Is Duplicate Code More Frequently Modified than Non-duplicate Code in Software Evolution?: An Empirical Study on Open Source Software

Keisuke Hotta[1] Yukiko Sano[1] Yoshiki Higo[1] Shinji Kusumoto[1]
[1]Graduate School of Information Science and Technology, Osaka University
{k-hotta,y-sano,higo,kusumoto}@ist.osaka-u.ac.jp

ABSTRACT

Various kinds of research efforts have been performed on the basis that the presence of duplicate code has a negative impact on software evolution. A typical example is that, if we modify a code fragment that has been duplicated to other code fragments, it is necessary to consider whether the other code fragments have to be modified simultaneously or not. In this research, in order to investigate how much the presence of duplicate code is related to software evolution, we defined a new indicator, **modification frequency**. The indicator is a quantitative measure, and it allows us to objectively compare the maintainability of duplicate code and non-duplicate code. We conducted an experiment on 15 open source software systems, and the result showed that the presence of duplicate code does not have a negative impact on software evolution.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*Complexity measures, Process metrics, Software science*; D.2.9 [**Software Engineering**]: Management—*Productivity, Software quality assurance*

General Terms

Measurement, Management

Keywords

Duplicate code, Software maintenance, Empirical study

1. INTRODUCTION

Recently, duplicate code is attracting a great deal of attention in software engineering. Duplicate code is generated by various reasons such as copy and paste programming or stereotyped code. The presence of duplicate code makes it more difficult to maintain consistency of the code. For example, if a bug is found in a duplicate code, we have to fix the same bug in its correspondents. In a book dedicated to refactoring, Fowler stated that:

Number one in the stink parade is duplicate code. If you see the same code structure in more than one place, you can be sure that your program will be better if you find a way to unify them [10].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
IWPSE-EVOL'10 September 20-21, 2010 Antwerp, Belgium
Copyright 2010 ACM 978-1-4503-0128-2/10/09 ...$10.00.

Various kinds of research efforts have been performed for resolving or improving the problems caused by the presence of duplicate code [2]. For example, there are currently a variety of duplicate code detection techniques available [5]. In addition, there are many research efforts for merging duplicate code as a single function or method, or for preventing duplications from being overlooked in modification [8, 21]. However, only a few research efforts have investigated how much the presence of duplicate code actually has a negative impact on software evolution.

Monden et al. and Lozano et al. investigated the influence of duplicate code on file unit or method unit, respectively [19, 20]. They found some case where the presence of duplicate code has a negative impact on software maintenance. However, their investigations remain a matter of improvement. The units (files and methods) are larger than duplicate code, so that it is possible that modifications are incorrectly counted. For example, if modifications are performed on a method where a fragment of duplicate code exists, all the modifications are assumed as performed on the duplicate code even if they are performed on non-duplicate code of the method. Krinke compared stability of duplicate code and non-duplicate code [16]. In his experiment, a line-based barometer was used. However, it was not possible that the barometer appropriately indicates the cost of maintainability. That is, the indicator cannot distinguish the following two cases: the first case is that *consecutive 10 lines of code was modified for fixing a bug*; the second case is that *1 line modification was performed on different 10 pieces 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 the buggy module, informing the maintainer about the bug, identifying buggy instructions and so on.

The present paper proposes a new investigation method, which compares duplicate code to non-duplicate code from a different standpoint, and reports the experimental result on open source software. The features of the proposed method are as follows:

- every line of code is investigated whether it is duplicate code or not. Such a fine-grained investigation can accurately judge whether every modification is conducted to duplicate code or to non-duplicate code;
- maintenance cost consists of not only source code modification but also several phases prior to the source code modification. 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 pieces;
- we evaluate and compare modifications of duplicate code and non-duplicate 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.

The remainder of this paper is organized as follows: Section 2 describes the previous studies on modification comparison between duplicate code and non-duplicate code. Section 3 introduces detection tools that are used in this study, and Section 4 explains how we compare modifications between duplicate code and non-duplicate code. Section 5 shows the comparison result, and Section 6 discusses threat to validity of the experimental result. Finally, Section 7 concludes this paper.

2. RELATED WORK

At present, there is a huge body of work on empirical evidence on code clones, starting with Kim et al.'s report on clone genealogies [15]. Their empirical studies on two open source software systems 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 [14]. 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 source systems. The empirical study 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 [20]. In their investigation, the number of revisions of every file was used 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 [18]. They developed a software 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 the result showed that: duplicate methods tend to be more frequently modified than non-duplicate methods; however, duplicate methods tend to be modified less simultaneously than non-duplicate 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 [17]. Three barometers were used in the investigation. The first one is *likelihood*, which indicates 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 [19]. Changeability is the ease of modification. If changeability is decreased, it will be a bottleneck of software maintenance. The result showed that it is possible that the presence of duplicate code decreases changeability. However 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 non-duplicate code, maintenance cost for duplicate code is greater than for non-duplicate code, and he conducted a case study in order to investigate whether the hypothesis is true or not [16]. The targets are 200 revisions (a revision per week) of source code of 5 large scale open source systems. He measured *added*, *deleted*, and *changed* LOCs on duplicate code and non-duplicate code, and compared them. The result showed that non-duplicate 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.

Eick et al. investigated whether source code decays when it is operated and maintained for a long time [9]. 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-years-operated large system showed that cost required for completing a single requirement tends to increase.

Göde modeled how type-1 code clones are generated and how they evolved [11]. 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 are 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.

As described above, some empirical studies reported that duplicate code should have a negative impact on software evolution meanwhile the others reported that it should not. 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.

3. DUPLICATE CODE DETECTION TOOLS

There are currently various kinds of duplicate code detection tools. The detection tools take the source code, and they provide the position of the detected duplicate code in it. 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.

The detection tools can be categorized based on their detection

```
 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

Figure 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)

techniques. Major categories should be line-based, token-based, metrics-based, AST[1]-based, and PDG[2]-based. Each technique has merits and demerits, and there is no technique that is superior to any other techniques in every way [5, 7]. The remainder of this section describes 4 detection tools that are used in our experiment.

3.1 CCFinder

CCFinder is a token-based detection tool [13]. The major features of CCFinder are as follows:

- 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.
- CCFinder detection speed is very fast. CCFinder can detects duplicate code from millions lines of code within an hour.
- CCFinder can handle multiple popular programming languages such as C/C++, Java, and COBOL.

3.2 CCFinderX

CCFinderX is a major version up from CCFinder [1]. CCFinderX is a token-based detection tool as well as CCFinder although the detection algorithm was changed to *bucket sort* from *suffix tree*. CCFinderX can handle more programming languages than CCFinder. Moreover, it can effectively use resources of multi-core CPUs for faster duplicate code detection.

3.3 Simian

Simian is a line-based detection tool [4]. As well as CCFinder family, Simian can handle multiple programming languages. Line-based techniques realized duplicate code detection on small memory usage and short running time. Also, Simian allows very fine-grained settings. For example, we can configure that duplicate code is not detected from import statements in the case of Java language.

3.4 Scorpio

Scorpio is a PDG-based detection tool [3, 12]. Scorpio builds a special PDG for duplicate code detection, not traditional one, in which there are two types of edge representing data dependency and control dependency. The special PDG has one more edge, execution dependency. The execution dependency 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.

[1]Abstract Syntax Tree
[2]Program Dependency Graph

4. MEASURING MODIFICATION FREQUENCY

This section proposes a method to estimate the influence of the presence of duplicate code on software evolution. The influence estimation is performed by a relative comparison of cost required for maintaining duplicate code and non-duplicate code. The proposed comparison has a different standpoint from the previous studies described in Section 2. The comparison is performed by using a new indicator, **modification frequency** (in short, **MF**). The MF of duplicate code (in short, MF_d) and the MF of non-duplicate code (in short, MF_n) are measured using the following steps:

STEP1: Identifies revisions where one or more source files are modified, added, or deleted. The identified revisions are called **target revisions** in the remainder of this section. Then, all the target revisions are checked out into the local storage.

STEP2: Normalizes all the source files in every target revision.

STEP3: 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.

STEP4: Identifies differences between two consecutive revisions. The start lines and the end lines of all the differences are stored.

STEP5: Counts the number of modifications on duplicate code and non-duplicate code.

STEP6: Calculates MF_d and MF_n based on the duplicate code detection results and difference identification results.

The remainder of this section explains each step of the measurement in detail.

STEP1: Obtains target revisions

In order to measure MF_d and MF_n for a target system, it is necessary to obtain the historical data of the source code. In this research, we used a version control system, subversion to obtain the historical data. Due to the limit of implementation, we restrict the target

Table 1: Target software systems

(a) Experiment 1

Software name	Domain	Programming language	# of revisions	LOC of the newest 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 2

Software name	Domain	Programming language	# of revisions	LOC of the newest 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

version control system to subversion. However, it is possible to use other version control systems such as CVS.

Firstly, we identify which files are modified, added, or deleted in each revision, and find out whether the files are source files or not by checking their extensions. If there are 1 or more source files in the modified files, its revision is regarded as a **target revision**. After identifying all the target revisions from the historical data, they are checked out into the local storage.

STEP2: Normalizes source code

In the STEP2, every source file in all the target revisions is normalized with the following rules:

- deletes blank lines, code comments, and indents,
- deletes lines including 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 comment located in the border of duplicate code and non-duplicate code should be treated, so that a certain detection tool regards such a code comment as duplicate code meanwhile another tool regards it as non-duplicate code.

As mentioned above, the presence of code comments makes it more difficult to accurately identify the start line and end line of duplicate code. Consequently, all the code comments are removed completely. As well as code comment, 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 including only a single open or close brace are removed and the open or close brace is added to the end of the previous line.

STEP3: Detects duplicate code

Duplicate code is detected from every target revision, and the detection results are stored into database. Each detected duplicate code is identified by 3-tuple, (v,f,l). Every element of the 3-tuple is as follows: 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 (Type-3 code clone).

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-automatically processing, and no manual work is required.

STEP4: Identifies differences between two consecutive revisions

In this research, we count the number of pieces of modified code, not lines of modified code. That is, even if multiple consecutive lines are modified, we regard it as a single modification. This is because that, in the process of fixing bugs or adding new functionalities, the cost required before source code modification is much greater than the cost of the modification cost itself. For example, if we regard every modified line as a single modification, we cannot distinguish *1 line modification at 10 pieces of code for fixing 10 different bugs* from *10 lines modification at 1 piece of code for fixing a bug*. In the actual process, the former case will require much more cost than the latter case. Counting based on the number of pieces of modified code can distinguish the two cases.

In order to identify the number of modifications in the above manner, we use UNIX diff command. Figure 1 shows an example of diff output. As shown in Figure 1, it is very easy to identify multiple consecutive lines modification 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.

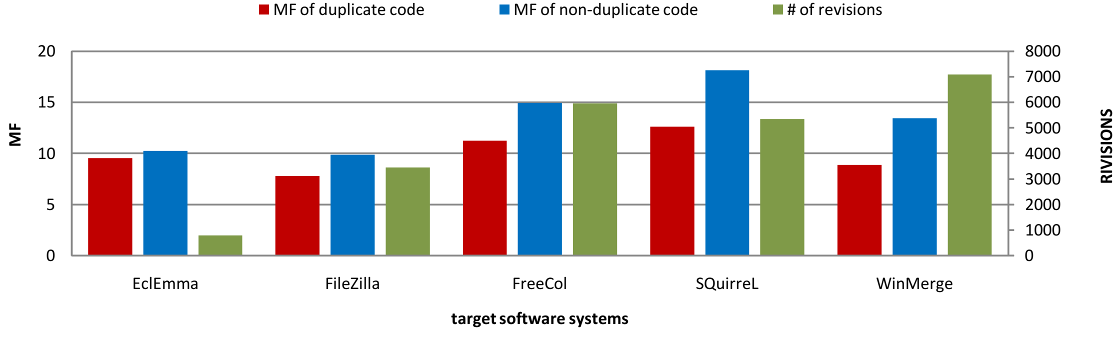


Figure 2: Result of Item A on Experiment 1

STEP5: Counts the number of modifications

In this step, the number of modifications of duplicate code and non-duplicate code is counted with the result of the previous step. Here we assume that the variable for the number of modifications of duplicate code is MC_d and, the variable for non-duplicate code is MC_n. Firstly, MC_d and MC_n are initialized with 0, then they are increased as follows: if the range of a specified modification is completely included in duplicate code, MC_d is incremented; if it is completely included in non-duplicate code, MC_n is incremented; if it is located across the border of duplicate code and non-duplicate code, both MC_d and MC_n are incremented. All the modifications are processed with the above algorithm.

STEP6: Calculates MF_d and MF_n

In this step, MF_d and MF_n are calculated with the result of duplicate code detection and the result of modification counting. Here we assume that:

- R is a set of target revisions,
- $MC_d(r)$ is the number of modifications on duplicate code between revision r and $r+1$.

Using this assumption, MF_d is formalized as follows:

$$MF_d = \frac{\sum_{r \in R} MC_d(r)}{|R|} \tag{1}$$

MF_d means the average number of modifications on duplicate code per revision. As well as MF_d, MF_n is formalized as follows:

$$MF_n = \frac{\sum_{r \in R} MC_n(r)}{|R|} \tag{2}$$

where:

- $MC_n(r)$ is the number of modifications on non-duplicate code between revision r and $r+1$.

MF_n means the average number of modifications on non-duplicate 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 MC_d is much smaller than MC_n. Therefore, we cannot fairly compare the MF_d and MF_n. In order to eliminate the bias of the amount of duplicate code, we normalize the formulae (1) and (2) using the LOCs of duplicate code and non-duplicate code. Here, we assume that:

- $LOC_d(r)$ is the total lines of duplicate code in revision r.
- $LOC_n(r)$ is the total lines of non-duplicate code on r.
- $LOC(r)$ is the total lines of code on r, so that the following formula is satisfied:

$$LOC(r) = LOC_d(r) + LOC_n(r)$$

Using these assumptions, the normalized MF_d and MF_n are defined as follows:

$$normalized\ MF_d = \frac{\sum_{r \in R} MC_d(r)}{|R|} \times \frac{\sum_{r \in R} LOC(r)}{\sum_{r \in R} LOC_d(r)} \tag{3}$$

$$normalized\ MF_n = \frac{\sum_{r \in R} MC_n(r)}{|R|} \times \frac{\sum_{r \in R} LOC(r)}{\sum_{r \in R} LOC_n(r)} \tag{4}$$

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

5. MF COMPARISON BETWEEN DUPLICATE AND NON-DUPLICATE CODE

This section describes the experiment for comparing MF_d and MF_n. In this experiment, we selected 15 software systems that are open to the public in SourceForge (see Table 1). The selection was performed based on the following criteria:

- the source code is managed with subversion;
- the source code is written with C/C++ or Java;
- we took care not to bias the domains of the targets.

The experiment consists of the following two sub-experiments.

Experiment 1: We compare MF_d and MF_n on various size software systems with a scalable detection tool, CCFinder. The purpose of this experiment is to reveal the relation between the number of revisions and the impact of duplicate code.

Experiment 2: We compare MF_d and MF_n on small size software with the 4 detection tools, described in Section 3. The purpose of this experiment is to investigate how MF comparison results are different from detection tools.

The following items are investigated in each sub-experiment.

Item A: This is for investigating whether duplicate code is modified more frequently than non-duplicate code. In this investigation, we calculate MF_d and MF_n on the entire period.

Item B: We expect that there are different MF tendencies between the early period and the aging period of development. Consequently, we divide the entire period into 10 sub-periods and calculate MF on every of the sub-periods.

5.1 Experiment1: Result and Discussion

In experiment 1, we selected 5 software systems that are various size and varying numbers of revisions. The ratio of duplicate code for each target software is between 13% and 29% (see Table 2). Figure 2 shows the result of Item A. As shown in Figure 2, MF_d is lower than MF_n on all the target systems. It is generally said that duplicate code is more frequently modified than non-duplicate code because the same modifications have to be applied to multiple pieces of code. However, the experimental result is different from the common belief.

Figure 3 shows the result of Item B. 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 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 non-duplicate code. However, the shapes of MF transitions are different from every software system.

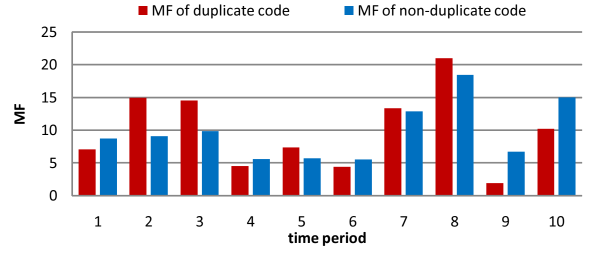
For WinMerge, we investigated period '2', where MF_n is much greater than MF_d, and period '10', where is only the period that MF_d is greater than MF_n. In period '10', there are many modifications on test cases. The number of revisions that test cases are modified is 49, and the ratio of duplicate 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.

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

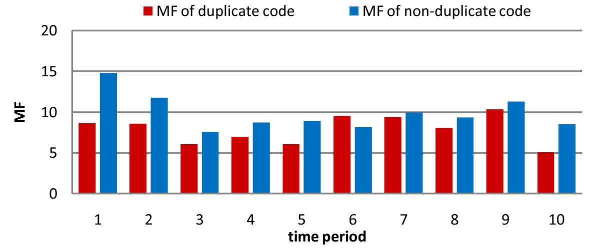
5.2 Experiment2: Result and Discussion

As shown in Table 1, the number of revisions of the target software in experiment 2 is smaller than the target software of experiment 1. This is because, in this experiment, 4 detection tools were used for calculating MFs. It took much time to detect duplicate code from every revision.

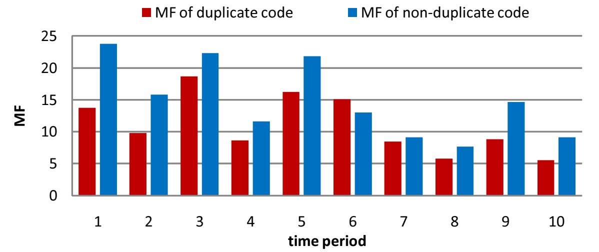
Figure 4 shows the result of Item A. In Figure 4, the detection tools are abbreviated as follows: CCFinder $\rightarrow C$; CCFinderX $\rightarrow X$; Simian $\rightarrow S_i$; Scorpio $\rightarrow S_c$. Scorpio does not handle C/C++, so that there are the results of the other 3 detection tools on C/C++ systems (Figure 4(b)). MF_d is less than MF_n in the 22 comparison results out of 35. In the 5 target systems out of 10, duplicate code is modified less frequently than non-duplicate code in the cases of all the detection tools. In the case of the other 2 systems, MF_d is



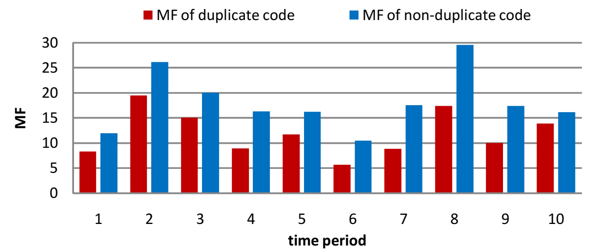
(a) EclEmma



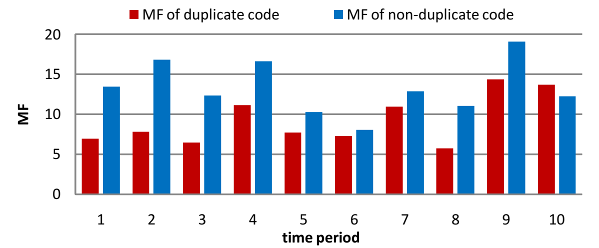
(b) FileZilla



(c) FreeCol

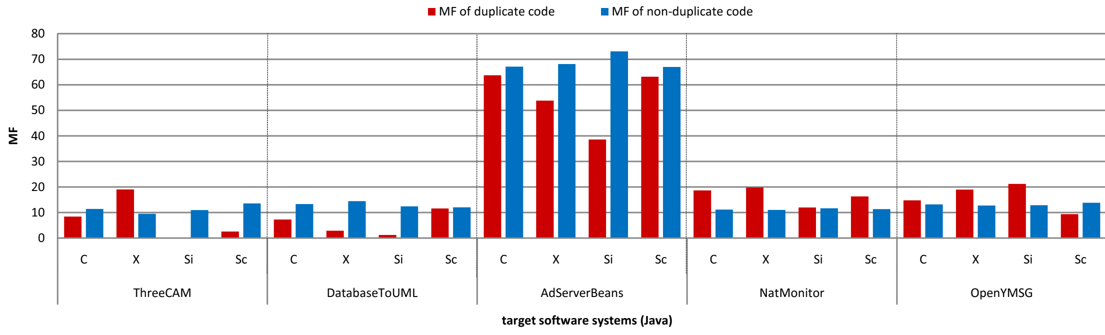


(d) SQuirreL

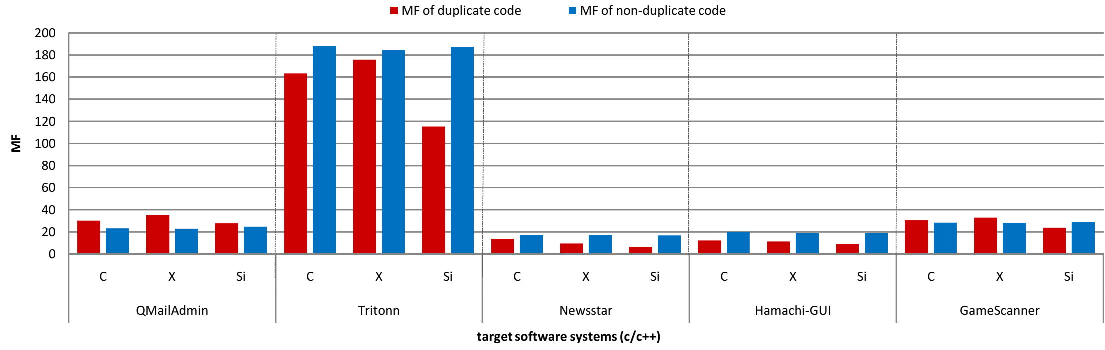


(e) WinMerge

Figure 3: Result of Item B on Experiment 1



(a) Java software



(b) C/C++ software

Figure 4: Result of Item A on Experiment 2

greater than MF_n in the cases of all the detection tools. And in the remaining software, the comparison result is different for the detection tools. Also, we compared MFs based on programming language and detection tool. The comparison result is shown in Table 3. The result shows that MF_d is less than MF_n on all the programming language and all the detection tools.

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. It is generally said that the presence of duplicate code makes it more difficult to maintain

Table 2: Ratio of duplicate code

(a) Experiment 1

Software name	CCFinder	CCFinderX	Simian	Scorpio
EclEmma	13.1%	-	-	-
FileZilla	22.6%	-	-	-
FreeCol	23.1%	-	-	-
SQuirrel	29.0%	-	-	-
WinMerge	23.6%	-	-	-

(b) Experiment 2

Software name	CCFinder	CCFinderX	Simian	Scorpio
TreeCAM	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%	-

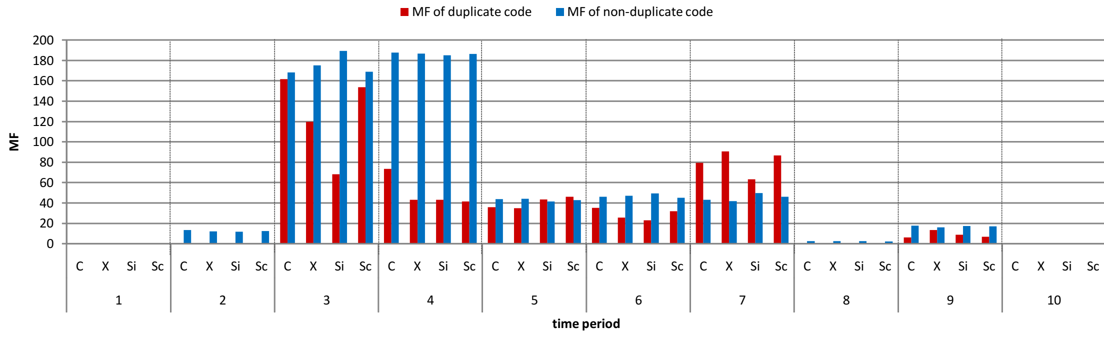
Table 3: Comparing MFs based on programming language

(a) Comparison on Programming Language

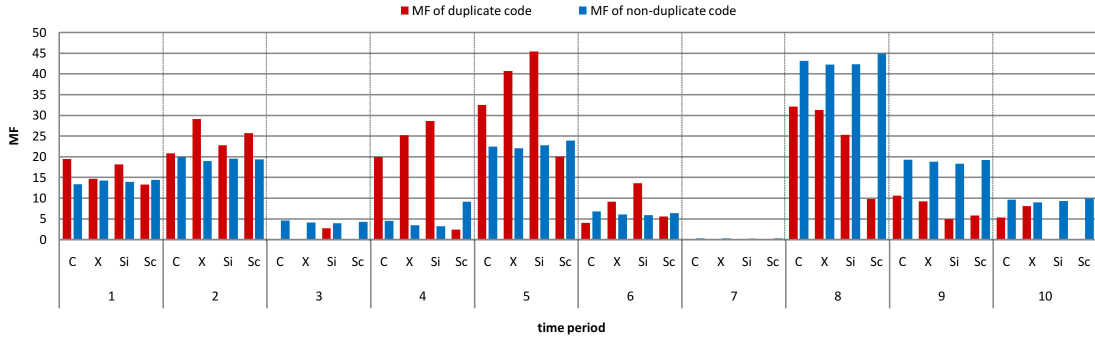
Programming language	MF	
	duplicate code	non-duplicate code
Java	20.1547	23.5330
C/C++	46.4531	55.0157
ALL	32.1764	38.0545

(b) Comparison on Detection Tool

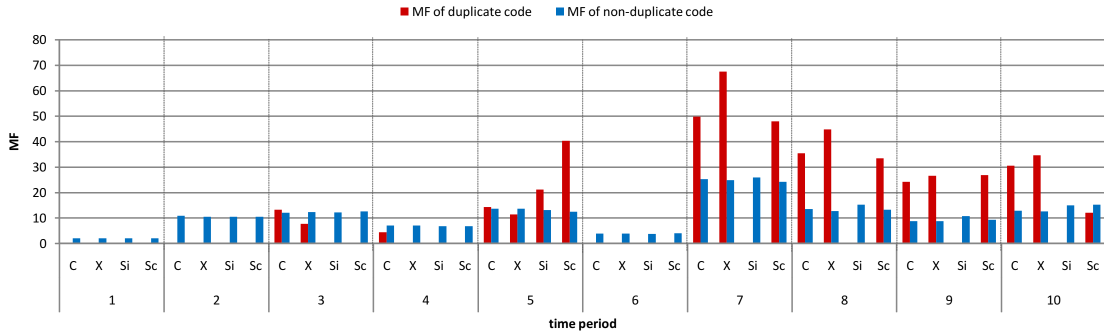
Detection tool	MF	
	duplicate code	non-duplicate code
CCFinder	36.2851	39.2862
CCFinderX	37.8827	38.7334
Simian	25.5285	39.7959
Scorpio	20.5851	23.5483
ALL	32.1764	38.0545



(a) AdServerBeans



(b) OpenYMSG



(c) NatMonitor

Figure 5: Result of Item B on Experiment 2

software. However, we cannot see such a tendency in the result of this comparison.

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 non-duplicate 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 most duplicate code is the generated code, MF_d becomes less than MF_n.

Figure 6 is an instance of this case. The whole of the method is duplicate code, and there are 16 correspondents in 4 source files. Every method adds an object received by a parameter into a collection. Their names are in a common form 'addXXXPropertyDescriptor' (XXX is different from every method). All the 17 methods were added in the 10th revision, and none of them was modified until the most recent revision. Annotation @generated was attached into all the method, which means that they were generated by a code generation tool.

We measured the ratio of duplicate code for each target software. The ratio is shown in Table 2(b). The ratio of duplicate code is very different for each detection tool on the same software. However, as well as in experiment 1, we cannot find out any relation between the ratio and MF.

```
@generated
protected void addNamePropertyDescriptor(Object object) {
  itemPropertyDescriptors.add(
    createItemPropertyDescriptor(
      ((ComposeableAdapterFactory)adapterFactory).
        getRootAdapterFactory(),
      getResourceLocator(),
      getString("_UI_NamedElement_name_feature"),
      getString("_UI_PropertyDescriptor_description",
                "_UI_NamedElement_name_feature",
                "_UI_NamedElement_type"),
      MetadataPackage.Literals.NAMED_ELEMENT__NAME,
      true, false, false,
      ItemPropertyDescriptor.GENERIC_VALUE_IMAGE,
      null, null));
}
```

Figure 6: An instance of stable duplicate code

Figure 5 shows the result of Item B. Figure 5 shows the result of only the 3 systems due to space limitation. In Figure 5(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 non-duplicate code. We investigated the source code of period '4'. In this period, many source files were created by copy-and-paste, and a large amount of duplicate code was detected by each detection tool. The copy-and-pasted code 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 5(b) shows that duplicate code tends to be modified more frequently than non-duplicate 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 5(c) shows the opposite result of Figure 5(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 detects only the exact-match duplicate code meanwhile the other tools detect extract-match and renamed duplicate code in the default setting.

The summary of experiment 2 is as follows: we found some instances that duplicate code was modified more frequently than non-duplicate code in a short period on each detection tool; however, in the entire period, duplicate code was modified less frequently than non-duplicate code on every target software with all the tools. Consequently, we conclude that the presence of duplicate code does not have a seriously-negative impact on software evolution.

The conclusion is contradictory with the previous studies [19, 20]. As described in Section 2, they investigated the influence of duplicate code on file unit or method unit. However, the units (files and methods) are larger than duplicate code, so that modifications can be incorrectly counted: if modifications were performed on a method where a duplicate code exists, all the modifications are assumed as performed on duplicate code even if they were actually performed on non-duplicate code. Thus, their studies might count more the number of modification on duplicate code than it really was. On the other hand, in the proposed method, the unit is line, which is equal to or smaller than duplicate code. Consequently, such an incorrect count was not performed on the proposed method. The incorrect count on duplicate code may introduce the previous works to the different conclusion from the present paper.

6. THREATS TO VALIDITY

This section describes threats to validity of this empirical study.

Cost required for every modification

In this empirical study, we assume that cost required for every modification is equal to one another. However, in the actual software evolution, the cost is different between every modification. Consequently, it is possible that the comparison based on MF may not appropriately represent the cost required for modifying duplicate code and non-duplicate 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 re-modify them for repairing the consistency. The effort for consistency is not necessary for modifying non-duplicate code. Consequently, the average cost required for duplicate code may be different from the one required for non-duplicate code. In order to compare them more appropriately, we have to consider the cost for maintaining consistency.

Identifying the number of modifications

In this empirical study, modifying consecutive multiple lines is 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.

Beside, we investigated how many the identified modifications occurred across the boundary of duplicate code and non-duplicate code. If this number is high, then the analysis suspect because such modifications increase both the counts at 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 non-duplicate code.

Category of modifications

In this empirical 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. 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.

Property of target software

In this empirical study, we used only open source software systems, so that different results may be shown with industrial software systems. It is generally said that industrial software includes more duplicate code than open source software. 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 investi-

gate the impact of duplicate code on industrial software, we have to compare *MF* on industrial software itself.

Division of development period

In this empirical study, we divided the development period in an automatic manner based on the number of revisions. However, different division may yield different results. For example, if we divide the period based on the border of versions, we may be able to grasp the properties of every version. Or, more fine grained division, that is, the period of every version is divided into multiple sub-periods, which will let us how duplicate code and non-duplicate code are modified from a start of a version to the end of the version.

Settings of detection tools

In this empirical study, we used default settings for all the detection tools. If we change the settings, different results will be shown.

7. CONCLUSION

The present paper experimentally evaluated the impact of the presence of duplicate code in open source software systems. In order to compare cost required for modifying duplicate code and non-duplicate code, we defined a new marker, **modification frequency**. In the experiment, we used 4 duplicate code detection tools for reducing the bias of detection tool. The empirical result from 15 software systems showed that duplicate code tends to be modified less frequently than non-duplicate code. However, in the face of statistics, there is no difference of significance between them where the level of significance is 5%. That is, the presence of duplicate code does not have made it more difficult to develop and maintain software systems. At the same time, we detected some duplicate code that was repeatedly modified simultaneously with its correspondents. In the future, we will conduct more experiments with a new condition that overcomes the threats to validity described in Section 6. Moreover, we will try to automatically identify duplicate code that should have a seriously-negative impact on future development and maintenance.

Acknowledgment

The present 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 Grants-in-Aid for Scientific Research (A) (21240002) and (C) (20500033) from the Japan Society for the Promotion of Science, and Grand-in-Aid for Young Scientists (B) (22700031) from Ministry of Education, Science, Sports and Culture.

8. REFERENCES

[1] CCFinderX. `http://www.ccfinder.net/`.

[2] Clone Detection Literature. `http://www.cis.uab.edu/tairasr/clones/literature/`.

[3] Scorpio. `http://www-sdl.ist.osaka-u.ac.jp/~higo/cgi-bin/moin.cgi/scorpio-e/`.

[4] Simian-similarity analyser. `http://www.redhillconsulting.com.au/products/simian/`.

[5] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on Software Engineering*, 31(10):804–818, Oct. 2007.

[6] N. Bettenburg, W. Shang, W. Ibrahim, B. Adams, Y. Zou, and A. E. Hassan. Inconsistent Changes to Code Clones at Release Level. In *Proc. of the 16th Working Conference on Reverse Engineering*, pages 85–94, Oct. 2009.

[7] E. Burd and J. Bailey. Evaluating Clone Detection Tools for Use during Preventative Maintenance. In *Proc. of the 2nd IEEE International Workshop on Source Code Analysis and Manipulation*, pages 36–43, Oct. 2002.

[8] M. de Wit, A. Zaidman, and A. van Deursen. Managing Code Clones Using Dynamic Change Tracking and Resolution. In *Proc. of the 25th IEEE International Conference on Software Maintenance*, pages 169–178, Sep. 2009.

[9] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus. Does Code Decay? Assessing the Evidence from Change Management Data. *IEEE Transactions on Software Engineering*, 27(1):1–12, Jan. 2001.

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

[11] N. Göde. Evolution of type-1 clones. In *Proc. of the 9th International Working Conference on Source Code Analysis and Manipulation*, pages 77–86, Sep. 2009.

[12] Y. Higo and S. Kusumoto. Significant and Scalable Code Clone Detection with Program Dependency Graph. In *Proc. of the 16th Working Conference on Reverse Engineering*, pages 315–316, Oct. 2009.

[13] 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*, 28(7):654–670, July 2002.

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

[15] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy. An Empirical Study of Code Clone Genealogies. In *Proc. of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 187–196, Sep. 2005.

[16] J. Krinke. Is cloned code more stable than non-cloned code? In *Proc. of the 8th International Working Conference on Source Code Analysis and Manipulation*, pages 57–66, Sep. 2008.

[17] A. Lozano and M. Wermelinger. Assessing the effect of clones on changeability. In *Proc. of the 24th International Conference on Software Maintenance*, pages 227–236, Sep. 2008.

[18] A. Lozano, M. Wermelinger, and B. Nuseibeh. Evaluating the harmfulness of cloning: a change based experiment. In *Proc. of the 4th International Workshop on Mining Software REpositories*, May 2007.

[19] A. Lozano, M. Wermelinger, and B. Nuseibeh. Evaluating the relation between changeability decay and the characteristics of clones and methods. In *Proc. of the 23rd International Conference on Automated Software Engineering*, pages 100–109, Sep. 2008.

[20] A. Monden, D. Nakae, T. Kamiya, S. Sato, and K. Matsumoto. Software Quality Analysis by Code Clones in Industrial Legacy Software. In *Proc. of the 8th IEEE International Software Metrics Symposium*, pages 87–94, June 2002.

[21] M. P. Robillard, W. Coelho, and G. C. Murphyy. How effective developers investigate source code: An exploratory study. *IEEE Transactions on Software Engineering*, 30(12):889–903, Dec. 2004.