On Refactoring for Open Source Java Program

Yoshiki Higo[1],Toshihiro Kamiya[2], Shinji Kusumoto[1], Katsuro Inoue[1] and Yoshio Kataoka[3]

[1]Graduate School of
Information Science and Technology,
Osaka University
{y-higo, kusumoto, inoue}@ist.osaka-u.ac.jp

[2]PRESTO, Japan Science and Technology Corp.
kamiya@ist.osaka-u.ac.jp

[3]System Engineering Laboratory, Toshiba Corp.
yoshio.kataoka@toshiba.co.jp

Abstract

Software maintenance is the most expensive activity in software development process. It have been reported that many software companies spent a large amount of cost to maintain the existing systems. 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." Code clone makes software maintenance difficult and so is called a typical 'bad-smell'. 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. This paper describes the application of the refactoring to an open source Java program, ANTLR, using a code clone analysis. First, we extract some code clones from the source code with a code clone detection tool. Second, we identify some code clones that some refactoring patterns could be applied to. Finally, we apply the refactoring patterns to the code clones and rewrite the code. The result shows that the applied two refactoring patterns "Extract Method" and "Pull Up Method" reduce the code size by 1000LOC. Also we confirm that the behavior of the ANTLR after the refactoring is the same as before.

1 Introduction

Software maintenance is the most expensive activity during the entire software development process. It have been reported that many software companies spent a large amount of cost to maintain the existing systems. One of the maintenance activities is a perfective maintenance [7], that is defined as a modification to a software product after delivery to improve its performance and/or maintainability.

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[5]." In [5], several refactoring patterns are described. It is necessary to identify the refactoring candidates that contain "bad-smells" in order to apply refactoring patterns.

Code clone is one of the typical bad-smells that makes software maintenance very difficult[5]. A code clone is a code fragment in a source file 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 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][4][6].

We have been developing a code clone detection tool CCFinder[8]. CCFinder detects code clones from program source codes and outputs the locations of the clone pairs in source codes. The clones detected by CCFinder were not, however, necessarily appropriate for refactoring.

In [12], we have proposed a method to extract the part of code clones which are appropriate for refactoring, or "refactoring–oriented code clones." In this paper, we intend to apply the proposed method to an actual software product and evaluate its effectiveness. The target open source program is ANTLR. We found 174 code clones in total and performed two refactoring patterns: "Extract Method" and "Pull Up Method." As a result, we could decrease 1000 LOC from the original ANTLR without changing its behavior.

2 Refactoring Process based on Code Clone

2.1 Definition of code clone

A clone relation is defined as an equivalence relation (i.e., reflexive, transitive, and symmetric relation) between two code fragments[8]. 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 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.

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

2.2 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 1 illustrates the types of the code clones. CCFinder extracts the following two types of code clone corresponds to a code fragment C:

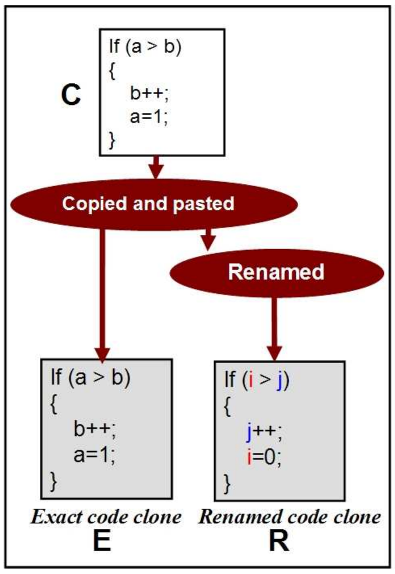


Figure 1. exact and renamed code clone

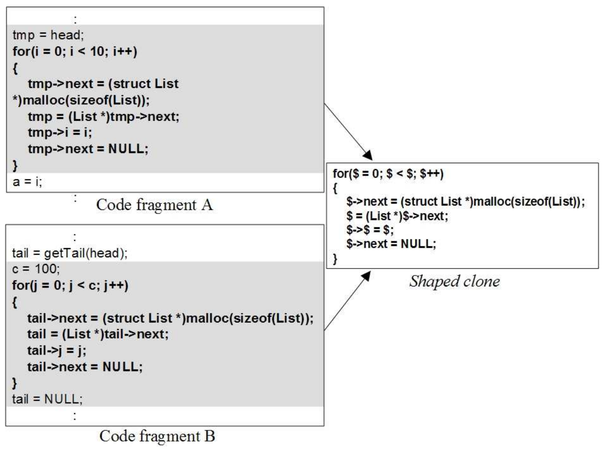
Exact code clone: E is a code fragment that is the same as C except for the difference about blank, new line and comments.

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.

2.3 Extraction of Refactoring–Oriented Code Clones

For the purpose of procedure extraction, code clone detection method for semantically cohesive ones using PDG(program dependence graph) have been proposed[10][11]. 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.

On the other hand, 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 merged into one module (subroutine, function etc.). Some of them are not suitable for refactoring. In order to deal with the problem, we proposed a method that extract refactoring–oriented code clones from the whole set of code clones detected by CCFinder [12]. Figure 2 describes the key idea.



```
            :
tmp = head;
for(i = 0; i < 10; i++)
{
   tmp->next = (struct List
*)malloc(sizeof(List));
   tmp = (List *)tmp->next;
   tmp->i = i;
   tmp->next = NULL;
}
a = i;
            :
```

```
            :
tail = getTail(head);
c = 100;
for(j = 0; j < c; j++)
{
   tail->next = (struct List *)malloc(sizeof(List));
   tail = (List *)tail->next;
   tail->j = j;
   tail->next = NULL;
}
tail = NULL;
            :
```

```
for($ = 0; $ < $; $++)
{
   $->next = (struct List *)malloc(sizeof(List));
   $ = (List *)$->next;
   $->$ = $;
   $->next = NULL;
}
```

Figure 2. Example of merging two code fragments

In Figure 2, there are two code fragments A and B from a program, and the code fragments with hatching are maximal clones among 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.

The proposed method is implemented as a filter for the output of CCFinder. We named the filter CCShaper. The extracting process using CCShaper consists of the following three steps:

Step1: Detect clone pairs using CCFinder

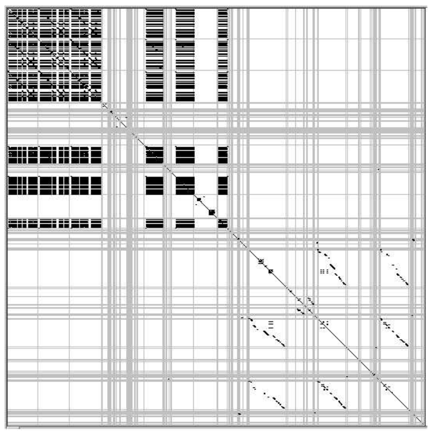
Step2: Provide semantic information (body of method, loop and so on) to each block by parsing the input source files and investigating the positions of blocks.

Step3: Extract meaningful blocks in the code clone using the information of location of clone pairs and semantics of blocks. Intuitively, meaningful block indicates the part of code clone that is easy to merge.

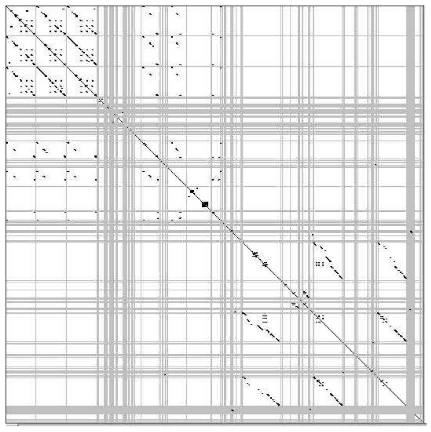
The details of CCShaper algorithm are shown in [12].

3 Case Study

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].



(a) Before refactoring

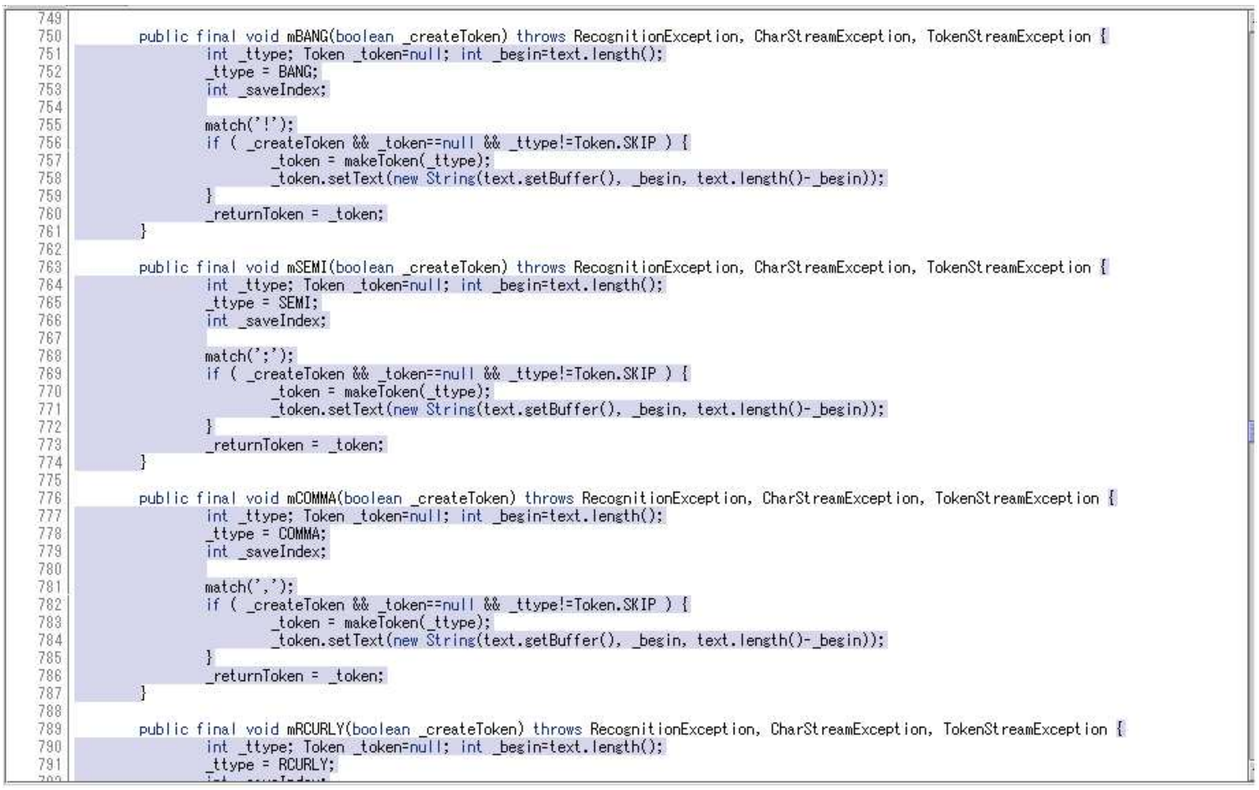


(b) After refactoring

Figure 3. Comparing on scatter plot

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.

First, we applied CCFinder to ANTLR. The result is shown in Figure 3(a). This figure is called “Scatter plot.” It shows visually where clone pairs exist in ANTLR. Both the vertical and horizontal axes represent code fragments of ANTLR. A clone pair is shown as a diagonal line segment. In this figure, the same code fragments 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



```
	public final void mBANG(boolean _createToken) throws RecognitionException, CharStreamException, TokenStreamException {
		int _ttype; Token _token=null; int _begin=text.length();
		_ttype = BANG;
		int _saveIndex;
		
		match('!');
		if ( _createToken && _token==null && _ttype!=Token.SKIP ) {
			_token = makeToken(_ttype);
			_token.setText(new String(text.getBuffer(), _begin, text.length()-_begin));
		}
		_returnToken = _token;
	}
	
	public final void mSEMI(boolean _createToken) throws RecognitionException, CharStreamException, TokenStreamException {
		int _ttype; Token _token=null; int _begin=text.length();
		_ttype = SEMI;
		int _saveIndex;
		
		match(';');
		if ( _createToken && _token==null && _ttype!=Token.SKIP ) {
			_token = makeToken(_ttype);
			_token.setText(new String(text.getBuffer(), _begin, text.length()-_begin));
		}
		_returnToken = _token;
	}
	
	public final void mCOMMA(boolean _createToken) throws RecognitionException, CharStreamException, TokenStreamException {
		int _ttype; Token _token=null; int _begin=text.length();
		_ttype = COMMA;
		int _saveIndex;
		
		match(',');
		if ( _createToken && _token==null && _ttype!=Token.SKIP ) {
			_token = makeToken(_ttype);
			_token.setText(new String(text.getBuffer(), _begin, text.length()-_begin));
		}
		_returnToken = _token;
	}
	
	public final void mRCURLY(boolean _createToken) throws RecognitionException, CharStreamException, TokenStreamException {
		int _ttype; Token _token=null; int _begin=text.length();
		_ttype = RCURLY;
```

Figure 4. Source code of C1

of token with itself, and the dots are symmetrical with respect to the diagonal line. 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. As a result, we identified the following two types of such code clones. A clone class (C1), which is one of the types, includes 24 code fragments. Each code fragment of C1 includes 82 tokens and implements the same algorithm to parse different lexical entities, such as the comma, the semicolon, or the operators. We applied "Extract Method" pattern [5] to this clone class and merged the 17 code fragment into a single method. Figure 4 shows some of the code fragment of C1 and Figure 5 shows the merged method. As for the second type, a clone class (C2) includes 150 code fragments and each of them includes 33 tokens. We applied two refactoring patterns "Extract Method" and "Pull Up Method" to the clone class and merged all the code fragments into a single method. The refactoring process to (C1) and (C2) resulted in 1000LOC reduction of the source code.

Finally, we have to confirm that the functionality is not changed by the above refactoring process to ANTLR. We 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 same.

```
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;
}
```

Figure 5. Extracted method for C1

Figure 3(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 3(a) have been removed.

4 Conclusion

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

It is necessary to evaluate the refactoring effect[9] 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).

[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] S. Ducasse , M. Rieger, and S. Demeyer: “A Language Independent Approach for Detecting Duplicated Code,” Proc. of ICSM99, pp. 109-118 (1999).

[5] M. Fowler: Refactoring: improving the design of existing code, Addison Wesley (1999).

[6] 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).

[7] IEEE Std 1219-1992: Standard for Software Maintenance, (1992).

[8] 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).

[9] Y. Kataoka, T. Imai, H. Andou and T. Fukaya: “A quantitative evaluation of maintainability enhancement by refactoring,” Proc. of ICSM2002, pp. 576-585 (2002).

[10] 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).

[11] Jens Krinke: “Identifying Similar Code with Program Dependence Graphs ”, Proc. of the 8th Working Conference on Reverse Engineering, pp. 562-584(2001).

[12] 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).