Refactoring Support Based on Code Clone Analysis

Yoshiki Higo[1],Toshihiro 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,kusumoto,inoue}@ist.osaka-u.ac.jp
[2] PRESTO,Japan Science and Technology Agency
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. Software maintenance is the most expensive activity in software development. Many software companies spent a large amount of cost to maintain the existing software systems. In perfective maintenance, refactoring has often been applied to the software to improve the understandability and complexity. One of the targets of refactoring is code clone. A code clone is a code fragment in a source code that is identical or similar to another. In an actual software development process, code clones are introduced because of various reasons such as reusing code by 'copy-and-paste' and so on. Code clones are one of the factors that make software maintenance difficult. In this paper, we propose a method which removes code clones from object oriented software by using existing refactoring patterns, especially "Extract Method" and "Pull Up Method". Then, we have implemented a refactoring supporting tool based on the proposed method. Finally, we have applied the tool to an open source program and actually perform refactoring.

1 Introduction

It is well known that software systems are evolving by adding new functions and modifying existing functions over time. On the other hand, through the evolution, the structure of the software becomes more complex. Then, the understandability and maintainability are deteriorating. So, perfective maintenance [9], that is defined as a modification to a software product after delivery to improve its performance and/or maintainability, is an important maintenance activity.

Refactoring is an effective technique to conduct the perfective maintenance. It is defined as "the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure[7]." In [7], several refactoring patterns are described. It is necessary to

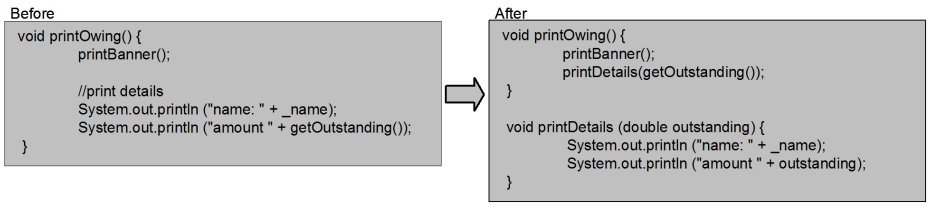


Fig. 1. Example of Extract Method

identify the refactoring candidates that contain "bad-smells" in order to apply refactoring patterns.

Code clone is one of the typical bad-smells that make software maintenance very difficult[7]. A code clone is a code fragment in a source file that is identical or similar to another. Code 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 be modified consistently. Hence effective code clone detection will support the refactoring activities of perfective maintenance. Up to the present, several code clone detection methods have been proposed [2][3][6][8].

In this paper, we show that the existing refactoring patterns[7] can be used to remove code clones. Then, we propose a method to support refactoring activity by applying code clone detection techniques. Furthermore, we implement a tool supporting our proposed method. This tool uses CCFinder[10], which is a code clone detection tool, and Gemini[16], which is code clone analysis environment. The function of this tool is to find certain code clones to which the refactoring patterns can be applied to. User can get code clone information for refactoring graphically.

2 Preliminaries

In this section, we briefly explain two refactoring patterns, "Extract Method" and "Pull Up Method" that are related to the code clone. Also, we define the code clone.

These patterns can be regarded as one typical activity to remove code clones. In other words, these patterns get code clones into common routine like method by using distinctive functions of object oriented programming language.

2.1 Refactoring Pattern

Extract Method To put it plainly, "Extract Method" means extraction of a part of existing method as a new method, and the extracted part is replaced by a new method caller shown in Figure 1. In general, this pattern is applied to the case that there is a too long method. In applying the pattern to code clones, a new method, that is a code fragment of code clone, is defined and the original

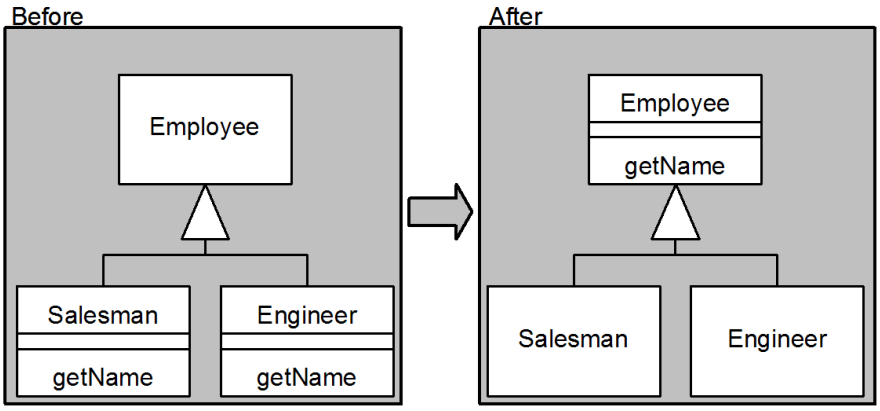


Fig. 2. Example of Pull Up Method

code clones are replaced by the new method caller. As the result, we can remove the code clones.

Pull Up Method "Pull Up Method" is a simple refactoring pattern. It means pulling up a method which defined in child class to its parent class. If the parent class has several child classes and some of them have the same method (that is, code clone), pulling up the method can remove the code clone.

2.2 Code Clone

Definition of code clone A clone relation is defined as an equivalence relation (i.e., reflexive, transitive, and symmetric relation) between two code fragments[10]. A clone relation holds between two code fragments 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 fragments is called a clone pair if the clone relation holds between the fragments. An equivalence class of

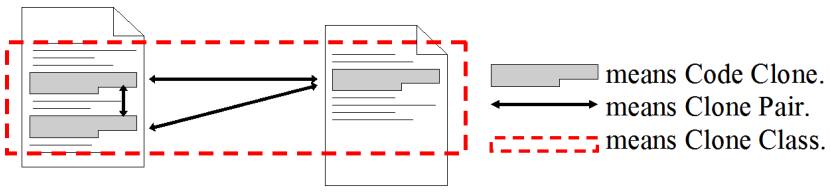


Fig. 3. Clone Relationship

clone relation is called a clone class. That is, a clone class is a maximal set of code fragments in which a clone relation holds between any pair of the code fragments as shown in figure 3.

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

Code Clone Detection Tool : CCFinder CCFinder detects code clones from program source codes and outputs the locations of the clone pairs in the source codes. The length of minimum clone is specified by a user beforehand. The length of minimum clone is the minimal size of the code fragment that CCFinder detect as a code clone.

Clone detection process of CCFinder consists of the following four steps:

Step1 Lexical analysis: Each line of source files is divided into tokens according 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 among multiple files is performed in the same way as a single file analysis.

Step2 Transformation: The token sequence is transformed, i.e., tokens are added, removed, or changed based on the transformation rules that aim 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 fragments including 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 the clone pair is converted into line numbers on the original source files.

Figure 4 illustrates the types of the code clones. CCFinder extracts the following two types of code clone corresponds to a code fragment C:

Exact code clone: E is a code fragment that is the same as C except for the difference about blank, new line and comments.

Renamed code clone: R is a code fragment that is the same as C except for the difference about the corresponded names of user-defined identifier (name of variables, constant, class, method and so on). Also, the reserved words and the sentence structures are the same between R and C.

3 Extraction of Refactoring–Oriented Code Clones

3.1 Filtering approach

As described in Section 2.2, the clone detection process of CCFinder is very fast but only lexical analysis is performed. Since code clones detected by CCFinder are sequences of tokens, they are not necessarily appropriate to be directly

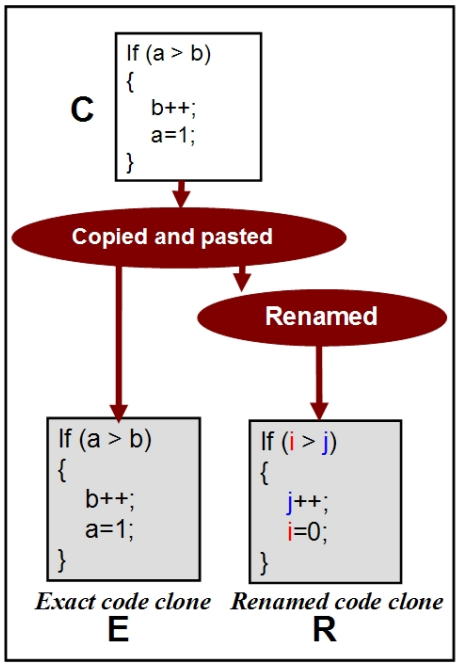


Fig. 4. Exact and renamed code clone

merged into one module (subroutine, function etc.). Some of them are not suitable for refactoring. In order to deal with the problem, we propose a method that extracts refactoring–oriented code clones from the whole set of code clones detected by CCFinder.

We have already proposed the key idea of this approach in [15]. In this paper, we extend the idea, and propose more practical approach to detect distinctive code clones to which these refactoring patterns are easy to apply. In the followings, we explain the detail of the approach.

The key idea is to find a kind of cohesive code fragment (like compound block or method bodies) from the code clone fragments. Figure 5 illustrates an example. In Figure 5, there are two code fragments A and B from a program, and the code fragments with hatching are maximal clones between them. In code fragment A, some data are substituted to list data structure from the head successively. In code fragment B, they are done so from the tail successively. The `for` blocks in A and B have a common logic that handles a list data structure. There are, however, sentences before and after `for` block, that are not necessarily related with the `for` block from semantic point of view. Such semantically unrelated sentences often obstruct refactoring. In other word, extracting only `for` block as a code clone is more preferable from refactoring viewpoint in this example.

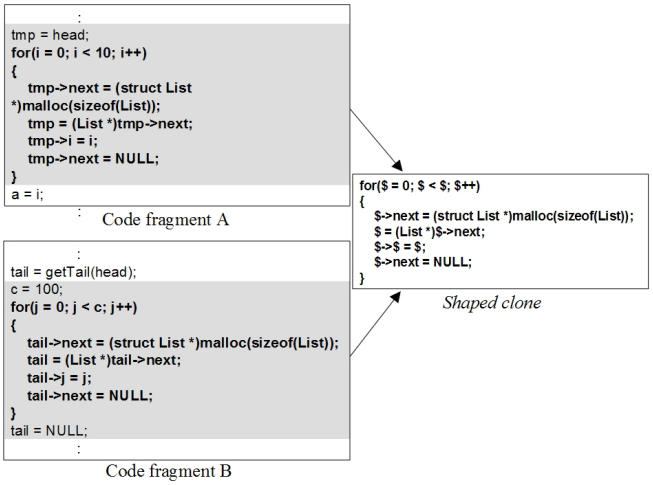


Fig. 5. Example of merging two code fragments

The proposed method is implemented as a filter for the output of CCFinder. We named the filter CCShaper. Currently CCShaper can be applied to only Java language. The extracting process using CCShaper consists of the following three steps:

Step1: Detect clone pairs using CCFinder
Step2: Provide semantic information (body of method, loop and so on) to each block by parsing the source files where clone pair are detected in Step1 and investigating the positions of blocks.
Step3: Extract structural blocks in the code clone using the information of location of clone pairs and semantics of blocks. Intuitively, structural block indicates the part of code clone that is easy to move and merge.

CCShaper performs Step 2 and Step 3. CCShaper extracts the following kinds of code clone as a refactoring-oriented code clone.

`Declaration` : class { }, interface { }
`Method` : method body, constructor, static initializer
`Statement` : if-statement, for-statement, while-statement, do-statement, switch-statement, try-statement, synchronized-statement
`Block` : range surrounded with '{' and '}'

3.2 Application of refactoring patterns

Here, we explain how these refactoring patterns are applied to the extracted code clones by CCShaper. At first, we have to say that there are two types of

code clone in which we are interested. One is "method-unit clone", ant the other is "statement-unit code clone".

For example, if all fragments of a given clone class are in the same class and the type is statement-unit, we can extract the clone statements as a new private method. In the other case, if all classes which have some fragments of a given clone class succeed to the same parent class and the type is method-unit, pulling up these fragments to the common parent class removes the code clones. In the similar case, if all classes which have some fragments of a given clone class succeed to the same parent class and type is statement-unit, each fragment could be extracted as a new method by applying "Extract Method", and in addition, each new method could be pulled up to the common parent class.

4 Case Study

4.1 Target Software

In order to evaluate the usefulness of the proposed method, we have applied it to a famous Java software:ANTLR(Version 2.7.1)[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. ANTLR includes 189 files and the size is 42000LOC.

4.2 Code Clone Analysis Environment : Gemini

In this case study, we used Gemini[16], which is graphical code clone analysis environment. Gemini uses CCFinder as a code clone detector, and greatly helps user to analysis code clone and modify source code. Currently, the function of CCShaper is included in Gemini, and the three steps of CCShaper(written in 3.1) are performed automatically. The followings explain some functions of Gemini, which were used in this case study.

Scatter Plot Figure 6 shows an example 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 6, 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.

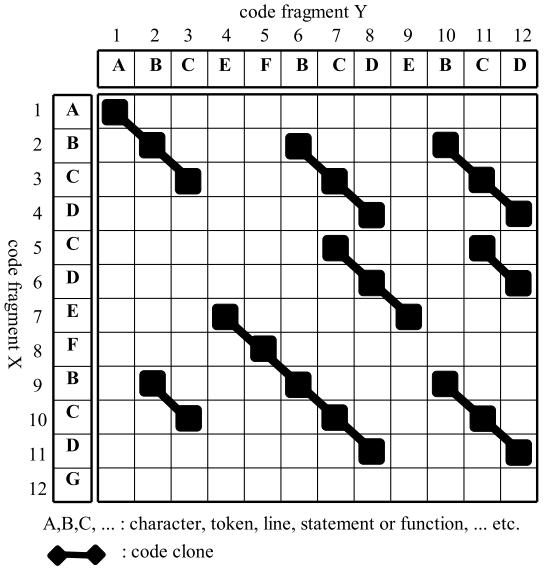


Fig. 6. Scatter Plot Model

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 code clones exist among two source files, the nearer the files are to be located in each axis. The details are described in [16].

Metric Graph Figure 7 shows the model of metric graph. A polygonal line is drawn per a clone class. Besides, each of five vertical bars represents the each metric. The followings are the metrics.

RAD :Represent the range of the source code fragments of a code clone in the directory hierarchy, when the source code is supposed to be stored in the hierarchical directory. When all the code fragments of a clone class are located in one source file, the RAD value of the clone class is equivalent to 0. When the code fragments of the clone class are located in multiple source files stored in one directory, the RAD is 1. If those sources files are stored in different directories, then RAD is the maximum depth of those sources measured from their common parent director.

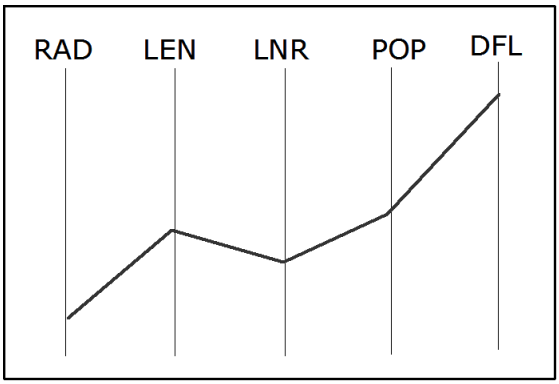


Fig. 7. Metric Graph Model

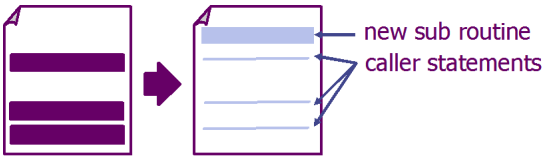


Fig. 8. Replace cloned portions with new identical routine

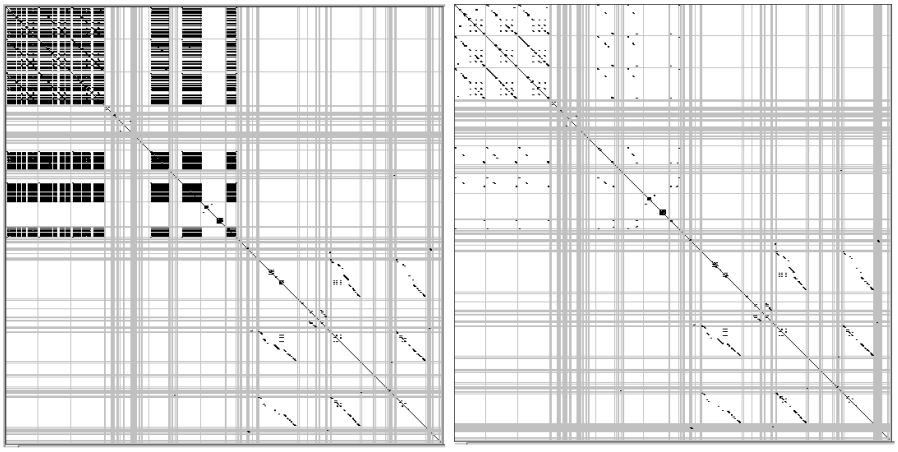
LEN :Represent the maximum length of element(code fragment) in a given clone class.
LNR :Represent the number of tokens without repeated part of the clone code fragments. each LNR is the same or smaller than LEN.
POP :Represent the number of element(code fragment) for a given clone class. If this value is high, it means that the similar code fragment appears in many places.
DFL :Represent an estimation of how many tokens would be removed from source files when all element(code fragment) of a given clone class are replaced with caller statements of a new identical routine like figure 8.

Metric graph allows user to set upper and lower bound on each metric, which enables user to select arbitrary clone class which he or she is interested in.

Source Code View User can browse source codes of code clones which are selected in scatter plot or metric graph. In this view, code clones are highlighted, so he or she can easily recognize the range of them.

4.3 Refactoring Activity

First, we have applied CCFinder to ANTLR. In this case study, we specified 50 as the length of minimum clone of CCFinder and 30 as the length of minimum



(a) Before refactoring (b) After refactoring

Fig. 9. Comparing on scatter plot

block of CCShaper. The result of scatter plot is shown in Figure 9(a). We can see there are many code clones in ANTLR. Totally, there are 276 clone classes and 13290 clone pairs.

Then, we extracted several code clones using CCShaper and investigated their appropriateness for refactoring. In this selection, we used metric graph. As a result, we identified the following two types of such code clones.

A clone class (C1), which was one of the types, included 24 code fragments. Each code fragment of C1 included 82 tokens and implemented the same algorithm to parse different lexical entities, such as the comma, the semi-colon, or the operators. All code fragments of (C1) were methods, and 17 code fragments of them were in same class. So, we applied "Extract Method" pattern [7] to (C1) and merged the 17 code fragments into a single method. Figure 10 shows some of the code fragments of C1 and Figure 11 shows the merged method.

As for the second type, a clone class (C2) included 150 code fragments and each of them included 33 tokens. All code fragments of (C2) were if-statements, and appeared in three classes. Furthermore these three classes inherited the same class as its parent. So, we firstly applied "Extract Method" pattern to (C2), which extracted the cloned portions as new methods. And then, we applied "Pull Up Method" to the new method. As the result, cloned if-statements are pulled up to common parent class as a new identical method shown in Figure 12.

The refactoring process to (C1) and (C2) resulted in 1000LOC reduction of the source code.

```
749
750     public final void mBANG(boolean _createToken) throws RecognitionException, CharStreamException, TokenStreamException {
751             int _ttype; Token _token=null; int _begin=text.length();
752             _ttype = BANG;
753             int _saveIndex;
754
755             match('!');
756             if ( _createToken && _token==null && _ttype!=Token.SKIP ) {
757                     _token = makeToken(_ttype);
758                     _token.setText(new String(text.getBuffer(), _begin, text.length()-_begin));
759             }
760             _returnToken = _token;
761     }
762
763     public final void mSEMI(boolean _createToken) throws RecognitionException, CharStreamException, TokenStreamException {
764             int _ttype; Token _token=null; int _begin=text.length();
765             _ttype = SEMI;
766             int _saveIndex;
767
768             match(';');
769             if ( _createToken && _token==null && _ttype!=Token.SKIP ) {
770                     _token = makeToken(_ttype);
771                     _token.setText(new String(text.getBuffer(), _begin, text.length()-_begin));
772             }
773             _returnToken = _token;
774     }
775
776     public final void mCOMMA(boolean _createToken) throws RecognitionException, CharStreamException, TokenStreamException {
777             int _ttype; Token _token=null; int _begin=text.length();
778             _ttype = COMMA;
779             int _saveIndex;
780
781             match(',');
782             if ( _createToken && _token==null && _ttype!=Token.SKIP ) {
783                     _token = makeToken(_ttype);
784                     _token.setText(new String(text.getBuffer(), _begin, text.length()-_begin));
785             }
786             _returnToken = _token;
787     }
788
789     public final void mRCURLY(boolean _createToken) throws RecognitionException, CharStreamException, TokenStreamException {
790             int _ttype; Token _token=null; int _begin=text.length();
791             _ttype = RCURLY;
```

Fig. 10. Source code of C1

Finally, we have to confirm that the functionality is not changed by the above refactoring process to ANTLR. We have checked the behavior of ANTLR after refactoring using all sample programs included in ANTLR package. For the 84 sample programs, the outputs from ANTLR before and after refactoring are exactly the same ones.

Figure 9(b) shows the scatter plot of ANTLR after refactoring. You can see that most of the clones located on the upper left side of Figure 9(a) have been removed.

5 Related Works

For the purpose of procedure extraction, code clone detection method for semantically cohesive ones using PDG(program dependence graph) have been proposed[12][14]. Also, Baxter et al. proposed a method to detect code clones using control/data flow dependencies from AST(abstract syntax trees)[3]. However, it is difficult to apply their approach to large scale softwares since the cost to create PDG/AST is very high.

Other interested approach is proposed by Balazinska et al[5]. They find function level clones by using 21 metrics[8], and apply different analysis[4][13] and context analysis[5] to detected clones, which means an estimation of refactoring applicability for each clone.

```
public final void ExtractedMethod(boolean _createToken, int _new_int, char _new_char)
        throws RecognitionException, CharStreamException, TokenStreamException {

        int _ttype; Token _token=null; int _begin=text.length();
        _ttype = _new_int;
        int _saveIndex;

        match(_new_char);
        if ( _createToken && _token==null && _ttype!=Token.SKIP ) {
                _token = makeToken(_ttype);
                _token.setText(new String(text.getBuffer(), _begin, text.length()-_begin));
        }
        _returnToken = _token;
}
```

Fig. 11. Extracted method for C1

6 Conclusion

We have applied CCShaper with CCFinder to a practical Java software ANTLR. We found two clone classes that refactoring patterns can be applied and actually conducted refactoring to them. As the result, we could reduce the code size by 1000 LOC without changing its original functionality.

But, since the current approach just extracts structural code clones from detected ones by CCFinder, it does not guarantee that all extracted structural code clones can be removed. So, as one of the future works, we will perform two types of analyses to get more precise result. One is to analyze variables which are referred in the code clone portions. For example, if all referred variables are declared in the same code clone portion, the portion can be easily moved to the parent class and so on. Otherwise, some referred variables are declared outside of the code clone portion, it is difficult to move the cloned portion to other class.

The other is to analyze the portion relationship between code fragments belonging to same clone class in class hierarchy. For example, all code fragments of a given clone class are in the plural classes, if these classes inherit the same parent class, this clone class may be removed. But, if they don't have common parent class, it is difficult to remove them. We consider that above two types of analyses can help us to perform refactoring.

It is also necessary to evaluate the refactoring effect[11] after detecting the bad-smell part in the actual refactoring process. Without knowing the effect, we cannot judge whether we should go for refactoring or not because we have to be cost sensitive. We are going to examine it in the refactoring process based on the code clone information.

References

1. ANTLR, `http://www.antlr.org/`, (2000).
2. B. S. Baker: "Parameterized Duplication in Strings: Algorithms and an Application to Software Maintenance," SIAM Journal on Computing, 26, 5, pp. 1343-1362 (1997).

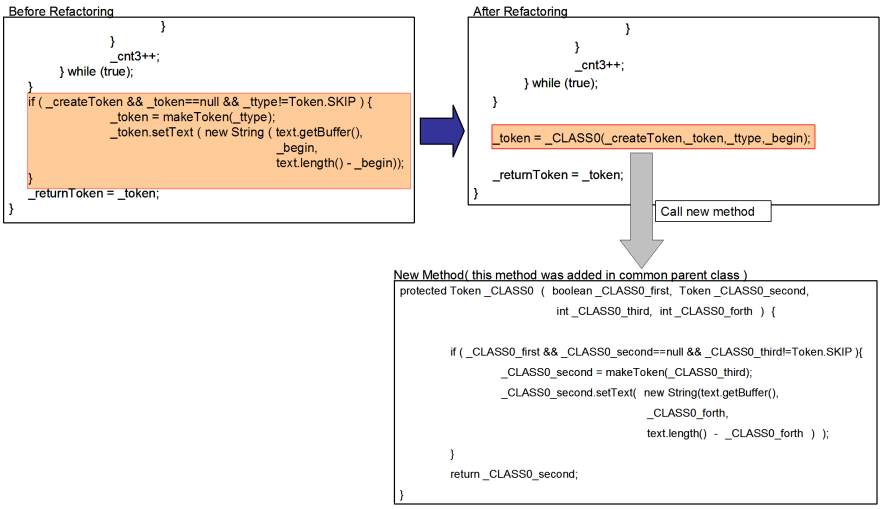


Fig. 12. Refactoring to C2

3. I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier: "Clone Detection Using Abstract Syntax Trees," Proc. of ICSM98, pp. 368-377 (1998).
4. M. Balazinska, E. Merlo, M.Dagenais, B. Lagu, and K. Kontogiannis: "Measuring clone based reengineering opportunities," Proc. of METRICS99, pp. 292-303 (1999).
5. M. Balazinska, E. Merlo, M. Dagenais, B. Lagu, and K. Kontogiannis: "Advanced Clone-Analysis to Support Object-Oriented System Refactoring," Proc. of WCRE2000, pp. 98-107 (2000)
6. S. Ducasse , M. Rieger, and S. Demeyer: "A Language Independent Approach for Detecting Duplicated Code," Proc. of ICSM99, pp. 109-118 (1999).
7. M. Fowler: Refactoring: improving the design of existing code, Addison Wesley (1999).
8. J. Mayland, C. Leblanc, and E. Merlo: "Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics," Proc. of ICSE96, pp. 244-253 (1996).
9. IEEE Std 1219-1992: Standard for Software Maintenance, (1992).
10. T. Kamiya, S. Kusumoto, and K. Inoue: "CCFinder: A multi-linguistic token-based code clone detection system for large scale source code," IEEE Trans. on Software Engineering, 28, 7, pp. 654-670 (2002).
11. Y. Kataoka, T. Imai, H. Andou and T. Fukaya: "A quantitative evaluation of maintainability enhancement by refactoring," Proc. of ICSM2002, pp. 576-585 (2002).
12. R. Komondoor and S. Horwitz: "Using slicing to identify duplication in source code ", Proc. of the 8th International Symposium on Static Analysis, pp. 40-56 (2001).
13. K. Kontogiannis, R. DeMori, E. Merlo, M. Galler, and M.Bernstein: "Pattern Matching Techniques for Clone Detection," Journal of Automated Software Engineering, Kluwer Academic Publishers, Vol. 3. pp.77-108, 1996.
14. Jens Krinke: "Identifying Similar Code with Program Dependence Graphs ", Proc. of the 8th Working Conference on Reverse Engineering, pp. 562-584(2001).

15. Y. Higo, Y. Ueda, T. Kamiya, S. Kusumoto and K. Inoue: "On software maintenance process improvement based on code clone analysis," Proc. of Profes 2002, pp. 185-197 (2002).
16. Y. Ueda, T. Kamiya, S. Kusumoto, K. Inoue, *Gemini: Maintenance Support Environment Based on Code Clone Analy sis*, 8th International Symposium on Software Metrics, June 4-7, 2002.