INFORMATION
SCIENCE
TECHNICAL
REPORT

NAIST-IS-TR2008005
ISSN 0919-9527

SHINOBI: A Real-Time Code Clone Detection Tool for Software Maintenance

Takanobu Yamashina, Hidetake Uwano, Kyohei Fushida, Yasutaka Kamei, Masataka Nagura, Shinji Kawaguchi, Hajimu Iida

March 2008

NAIST

〒 630-0192
奈良県生駒市高山町 8916-5
奈良先端科学技術大学院大学
情報科学研究科

Graduate School of Information Science
Nara Institute of Science and Technology
8916-5 Takayama, Ikoma, Nara 630-0192, Japan

SHINOBI: A Real-Time Code Clone Detection Tool for Software Maintenance

Takanobu Yamashina Hidetake Uwano Kyohei Fushida Yasutaka Kamei
Masataka Nagura Shinji Kawaguchi Hajimu Iida
Nara Institute of Science and Technology
8916-5, Takayama, Ikoma Nara, Japan
{takanobu-y, hideta-u, kyohei-f, yasuta-k, nag, kawaguti}@is.naist.jp iida@itc.naist.jp

ABSTRACT

Recent research describes how code clones in source code decrease reliability of the program and require more development cost. To solve the problem, several code clone detection methods and tools have been implemented. In this paper, we propose a novel code clone detection/modification tool to support the software maintenance process. The proposed tool, SHINOBI, indicates code clones in source code immediately by real-time clone detection. The results of an evaluation experiment showed the system had sufficient performance to support programmers in a large-scale maintenance project.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques – Program editors; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement – Restructuring, reverse engineering, and reengineering.

General Terms

Performance, Experimentation, Human Factors.

Keywords

Development Support Tool, Code-clone, Automatic Detection, Real-Time Detection, Legacy Software, Maintenance

1. INTRODUCTION

Legacy software is a program that is still well used by the community but was developed years ago. Many functions of legacy software have been modified and corrected over the years. Monden et al. show many modules in legacy software have code clones in their source code [6]. A code clone (hereafter "clone") is duplicated code in the source code. A clone increases the maintenance cost of the software because if a defect is found in one of the clones, all relative clones must be inspected one-by-one and be corrected if necessary. Also the clones decrease the reliability of the entire system [4]. Clones that are not revised in the clone modification because of clone detection omission cause remaining defects. To support the clone detection process, many studies focus on improvement of clone detection methods and evaluation of the methods [1].

However, there are few studies that analyze programmers' behavior in the software maintenance process, for instance, how programmers detect clones in their development environment, and/or what is a problem of clone detection in the maintenance process. We believe understanding programmers' behavior in clone detection is useful knowledge for development of a more efficient clone detection tool.

In this paper, we propose a clone detection/modification tool to support the software maintenance process. First, we interview programmers working on a large-scale legacy software maintenance project to understand problems in the clone detection phase and clone modification phase. Next, we propose a real-time clone detection/modification tool, SHINOBI, to resolve the problems found in the interviews. Finally, we apply SHINOBI to a legacy software maintenance project to evaluate clone detection performance and effectiveness.

2. PRE-EXPERIMENT

2.1 Overview

To study actual programmers' behavior, we conducted two preliminary experiments. In the first experiment, we investigated how many revised files contained clones when one added functionalities or fixed faults, and we confirmed how often the clones were actually fixed at the same time. In the other experiment, we conducted an ethnographic study by observing 3 programmers' coding, and we interviewed 8 programmers to clarify their motivations, method and process in maintenance behavior of handling clones.

The target system is legacy software that is a commercial CAD application. It has been developed for more than 10 years, and it consists of 4,400 files and about 1,600,000 lines. The rate of CVR[1] is about 29% when measured by CCFinderX [3] (smallest clone length is 50).

2.2 Results

2.2.1 Rate of Modified Code Clone

At first, we retrieve transactions from the CVS using the sliding window approach in [7]: two subsequent commits by the same author and with the same rationale are part of one transaction if they are at most 300 seconds apart. Then, we calculate $\boldsymbol{R_C}$ and $\boldsymbol{R_S}$ for each transaction. $\boldsymbol{R_C}$ is the rate of transactions revising files that have some clones. $\boldsymbol{R_S}$ is the rate of transactions revising files that share the same clones. $\boldsymbol{R_C}$ and $\boldsymbol{R_S}$ are calculated using the following formulas:

$$R_c = C_C / C_{all.} \qquad R_s = C_S / C_{all.}$$

[1] Ratio of tokens that are covered by any code clone

(C_C: number of transactions including modifications of a file with clones, C_S: number of transactions including modifications of files with same clones, C_{all}: number of all transactions)

If R_c is extremely low in comparison with R_s, it shows that the revised files having the same clones were not committed at the same time. It indicates that other revised files including the same clones may have been overlooked, forgotten, or unknown. Figure 1 shows a simple example of the calculation of R_c and R_s. In this example, the number of C_C is 5 (T_1,T_3,T_4,T_5,T_6), the number of C_S is 2 (T_4,T_5) ,and C_{all} is 6 (T_1 - T_6). Therefore, R_c is calculated as 5/6 and R_s is calculated as 2/6.

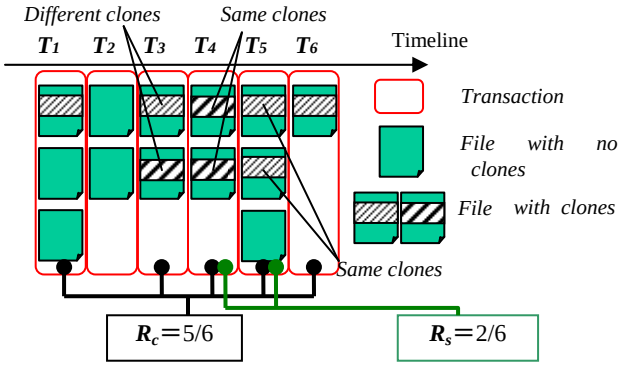

Figure 1. Simple example of calculation of R_c and R_s.

Table 1 shows the results. R_C and R_S were each calculated as 79.3% and 9.7%. This result indicates that the modification of the files with code clones during the software maintenance phase is high, but it also indicates that the developer does not necessarily modify other files related to this modification. This means that developers may not be aware of the existence of code clones during the software maintenance phase (Problem 1).

Table 1. Analysis result of modification of file with code clones

name	value
Number of all transactions (C_{all})	1890
Number of transactions including modifications of a file with clones (C_C)	1498 R_c = 79.3%
Number of transactions including modifications of files with same clones. (C_S)	183 R_s = 9.7%

2.2.2 Observation and Interviews

MOTIVATION

Compared to veteran programmers, novice programmers tended to revise only defective codes they found first, and not to search and revise their clones (Problem 2). They seldom knew whether clones of revised code existed. As they did not know where and how to search for clones, it would take a lot of time if they had to do so. Moreover, it is difficult to decide whether they should revise the files even if they are able to search for those files (Problem 3).

METHOD

All programmers used GREP very frequently when they revised source code, and they used it for searching for similar codes like clones. However, providing appropriate keywords for GREP and limiting the search target requires deep understanding of the structure and terminology of the whole system. Therefore, it is difficult for a novice to search for his target precisely with low cost. It is more difficult to do it in large-scale legacy software because the target is vast and the terms are not standardized (Problem 4). Furthermore, by a questionnaire, it was difficult for a veteran to find a clone, especially if their variable names are changed (Problem 5).

PROCESS

In the maintenance process, a veteran's code revising process was different from a novice's. The veteran started with confirming the range where the revised codes and their clones would have influence. Then, he decided a revision strategy and revised fault. On the other hand, the novice revised the fault first, and then began to search where the change had effect. Such blinkered, ad-hoc changing may lack completeness, and novices could not realize their mistake until they searched other influenced regions. It makes software maintenance more time-consuming. In the maintenance of a system that is already operating, we should decide a revision strategy before revising the clones (Problem 6).

2.3 Requirements for Code Clone Detection in Software Maintenance

To solve the problems in Section 2.2.1 and Section 2.2.2, we propose the following requirements that clone detection tool should fulfill.

Requirement 1: From Problems 1, 2 and 4, it is necessary to detect clones without the programmer's clear intention. In addition, from Problem 6, it is necessary to detect clones before programmers revise the source code.

Requirement 2: From Problem 3, it is difficult to make a decision as to whether clones should be revised only by looking at them. To support decision-making, it is necessary to display additional information.

Requirement 3: From Problem 5, it is necessary to detect clones even when a variable name is changed.

Requirement 4: It is necessary to detect clones fast in large-scale legacy software.

3. SHINOBI: REAL-TIME CLONE DETECTOR

We suggest SHINOBI (Rapid and Runtime Duplication Detector) as a new tool for solving the problems acquired through the pre-experiment.

3.1 Feature

We implement the following functions to satisfy requirements described in Section 2.3

- From Requirement 1, SHINOBI is supported as an Add-In of Microsoft Visual Studio 2005. It automatically detects clones without the programmer's clear intention at the time of opening and editing source code, and constantly displays the detected clones on the view in IDE. Whenever the programmer moves the cursor on the source code editor,

SHINOBI automatically detects clones before programmers revise the source code.

- From Requirement 2, SHINOBI displays the detected clones in order of the ranking f the similarity between the source code on the cursor and the detected clones, and the information in the CVS repository, such as message logs and committed dates of the source code that is detected clones.
- From Requirements 3 and 4, SHINOBI has a token-based clone detection engine. This engine is nearly unaffected by the change of a variable identifier, and it works fast in large-scale software.

3.2 Tool Overview

Figure 2 shows the architecture and data flow of all of SHINOBI. This tool is a Windows application developed on C++ and C#. SHINOBI consists of the SHINOBI Server and SHINOBI Client.

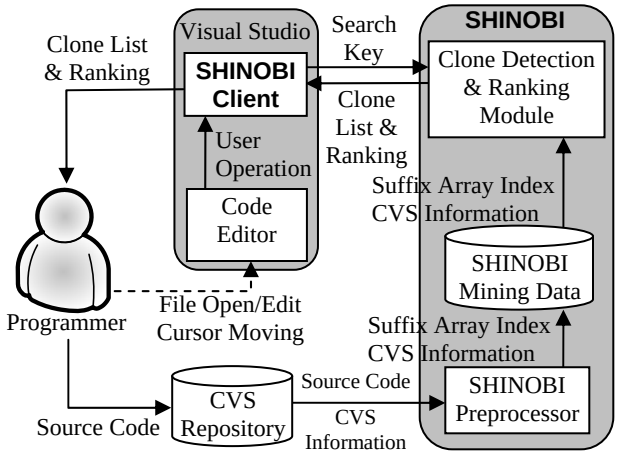

Figure 2. SHINOBI architecture.

3.3 SHINOBI Server

The SHINOBI server performs clone retrieving and ranking. At first, the SHINOBI server parses the CVS repository and prepares SHINOBI Mining Data. After that, whenever a clone search request arrives from the SHINOBI Client, the SHINOBI Server detects clones and sends the results to the SHINOBI Client.

The SHINOBI Preprocessor automatically acquires source code and history information from the CVS Repository whenever the CVS Repository is updated. It parses all the latest source code and old revision source code from the CVS Repository and creates an index using the Suffix Array [5] technique. We call it the Suffix Array Index. The Suffix Array Index is essential to detect clones very quickly. As a parsing tool, we use CCFinderX's preprocessor, which unifies identifiers to ignore differences of identifiers in source code. The SHINOBI Preprocessor also analyzes commit information and source-code differences. Such CVS Information is also stored in the SHINOBI Mining Data.

The Clone Detection & Ranking Module searches for clones with the Search Key sent from the SHINOBI client using the Suffix Array Index. The order of returned clones is determined by Ranking Value. The Ranking Value is the sum of two values: 1) the ratio of files committed at the same time and 2) the ratio of files opened or edited at the same period in Visual Studio.

3.4 SHINOBI Client

The SHINOBI client, an Add-In for Visual Studio, always displays clones that are similar to the code where the cursor is located.

3.4.1 IDE Add-In

Figure 3 shows our SHINOBI as an Add-In for the Visual Studio programming environment. The right pane in Figure 3 is the Code Clone View. Code Clone View always shows the clones related with the region around the cursor. When the cursor position is changed or the region around the cursor is edited, the SHINOBI Client detects such interaction and automatically updates the listed clones.

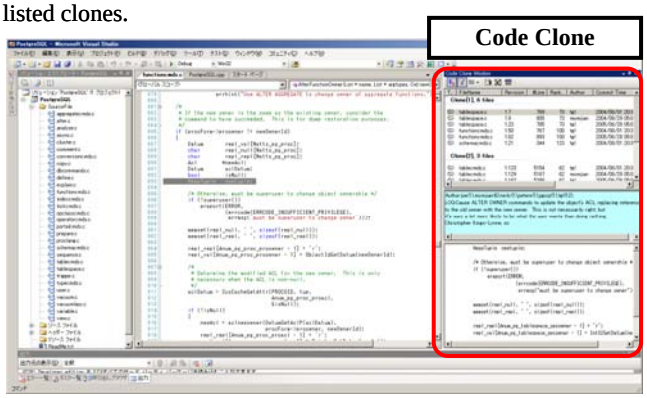

Figure 3. Screenshot of SHINOBI Add-In for Visual Studio.

3.4.2 Code Clone View

Figure 4 shows a screenshot of Code Clone View. It consists of two views, Clone List View and File Information View.

In a toolbar, some filter control buttons are arranged. You can filter out some clones by toggling some conditions. The conditions you can choose are described in Figure 4.

Clone List View displays the list of the clones detected by SHINOBI. For each listed clone, Clone List View shows a file name, revision, line number, Ranking Value, the author of the last commitment, update day, comment, and the revision of creating the clone.

When you choose a file in Clone List View, SHINOBI displays the contents of the selected file and opens the file in Visual Studio.

File Information View displays detailed contents of the clone that you chose in Clone List View. It shows the actual source code of the clone and additional information such as authors, the number of times grouped by them, and comment contents.

3.5 Comparison to Existing Tools

Existing clone detection tools like CCFinderX or ICCA [2] analyze the entire software at once. Although it is useful when you analyze clones in the entire software, many programmers need clones only related to fixed code sections and the remaining sections are ignored. In addition, existing tools require the user to install stand-alone applications and learn how to use them. That has a very high cost if there are many programmers and all of them need to use it. However, they are suitable for analyzing clones in the entire software by a skilled analyzer, but they are not intended for use by many programmers in software maintenance.

Compared to existing tools, SHINOBI is easy to introduce and familiar for Visual Studio programmers because it is implemented as a Visual Studio Add-In. SHINOBI automatically tells a programmer where the clones exist without any user operations. Thus if a programmer fixes some source code, they can know where the duplicated codes exist without any operations. Since SHINOBI detects clone only related to the source code that the user is looking at, SHINOBI can react very quickly. Moreover, SHINOBI provides a great deal of useful information such as a revision history or indication of the commitment information. Furthermore, SHINOBI always stores operations of the users and precision of ranking will rise as much as you use it by reflecting users' decisions. Because you can confirm the timing when clones are made or refactored, it is easy to make a correction strategy. Therefore, SHINOBI would considerably improve maintenance efficiency.

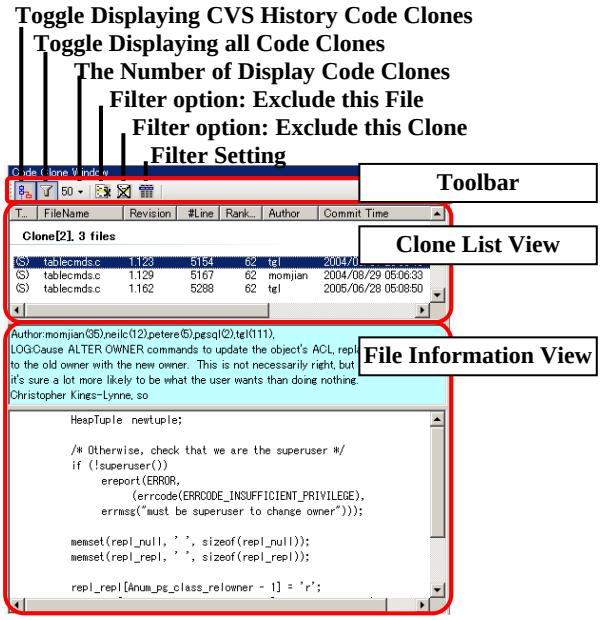

Figure 4. Screenshot of Code Clone View.

4. EVALUATION

We performed an experiment with SHINOBI. We analyzed how many resources were necessary and how much time was necessary for detecting clones. About both experiments, we used source code of the commercial application that we described in Section 2.1. We decided the Ranking Value using the rule described in Section 3.3. However, because SHINOBI has not been used in an actual development environment, we did not consider the ratio of files referred to with an opened or edited file in Visual Studio.

4.1 Evaluation Setting

We measured execution time to evaluate scalability and resource consumption. At this time, we increased step-by-step the number of subsystems to be analyzed by SHINOBI.

Then we evaluated the effectiveness by an intentional debugging to measure detection accuracy. We prepared 24 modification points to be debugged beforehand, and SHINOBI detected the points. On the other hand, we detected the points by using GREP command. We compared these detected points. In addition, we obtained the values of Recall and Precision.

All measuring was run on a 3.6 GHz Pentium IV Windows XP machine with 1 GB memory.

4.2 Results

Table 2 shows the results of measuring execution time. On average, SHINOBI can detect clones within less than 0.5 sec even in the case that it detects from more than 3,500,000 LOC legacy code. Suffix Array Index Loading is the processing to load indexes into memory and its time is increased linearly with the amount of code. However, Clone Detection time is not affected so much by the increases of LOC. Clone Detection time is proportional to the log of LOC ideally because the searching of the Suffix Array is performed as a binary search. Note that Suffix Array Index Loading Time can be decreased when the SHINOBI server always loads indexes into memory.

Table 2. Execution time

Num of subsystem		12	4	1
Num. of files		13,844	9,260	4,377
Num. of Tokens(M)		21.9	16.7	8.4
File Size (Mbyte)		401	366	186
Lines of code(MLOC)		4.5	3.2	1.7
Suffix Array Index Size (Mbyte)		110	84	42
Execution Time	**Suffix Array Index Loading (ms)**	250	187	156
	Token Extraction (ms)	140	172	156
	Clone Detection (ms)	14	6	1
	Total (ms)	404	365	313

Table 3 shows the detection accuracy. The number of detections using GREP is much higher than that using SHINOBI. It causes values using GREP to have high Recall value and very low Precision value, namely, detected points using GREP contain many misses. Therefore, detection accuracy using SHINOBI is much higher than using GREP command in general.

Table 3. Detection accuracy

	# Detection	Time	Recall	Precision
GREP	205	62.0sec	96%	11%
SHINOBI	20	0.7sec	83%	100%

4.3 Discussion

SHINOBI is superior with both execution time and detection accuracy. SHINOBI needs extra volume of the Suffix Array Index to detect clones fast. About a quarter of File Size is needed for the Suffix Array Index. When File Size is too large, it may also be so large. However, this indexing technique gives a very high advantage for execution time and File Size will be limited. Considering this advantage, we think this problem is not important.

5. CONCLUSION AND FUTURE WORK

In this article, we performed observation and interviews of the maintenance work from the viewpoint of clones, and we analyzed requirements for the clone detection tool. Then, we implemented it as SHINOBI. This is executed as an Add-in of Visual Studio, and always automatically displays the clone information that is useful for programmer. In addition, we confirmed SHINOBI performed rapidly for commercial large-scale legacy software with certain accuracy. Furthermore, SHINOBI would reduce

unthinking copy & paste because they are always aware where clones exist.

In a future study, we will confirm the effectiveness of SHINOBI. To confirm it, we will examine whether introducing SHINOBI has a good effect for the ratio of revising clone files and the ratio of increasing clones. In addition, we will observe whether SHINOBI changes novice programmer behavior to search for clones and to make a revision strategy at first. We also need to evaluate the ranking of clones and improve the ranking calculation algorithm if we need it. Subsequently, we want to extend SHINOBI to suggest other useful information to support understanding source code.

6. ACKNOWLEDGMENTS

We thank to employees at Nihon Unisys Ltd. for their cooperation in our experiments. This work was supported by the Comprehensive Development of e-Society Foundation Software program of MEXT, and Stage Project, the Development of Next Generation IT Infrastructure, supported by MEXT. This work was partially supported by JSPS KAKENHI (18800024).

7. REFERENCES

[1] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, " Comparison and Evaluation of Clone Detection Tools, " IEEE Trans. on Software Engineering, vol. 33, no. 9, pp. 577-591, 2007.

[2] http://sel.ist.osaka-u.ac.jp/icca/index-e.html

[3] http://www.ccfinder.net/ccfinderx.html

[4] B. Lague, D. Proulx, J. Mayrand, E. M. Merlo, and J. Hudepohl, " Assessing the Benefits of Incorporating Function Clone Detection in a Development Process," Proc. of IEEE Int'l Conf. on Software Maintenance, pp. 314-321, 1997.

[5] U. Manber and G. Myers, “Suffix Arrays: A New Method for On-Line String Searches, " SIAM J. Comput. 22(5), pp. 935-948, 1993.

[6] A. Monden, D. Nakae, T. Kamiya, S. Sato, and K. Matsumoto, "Software Quality Analysis by Code Clones in Industrial Legacy Software, " IEEE Symposium on Software Metrics, pp. 87-94, 2002.

[7] T. Zimmermann, P. Weissgerber, S. Diehl, and A. Zeller. "Mining Version Histories to Guide Software Changes, " Proc. of IEEE Int'l Conf. on Software Engineering, pp. 563–572, 2004.