On Software Maintenance Process Improvement Based On Code Clone Analysis

Yoshiki Higo[1],Yasushi Ueda[1],Toshihro Kamiya[2],
Shinji Kusumoto[1] and Katsuro Inoue[1]

[1] Graduate School of Information Science and Technology, Osaka University,
Toyonaka, Osaka 560-8531, Japan
Phone:+81-6-6850-6571,Fax:+81-6-6850-6574
{y-higo,y-ueda,kusumoto,inoue}@ist.osaka-u.ac.jp
[2] PRESTO,Japan Science and Technology Corp.
Current Address:Graduate School of Information Science and Technology, Osaka University,
Toyonaka, Osaka 560-8531, Japan
Phone:+81-6-6850-6571,Fax:+81-6-6850-6574
kamiya@ist.osaka-u.ac.jp

Abstract. Maintaining software systems is getting more complex and difficult task. Code clone is one of the factors that make software maintenance more difficult. A code clone is a code portion in source files that is identical or similar to another. If some faults are found in a code clone, it is necessary to correct the faults in its all code clones. We have developed a maintenance support environment, Gemini, which provides the user with the useful functions to analyze the code clones and modify them. However, through case studies, several problems were reported. That is, the clones provided by Gemini were not appropriate to merge into one module. In this paper, we intend to extend the functionality of Gemini to cope with the problems. Finally, we apply the extended Gemini to several software and evaluate the applicability of the new functions.

1 Introduction

As the size and the complexity of software increase, it becomes important to develop high-quality software cost-effectively within a specified period. Software process improvement is one of the promising method to attain it.

Recently, it is pointed out that maintenance phase is the most expensive one in the entire software development process. Many research studies have reported that large software companies spent a lot of cost to maintaining the existing systems. Maintenance of software system is defined as modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the products to a modified environment[20].

Code clone is one of the factors that make software maintenance more difficult[8]. A code clone is a code portion in source files that is identical or similar to another. Clones are introduced because of various reasons such as reusing code

by 'copy-and-paste' and so on. Code clones make the source files very hard to modify consistently. For example, when a fault is found in one clone, it must be carefully modified all the clones. So, effective code clone detection will support the improvement of software maintenance process. In order to detect the code clones effectively, various clone detection methods have been proposed.

We have developed a maintenance support environment, Gemini, which provides the user with the useful functions to analyze the code clones and modify them[22]. CCFinder[13] is one of the components of Gemini and used to detect code clones. Gemini primarily provides two diagrams: scatter plot and metrics graph. The scatter plot graphically shows the locations of code clones among source codes. The metrics graph shows metric value of each clone and has a feature to identify the distinctive code clones. Using the diagrams, we expected that maintenance process can be improved.

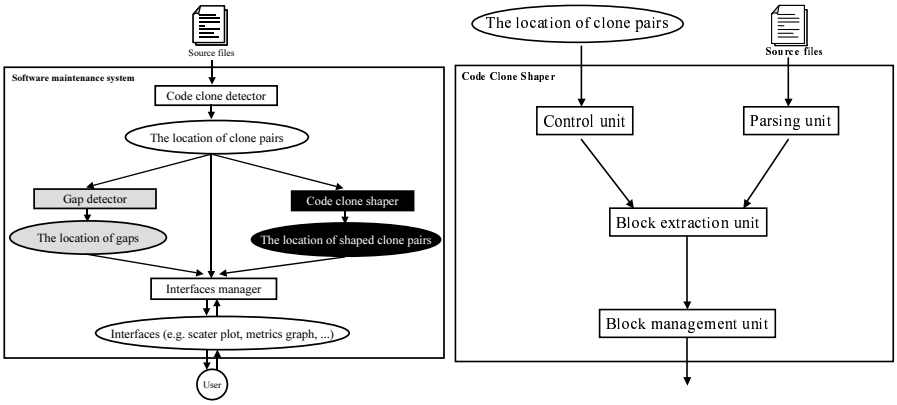
We have delivered Gemini to several software companies and evaluated it through case studies. In the case studies, we have received several practical problems. First one has been appeared in applying Gemini to refactoring activities[8]. Usually, code clones are merged into one module(procedure,function,macro etc). The clones detected by Gemini were not appropriate to be merged, since it detects the maximal code clones that often include excessive tokens that should be omitted in merging the clones into one routine. Second is one how to identify the modified code portions as clone. As described above, code clone is introduced copy-and-paste programming. But, in most case, the copied-and-pasted code portion is not used as it was. Usually, some statements are inserted to the code portion or deleted from it. The practitioners in the company want to extract such modified code clones (called gapped clone) but Gemini cannot find them.

In this paper, we intend to solve the above issues to extend the functionality of Gemini. For the former issue, we have added the new function to extract the part of code clone which is easy to merge one module. For the latter issue, we propose a method to show all the candidates of gapped code clones. As spaces are limited, we mainly explain the first topics. Finally, we apply Gemini to several software and evaluate the applicability of the proposed method.

2 Code Clone Analysis

2.1 Definitions on code clone

A clone relation is defined as an equivalence relation (i.e., reflexive, transitive, and symmetric relation) on code portions[13]. A clone relation holds between two code portions if (and only if) they are the same sequences. (Sequences are sometimes original character strings, strings without white spaces, sequences of token type, and transformed token sequences.) For a given clone relation, a pair of code portions is called clone pair if the clone relation holds between the portions. An equivalence class of clone relation is called clone class. That is, a clone class is a maximal set of code portions in which a clone relation holds between any pair of code portions.



(a) Architecture

(b) Code Clone Shaper

Fig. 1. Overview of Gemini

A code portion in a clone class of a program is called a code clone or simply a clone.

2.2 Maintenance support environment: Gemini

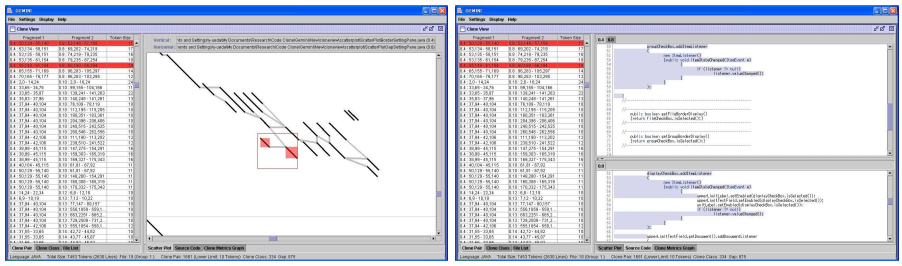
In [21], we have developed a maintenance support environment based on code clone analysis (called Gemini). Figure 1(a) shows the system architecture. In Figure 1(a), the gray parts (a gray quadrilateral and ellipse) have been proposed in [22] and the black parts(, which is enlarged in Figure 1(b).) will be proposed in Section 3 in this paper. Basically, Gemini delivers the source files to the code clone detector, CCFinder[13], and then shows the information of the detected code clones to the user through various GUIs.

In this Section, we briefly explain the characteristic of CCFinder and Gemini.

Tool: CCFinder CCFinder detects code clones from programs and outputs the locations of the clone pairs on the programs. The length of minimum clone is set by user before.

Clone detection of CCFinder is a process in which the input is source files and the output is clone pairs. The process consists of four steps:

Step1 Lexical analysis: Each line of source files is divided into tokens corresponding to a lexical rule of the programming language. The tokens of all source files are concatenated into a single token sequence, so that finding clones in multiple files is performed in the same way as single file analysis.



(a) scatter plot (b) corresponding code

Fig. 2. Gemini snap shots

Step2 Transformation: The token sequence is transformed, i.e., tokens are added, removed, or changed based on the transformation rules that aims at regularization of identifiers and identification of structures. Then, each identifier related to types, variables, and constants is replaced with a special token. This replacement makes code portions with different variable names clone pairs.

Step3 Match Detection: From all the sub-strings on the transformed token sequence, equivalent pairs are detected as clone pairs.

Step4 Formatting: Each location of clone pair is converted into line numbers on the original source files.

Details of CCFinder have been shown in [13].

Tool: Gemini Gemini is a GUI-based code clone analysis environment which uses CCFinder as a code clone detector. Gemini provides to the users the following view windows that enable an interactive code clone analysis:

- Scatter plot view,
- Metric graph view, and
- Source code view.

Scatter plot view shows visually where clone pairs exist in source files. It is very effective mechanism in early phase of code clone analysis since the state of distribution of code clone can be grasped at a glance. In the view,user can select clone pairs by mouse dragging. Figure 2(a) shows an example of scatter plot view. The detail of scatter plot will be described later.

Metric graph view is designed for enabling the users to select clones by the quantitative characteristics of them. In metric graph view, user can select clone pairs or classes by the values of metric for each clone class to easily select the distinctive ones.

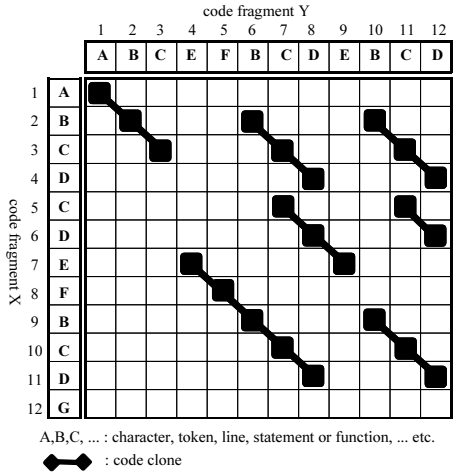


Fig. 3. Scatter plot of code clones

The source code view works cooperating the scatter plot view on the metric graph view. The user can obtain the actual source code corresponding to clones selected in the other views. Figure 2(b) shows an example of source code view.

Scatter plot Figure 3 shows examples of scatter plot. Both the vertical and horizontal axes represent code portions of source files. The following two sequences are used as sample code portions in the scatter plot.

code portion X: "`ABCDCDEFBCDG`",
code portion Y: "`ABCEFBCDEBCD`"

Here, symbols "`A`","`B`","`C`",... are code portions in an unit such as character, token, line, statement, function, etc (In Gemini, it is token). In Figure 3, each small black square means that corresponding two elements on the two axes are the same. So, a clone pair is shown as a diagonal line segment. If the same code portions are arranged on the two axes, naturally, a diagonal line from the upper left to the lower right is drawn since such dot means comparison of token with itself, and the dots are symmetrical with a diagonal line.

The state of distribution of code clone can be grasped at a glance. However, as for large scale software in which there are many code clones, it is very difficult to decide which plot (that is code clone) in the huge scatter plot should be kept our eyes on. That is, if many files are located on the axis of coordinate in naive order, such as alphabetical order with file name, the distribution of code clones is occasionally spread widely without conspicuous deviation. So, Gemini has the function to sort the order of files on the two axes. It causes code clones not to distribute all over a scatter plot as much as possible. As a basic idea, the more

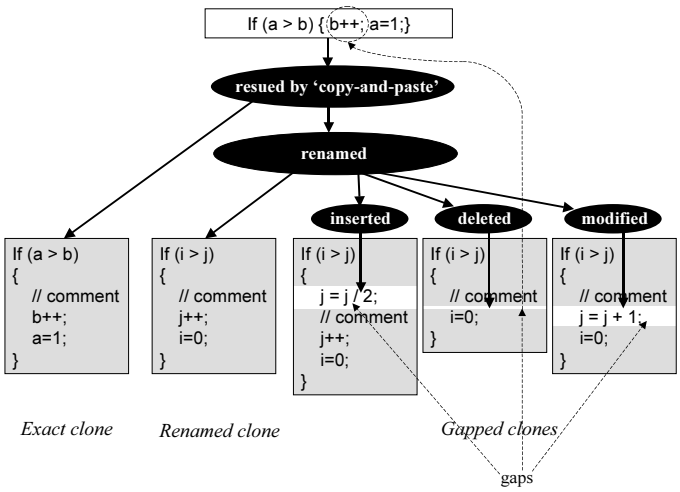


Fig. 4. Gapped clone

code clones exist among two source files, the nearer the files are to be located in each axis. The details is described in [21].

3 Proposed Method

3.1 Problems found in case studies

We have applied Gemini (and CCFinder) to several commercial software products. In the case studies, the users reported some problems as feedback. Among them, the following two problems are repeatedly reported and serious ones.

As for the first one, in the case of 'copy-and-paste' reuse, the developers usually do not reuse the code portion as it was but partially modifies and then reuse it. Moreover, in the modification, they do not only replace the user-defined identifiers in the code portion but also modify it. For example, additional statements would be inserted into it. Thus, some differences exist between the original code portion and the copied-and-pasted one. Here, we call the each difference "gap" and such code clone as "gapped clone". From a viewpoint of how to reuse code, we classify code clone into five categories shown in Figure 4. Then, CCFinder can only detect exact clones and renamed clones.

In such case, the developers can subjectively identify the code clones even if they include some gaps among them. On the other hand, CCFinder detects the clone as several short code clones separately. Or, since the minimum length of a code clone must be set in CCFinder beforehand, if the code portion is too short, CCFinder does not identify it as a code clone. Conversely, if we set a small value to the minimum length, then a lot of code clones are detected and the information is practically useless.

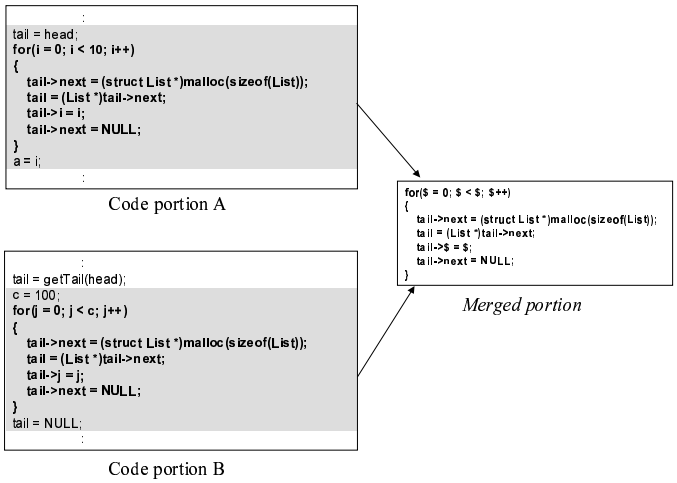


Fig. 5. Example of merging two code portions

In [22], we proposed the solution of this problem. In the paper, we could refer to a certain set of gapped clones by representing visually exact/renamed clones and gaps themselves on scatter plot. In fact, the complexity of detecting all gapped clones one by one is massive (square of number of exact/renamed clones). So, we took the alternative solution.

Next, as for the second problem, if we detect code clones for refactoring[8], sometimes semantically cohesive ones has more important meaning than maximal (just longest in local) ones although the formers may be shorter than the latters. In our experiments, we found many clones that have not only primary logic statements but also the other coincidental clone statements before (and/or behind) them, since simple statements, such as assignment or variable declaration, tend to become clones coincidentally. Figure 5 shows an example. In Figure 5, there are two code portions A and B from a program, and the code portions with hatching are maximal clones among them. In code portion A, some data are substituted to list data structure from the head successively. In code portion B, they are done so from the tail successively. There is a common logic between these two processes that is code portions handling list data structure (in `for` block). However before and behind `for` block there are sentences that are identified as a part of code clones coincidentally. It can be said that such blocks without coincidental portions are preferred to whole portions with hatching in the figure in the view point of refactoring.

In [14] and [15], they detect semantically cohesive code clones using program dependence graph (PDG) for the purpose of procedure extraction and so on. However, currently, there are no examples of the application of their approaches to large scale softwares since the cost to create PDG is very high. On the other hand, the clone detection process of CCFinder is very fast but only

lexical analysis is performed. So, the detected clones are just maximal and not always semantically cohesive. Hence it is necessary for the user of CCFinder to extract semantically cohesive portions manually from the maximal.

To solve this problem, we take a two-step approach in which we firstly detect maximal clones and secondly extract semantically cohesive ones from the results. By this approach, in practical time, we can detect code clones that are easy to be reused. The details are explained in next section.

3.2 Approach

Here, we define Shaped Clone as the merge-oriented code clone extracted from the clones detected by CCFinder. We explain the way how to extract the Shaped Clone. The extracting process consists of the following three steps:

STEP1: CCFinder is performed and clone pairs are detected.
STEP2: By parsing the inputted source files and investigating the positions of blocks, semantic information (body of method, loop and so on) is given to each block.
STEP3: Using the information of location of clone pairs and semantics of blocks, meaningful blocks in the code clone are extracted. Here, intuitively, meaningful block indicates the part of code clone that is easy to merge.

3.3 Implementation

We have implemented the shaped clone detection function(Code Clone Shaper in Figure 1(a)) in Gemini. The size of the function is about 10KLOC and implemented in Java. The target source files are also Java programs.

We explain the implementation of the proposed shaped clone detection method. The implementation includes the following units shown in Figure 1(b):

- Control unit
- Parsing unit
- Block extraction unit
- Block management unit

Control unit Control unit invokes the Parsing unit, Block extraction unit, and Block management unit through reading the code clone information (output from CCFinder).

Parsing unit Parsing unit conducts lexical and syntax analysis for the inputted source files. Here, we define **Block** as code portion enclosed by a pair of brackets. So we use only the result of lexical analysis in this paper and the information about syntax will be taken into the consideration in our future research. Then, the location information of the extracted token is stored. It is implemented using JavaCC[11].

Block extraction unit Block extraction unit extracts the block from the code clones detected by CCFinder using the stored data and analysis results from CCFinder.

Block management unit Block management unit puts the blocks extracted by Block extraction unit in an appropriate order. It is necessary to obtain the consistency of the data used in Gemini.

4 Case Study

In order to evaluate the usefulness of the proposed shaped clone detection method, we have applied it to famous Java software:ANTLR[2] and Ant[1].

ANTLR(ANother Tool for Language Recognition,) is a language tool that provides a framework for constructing recognizers, compilers, and translators from grammatical descriptions containing C++ or Java actions.

Ant is another Java based build tool. Instead of a model where Ant is extended with shell based commands, it is extended using Java classes. Instead of writing shell commands, the configuration files are XML based calling out a target tree where various tasks get executed. Each task is run by an object which implements a particular task interface.

In the evaluation, we have applied Gemini without using Code Clone Shaper and Gemini with it to the data, independently. Then, we compare the results. In this case study, we have set the minimum length of a code clone as 50 tokens.

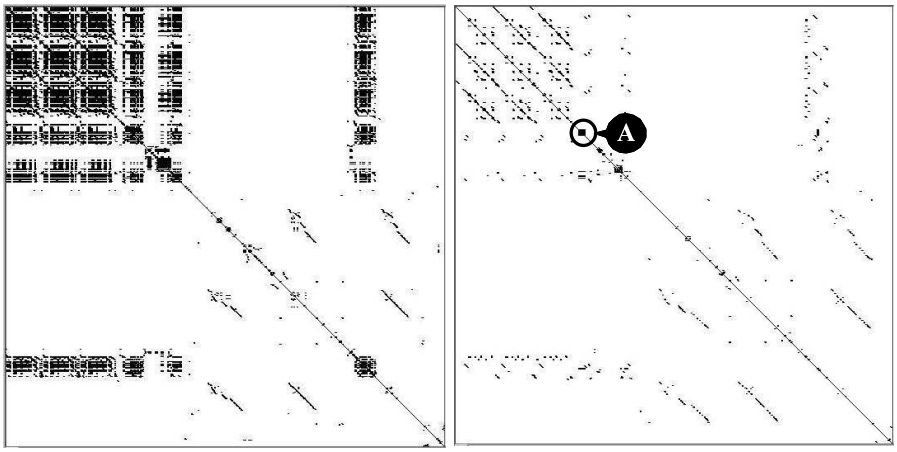
Table 1. Source code size

	Number of files	Lines of code	Number of tokens
ANTLR	239	43548	140802
Ant	508	141254	221203

4.1 ANTLR

ANTLR includes 239 files and the size is about 44KLOC(see in Table 1). Figure 6(a) shows the results of applying the Gemini without Code Clone Shaper. You can see that there are a lot of clones in ANTLR. Here, we can find 338574 clone pairs and 1072 clone classes. So, it is very difficult to extract the clones that can be merged into one module.

On the other hand, Figure 6(b) shows the results of applying the Gemini with Code Clone Shaper. You can see that non-meaningful clones are omitted. Here, we can find 972 clone pairs and 142 clone classes. The reduction rate of the number of clone pairs and clone classes are about 1/350 and 1/8, respectively(see in Figure 6(c)).



(a) Result without Code Clone Shaper (b) Result with Code Clone Shaper

	without Code Clone Shaper	with Code Clone Shaper
Number of Clone Pair	338574	972
Number of Clone Class	1072	142

(c) Numbers of clone

Fig. 6. Result of ANTLR analysis

Then, we checked the part labeled A in Figure 6(b) and found distinctive code clones. There are 28 clones and each of them include 82 tokens. We can easily merge the clones to one method by adding two parameters shown in Figure 7. Code portions on the left side are clones provided by Gemini with Code Clone Shaper. If they are merge into one method,it will be like the code portion on the right side.

4.2 Ant

Next, we applied Gemini to Ant. Ant includes 508 files and the size is 141KLOC(see in Table 1). Figure 8(a) shows the results of applying the Gemini without Code Clone Shaper. You can see that code clones spread over the scatter plot. Here, 12033 clone pairs and 856 clone classes were detected. On the other hand, Figure 8(b) shows the results of applying the Gemini with Code Clone Shaper. Here, 103 clone pairs and 53 clone classes were detected .

You can see that most of the clones are omitted and the part labeled B stands out. Figure 8(d) shows the actual code clones of it. We found seven separate

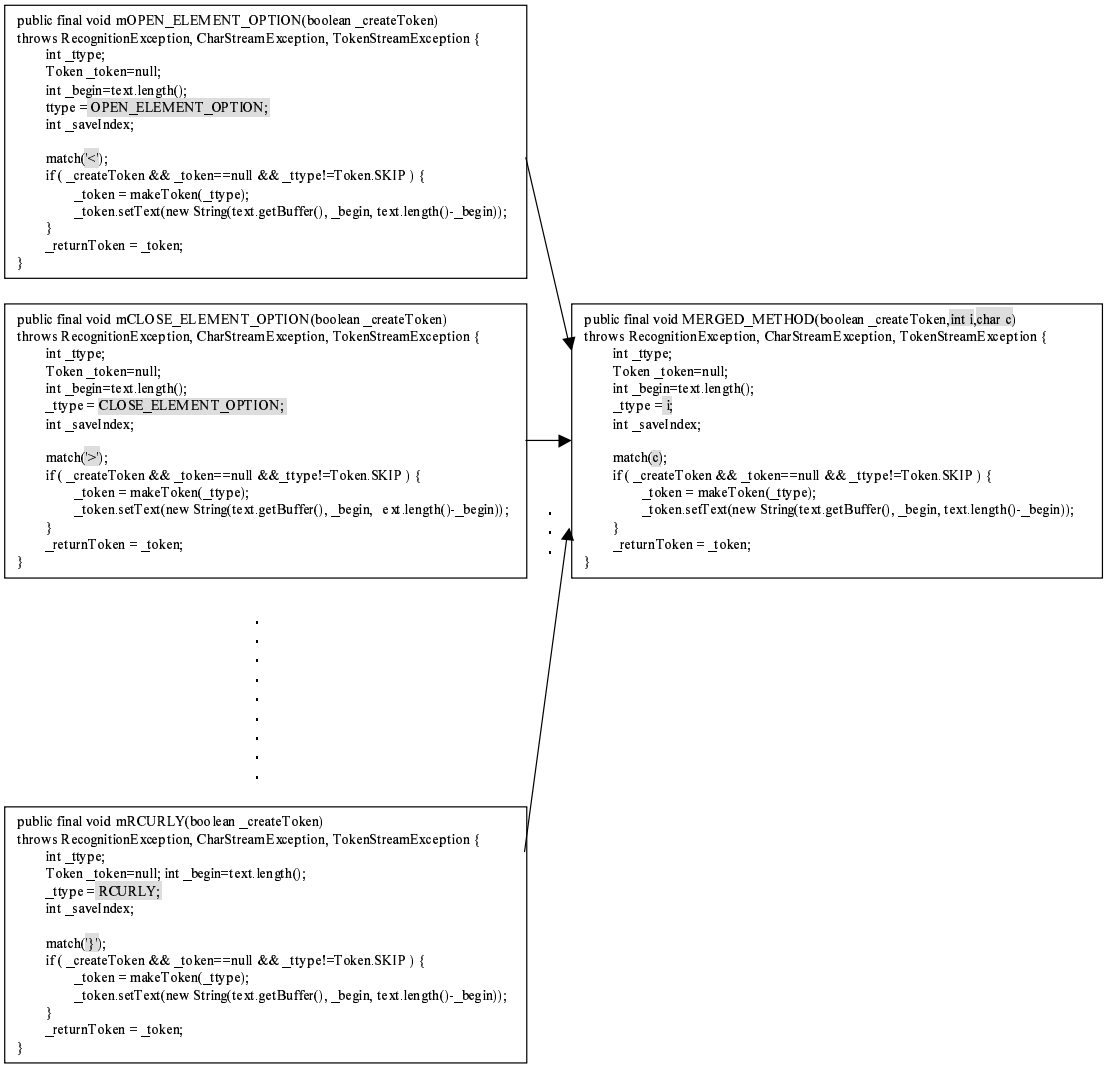
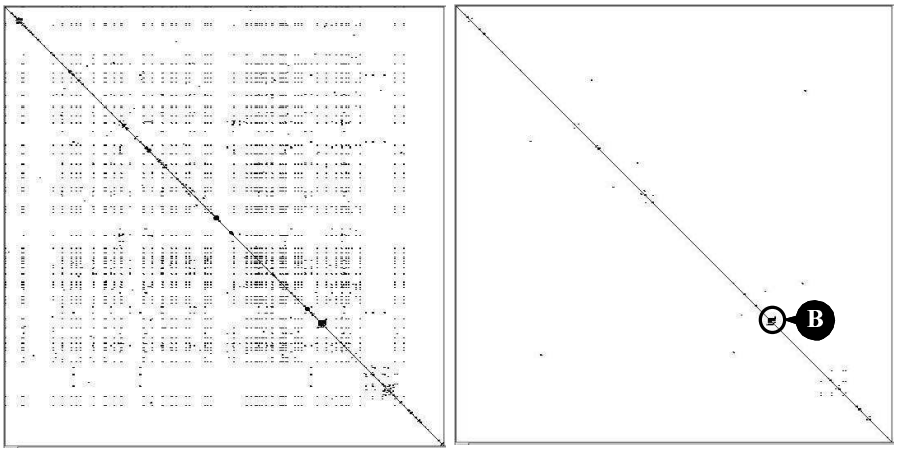


Fig. 7. Merged clone sample in ANTLR



(a) Result without Code Clone Shaper (b) Result with Code Clone Shaper

	without Code Clone Shaper	with Code Clone Shaper
Number of Clone Pair	12033	103
Number of Clone Class	856	53

(c) Numbers of clone

```
public void getAutoresponse(Commandline cmd) {
  if (m_AutoResponse == null) {
    cmd.createArgument().setValue(FLAG_AUTORESPONSE_DEF);
  } else if (m_AutoResponse.equalsIgnoreCase("Y")) {
    cmd.createArgument().setValue(FLAG_AUTORESPONSE_YES);
  } else if (m_AutoResponse.equalsIgnoreCase("N")) {
    cmd.createArgument().setValue(FLAG_AUTORESPONSE_NO);
  } else {
    cmd.createArgument().setValue(FLAG_AUTORESPONSE_DEF);
  } // end of else
}
```

(d) Entirely same clone in Ant

Fig. 8. Result of Ant analysis

methods in the several files. Since the methods inherit the same super class, we can remove the clones easily by moving the method to the super class.

Also, the reduction rate of the number of clone pairs and clone classes are about 1/120 and 1/16, respectively(see in Figure 8(c)).

5 Conclusion

In this paper, we have extended the functionality of a maintenance support environment Gemini to easily merge code clones into one code portion. We have applied Gemini with Code Clone Shaper to two practical Java software ANTLR and Ant. By using Code Clone Shaper, we can dramatically reduce the number of clone pairs and clone classes. The clones removed by Code Clone Shaper has no meaningful block (not including the pair of brackets) and are difficult to merge as one method. Moreover, as shown in Figures 7 and 8(d), the selected clones are easy to merge into one code portion. So, we consider that Gemini achieves the evolution to support the maintenance activity more efficiently.

Of course, we have to continue applying Gemini to actual software maintenance process and improving/refining the functionality.

References

1. Ant, `http://jakarta.apache.org/ant/`, 2002.
2. ANTLR, `http://www.antlr.org/`, 2000.
3. B. S. Baker, *A Program for Identifying Duplicated Code*, Computing Science and Statistics, 24:49-57, 1992.
4. B. S. Baker, *On Finding Duplication and Near-Duplication in Large Software Systems*, IN Proc. IEEE Working Conf. on Reverse Engineering, pages 86-95, July 1995.
5. B. S. Baker, *Parameterized Duplication in Strings: Algorithms and an Application to Software Maintenance*, SIAM Journal on Computing, 26(5):1343-1362, 1997.
6. I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, *Clone Detection Using Abstract Syntax Trees*, Proc. IEEE Int'l Conf. on Software Maintenance (ICSM) '98, pages 368-377, Bethesda, Maryland, Nov. 1998.
7. S. Ducasse, M. Rieger, and S. Demeyer, *A Language Independent Approach for Detecting Duplicated Code*, Proc. of IEEE Int'l Conf. on Software Maintenance(ICSM) '99, pages 109-118, Oxford, England, Aug. 1999.
8. M. Fowler, *Refactoring: improving the design of existing code*, Addison Wesley, 1999.
9. D. Gusfield, *Algorithms on Strings, Trees, And Sequences*, Cambridge University Press, 1997.
10. J. Helfman, *Dotplot Patterns: a Literal Look at Pattern Languages*, TAPOS, 2(1):31-1,1995.
11. JavaCC, `http://www.webgain.com/products/java_cc/`, 2000.
12. J. H. Johnson, *Identifying Redundancy in Source Code using Fingerprints*, Proc. of CASCON '93, pages 171-183, Toronto, Ontario, 1993.
13. T. Kamiya, S. Kusumoto, and K. Inoue, *CCFinder: A multi-linguistic token-based code clone detection system for large scale source code* IEEE Transactions on Software Engineering, 28(7):654-670, 2002.

14. R. Komondoor and S. Horwitz, *Using slicing to identify duplication in source code*, In Proc. of the 8th International Symposium on Static Analysis, Paris, France, July 16-18, 2001.
15. Jens Krinke, *Identifying Similar Code with Program Dependence Graphs* , In Proc. of the 8th Working Conference on Reverse Engineering, 2001.
16. J. Mayland, C. Leblanc, and E. M. Merlo *Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics*, Proc. of IEEE Int'l Conf. on Software Maintenance (ICSM) '96, pages 244-253, Monterey, California, Nov. 1996.
17. L. Prechelt, G. Malpohl, M. Philippsen, *Finding plagiarisms among a set of programs with JPlag*, submitted to Journal of Universal Computer Science, Nov. 2001, taken from `http://wwwipd.ira.uka.de/~prechelt/Biblio/`
18. M. Rieger, S. Ducasse, *Visual Detection of Duplicated Code*, 1998.
19. Duploc, `http://www.iam.unibe.ch/~rieger/duploc/`, 1999.
20. Pigoski T. M, *Maintenance*, Encyclopedia of Software Engineering, 1, John Wiley & Sons, 1994.
21. Y. Ueda, T. Kamiya, S. Kusumoto, K. Inoue, *Gemini: Maintenance Support Environment Based on Code Clone Analysis*, 8th International Symposium on Software Metrics, pages 67-76, June 4-7, 2002.
22. Y. Ueda, T. Kamiya, S. Kusumoto, K. Inoue, *On Detection of Gapped Code Clones using Gap Locations*, 9th Asia-Pacific Software Engineering Conference, 2002, (to appear).
23. S. W. L. Yip and T. Lam, *A software maintenance survey*, Proc. of APSEC '94, pages 70-79, 1994.