. For RQ3, we use Item B of Experiment 1.1 and Experiment 1.2. Finally, for RQ4, we use Item A of Experiment 1.1 and Experiment 1.2.

5.1.2. Target Software Systems. In Experiment 1, we select 15 open source software systems shown in Table 2 as investigation targets. 5 software systems are investigated in Experiment 1.1, and the other software systems are investigated in Experiment 1.2. The criteria for these target software systems are as follows:

(i) the source code is managed with Subversion;

(ii) the source code is written in C/C++ or Java;

(iii) we took care not to bias the domains of the targets.

Table 5: Ratio of duplicate code—Experiment 1.

(a) Experiment 1.1

Software Name	ccf	ccfx	sim	sco
EclEmma	13.1%	—	—	—
FileZilla	22.6%	—	—	—
FreeCol	23.1%	—	—	—
SQuirrel	29.0%	—	—	—
WinMerge	23.6%	—	—	—

(b) Experiment 1.2

Software Name	ccf	ccfx	sim	sco
ThreeCAM	29.8%	10.5%	4.1%	26.2%
DatabaseToUML	21.4%	25.1%	7.6%	11.8%
AdServerBeans	22.7%	18.2%	20.3%	15.9%
NatMonitor	9.0%	7.7%	0.7%	6.6%
OpenYMSG	17.4%	9.9%	5.8%	9.9%
QMailAdmin	34.3%	19.6%	8.8%	—
Tritonn	13.8%	7.5%	5.5%	—
Newsstar	7.9%	4.8%	1.5%	—
Hamachi-GUI	36.5%	23.1%	18.5%	—
GameScanner	23.1%	13.1%	6.6%	—

5.2. Experiment 2

5.2.1. Outline. In Experiment 2, we compare the results of the proposed method and two previously described investigation methods on the same targets. The purpose of this experiment is to reveal whether comparisons of duplicate code and nonduplicate code with different methods always introduce the same result. Also, we evaluate the efficacy of the proposed method comparing to the other methods.

5.2.2. Investigation Methods to Be Compared. Here, we describe 2 investigation methods used in Experiment 2. We choose investigation methods proposed by Krinke [11] (in short, Krinke's method) and proposed by Lozano and Wermelinger [9] (in short, Lozano's method). Table 3 shows the overview of these methods and the proposed method. The selection was performed based on the following criteria.

(i) The investigation is based on the comparison some characteristics between duplicate code and nonduplicate code.

(ii) The method has been published at the time when our research started (at 2010/9).

In the experiments of Krinke's and Lozano's papers, only a single detection tool `Simian` or `CCFinder` was selected. However, in this experiment, we selected 4 detection tools for bringing more valid results.

We developed software tools for Krinke's and Lozano's methods based on their papers. We describe Krinke's method and Lozano's method briefly.

Krinke's Method. Krinke's method compares stability of duplicate code and nonduplicate code [11]. Stability is calculated based on ratios of modified duplicate code and modified nonduplicate code. This method uses not all the revisions but a revision per week.

First of all, a revision is extracted from every week history. Then, duplicate code is detected from every of the extracted revisions. Next, every consecutive two revisions are compared for obtaining where added lines, deleted lines, and changed lines are. With this information, the ratios of added lines, deleted lines, and changed lines on duplicate and nonduplicate code are calculated and compared.

Lozano's Method. Lozano's method categorized Java methods, then compare distributions of maintenance cost based on the categories [9].

Firstly, Java methods are traced based on their owner class's full qualified name, start/end lines, and signatures. Methods are categorized as follows:

AC-Method. Methods that always had duplicate code during their lifetime;

NC-Method. Methods that never had duplicate code during their lifetime;

SC-Method. Methods that sometimes had duplicate code and sometimes did not.

Lozano's method defines the followings where m is a method, P is a period (a set of revisions), and r is a revision.

(i) $ChangedRevisions(m, P)$: a set of revisions that method m is modified in period P,

(ii) $Methods(r)$: a set of methods that exist in revision r,

(iii) $ChangedMethods(r)$: a set of methods that were modified in revision r,

(iv) $CoChangedMethods(m, r)$: a set of methods that were modified simultaneously with method m in revision r. If method m is not modified in revision r, it becomes 0. If modified, the following formula is satisfied:

$$ChangedMethod(r) = m \cup CoChangedMethod(m, r). \tag{5}$$

Table 6: Overall results—Experiment 1.

(a) Experiment 1.1

Software Name	ccf	ccfx	sim	sco
EclEmma	N	—	—	—
FileZilla	N	—	—	—
FreeCol	N	—	—	—
SQuirrel	N	—	—	—
WinMerge	N	—	—	—

(b) Experiment 1.2

Software Name	ccf	ccfx	sim	sco
ThreeCAM	N	C	N	N
DatabaseToUML	N	N	N	N
AdServerBeans	N	N	N	N
NatMonitor	C	C	N	C
OpenYMSG	C	C	C	N
QMailAdmin	C	C	C	—
Tritonn	N	C	N	—
Newsstar	N	N	N	—
Hamachi-GUI	N	N	N	—
GameScanner	C	C	N	—

Then, this method calculates the following formulae with the above definitions. Especially, *work* is an indicator of the maintenance cost:

$$\begin{aligned} likelihood(m, P) &= \frac{ChangedRevisions(m, P)}{\sum_{r \in P} |ChangedMethods(r)|}, \\ impact(m, P) & \\ &= \frac{\sum_{r \in P} |CoChangedMethods(m, r)| / |Methods(r)|}{|ChangedRevisions(m, P)|}, \\ work(m, P) &= likelihood(m, P) \times impact(m, P). \end{aligned} \tag{6}$$

In this research, we compare *work* between AC-Method and NC-Method. In addition, we also compare SC-Methods' *work* on duplicate period and nonduplicate period.

5.2.3. Target Software Systems. We chose 5 open-source software systems in Experiment 2. Table 4 shows them. Two targets, OpenYMSG and EclEmma, are selected as well as Experiment 1. Note that the number of revisions and LOC of the latest revision of these two targets are different from Table 2. This is because they had been being in development between the time-lag in Experiments 1 and 2. Every source file is normalized with the rules described in Section 4.2.2 as well as Experiment 1. In addition, automatically generated code and testing code are removed from all the revisions before the investigation methods are applied.

6. Experiment 1—Result and Discussion

6.1. Overview. Table 5 shows the average ratio for each target of Experiment 1. Note that "ccf," "ccfx," "sim," and "sco" in

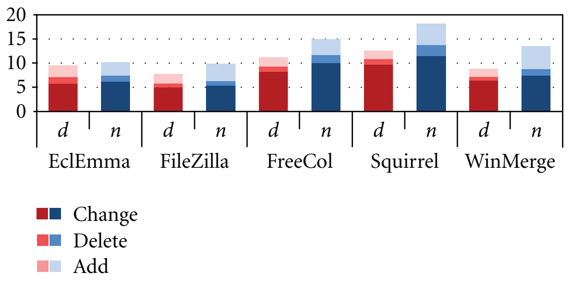


FIGURE 2: Result of Item A on Experiment 1.1.

TABLE 7: The average values of MF in Experiment 1.1.

Modification Type	MF	
	Duplicate code	Nonduplicate code
Change	7.0337	8.1039
Delete	1.0216	1.4847
Add	1.9539	3.7378
ALL	10.0092	13.3264

the table are the abbreviated form of `CCFinder`, `CCFinderX`, `Simian`, and `Scorpio`, respectively.

Table 6 shows the overall result of Experiment 1. In this table, "C" means $MF_d > MF_n$ in that case, and "N" means the opposing result. For example, the comparison result in ThreeCAM with `CCFinder` is $MF_d < MF_n$, which means duplicate code is not modified more frequently than nonduplicate code. Note that "—" means the cases that we do not consider because of the following reasons: (1) in Experiment 1.1, we use only `CCFinder`, so that the cases with other detectors are not considered; (2) `Scorpio` can handle only Java, so that the cases in software systems written in C/C++ with `Scorpio` are not considered.

We describe the results in detail in the following subsections.

6.2. Result of Experiment 1.1. Figure 2 shows all the results of Item A on Experiment 1.1. The labels "*d*" and "*n*" in *X*-axis means MF in duplicate code and nonduplicate code, respectively, and every bar consists of three parts, which means *change*, *delete*, and *add*. As shown in Figure 2, MF_d is lower than MF_n on all the target systems. Table 7 shows the average values of MF based on the modification types. The comparison results of MF_d and MF_n show that MF_d is less than MF_n in the cases of all the modification types. However, the degrees of differences between MF_d and MF_n are different for each modification type.

For Item B on Experiment 1.1, first, we divide the entire period into 10 sub-periods and calculate MF on every of the sub periods. Figure 3 shows the result. *X*-axis is the divided periods. Label "1" is the earliest period of the development, and label "10" is the most recent period. In the case of EclEmma, the number of periods that MF_d is greater than MF_n is the same as the number of periods that MF_n is greater than MF_d. In the case of FileZilla, FreeCol, and WinMerge, there is only a period that MF_d is greater than MF_n. In

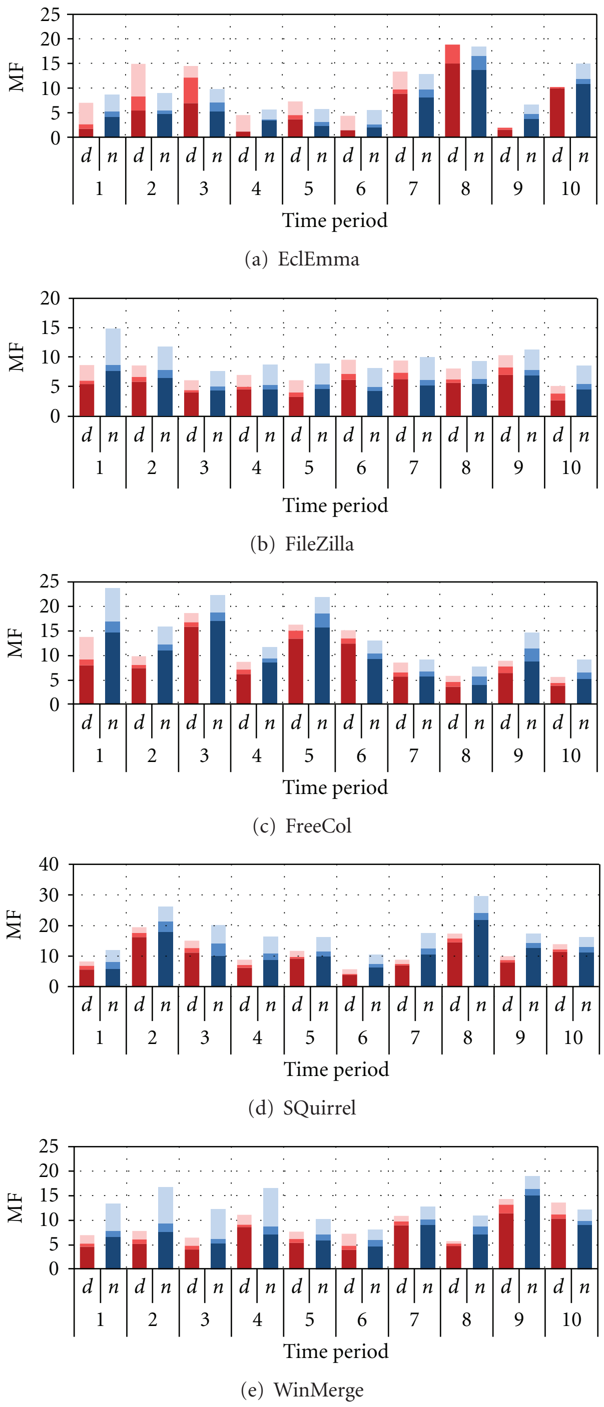


FIGURE 3: Result of Item B on Experiment 1.1 (divided into 10 periods).

the case of Squirrel SQL Client, MF_n is greater than MF_d in all the periods. This result implies that if the number of revisions becomes large, duplicate code tends to become more stable than nonduplicate code. However, the shapes of MF transitions are different from every software system.

For WinMerge, we investigated period "2," where MF_n is much greater than MF_d, and period "10," where is only the

Table 8: Comparing MFs based on programming language and detection tool.

(a) Comparison on programming language

Programming language	MF	
	Duplicate code	Nonduplicate code
Java	20.4370	24.1739
C/C++	49.4868	57.2246
ALL	32.8869	38.3384

(b) Comparison on detection tool

Detection tool	MF	
	Duplicate code	Nonduplicate code
`CCFinder`	38.2790	40.7211
`CCFinderX`	40.3541	40.0774
`Simian`	26.0084	42.1643
`Scorpio`	20.9254	24.1628
ALL	32.8869	38.3384

Table 9: The average values of MF in Experiment 1.2.

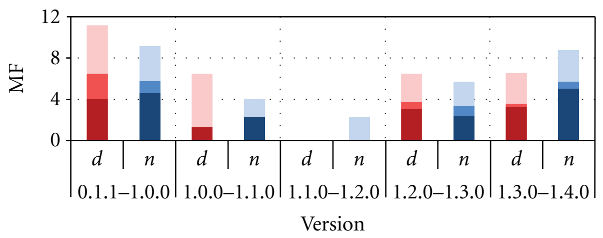
Modification type	MF	
	Duplicate code	Nonduplicate code
Change	26.8065	29.2549
Delete	3.8706	3.5228
Add	2.2098	5.5608
ALL	32.8869	38.3384

period that MF_d is greater than MF_n. In period "10," there are many modifications on test cases. The number of revisions that test cases are modified is 49, and the ratio of duplicate code in test cases is 88.3%. Almost all modifications for test cases are performed on duplicate code, so that MF_d is greater than MF_n. Omitting the modifications for test cases, MF_d and MF_n became inverted. However, there is no modification on test cases in period "2," so that MF_d is less than MF_n in this case.

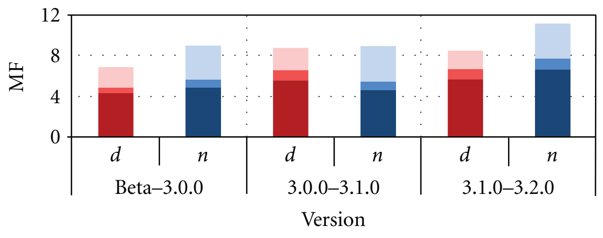
Moreover, we divide the entire period by release dates and calculate MF on every period. Figure 4 shows the result. As the figure shows, MF_d is less than MF_n in all the cases for FileZilla, FreeCol, SQuirrel, and WinMerge. For EclEmma, there are some cases that $\mathrm{MF}_d > \mathrm{MF}_n$ at the release level. Especially, duplicate code is frequently modified in the early releases.

Although MF_d is greater than MF_n in the period "6" in Freecol and the period "10" in WinMerge, MF_d is less than MF_n in all cases at the release level. This indicates that duplicate code is sometimes modified intensively in a short period, nevertheless it is stable than nonduplicate code in a long term.

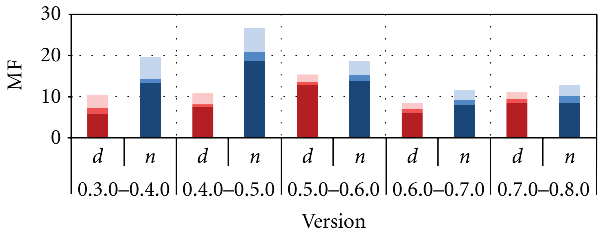
The summary of Experiment 1 is that duplicate code detected by `CCFinder` was modified less frequently than nonduplicate code. Consequently, we conclude that duplicate code detected by `CCFinder` does not have a negative impact on software evolution even if the target software is large and its period is long.



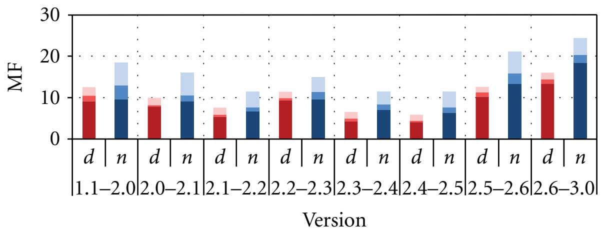
(a) EclEmma



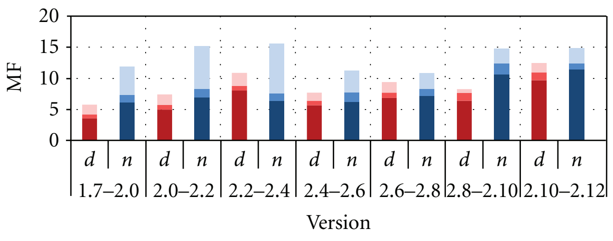
(b) FileZilla



(c) FreeCol



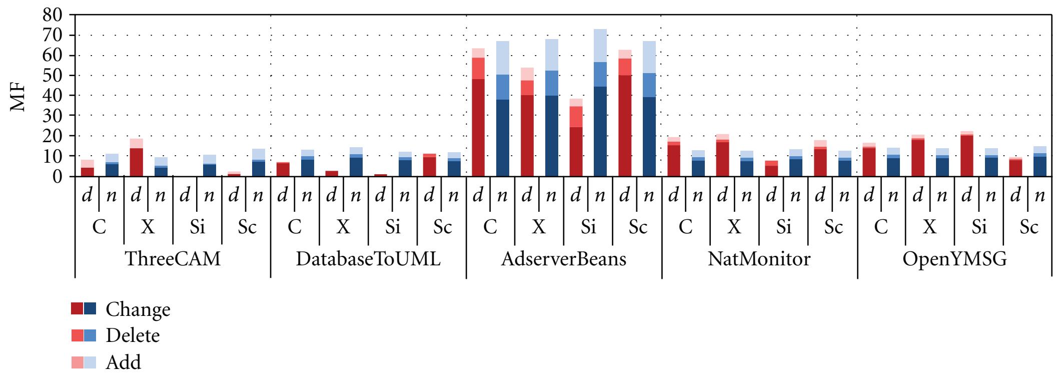
(d) SQuirrel



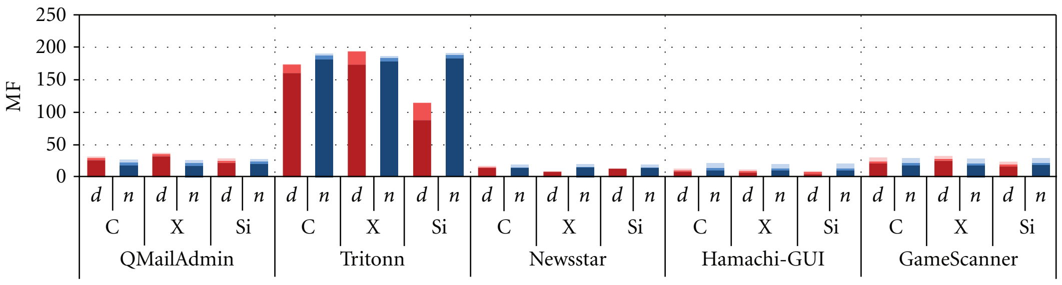
(e) WinMerge

Figure 4: Result of Item B on Experiment 1.1 (divided by releases).

6.3. Result of Experiment 1.2. Figure 5 shows all the results of Item A on Experiment 1.2. In Figure 5, the detection tools are abbreviated as follows: `CCFinder` → C; `CCFinderX` → X; `Simian` → `Si`; `Scorpio` → `Sc`. There are the results of 3 detection tools except `Scorpio` on C/C++ systems, because `Scorpio` does not handle C/C++. MF_d is less than MF_n in the 22 comparison results out of 35. In the 4 target systems out of 10, duplicate code is modified less frequently than



(a) Java software



(b) C/C++ Software

Figure 5: Result of Item A on Experiment 1.2.

nonduplicate code in the cases of all the detection tools. In the case of the other 1 target system, MF_d is greater than MF_n in the cases of all the detection tools. In the remaining systems, the comparison result is different for the detection tools. Also, we compared MF_d and MF_n based on programming language and detection tools. The comparison result is shown in Table 8. The result shows that MF_d is less than MF_n on all the programming language, and MF_d is less than MF_n on the 3 detectors, `CCFinder`, `Simian`, and `Scorpio`, meanwhile the opposing result is shown in the case of `CCFinderX`. We also compared MF_d and MF_n based on modification types. The result is shown in Table 9. As shown in Table 9, MF_d is less than MF_n in the cases of change and addition, meanwhile the opposing result is shown in the case of deletion.

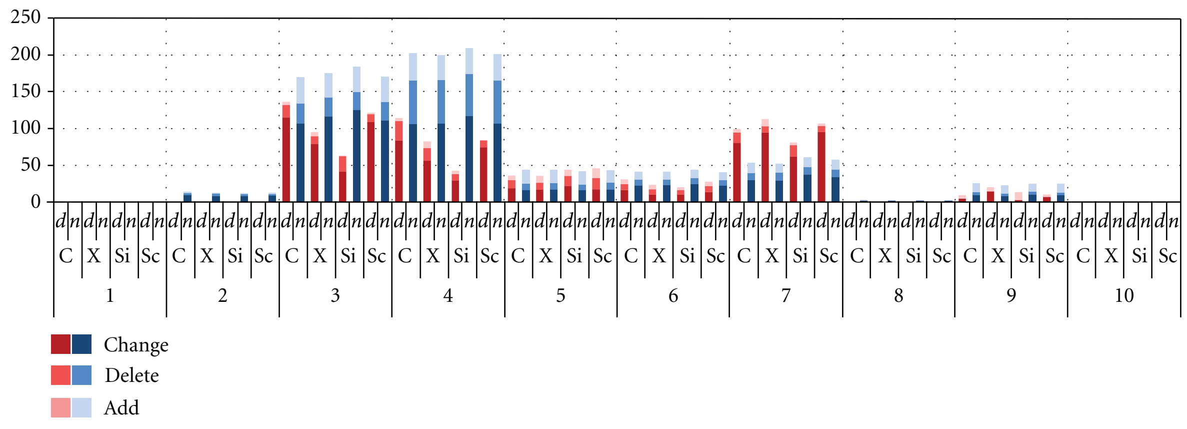
We investigated whether there is a statistically significant difference between MF_d and MF_n by t-test. The result is that, there is no difference between them where the level of significance is 5%. Also, there is no significant difference in the comparison based on programming language and detection tool.

For Item B on Experiment 1.2, we divide the whole period into 10 subperiods likewise Experiment 1.1. Figure 6 shows the result. In this experiment, we observed that the tendencies of MF transitions loosely fall into three categories: (1) MF_d is lower than MF_n almost of all the divisions; (2) MF_d is greater than MF_n in the early divisions, meanwhile the opposite tendency is observed in the late divisions; (3) MF_d is less than MF_n in the early divisions, meanwhile the opposite tendency is observed in the late divisions. Figure 6 shows the result of the 3 systems on which we observed remarkable tendencies of every category.

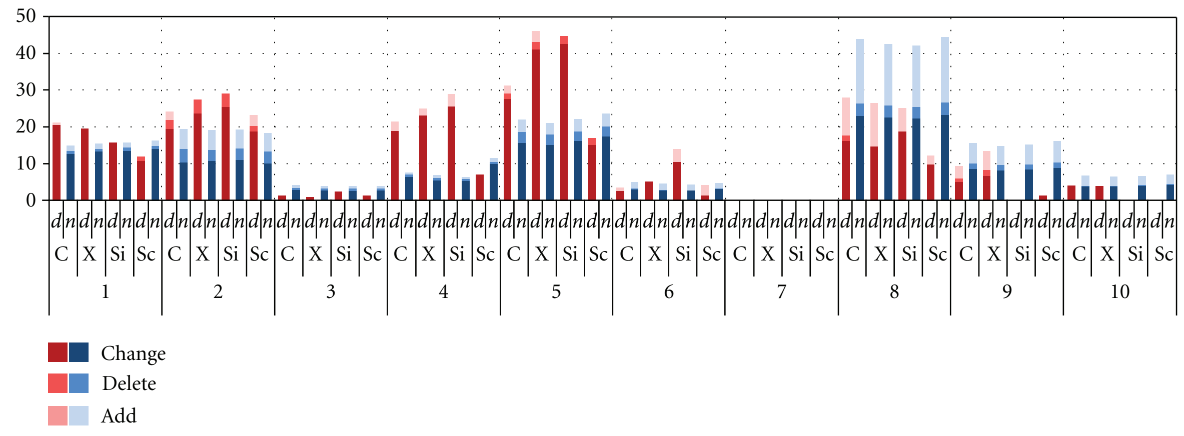
In Figure 6(a), period "4" shows that MF_n is greater than MF_d on all the detection tool meanwhile period "7" shows exactly the opposite result. Also, in period "5," there are hardly differences between duplicate code and nonduplicate code. We investigated the source code of period "4." In this period, many source files were created by copy-and-paste operations, and a large amount of duplicate code was detected by each detection tool. The code generated by copy-and-paste operations was very stable meanwhile the other source files were modified as usual. This is the reason why MF_n is much greater than MF_d in period "4."

Figure 6(b) shows that duplicate code tends to be modified more frequently than nonduplicate code in the anterior half of the period meanwhile the opposite occurred in the posterior half. We found that there was a large number of duplicate code that was repeatedly modified in the anterior half. On the other hand, there was rarely such duplicate code in the posterior half.

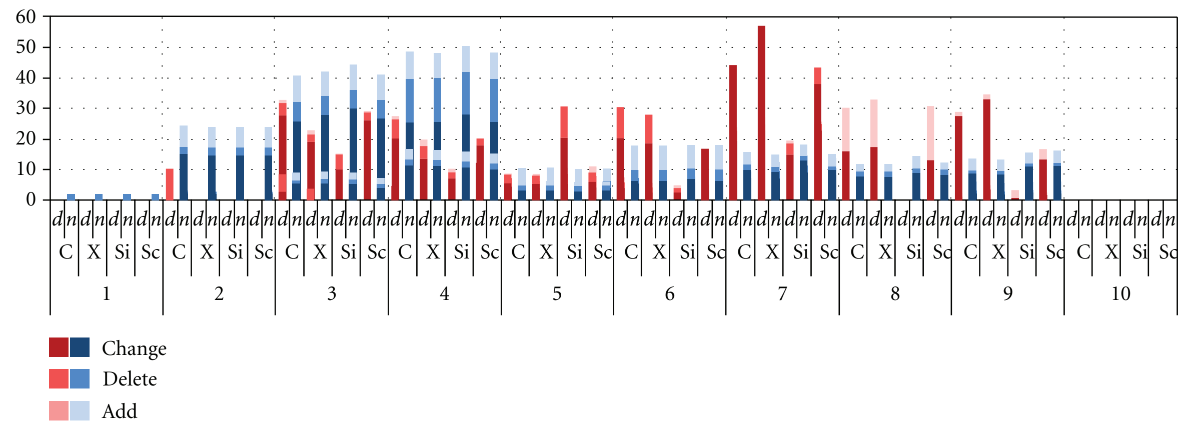
Figure 6(c) shows the opposite result of Figure 6(b). That is, duplicate code was modified more frequently in the posterior half of the period. In the anterior half, the amount of duplication was very small, and modifications were rarely performed on it. In the posterior half, amount of duplicate code became large, and modifications were performed on it repeatedly. In the case of `Simian` detection, no duplicate code was detected except period "5." This is because `Simian`



(a) AdServerBeans



(b) OpenYMSG



(c) NatMonitor

FIGURE 6: Result of Item B on Experiment 1.2 (divided into 10 periods).

detects only the exact-match duplicate code meanwhile the other tools detect exact match and renamed duplicate code in the default setting.

In Experiment 1.1, we investigate MF tendencies at the release level. However, we cannot apply the same investigation way to Experiment 1.2. This is because the target software systems in Experiment 1.2 is not enough mature to have multiple releases. Instead, we investigate MF tendencies at the most fine-grained level, at the revision level. Figure 7 shows the result of the investigation at the revision level for AdServerBeans, OpenYMSG, and NatMonitor. The X-axis of each graph indicates the value of $\mathrm{MF}_d - \mathrm{MF}_n$. Therefore, if the value is greater than 0, MF_d is greater than MF_n at the revision and vice versa. For AdServerBeans, MF tendencies are similar for every detection tool except revision 21 to 26. For other 2 software systems, MF comparison results differ

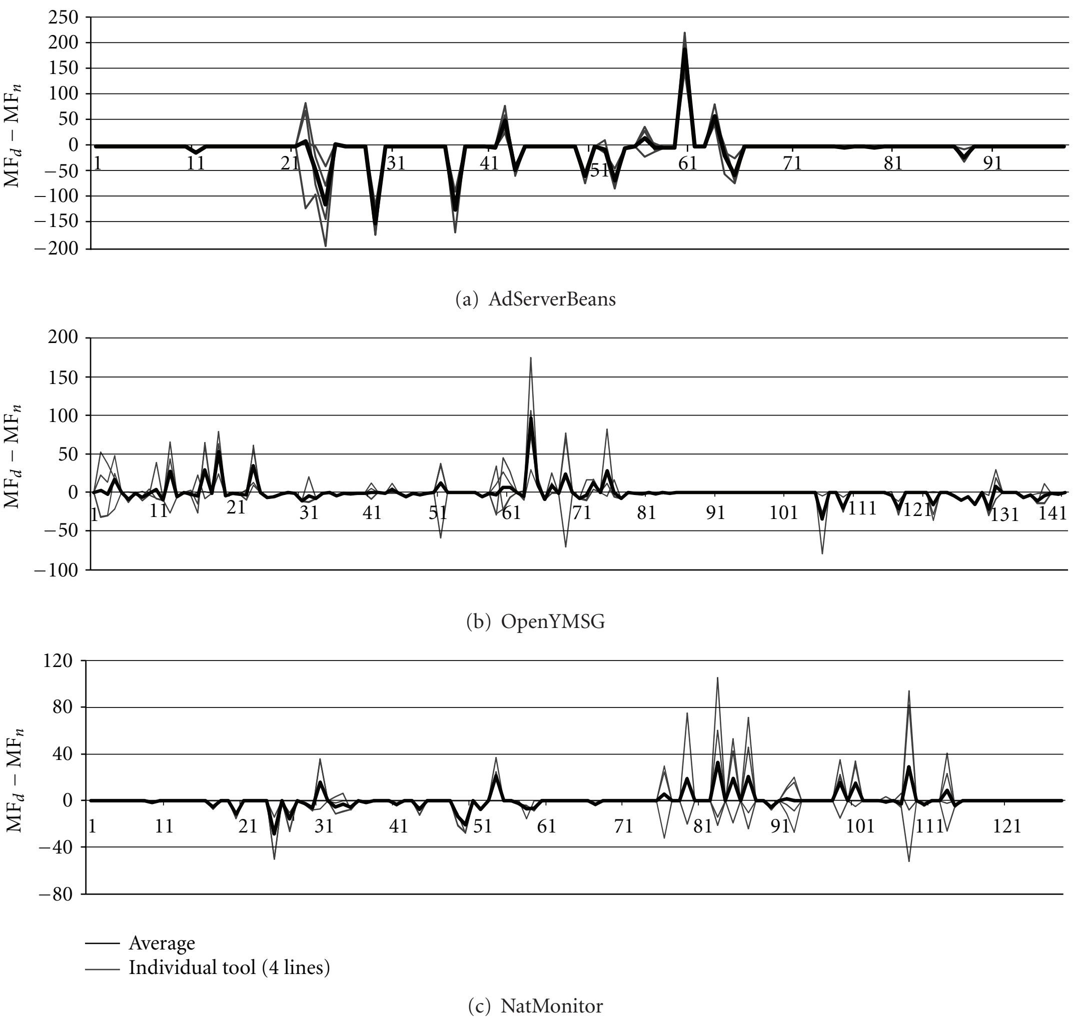


Figure 7: Result of Item B on Experiment 1-2 (for each revision).

from each detection tool in most of the revisions. As the figures show, tendencies of MF transition differ from clone detectors nevertheless there seems to be small differences between clone detectors in 10 sub-periods division. However, these graphs do not consider modification types. Therefore, we cannot judge what type of modification frequently occurred from the graphs.

The summary of Experiment 1.2 is as follows: we found some instances that duplicate code was modified more frequently than nonduplicate code in a short period on each detection tool; however, in the entire period, duplicate code was modified less frequently than nonduplicate code on every target software with all the detection tools. Consequently, we conclude that the presence of duplicate code does not have a seriously-negative impact on software evolution.

6.4. Answers for RQs

RQ1: Is duplicate code more frequently modified than nonduplicate code? The answer is *No*. In Experiment 1.1, we found that MF_d is lower than MF_n in all the target systems. Also, we found a similar result in Experiment 1.2: 22 comparison results out of 35 show that MF_d is lower than MF_n, also MF_d is lower than MF_n in average. This result indicates that the presence of duplicate code does not seriously affect software evolution, which is different from the common belief.

RQ2: Are the comparison results of stability between duplicate code and nonduplicate code different from multiple detection tools? The answer is *Yes*. In Experiment 1.2, the comparison results with `CCFinderX` are different from the results with other 3 detectors. Moreover, MF_n is much greater than MF_d in the case of `Simian`. At present, we cannot find the causes of the difference of the comparison results. One of the causes may be the ratio of duplicate code. The ratio of duplicate code is quite different for each detection tool on the same software. However, we cannot see any relation between the ratio of duplicate code and MF.

RQ3: Is duplicate code modified uniformly throughout its lifetime? The answer is *No*. In Item B of Experiments 1.1

and 1.2, there are some instances that duplicate code was modified more frequently than nonduplicate code in a short period though MF_d is less than MF_n in the whole period. However, these MFs tendencies depend on target software systems, so that we cannot find characteristics of such variability.

RQ4: Are there any differences in the comparison results on modification types? The answer is *Yes*. In Experiment 1.1, MF_d is less than MF_n on all the modification types. However, there is a small difference between MF_d and MF_n in the case of deletion, meanwhile there is a large difference in the case of addition. In Experiment 1.2, MF_d is less than MF_n in the cases of change and addition. Especially, MF_n is more than twice as large as MF_d in the case of addition. However, MF_d is greater than MF_n in the case of deletion. These results show that deletion tends to be affected by duplicate code, meanwhile addition tends not to be affected by duplicate code.

6.5. Discussion. In Experiment 1, we found that duplicate code tends to be more stable than nonduplicate code, which indicates that the presence of duplicate code does not have a negative impact on software evolution. We investigated how the software evolved in the period, and we found that the following activities should be a part of factors that duplicate code is modified less frequently than nonduplicate code.

Reusing Stable Code. When implementing new functionalities, reusing stable code is a good way to reduce the number of introduced bugs. If most of duplicate code is reused stable code, MF_d becomes less than MF_n.

Using Generated Code. Automatically generated code is rarely modified manually. Also, the generated code tends to be duplicate code. Consequently, if the amount of generated code is high, MF_d will become less than MF_n.

On the other hand, there are some cases that duplicate code was more frequently modified than nonduplicate code in a short period. The period "7" on AdServerBeans (Experiment 1.2, Item B) is one of these instances. We analyzed the source code of this period to detect why MF_d was greater than MF_n in this period though the opposite results were shown in the other periods. Through the analysis, we found that there are some instances that the same modifications were applied to multiple places of code.

Algorithm 2 shows an example of unstable duplicate code. There are 5 code fragments that are similar to this fragment. Firstly, lines labeled with "%" (shown in Algorithm 2(b)) were modified to replace the getter methods into directly accesses to fields. In the next, a line labeled with "#" is removed (shown in Algorithm 2(c)). These two modifications were concentrically conducted in period "7." Reusing unstable code like this example can cause additional costs for software maintenance. Moreover, a code fragment was not simultaneously changed with its correspondents at the second modification. If this inconsistent change was introduced unintentionally, it might cause a bug. If so, this is a typical situation that duplicate code affects software evolution.

Table 10: Ratio of duplicate code—Experiment 2.

Software Name	ccf	ccfx	sim	sco
OpenYMSG	12.4%	6.2%	2.7%	5.5%
EclEmma	6.9%	4.8%	2.0%	3.7%
MASU	25.6%	26.5%	11.3%	15.4%
TVBrowser	13.6%	10.9%	5.4%	19.0%
Ant	13.9%	12.1%	6.2%	15.6%

Table 11: Overall results—Experiment 2.

Software Name	Method	Tools			
		ccf	ccfx	sim	sco
OpenYMSG	Proposed	N	C	C	N
	Krinke	N	C	C	N
	Lozano	—	—	N	—
EclEmma	Proposed	N	N	N	N
	Krinke	N	N	N	C
	Lozano	N	N	—	—
MASU	Proposed	C	N	C	C
	Krinke	C	C	C	C
	Lozano	C	C	C	C
TVBrowser	Proposed	N	N	N	N
	Krinke	C	C	C	C
	Lozano	C	C	C	C
Ant	Proposed	N	N	N	N
	Krinke	C	C	C	C
	Lozano	C	C	C	C

7. Experiment 2—Result and Discussion

7.1. Overview. Table 10 shows the average ratios of duplicate code in each target, and Table 11 shows the comparison results of all the targets. In Table 11, "C" means that duplicate code requires more cost than nonduplicate code, and "N" means its opposite. The discriminant criteria of "C" and "N" are different in each investigation method.

In the proposed method, if MF_d is lower than MF_n, the column is labeled with "C," and the column is labeled with "N" in its opposite case.

In Krinke's method, if the ratio of *changed* and *deleted* lines of code on duplicate code is greater than *changed* and *deleted* lines on nonduplicate code, the column is labeled with "C," and in its opposite case the column is labeled with "N." Note that herein we do not consider *added* lines because the amount of *add* is the lines of code added in the next revision, not in the current target revision.

In Lozano's method, if *work* in AC-Method is statistically greater than one in NC-Method, the column is labeled with "C." On the other hand, if *work* in NC-Method is statistically greater than one in AC-Method, the column is labeled with "N." Here, we use Mann-Whitney's U test under setting

```
(a) Before Modification
    int offsetTmp = dataGridDisplayCriteria
             .getItemsPerPage() *
                  (dataGridDisplayCriteria.getPage() -1);
        if (offsetTmp > 0) --offsetTmp;
        if (offsetTmp < 0) offsetTmp = 0;
    final int offset = offsetTmp;
     String sortColumn =
              dataGridDisplayCriteria.getSortColumn();
    Order orderTmp =
             dataGridDisplayCriteria.getOrder()
                      .equals(AdServerBeansConstants.ASC) ?
                                Order.asc(sortColumn) :
                                      Order.desc(sortColumn);

(b) After 1st Modification
    int offsetTmp = dataGridDisplayCriteria
             .getItemsPerPage() *
                    (dataGridDisplayCriteria.getPage() -1);
        if (offsetTmp > 0) --offsetTmp;
        if (offsetTmp < 0) offsetTmp = 0;
    final int offset = offsetTmp;
    String sortColumn =
%             dataGridDisplayCriteria.sortColumn;
    Order orderTmp =
%                 dataGridDisplayCriteria.order
                          .equals(AdServerBeansConstants.ASC) ?
                                  Order.asc(sortColumn) :
                                        Order.desc(sortColumn);

(c) After 2nd Modification
    int offsetTmp = dataGridDisplayCriteria
             .getItemsPerPage() *
                     (dataGridDisplayCriteria.getPage() -1);
#
        if (offsetTmp < 0) offsetTmp = 0;
    final int offset = offsetTmp;
    String sortColumn =
             dataGridDisplayCriteria.sortColumn;
    Order orderTmp =
             dataGridDisplayCriteria.order
                      .equals(AdServerBeansConstants.ASC) ?
                               Order.asc(sortColumn) :
                                     Order.desc(sortColumn);
```

Algorithm 2: An example of unstable duplicate code.

5% as the level of significance. If there is no statistically significant difference in AC- and NC-Method, we compare *work* in duplicate period and nonduplicate period in SC-Method with Wlcoxon's singed-rank test. We also set 5% as the level of significance. If there is no statistically significant difference, the column is labeled with "—."

As this table shows, different methods and different tools brought almost the same result in the case of EclEmma and MASU. On the other hand, in the case of other targets, we get different results with different methods or different tools. Especially, in the case of TVBrowser and Ant, the proposed method brought the opposite result to Lozano's and Krinke's method.

7.2. Result of MASU. Herein, we show comparison figures of MASU. Figure 8 shows the results of the proposed method. In this case, all the detection tools except `CCFinderX` brought the same result that duplicate code is more frequently modified than nonduplicate code. Figure 9 shows the results of Krinke's method on MASU. As this figure shows, the comparison of all the detection detectors brought the same result that duplicate code is less stable than nonduplicate code. Figure 10 shows the results of Lozano's method on MASU with `Simian`. Figure 10(a) compares AC-Method and NC-Method. X-axis indicates maintenance cost (*work*) and Y-axis indicates cumulated frequency of methods. For readability, we adopt logarithmic axis on X-axis. In this

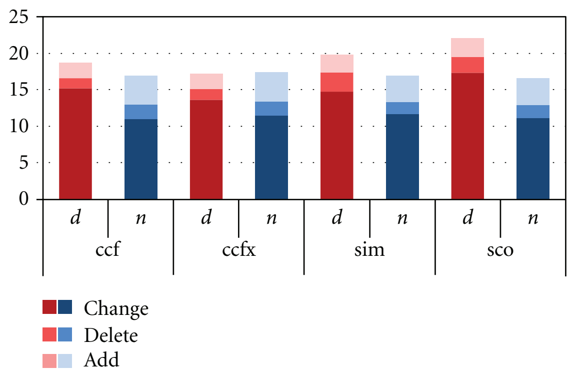


Figure 8: Result of the proposed method on MASU.

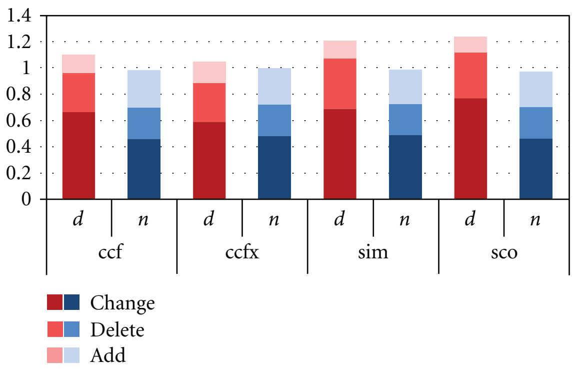
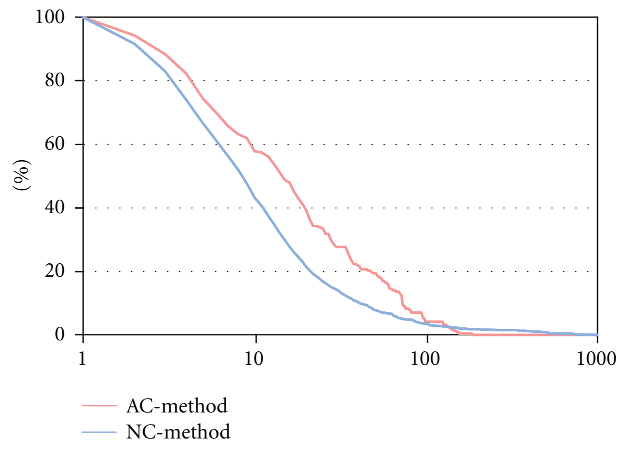
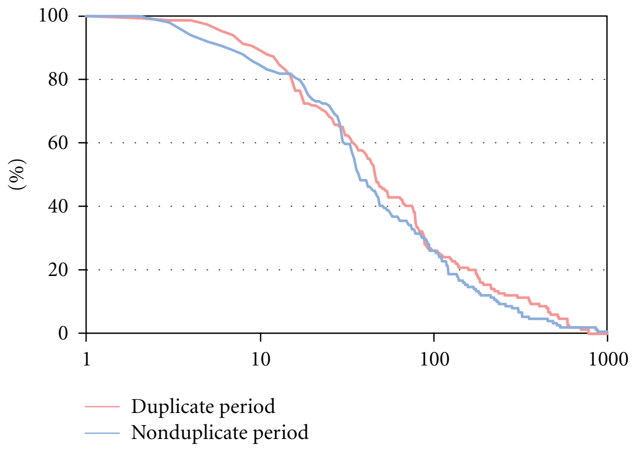


Figure 9: Result of Krinke's method on MASU.



(a) AC-Method versus NC-Method



(b) SC-Method

Figure 10: Result of Lozano's Method on MASU with `Simian`.

case, AC-Method requires more maintenance cost than NC-Method. Also, Figure 10(b) compares duplicate period and nonduplicate period of SC-Method. In this case, the maintenance cost in duplicate period is greater than in nonduplicate period.

In the case of MASU, Krinke's method and Lozano's method regard duplicate code as requiring more cost than nonduplicate code in the cases of all the detection tools. The proposed method indicates that duplicate code is more frequently modified than nonduplicate code with `CCFinder`, `Simian`, and `Scorpio`. In addition, there is little differences between MF_d and MF_n in the result of the proposed method with `CCFinderX`, which is the only case that duplicate code is more stable than nonduplicate code. Considering all the results, we can say that duplicate code has a negative impact on software evolution on MASU. This result is reliable because all the investigation methods show such tendencies.

7.3. Result of OpenYMSG. Figures 11, 12, and 13 show the result of the proposed method, Krinke's method, and Lozano's method on OpenYMSG. In the cases of the proposed method and Krinke's method, duplicate code is regarded as having a negative impact with `CCFinderX` and `Simian`, meanwhile the opposing results are shown with `CCFinder` and `Scorpio`. In Lozano's method with `Simian`, duplicate code is regarded as not having a negative impact. Note that we omit the comparison figure on SC-Method because there are only 3 methods that are categorized into SC-Method.

As these figures show, the comparison results are different for detection tools or investigation methods. Therefore, we cannot judge whether the presence of duplicate code has a negative impact or not on OpenYMSG.

7.4. Discussion. In the case of OpenYMSG, TVBrowser, and Ant, different investigation methods and different tools brought opposing results. Figure 14 shows an actual modification in Ant. Two methods were modified in this modification. The hatching parts are detected duplicate code and frames in them mean pairs of duplicate code between two methods. Vertical arrows show modified lines between this modification and the next (77 lines of code were modified).

This modification is a refactoring, which extracts the duplicate instructions from the two methods and merges them as a new method. In the proposed method, there are 2 modification places in duplicate code and 4 places in nonduplicate code, so that MF_d and MF_n become 51.13 and 18.13, respectively. In Krinke's method, DC + CC and

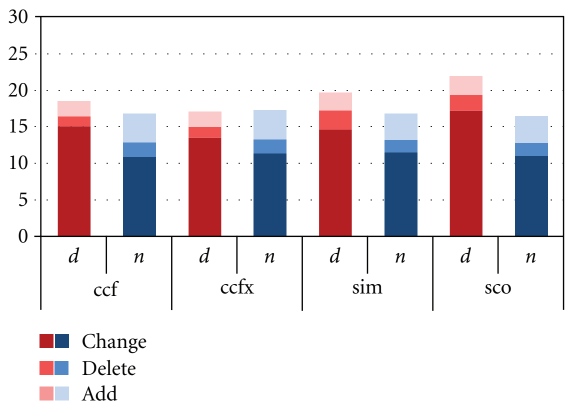


Figure 11: Result of the proposed method on OpenYMSG.

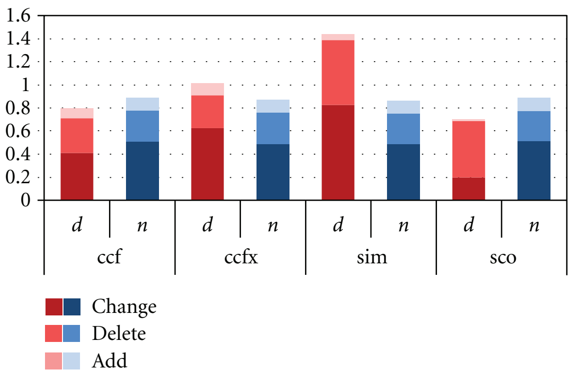


Figure 12: Result of Krinke's method on OpenYMSG.

DN + CN become 0.089 and 0.005, where DC, CC, DN, and CN indicate the ratio of deleted lines on duplicate code, changed lines on duplicate code, deleted lines on nonduplicate code, and deleted lines on nonduplicate code, respectively.

In this case, both the proposed method and Krinke's method regard duplicate code requiring more maintenance cost than nonduplicate code. However, there is a great difference in Krinke's method than the proposed method: in the proposed method, duplicate code is modified about 2.8 times as frequently as nonduplicate code; meanwhile, in Krinke's method, duplicate code is modified 17.8 times as frequently as nonduplicate code. This is caused by the difference of the barometers used in each method. In Krinke's method, the barometer depends on the amount of modified lines, meanwhile the barometer depends on the amount of modified places in the proposed method. This example is one of the refactorings on duplicate code. In Krinke's method, if removed duplicate code is large, duplicate code is regarded as having more influence. However, in the cases of duplicate code removal, we have to spend much effort if the number of duplicate fragments is high. Therefore, we can say that the proposed method can accurately measure the influence of duplicate code in this case.

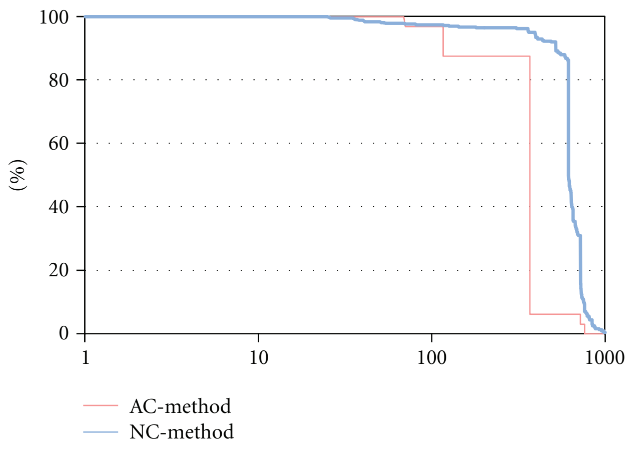


Figure 13: Result of Lozano's method on OpenYMSG with `Simian`.

This is an instance that is advantageous for the proposed method. However, we cannot investigate all the experimental data because the amount of the data is too vast to conduct manual checking for all the modifications. There is a possibility that the proposed method cannot accurately evaluate the influence of duplicate code in some situations.

In Experiment 2, we found that the different investigation methods or different detectors draw different results on the same target systems. In Experiment 1, we found that duplicate code is less frequently modified than nonduplicate code. However, the result of Experiment 2 shows that we cannot generalize the result of Experiment 1. We have to conduct more experiments and analyze the results of them in detail to gain more generic.

8. Threats to Validity

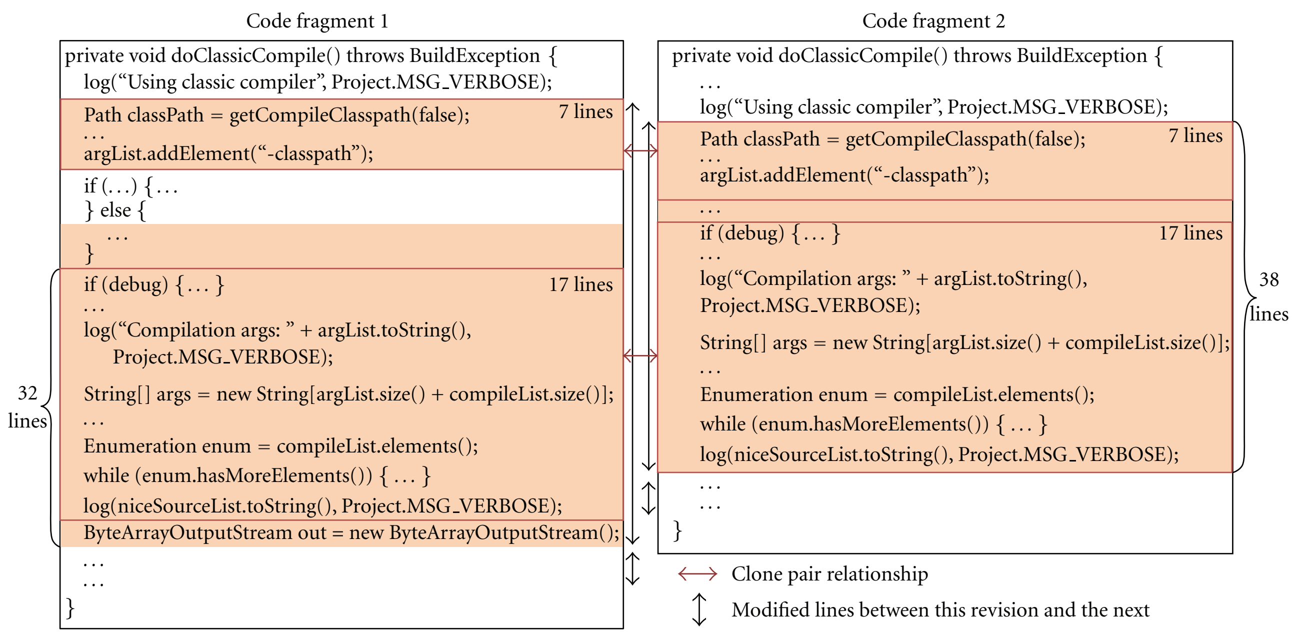
This section describes threats to validity of this study.

8.1. Features of Every Modification.

In this study, we assume that cost required for every modification is equal to one another. However, the cost is different between every modification in the actual software evolution. Consequently, the comparison based on MF may not appropriately represent the cost required for modifying duplicate code and nonduplicate code.

Also, when we modify duplicate code, we have to consider maintaining the consistency between the modified duplicate code and its correspondents. If the modification lacks the consistency by error, we have to remodify them for repairing the consistency. The effort for consistency is not necessary for modifying nonduplicate code. Consequently, the average cost required for duplicate code may be different from the one required for nonduplicate code. In order to compare them more appropriately, we have to consider the cost for maintaining consistency.

Moreover, distribution of source code that should be modified are not considered. However, it differs from



Code fragment 1

```
private void doClassicCompile() throws BuildException {
   log("Using classic compiler", Project.MSG_VERBOSE);
   Path classPath = getCompileClasspath(false);
   ...
   argList.addElement("-classpath");
   if (...) {...
   } else {
      ...
   }
   if (debug) {... }
   ...
   log("Compilation args: " + argList.toString(),
       Project.MSG_VERBOSE);
   String[] args = new String[argList.size() + compileList.size()];
   ...
   Enumeration enum = compileList.elements();
   while (enum.hasMoreElements()) { ... }
   log(niceSourceList.toString(), Project.MSG_VERBOSE);
   ByteArrayOutputStream out = new ByteArrayOutputStream();
   ...
   ...
}
```

Code fragment 2

```
private void doClassicCompile() throws BuildException {
   ...
   log("Using classic compiler", Project.MSG_VERBOSE);
   Path classPath = getCompileClasspath(false);
   ...
   argList.addElement("-classpath");
   ...
   if (debug) {... }
   ...
   log("Compilation args: " + argList.toString(),
   Project.MSG_VERBOSE);
   String[] args = new String[argList.size() + compileList.size()];
   ...
   Enumeration enum = compileList.elements();
   while (enum.hasMoreElements()) { ... }
   log(niceSourceList.toString(), Project.MSG_VERBOSE);
   ...
   ...
}
```

FIGURE 14: An Example of Modification.

every modification, thus we may get different results by considering the distribution of source code.

8.2. Identifying the Number of Modifications. In this study, modifying consecutive multiple lines are regarded as a single modification. However, it is possible that such an automatically processing identifies the incorrect number of modifications. If multiple lines that were not contiguous are modified for fixing a single bug, the proposed method presumes that multiple modifications were performed. Also, if multiple consecutive lines were modified for fixing two or more bugs by chance, the proposed method presumes that only a single modification was performed. Consequently, it is necessary to manually identify modifications if we have to use the exactly correct number of modifications.

Besides, we investigated how many the identified modifications occurred across the boundary of duplicate code and nonduplicate code. If this number is high, then the analysis suspects because such modifications increase both the counts at the same time. The investigation result is that, in the highest case, the ratio of such modifications is 4.8%. That means that almost all modifications occurred within either duplicate code or nonduplicate code.

8.3. Category of Modifications. In this study, we counted all the modifications, regardless of their categories. As a result, the number of modifications might be incorrectly increased by unimportant modifications such as format transformation. A part of unimportant modifications remained even if we had used the normalized source code described in Section 4.2.2. Consequently, manual categorization for the modifications is required for using the exactly correct number of modifications.

Also, the code normalization that we used in this study removed all the comments in the source files. If considerable cost was expended to make or change code comments on the development of the target systems, we incorrectly missed the cost.

8.4. Property of Target Software. In this study, we used only open-source software systems, so that different results may be shown with industrial software systems. Some researchers pointed out that industrial software systems include much duplicate code [24, 25]. Consequently, duplicate code may not be managed well in industrial software, which may increase MF_d. Also, properties of industrial software are quite different from ones of open source software. In order to investigate the impact of duplicate code on industrial software, we have to compare MF on industrial software itself.

8.5. Settings of Detection Tools. In this study, we used default settings for all the detection tools. If we change the settings, different results will be shown.

9. Conclusion

This paper presented an empirical study on the impact of the presence of duplicate code on software evolution. We assumed that if duplicate code is modified more frequently than nonduplicate code, the presence of duplicate code affects software evolution and compared the stability of duplicate code and nonduplicate code. To evaluate from a different standpoint from previous studies, we used a new indicator, modification frequency, which is calculated with the number of modified places of code. Also, we used 4 duplicate code detection tools to reduce the bias of duplicate code detectors. We conducted an experiment on 15 open-source software systems, and the result showed that duplicate code was less frequently modified than nonduplicate code. We also found some cases that duplicate code was intensively modified in a short period though duplicate code was stable than nonduplicate code in the whole development period.

Moreover, we compared the proposed method to other 2 investigation methods to evaluate the efficacy of the proposed method. We conducted an experiment on 5 open-source software systems, and in the cases of 2 targets, we got the opposing results to other 2 methods. We investigated the result in detail and found some instances that the proposed method could evaluate more accurately than other methods.

In this study, we found that duplicate code tends to be stable than nonduplicate code. However, more studies are required to generalize this result, because we found that different investigation methods may bring different results. As future work, we are going to conduct more studies with other settings to get the characteristics of harmful duplicate code.

Acknowledgments

This research is being conducted as a part of the Stage Project, the Development of Next Generation IT Infrastructure, supported by the Ministry of Education, Culture, Sports, Science, and Technology of Japan. This study has been supported in part by Grant-in-Aid for Scientific Research (A) (21240002) and Grant-in-Aid for Exploratory Research (23650014) from the Japan Society for the Promotion of Science, and Grant-in-Aid for Young Scientists (B) (22700031) from Ministry of Education, Science, Sports and Culture. This paper is an extended version of earlier conference papers [26, 27].

References

[1] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Transactions on Software Engineering*, vol. 32, no. 10, pp. 804–818, 2007.

[2] M. D. Wit, A. Zaidman, and A. V. Deursen, "Managing code clones using dynamic change tracking and resolution?" in *Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM '09)*, pp. 169–178, September 2009.

[3] M. P. Robillard, W. Coelho, and G. C. Murphy, "How effective developers investigate source code: an exploratory study," *IEEE Transactions on Software Engineering*, vol. 30, no. 12, pp. 889–903, 2004.

[4] C. J. Kapser and M. W. Godfrey, "'cloning considered harmful' considered harmful: patterns of cloning in software," *Empirical Software Engineering*, vol. 13, no. 6, pp. 645–692, 2008.

[5] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy, "An empirical study of code clone genealogies," in *Proceedings of the 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 187–196, September 2005.

[6] N. Bettenburg, W. Shang, W. M. Ibrahim, B. Adams, Y. Zou, and A. E. Hassan, "An empirical study on inconsistent changes to code clones at the release level," *Science of Computer Programming*, vol. 77, no. 6, pp. 760–776, 2012.

[7] A. Monden, D. Nakae, T. Kamiya, S. Sato, and K. Matsumoto, "Software quality analysis by code clones in industrial legacy software," in *Proceedings of the 8th IEEE International Software Metrics Symposium*, pp. 87–94, June 2002.

[8] A. Lozano, M. Wermelinger, and B. Nuseibeh, "Evaluating the harmfulness of cloning: a change based experiment," in *Proceedings of the 4th International Workshop on Mining Software Repositories (MSR '07)*, May 2007.

[9] A. Lozano and M. Wermelinger, "Assessing the effect of clones on changeability," in *Proceedings of the 24th International Conference on Software Maintenance*, pp. 227–236, September 2008.

[10] A. Lozano, M. Wermelinger, and B. Nuseibeh, "Evaluating the relation between changeability decay and the characteristics of clones and methods," in *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, pp. 100–109, September 2008.

[11] J. Krinke, "Is cloned code more stable than non-cloned code?" in *Proceedings of the 8th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2008*, pp. 57–66, September 2008.

[12] N. Göde and J. Harder, "Clone stability," in *Proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR '11)*, pp. 65–74, March 2011.

[13] J. Krinke, "Is cloned code older than non-cloned code?" in *Proceedings of the 5th International Workshop on Software Clones (IWSC '11)*, pp. 28–33, May 2011.

[14] F. Rahman, C. Bird, and P. Devanbu, "Clones: what is that smell?" in *Proceedings of the 7th IEEE Working Conference on Mining Software Repositories*, pp. 72–81, May 2010.

[15] N. Göde and R. Koschke, "Frequency and risks of changes to clones," in *33rd International Conference on Software Engineering (ICSE '11)*, pp. 311–320, May 2011.

[16] S. G. Eick, T. L. Graves, A. F. Karr, U. S. Marron, and A. Mockus, "Does code decay? Assessing the evidence from change management data," *IEEE Transactions on Software Engineering*, vol. 27, no. 1, pp. 1–12, 2001.

[17] N. Göde, "Evolution of type-1 clones," in *Proceedings of the 9th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM '09)*, pp. 77–86, September 2009.

[18] E. Burd and J. Bailey, "Evaluating clone detection tools for use during preventative maintenance," in *Proceedings of the 2nd IEEE International Workshop on Source Code Analysis and Manipulation*, pp. 36–43, October 2002.

[19] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[20] CCFinderX, http://www.ccfinder.net/ccfinderx.html/.

[21] Simian, http://www.harukizaemon.com/simian/.

[22] Y. Higo and S. Kusumoto, "Code clone detection on specialized PDGs with heuristics," in *Proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR '11)*, pp. 75–84, March 2011.

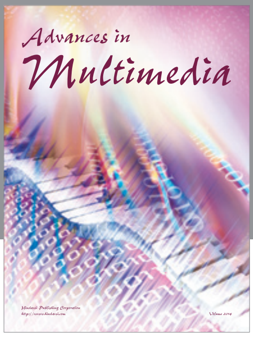
[23] Scorpio, http://sdl.ist.osaka-u.ac.jp/~higo/cgi-bin/moin.cgi/Scorpio/.

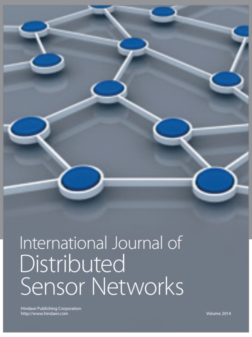
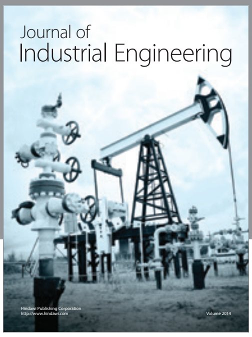
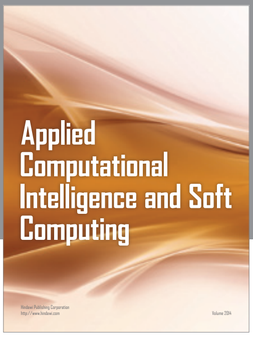
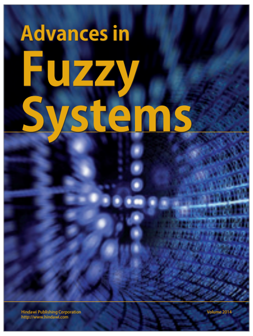
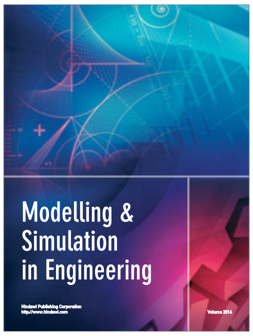
[24] S. Ducasse, M. Rieger, and S. Demeyer, "Language independent approach for detecting duplicated code," in *Proceedings of the 15th IEEE International Conference on Software Maintenance (ICSM '99)*, pp. 109–118, September 1999.

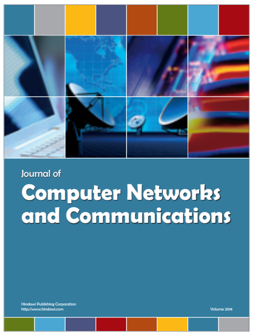
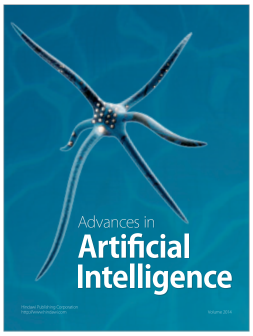
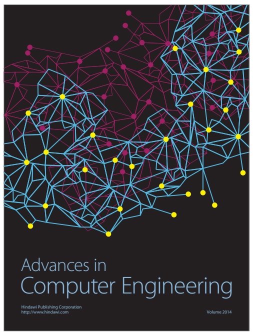
[25] S. Uchida, A. Monden, N. Ohsugi, T. Kamiya, K. I. Matsumoto, and H. Kudo, "Software analysis by code clones in open source software," *Journal of Computer Information Systems*, vol. 45, no. 3, pp. 1–11, 2005.

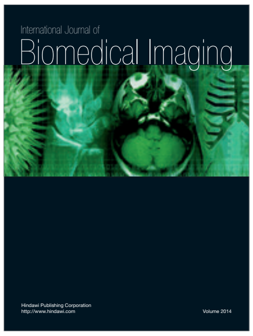
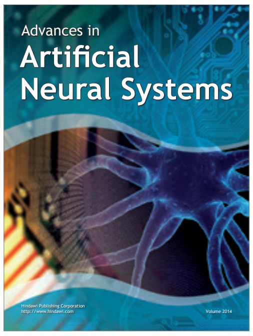
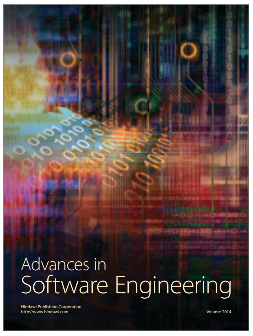
[26] K. Hotta, Y. Sano, Y. Higo, and S. Kusumoto, "Is duplicate code more frequently modified than non-duplicate code in software evolution?: an empirical study on open source software," in *Proceedings of the 4th the International Joint ERCIM Workshop on Software Evolution and International Workshop on Principles of Software Evolution*, September 2010.

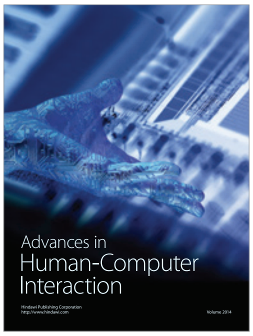
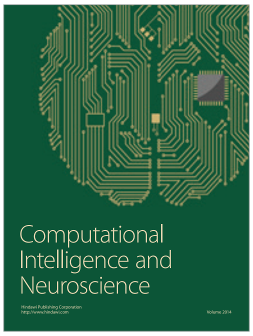
[27] Y. Sasaki, K. Hotta, Y. Higo, and S. Kusumoto, "Is duplicate code good or bad? an empirical study with multiple investigation methods and multiple detection tools," in *Proceedings of the 22nd International Symposium on Software Reliability Engineering (ISSRE '11)*, Hiroshima, Japan, November 2011.


Advances in
Multimedia

The Scientific World Journal

International Journal of
Distributed Sensor Networks

Journal of
Industrial Engineering

Applied Computational Intelligence and Soft Computing

Advances in
Fuzzy Systems

Modelling & Simulation in Engineering


Journal of
Computer Networks and Communications

Advances in
Artificial Intelligence

Advances in
Computer Engineering

International Journal of
Computer Games Technology

International Journal of
Biomedical Imaging

Advances in
Artificial Neural Systems

Advances in
Software Engineering

Journal of
Robotics

Advances in
Human-Computer Interaction

Computational Intelligence and Neuroscience

International Journal of
Reconfigurable Computing

Journal of
Electrical and Computer Engineering