Is Duplicate Code Good or Bad? An Empirical Study with Multiple Investigation Methods and Multiple Detection Tools

Yui Sasaki, Keisuke Hotta, Yoshiki Higo, Shinji Kusumoto
Graduate School of Information Science and Technology, Osaka University,
1-5, Yamadaoka, Suita, Osaka, 565-0871, Japan
Email: {s-yui,k-hotta,higo,kusumoto}@ist.osaka-u.ac.jp

***Abstract*—There are several research efforts that compare duplicate code and non-duplicate code for revealing that the presence of duplicate code actually has a negative impact on software development and maintenance. However, each of them compares them with a single method and a single detection tool. Consequently, their result may not be reliable. This paper reports an empirical study that duplicate code and non-duplicate code were compared with multiple investigation methods and multiple detection tools.**

***Keywords*-Duplicate Code, Replication, Maintenance**

I. Introduction

It is generally said that the presence of duplicate code has negative impacts on software development and maintenance. For example, it increases 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.

Recently, a variety of research efforts related to duplicate code has been conducted. Some of them have proposed methods that apply refactorings for removing duplication [6], meanwhile others said that duplicate code is a good choice for design of the source code [12].

In order to obtain a sight of truth, several research efforts have proposed comparison methods for duplicate code and non-duplicate code. Each of them compares a characteristic of duplicate code and non-duplicate code instead of directly investigating their maintenance cost because investigating actual maintenance cost is quite difficult. However, authors thought that their comparison results may not be reliable because every of them compares a single characteristic with a single detection tool. Herein some of characteristics are stability [13], age [14], work [15], and frequency [9].

This paper reports an experimental result with three comparison methods, four detection tools, and five target systems. A research question that we want to reveal in this paper is the following: every method compares duplicate code and non-duplicate code from a single viewpoint, and concludes the presence of duplicate code is good or bad; *the purpose of the experiment is to reveal whether comparisons of duplicate code and non-duplicate code with different methods and different detection tools always introduce the same result*[1]. If so, all we have to do is using a single investigation method. If not, it is quite difficult to judge it.

II. Duplicate Code Detection Tools

There are currently various kinds of duplicate code detection tools. The detection tools can be categorized based on their detection techniques. Major categories should be line-based, token-based, metrics-based, AST[2]-based, and PDG[3]-based. Each technique has merits and demerits, and there is no technique that is superior to any other techniques in every way [3], [4]. The remainder of this section describes 4 detection tools that are used in our experiment. We uses two token-based detection tools, which is for investigating whether both the token-based detection tools always introduce the same result or not.

A. CCFinder

CCFinder is a token-based detection tool [11]. 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. Also, CCFinder detection speed is very fast. Moreover, CCFinder can detect duplicate code from millions lines of code within an hour. CCFinder can handle multiple popular programming languages such as C/C++, Java, and COBOL.

B. 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.

[1]Herein, result is whether duplicate code is good or bad
[2]Abstract Syntax Tree
[3]Program Dependency Graph

C. Simian

Simian is a line-based detection tool [2]. 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.

D. Scorpio

Scorpio is a PDG-based detection tool [7]. 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-next link, which allows detecting more duplicate code than traditional PDG [7]. Also, Scorpio adopts some heuristics for filtering out false positives. Currently, Scorpio can handle only Java language.

III. Investigation Methods

In this research, we use three investigation methods, Krinke's [13], Lozano's [15], and Hotta's methods [9]. The remainder of this section describes an overview of them. Due to space limitation, we cannot explain the details of them. Please see their papers if you are interested in them.

A. Krinke's Method

Krinke's method compares ratios of modified duplicate code and modified non-duplicate code. This method uses not all the revisions but a revision per week [13].

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. By using the information, the ratios of added lines, deleted liens, and changed lines on duplicate and non-duplicate code are calculated and compared.

B. Lozano's Method

Lozano's method categorizes Java methods, then compares distributions of maintenance cost based on the categories [15].

Firstly, Java methods are traced based on their owner-class's full qualified name, start/end lines, and signatures. An identified method trace is called *method-chain*. Method-chains are categorized as follows:

- **AC-Method**: method-chains whose every revision include duplicate code;
- **NC-Method**: method-chains whose any revision does not include duplicate code;
- **SC-Method**: method-chains whose some revisions include duplicate code but the others does not.

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

- $ChangedRevisions(m,P)$: a set of revisions that method-chain m is modified in period P,
- $Methods(r)$: a set of methods that exist in revision r,
- $ChangedMethods(r)$: a set of methods that were modified in revision r,
- $CoChangedMethods(m,r)$: a set of methods that were modified simultaneously with method m in revision r. if method m is not modified in revision r, it become $\emptyset$. If modified, the following formula is satisfied.

$$ChangedMethod(r) = m \cup CoChangedMethod(m,r)$$

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

$$likelihood(m,P) = \frac{ChangedRevisions(m,P)}{\sum_{r\in P}|ChangedMethods(r)|} \quad (1)$$

$$impact(m,P) = \frac{\sum_{r\in P}\frac{|CoChangedMethods(m,r)|}{|Methods(r)|}}{|ChangedRevisions(m,P)|} \quad (2)$$

$$work(m,P) = likelihood(m,P) \times impact(m,P) \quad (3)$$

In this research, we compare *work* between AC-Method and NC-Method, and compare SC-Method's work on duplicate period and non-duplicate period.

C. Hotta's Method

Hotta's method compares modification frequencies between duplicate code and non-duplicate code [9].

Hotta's method identifies modified places, and calculates modification frequencies on them. A modified place is a set of consecutive lines. If all the consecutive lines are in duplicate/non-duplicate code, it is regarded as a modification on duplicate/non-duplicate code. If a part of consecutive lines is duplicate code. it is regarded as modification on both duplicate and non-duplicate code.

Herein we assume that R is a set of revisions, $l(r)$, $lc(r)$, $ln(r)$ are total-lines-of-code, lines-of-duplicate-code, lines-of-non-duplicate-code on revision $r \in R$. Also, $mc(r)$ and $mn(r)$ are the numbers of modified places on duplicate code and non-duplicate code in r. By using the above terms, MF_c and MF_n, which are modification frequencies on duplicate code and non-duplicate code, are calculated as follows:

$$MF_c = \frac{\sum_{r\in R} mc(r)}{|R|} \times \frac{\sum_{r\in R} l(r)}{\sum_{r\in R} lc(r)} \quad (4)$$

$$MF_n = \frac{\sum_{r\in R} mn(r)}{|R|} \times \frac{\sum_{r\in R} l(r)}{\sum_{r\in R} ln(r)} \quad (5)$$

IV. Experiment

A. Experimental Setup

We compare duplicate code and non-duplicate code with the three methods described in Section III. Note that there are many other methods that investigate characteristics of duplicate code [8]. However, this research focus on comparison between duplicate code and non-duplicate code, which is why Lozano's, Krinke's and Hotta's methods were selected.

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. All the selected tools are open to the public in Internet. Everyone can use these tools.

We already have an implementation of Hotta's method because he is a member of our research group. On the other hand, we developed software tools for Lozano's and Krinke's methods based on their papers.

We chose five software systems that are open to the public in SourceForge. Table I shows them. All the systems are written in Java language because Scorpio handles only Java language. Automatic generated code and testing code are removed from all the revisions before the investigation methods are applied.

B. Result of MASU

Due to space limitation, we cannot shows all the comparison figures of all the software. Herein, we show only comparison figures of MASU. MASU is a metrics measurement tool and is open to the public in http://sourceforge.net/projects/masu. Figure 1 shows the results of Hotta's method. Left bars (red) and right bars (blue) are modification frequencies on duplicate and non-duplicate code respectively. Herein, if the difference was more than 5%, we regarded it as a significant one. In this case, all the detection tools except CCFinderX brought the same result that duplicate code is more frequently modified than non-duplicate code.

Table I
TARGET SOFTWARE SYSTEMS

Software	# of revisions	LOC of the end revision
QpenYMSG	194	14,111
EclEmma	1,220	31,409
MASU	1,620	79,360
TVBrowser	6,829	264,796
Ant	5,412	198,864

Table II
RATIO OF DUPLICATE CODE

Software name	ccf	ccfx	sim	sco
OpenYMSG	12.41%	6.16%	2.66%	5.50%
EclEmma	6.94%	4.77%	2.03%	3.69%
MASU	25.62%	26.49%	11.31%	15.43%
TVBrowser	13.64%	10.86%	5.39%	18.96%
Ant	13.9%	12.12%	6.19%	15.55%

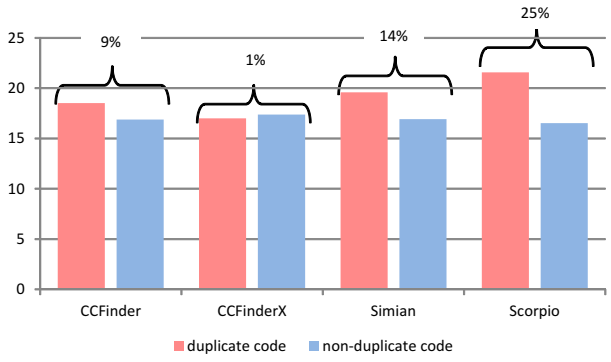


Figure 1. Result of Hotta's Method on MASU

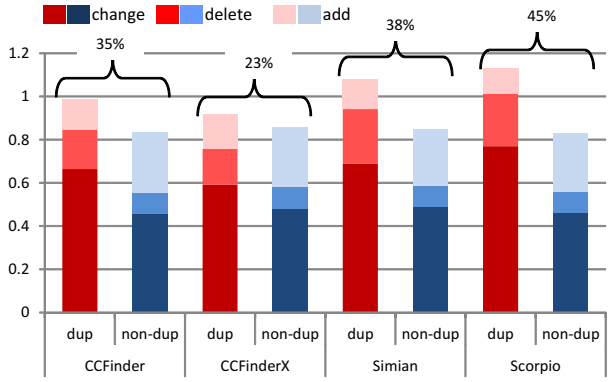
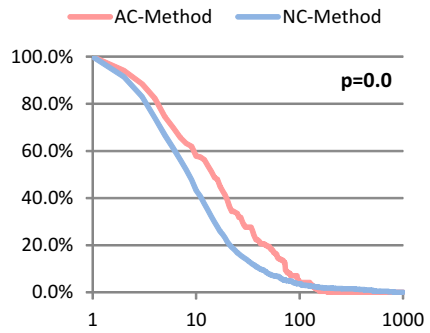


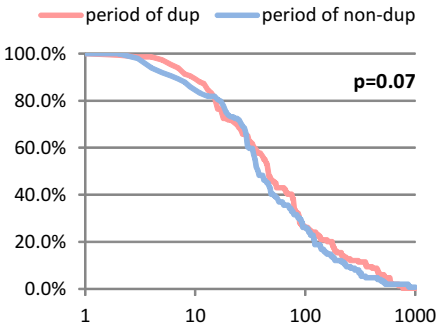
Figure 2. Result of Krinke's Method on MASU

Figure 2 shows the results of Krinke's method on MASU. Every bar consists of three parts, which means *changed*, *deleted*, and *added*. In this experiment, we set 5% as the threshold of significant difference in Krinke's method. Note that the difference was calculated based on *changed* and *deleted* because the amount of *add* is the lines of code added in the next revision. This figure shows that comparisons of all the tools brought the same result that duplicate code is less stable than non-duplicate code.

Figure 3 shows the results of Lozano's method on MASU. Figure 3(a) compares AC-Method and NC-Method. X-axis is maintenance cost (*work*) and Y-axis is cumulated frequency of methods. For readability, we adopt logarithmic axis on X-axis. The value of p, which locates on the upper right of the graph, means the result of Mann-Whitney's U-test. We set 5% as the level of significance. In this case, AC-Method requires more maintenance cost than NC-Method. Also, Figure 3(b) compares duplicate period and non-duplicate period of SC-Method. The value of p means the result of Wilcoxon's singed-rank test. We set 5% as the level of significance. In this case, the maintenance cost in duplicate period is greater than non-duplicate period.



(a) AC-Method and NC-Method



(b) SC-Method

Figure 3. Result of Lozano's Method on MASU with Simian

C. Overall Result

Table III shows the comparison results of all the systems. In table III, "*C*" means that duplicate code requires more cost than non-duplicate code, and "*N*" means its opposite. "–" means there is no significant difference between duplicate and non-duplicate code. This table shows the followings:

- In the case of EclEmma, there is no result that shows duplicate code is worse than non-duplicate code.

Table III
OVERALL RESULT

Software	Method	Tools			
		ccf	ccfx	sim	sco
OpenYMSG	Hotta	N	C	C	N
	Krinke	N	C	C	N
	Lozano	-	-	N	-
EclEmma	Hotta	N	N	-	N
	Krinke	N	N	N	-
	Lozano	N	N	-	-
MASU	Hotta	C	-	C	C
	Krinke	C	C	C	C
	Lozano	C	C	C	C
TVBrowser	Hotta	N	N	N	N
	Krinke	C	C	N	C
	Lozano	C	C	C	C
Ant	Hotta	N	N	N	N
	Krinke	C	C	C	C
	Lozano	C	C	C	C

- On the other hand, in MASU, almost all the results show duplicate code has a negative impact.
- For the other systems, there are opposing results. It is interesting that there are some cases CCFinder and CCFinderX introduce opposing results.

We would like to know that the presence of duplicate code has a negative impact on a given system. However, running all the investigation methods with all the detection tools consumes much time. Consequently, from Table III, we recommend to run Hotta's method and Krinke's method with CCFinder and CCFinderX. If all of the 4 results are the same, the result is probably reliable. If not, it is difficult to judge whether duplicate code is good or bad on the system. There is one reason that we recommend a paper of CCFinder and CCFinderX, not a pair of Simian of Scorpio. CCFinder family adopts token-based detection algorithm, so that their detection speed is very high meanwhile the detection speed of Scorpio is slow because it uses PDG-based algorithm.

D. Discussion

In the case of OpenYMSG, TVBrowser, and Ant, different detection methods and different tools brought opposing results. Those showed that investigating duplicate code with a single method and with a single detection tool has poor reliability and believing such a result is dangerous.

Figure 4 shows an actual modification in Ant. Two methods were modified in this modification. The hatching parts are detected duplicate code and frames in them means pairs of duplicate code between the two methods. Vertical arrows shows modified lines between this modification and the next (77 lines of codes were modified).

This modification is a refactoring, which extracts the duplicate instructions from the two methods and merges them as a new method. In Hotta's method, there are 2 modification places in duplicate code and 4 places in non-duplicate code, so that MF_C and MF_N become 51.13 and 18.13, respectively. The difference of them is about 65%. In Krinke's method, $DC + CC =$ and $DN + CN$ become 0.089 and 0.005, respectively. The difference between them is about 93%. This modification removed duplicate code in the two methods, so that this revision is the border between periods P_C and P_N. The values of *work* becomes the followings.

$$\begin{aligned} work(CodeFragment1, P_C) &= 5.71 \times 10^{-5} \\ work(CodeFragment1, P_N) &= 8.17 \times 10^{-6} \\ work(CodeFragment2, P_C) &= 5.93 \times 10^{-5} \\ work(CodeFragment2, P_N) &= 1.03 \times 10^{-5} \end{aligned}$$

These values reveal that *work* of duplicate code is higher than *work* of non-duplicate code, which implies that the refactoring improved the *work* measure. All the methods revealed different aspects of development history.

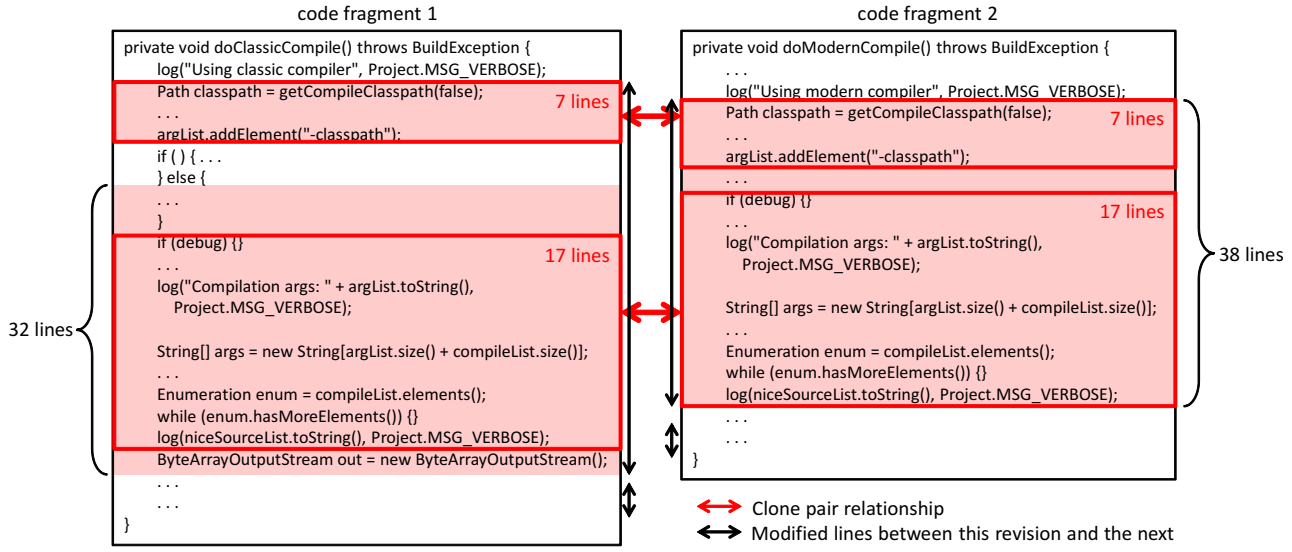


Figure 4. An example of modification

On the other hand, in the case of EclEmma and MASU, different methods and different tools brought almost the same result. For these systems, the comparison results should be a reliable marker. Consequently, if you investigate the impact of duplicate code in automatic way, you should do in multiple methods with multiple detection tools.

In the case of TVBrowser and Ant, Hotta's method brought the opposite result to Lozano's and Krinke's method. It is quite interesting because Hotta's method was proposed for overcoming the weaknesses of Lozano's and Krinke's methods. Unfortunately, at present, we cannot investigate which result is correct for these systems.

In this experiment, we used only five systems, so that different results may be brought out on other systems. More and more experiments are required for obtaining more reliable results. It may be a reasonable choice that observing the entire process of a small project and manually investigating the impact of duplicate code. Such a result can be an oracle for evaluating automatic investigation methods.

V. Related Work

Krinke compared ages of duplicate code and non-duplicate code [14]. Every source line of the latest version was checked for identifying in which revision it was lastly changed. Then, duplicate code was detected by using Simian. Average ages of duplicate lines and non-duplicate lines were calculated ant compared. In his experiment, 4 large-scale Java software systems were analyzed and the average age of duplicate code is older than the one of non-duplicate code. That implies duplicate code is more stable than non-duplicate code. However, the following issues seem to remain in his experiment,

- it did not consider non-contiguous code clones, which are not detected by line-based detection tools,
- it did not remove test code and generated code, which may affect the comparison result.

Rahman et al. investigated relationship between duplicate code and bugs [16]. They analyzed 4 software systems written in C language, and used bug information stored in Bugzilla. Declard, which is a AST-based detection tool, was used for detecting duplicate code. Their result showed that,

- only a small part of the bugs located on duplicate code,
- the presence of duplicate code did not dominate bug appearances.

However, there are some issues in their experiment,

- Declard is a AST-based detection tool, its ability to detect non-contiguous code clones is limited,
- they regarded all the code changed in the bug-fix revisions as buggy code instead of actual buggy code,
- they use only a single revision per month instead of using all the revisions.

Göde et al. replicated Krinke's experiment [5]. Krinke's original experiment detected line-based duplicate code meanwhile Göde experiment detected token-based duplicate code. The experimental result was the same as Krinke's one. Duplicate code is more stable than non-duplicate code in the viewpoint of add and changed. On the other hand, from the delete viewpoint, non-duplicate code is more stable than duplicate code. The same kinds of issues remains in the Göde's experiment.

- only a single revision per a week was analyzed instead of all the revisions,
- non-contiguous code clones were not counted,
- they did not normalize source code, so that trivial

modifications (e.g., format change, comment change) may have impacts on the comparison result.

Juergens et al. investigated inconsistencies in duplicate code and relationships between inconsistencies and bugs [10]. Their targets were 2 industrial and 1 open source systems. They applied token-based detection to the systems. Their token-based detection has ability to detect duplicate code including small gaps, which is a part of non-contiguous code clones. Their experiment revealed the followings:

- about a half of duplicate code includes inconsistences,
- 1/4 of the inconsistences were unintended ones,
- $3 \sim 23\%$ of inconsistences contains bugs. However, the 3% was a COBOL software system, if we do not care it, the average was increased to 18%.

In this paper, we focused on comparison between duplicate code and non-duplicate code. However, as described above, various research efforts have been performed on an open question, which is whether the presence of duplicate code actually has negative impact on software development and maintenance. Unfortunately, all the above research used only a single detection technique, and non-contiguous code clones are not considered or slightly considered. As shown in our experiment, difference detection tools may introduce different comparison results. Consequently, all the research described in this section should be replicated with multiple detection tools.

VI. Conclusion

This paper compared three duplicate code investigation methods. The result shows that different investigation methods with different detection tools bring different results. That implies existing investigation results with a single method and a single detection tool may not be reliable. However, we found that, if Krinke's method and Hotta's method with CCFinder and CCFinderX introduce the same result, the result are probably reliable. this finding is a shortcut for obtaining a reliable result.

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 Grant-in-Aid for Exploratory Research (23650014) 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.

References

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

[2] Simian. http://www.harukizaemon.com/simian/.

[3] 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.

[4] 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.

[5] N. Göde and J. Harder. Clone stability. In *Proc. of the Software Maintenance and Reengineering*, pages 65–74, Mar. 2011.

[6] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. ARIES: Refactoring Support Environment Based on Code Clone Analysis. In *Proc. of 8th IASTED International Conference on Software Engineering and Applications*, pages 222–229, Nov. 2004.

[7] Y. Higo and S. Kusumoto. Code Clone Detection on Specialized PDGs with Heuristics. In *Proc. of the 15th European Conference on Software Maintenance and Reengineering*, pages 75–84, Mar. 2011.

[8] W. Hordijk, M. L. Ponisio, and R. Wieringa. Harmfulness of Code Duplication - A Strctured Review of the Evidence. In *Proc. of the 13th International Conference on Evaluation and Assessment in Software Engineering*, Apr. 2009.

[9] K. Hotta, Y. Sano, Y. Higo, and S. Kusumoto. Is Duplicate Code More Frequently Modified than Non-duplicate Code in Software Evolution?: An Emprical Study on Open Source Software. In *Proc. of the 4th International Joint ERCIM/IWPSE Symposium on Software Evolution*, pages 73–82, Sep. 2010.

[10] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *Proc. of the 30th International Conference on Software Engineering*, May 2009.

[11] 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.

[12] C. Kapser and M. W. Godfrey. "Cloning Considered Harmful" Considered Harmful. In *Proc. of the 13th Working Conference on Reverse Engineering*, pages 19–28, Oct. 2006.

[13] J. Krinke. Is Cloned Code more stable than Non-Cloned Code? In *Proc. of the 8th IEEE International Working Conference on Source Code Analysis and Manupulation*, pages 57–66, Oct. 2008.

[14] J. Krinke. Is cloned code older than non-cloned code? In *Proc. of the 5th International Workshop on Software Clones*, pages 28–33, May 2011.

[15] 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.

[16] F. Rahman, C. Bird, and P. Devanbu. Clones: What is that smell? In *Proc. of the 7th IEEE Working Conference on Mining Software Repositories*, pages 72–81, May 2010.