An Experience Report on Analyzing Industrial Software Systems Using Code Clone Detection Techniques

Norihiro Yoshida
Graduate School of Infomation Science,
Nara Institute of Science and Technology
Email: yoshida@is.naist.jp

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

***Abstract*—A variety of application results of code clone detection and analysis has been reported. There are many reports of code clone detection and analysis on open source software whereas few reports on industrial systems are open to the public. This paper reports an experience of code clone analysis on a governmental project. In the project, a software system was developed by multiple Japanese vendors. We detected and analyzed code clones in the system, and found that there were many code clones in the project, however we concluded that the presence of the code clones did not have negative impacts on the maintenance of the system because of the following reasons: (1) when different modules are similar to each other in the design document, they also share many code clones in the source code; (2) code clones located in trusted modules, which are libraries maintained by one of the companies.**

I. Introduction

It is a general opinion that the presence of code clones makes software maintenance more difficult. A code clone is a code fragment that has identical or similar code fragments to it in the source code. Code clones are introduced by various reasons such as reusing code by 'copy and paste'. If we modify a code clone with many similar code fragments, it is necessary to consider whether or not we have to modify each of them. Especially, for large-scale software, such a process is very complicated and expensive. We sometimes overlook some of code fragments which should be modified simultaneously [4], [7], [8]. In order to detect code clones automatically, various detection techniques have been proposed [1], [11].

Several research efforts have empirically investigated whether the presence of duplicate code (code clones) is harmful or not . There are several reports on the experiences of such code clone analyses on open source software [3], [6], [9], [10]. Recently, code clone detections and analyses have started to be applied to industrial software systems in companies [12]: however, few reports on industrial software systems are open to the public because of security or other kinds of reasons.

In this paper, we report an empirical study of code clone investigation on a closed source software. The target system was developed in a governmental project. The system is a kind of *Probe Information System*, which is a system that regards a vehicle as a moving sensor. The results of sensing are transformed to the center. The center provides useful information by analyzing, accumulating and converting the sensing results.

The system was developed by five Japanese vendors. Every vendor was assigned a subsystem to be developed. The vendors hardly communicated one another during the development. The project manager, who was independent of any of the vendors, could not see the state of the source code, the number of man-hour, and the status of development process (e.g., outsourcing companies, required human resources) [1]. On the periodic meetings that the project manager held, each vendor manager reported only the followings:

- a brief summary of current development state, and
- difference between the actual progress and the plan.

For helping such a blind management of the manager, we conducted a code clone analysis. The purpose was for grasping the state of the black-boxed source code.

The remainder of this paper is organized as follows: Section II explains CCFinder and Gemini, which we used for analyzing the target system; Section III shows the experimental result; Section IV discusses what we got from the application; Section V concludes this paper.

II. Tools for Code Clone Detection and Analysis

This section explains the code clone tools that we used in the case study. Due to space limitation, we cannot describe all the features of CCFinder and Gemini. If you get interested in the tools, please refer to [2], [5].

CCFinder detects code clones from source programs, and it outputs the locations of the code clones on the source programs [5]. In the detection processing, CCFinder replaces user-defined identifiers such as variable names with special tokens, so that it can regard two similar code fragments as code clones even if they include different user-defined identifiers. The minimum size of code clones to be detected is set by a user in advance. CCFinder can complete code clone detection from systems of millions line scale in a practical timeframe.

Gemini is a code clone analysis tool [2]. Gemini takes an output of CCFinder, and visualizes the code clones included in it with several viewers. The followings are its main features.

RNR Metric: We had learned that automatic code clone detection by tools produces many false positives [2]. Herein, a

[1] Every of the five vendors are a competitor of the others in the many fields of home electronics, so that it did not report the details of its process and products to the project manager.

1530-1362/12 $26.00 © 2012 IEEE
DOI 10.1109/APSEC.2012.98

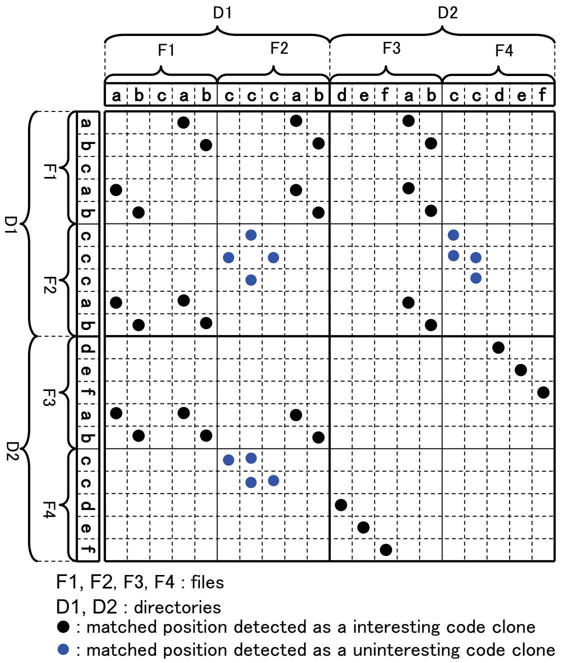
D1
D2
F1
F2
F3
F4
a b c a b c c c a b d e f a b c c d e f
F1, F2, F3, F4 : files
D1, D2 : directories
● : matched position detected as a interesting code clone
● : matched position detected as a uninteresting code clone

Fig. 1. Model of Scatter Plot

false positive means a code clone whose existence information is useless when using code clone information in software development or maintenance. For filtering out such code clones, we developed metric RNR, and concluded that an appropriate threshold of RNR is 0.5 [2]. If the RNR value of a code clone is lower than the threshold, it is regarded as a false positive.

Scatter Plot: Scatter plot is a bird's eye view visualization method for code clones. Figure 1 illustrates a model of scatter plot. Both the vertical and horizontal axes represent token sequences included in the source files that are sorted alphabetically by their file path. Source files in the same directory are close to each other. A pair of code clones is shown as a diagonal lime segment. Each dot on diagonal line segments means the corresponding tokens on the horizontal and vertical axes are identical. The dots are spread symmetrically with a diagonal line from the upper left corner to the bottom right. Using scatter plot, the distribution state of code clones can be grasped at a glance.

Code Clone Metrics and File Metrics: Gemini characterizes code clones and source files using several quantitative metrics. Also, Gemini has selective mechanisms of characterized code clones and files, so that we can easily select any of them that has caracteristics that we are interested in. For example, we used the number of quivalent code clones as a code clone metric, and the duplicate proportion of source files as a file metric.

III. Experimental Result

Herein, we describe the experimental result. The system is written in C/C++. The total LOC of the system is about hundreds thousand lines, and the source code after the combined test is 20 thousands greater than the one after the unit test. The analysis was conducted on each vendor's source code separately. In this application, we used 30 tokens as the minimum size of code clones that CCFinder detects. We also used 0.5 as the threshold of the RNR metric.

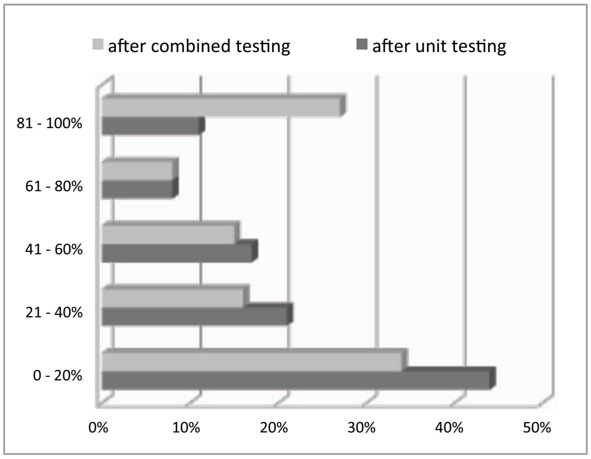

Fig. 2. Transition of the duplicated proportion of company Y

The analysis described in Subsections III-B, III-C, and III-D are for the source code after the combined testing.

A. Duplicate Proportion of Source Code

We analyzed how the amount of code clones after the unit testing is different from the one after the combined testing. Table I illustrates the number of code clones and the duplicate proportion of the subsystems after the unit testing and after the combined testing. In the subsystem developed by company Y, the number of code clones after the combined testing is remarkably greater than the one after the unit testing. Usually, after unit testing, no new function is added to the system. Thus we had predicted that the amounts of code clones between them were not so different.

Figure 2 shows how much source files had duplications. We can see that the number of high duplicated files was greatly increased. Also, Figure 3 shows code clones between the two versions and within each version of the source code of a subsystem developed by company Y. We can see that a part of the source files after combined testing is not included in the source code after unit testing (see area "E" in Figure 3). There are many code clones within the part of source files after combined testing (see area "D" in Figure). We thought

TABLE I
Amount of Code Clones in Subsystems

Co.	After unit testing		After combined testing	
	# of clones	Duplicate proportion	# of clones	Duplicate proportion
V	259	33.9%	259	33.4%
W	369	27.3%	379	26.2%
X	4,483	55.3%	4,768	50.8%
Y	6,747	42.6%	7,628	46.0%
Z	2,450	56.2%	2,505	56.3%

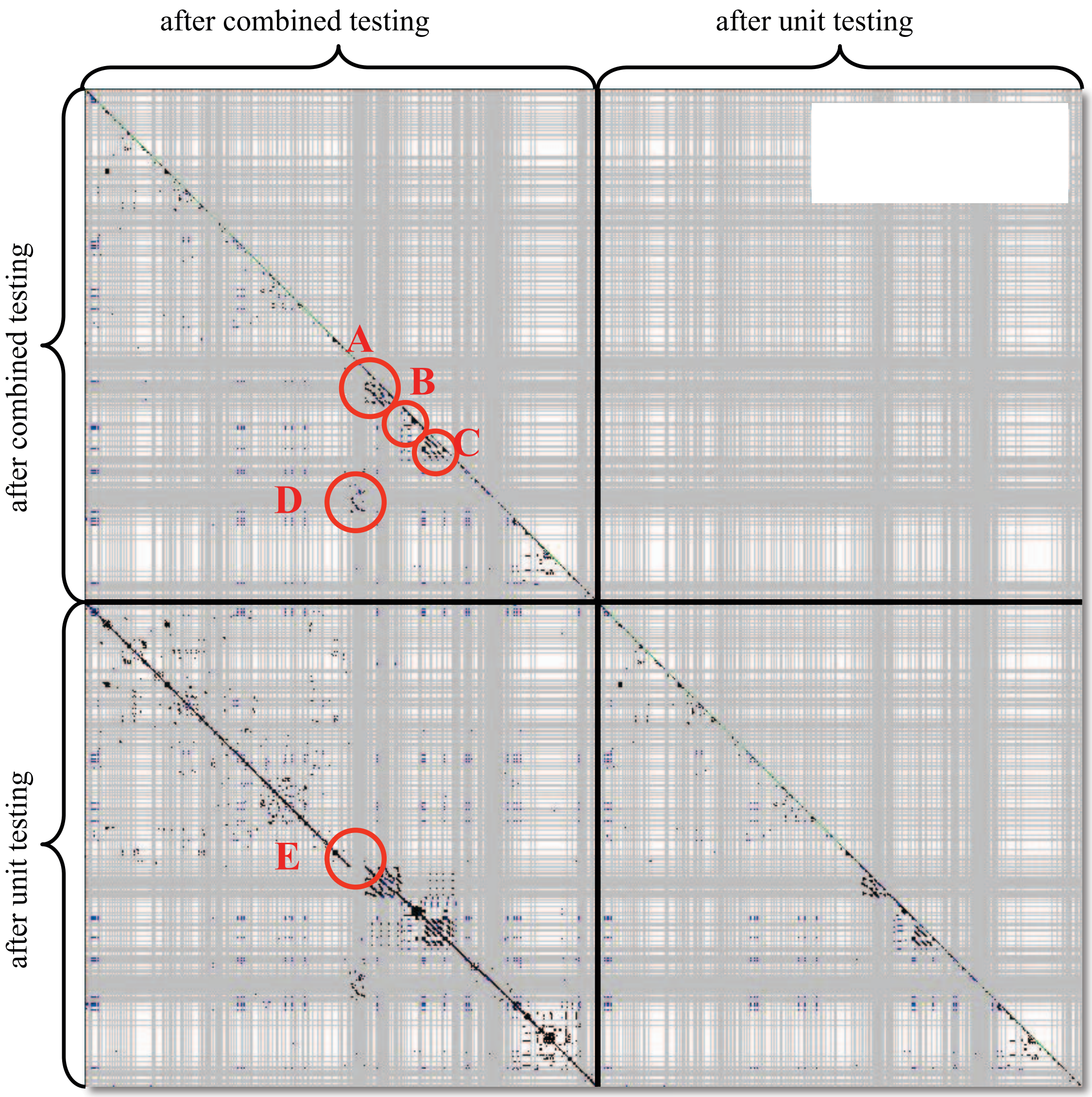

Fig. 3. Screen shot of Scatter Plot

that it was a very unusual case and interviewed the developers of company Y: they said that these files were added just before the combined testing to implement some new functions; the added files were library code managed in company Y, and they had used in many software developments; they contained many code clones but they were very stable because they had been managed in many projects.

B. Distribution of Code Clones

In Figure 3, there were many code clones in area “A”. These code clones were shared by different directories (different modules). These directories treated geographical information of vehicles, and each directory was for a kind of vehicles. The above means each directory treated different information, nevertheless, the logics were the same, which were detected as code clones.

In area “B”, we found that there were many operations related to database. Statements building SQL queries became code clones. Also, in area “C”, a large number of code clones were detected. In area “C ”, there were several directories, each of which included initialization function and finalization functions of a certain functionalities. Such initialization and finalization functions shared many code clones.

C. Subjective Evaluation by Project Manager

We herein introduce some discriminative code clones.

1) Longest code clone: In a subsystem, we detected a pair of code clones whose length (the number of tokens included in the code clone) is 154 lines. One is in file “AAXXBB.cpp” and the other is in “AYYBB.cpp”. In the code fragment of AAYYBB.cpp, some method names and comments included string "XX". This implies that a 'copy and paste' from AAXXBB.cpp to AAYYBB.cpp is performed, and forgot to modify some of names.

2) Most occurrence code clone: In all subsystems, most occurrence code clones are data validity check code (checking by using if-statement, and if not valid output error). The data formats are different from vendor to vendor, however the processes of validity checking were the same logic.

3) Code clone occurred in most source files: In a subsystem, we detected a set of code clones occurred in 8 source files. The code clones are implementations checking whether the target string ends with NULL or not. If not, Null is added to its end. These code clones deemed to be merged easily hence each of them is a whole function.

D. File Metrics Analysis

Herein, we describe some discriminative source files.

1) Source file containing most code clones: In a subsystem, we detected a source file containing 358 code clones. The code clones were scattered all over the file. They are both within-file code clones and across-files code clones. These code clones deemed not to be particularly problematical, nevertheless we thought that the maintainability of the file is not good.

2) Most duplicated source file: The duplicated ratio of two source files included in a subsystem was 96%. One is for online process, and the other is for offline process. We interviewed the developers of the system: they decided to separate implementations of the two processes before coding, and so they know the existence of the code clones.

3) Source files sharing code clones with most other source files: There was a source file sharing code clones with other 13 source files. The files included various input/output processings, each of which was duplicated with a part of the other files. Code clones were well understood logics, and they were not problematic.

IV. Discussion

This section describes what we got from the analysis of the industrial software system. Code clone analysis can be utilized for checking outsourced source code. If the amount of money to the outsourcing company is decided based on the size of the source code, the developers in the company may increase the size unnecessarily by using 'copy and paste'. There may be fully duplicated source files: we actually expose such a case in the past. If the scale of a software system is large, manual checking is unrealistic. However, automated code clone detection by tools can easily detect such inappropriate duplications.

Another usage is regarding duplicated modules as wrong designed parts, and utilizing the information so as not to repeat the same mistakes in the future. We think that we have to pay particular attentions to code clones that we didn't know their existences. Revealing why the code clones were created is an important activity for the next project.

We described two usages of code clones analysis, however there are other usages. Code clone analysis has a high general versatility because all it requires is only the source code.

V. Conclusion

In this paper, we reported a case study of code clone analysis on an industrial software system. We detect code clones from two versions of source code of the target system: one was source code after unit testing; the other was source code after combined testing. We found that 20 thousands LOC code was added to the system, and the added code included a large number of code clones: however, the added code was very stable (the code was a library used in the company) and the developers said that there was no problem in the added code. Also, we identified that there were many code clone from the source code after combined testing. Due to time limitation of analyzing code, we could not investigate all the detected code clones. Alternatively, we investigated some code clones that had discriminative features: however, we could not find problematic code clones that had a negative impact on software development. Currently, we conduct joint research on other companies. In the current research, we found some problematic code clones. Consequently, we are going to investigate characteristics of problematic and non-problematic code clones.

References

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

[2] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Method and Implementation for Investigating Code Clones in a Software System. *Information and Software Technology*, 49(9-10):985–998, Sep. 2007.

[3] 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.

[4] K. Inoue, Y. Higo, N. Yoshida, E. Choi, S. Kusumoto, K. Kim, W. Park, and E. Lee. Experience of Finding Inconsistently-Changed Bugs in Code Clones of Mobile Software. In *Proc. of the 6th International Workshop of Software Clones*, pages 94–95, June 2012.

[5] 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.

[6] 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.

[7] J. Li and M. D. Ernst. CBCD: Cloned Buggy Code Detector. In *Proc. of the 34th International Conference on Software Engineering*, pages 310–320, June 2012.

[8] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. *IEEE Transactions on Software Engineering*, 32(3):176–192, Mar. 2006.

[9] 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.

[10] M. Modal, C. K. Roy, M. S. Rahman, R. K. Saha, J. Krinke, and K. A. Schneider. Comparative Stability of Cloned and Non-cloned Code: An Empirical Study. In *Proc. of the 27th ACM Symposium on Applied Computing*, Mar. 2012.

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

[12] K. Yoshimura. Visualizing Code Clone Outbreak: An Industrial Case Study. In *Proc. of the 6th International Workshop of Software Clones*, pages 96–97, June 2012.