

Context-Aware Automatic Code Segment Extraction and Refactoring in Object-Oriented Systems

Wei Liu^{a,*}, Xindi Huang^{a,*}, Zhigang Hu^b, and Hong Phong Nguyen^c

^a*School of Informatics, Hunan University of Chinese Medicine, Hunan, 410208, China*

^b*School of Software, Central South University, Hunan, 410075, China*

^c*Faculty of Civil Engineering, University of Transport and Communications, Hanoi, 100000, Vietnam*

Abstract

Refactoring is a very important technology to improve the reusability and maintainability of existing code, and it is widely used in software development. In order to extract the code segment into a new method easily and cover the shortage of Eclipse in refactoring, the method of Context-Aware Automatic Code Segment Extraction and Refactoring (CAACSER) is proposed. By analyzing the context of the code, the input parameter class, and the output parameter class are introduced to handle complex code segments. The experimental results show that the CAACSER effectively solves some problems and drawbacks of many existing tools in code segment extraction, which acts as a basic step for realizing automatic and semi-automatic refactoring methods. The visualization tool of CAACSER can also carry out reasonable optimizations of the code without changing the systems' behaviors.

Keywords: refactoring; code segment extraction; extract method; Eclipse

(Submitted on March 29, 2018; Revised on April 27, 2018; Accepted on May 23, 2018)

© 2018 Totem Publisher, Inc. All rights reserved.

1. Introduction

Refactoring is a technique to restructure software by applying a series of methods without changing its observable behavior and to make software easier to understand and cheaper to modify [1]. Refactoring can improve the code quality of a software, including readability, reusability and maintainability [2-4]. Code extraction is one of the frequently-used refactoring skills and is also an important operating step for other complex refactoring methods, such as Extract Class, Replace Type Code with Subclasses, Replace Type Code with State/Strategy, Form Template Method, etc.

A method can be improved if it is too long or requires comments to be understood by separating it into several small methods, if duplicated codes exist by extracting them into an independent method, or if complex conditional statements exist by extracting each conditional branch to an independent method or breaking them down by polymorphism or strategy pattern. Martin Flower [1] recommended using short effective naming and fine-grained methods with better reusability and readability so it can more easily be understood, overloaded, and used by other methods. While the automatic extraction of source code is essential in the extraction and encapsulation of duplicated codes, in the separation of complex conditional statements, and in the decomposition of too large or too long methods, the easiest automatic extraction approach proposed by Martin Fowler [1] is Extract Method refactoring. This approach extracts specified code directly from the original code and encapsulates it into a new method. Code segment extraction is a very important step for many refactorings, such as the Extract Method [5-6], Move Method [7-8], etc.

At present, most of the existing code extraction methods are based on Mark Weiser's program slicing [9-10] and follow three principles proposed by Nikolaos Tsantalis [11], that is, (1) for declared variables, the code segment extracted must include their complete calculation procedures, (2) the behaviors of the program before and after extraction must be consistent, and (3) the code segment extracted should not reappear in the method after extraction. Nikolaos Tsantalis divided program slices into two types: the complete computation slices and the object state slices. Based on which different identification and

* Corresponding author.

E-mail address: xindih@126.com

extraction methods are provided, the complete computation and object state slices are extended to include the object state slice of any object reference that is used in the initial slice. Research by Tushar Sharma [12] and Katsuhisa Maruyama [13] presented algorithms for the identification of refactoring opportunities and for automatic extraction. Alaknanda Chandra et al. [14] proposed an approach to compute a static slice as well as a dynamic slice of a set of programs by creating an intermediary representation of the program. Isabella Mastroeni and Damiano Zanardini [15] formally defined the notion of abstract program slicing, a general form of program slicing where properties of data are considered instead of their exact value. This approach was applied to a language with numeric and reference values, relying on the notion of abstract dependencies between statements.

Simple refactoring is available in some popular IDEs (Integrated Development Environments), such as Eclipse, which includes the Extract Method option in its Refactor menu, but this function is limited and simple for complex situations. For instance, the Extract Method refactoring provided by Eclipse does not support multiple outputs. If Eclipse is used to refactor a code segment with multiple output variables that will be used by the subsequent code, it displays an error: Ambiguous return value: Selected block modifies more than one local variable used in the code. Affected variables are: {0}, {1}, ..., as shown in Figure 1.

Moreover, if multiple inputs are declared before the segment being extracted, Eclipse does not encapsulate these inputs as an object but uses a long parameter list, which affects readability and maintainability. Thus, it is necessary to remove this too long parameter list code smell [1], but the automatic refactoring provided does not meet the requirement to solve this problem.

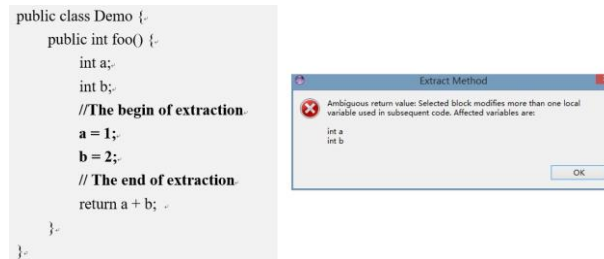


Figure 1. A failure sample for extracting multi-output code segment by Eclipse

This paper proposes the Context-Aware Automatic Code Segment Extraction and Refactoring (CAACSER) method capable of encapsulating specified code into a new method efficiently. CAACSER is the key part of refactoring to remove many code smells and acts as one of the basic steps in some design patterns oriented refactoring techniques. CAACSER, based on program slicing technique, simplifies the processing of variables and introduces appropriate automatic refactoring at the same time, supporting multiple inputs and outputs of the extracted code segments.

2. Principle and Implementation of CAACSER

CAACSER is concerned with the declaration and use of variables, the core of which focuses on parameters passing between the code being extracted and its context, providing automatic refactoring without changing the original behavior. This section will detail CAACSER with principles and implementation process.

2.1. Triple Segments Analysis

CAACSER, context-aware refactoring, has no influence on the logical integrity of the code after extracted. Thus, a comprehensive code analysis of the extracted code is required to consider its relation with the context and its influence on the original code. The Three Segment Analysis (TSA) is used to divide the original code into three parts: Pre-Segment (PrS), Extract Segment (ES) and Post-Segment (PoS), as shown in Figure 2. It is implied that the declaration and use of variables in different parts will have different influences on the code to be extracted. A number of terms and their abbreviations are defined in Table 1 for the concise expression of variables and process of CAACSER.

For variables in CASet, MPSet and PrsDefSet, if used in the extracted code ES, they need to be added to the parameter list of the new method as input. Variables ESUseSet and ESDefSet used or declared in ES may continue to be used in PoS, so they need to be passed back to the original method as outputs of the extracted code. In addition, methods defined in ESDefSet must be redefined in PoS for compilation. Thus, two definitions are given below to represent the input set and output set of the method being extracted.

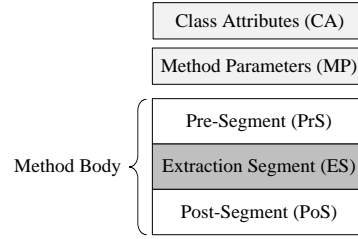


Figure 2. The schematic diagram of TSA

Table 1. A glossary of terms in CAACSER

Abbreviation	Description
CA	Class Attributes
MP	Method Parameters
PrS	Pre-Segment
ES	Extraction Segment
PoS	Post-Segment
CASet	Variable Set of CA
MPSet	Variable Set of MP
PrSDefSet	Variables declared in PrS
ESDefSet	Variables declared in ES
ESUseSet	Variables used in ES
PoSUseSet	Variables used in PoS
NOC	New Output Class
NIC	New Input Class
EM()	Extract Method

Definition 1: The input set InSet:

$$\text{InSet} \equiv \text{ESUseSet} \cap (\text{CASet} \cup \text{MPSet} \cup \text{PrSDefSet})$$

Definition 2: The output set OutSet:

$$\text{OutSet} \equiv \text{PoSUseSet} \cap (\text{ESDefSet} \cup \text{ESUseSet})$$

2.2. Extraction Rules and Processes

In the code extraction process, the size of InSet and OutSet, expressed as $\text{Size}(\text{InSet})$ and $\text{Size}(\text{OutSet})$, must be examined to determine whether to introduce the NIC class, NOC class, both, or neither. The corresponding extraction rules are as follows.

- (1) Code segment ES is moved into a new method named EM().
- (2) For three cases of $\text{Size}(\text{InSet})$, such as $\text{Size}(\text{InSet})=0$, $0 < \text{Size}(\text{InSet}) < T$, and $\text{Size}(\text{InSet}) \geq T$, the input of EM() is none, input set InSet, or New Input Class (NIC) object respectively, where T represents a user-defined threshold detailed in 2.3 *Determination of threshold* with default value of 6.
- (3) For three cases of $\text{size}(\text{OutSet})$, such as $\text{size}(\text{OutSet})=0$, $\text{Size}(\text{OutSet})=1$, and $\text{Size}(\text{OutSet}) > 1$, EM () returns void, a variable in the same type of the one in OutSet, or New Output Class (NOC) object respectively.

Flowchart in Figure 3 shows the process of extraction corresponding to the three rules as explained above.

2.3. Determination of Threshold

In the design and implementation of CAACSER, it is feasible to encapsulate inputs up to the specified threshold into a single NIC object automatically to avoid the bad smell of too long parameters list. In order to obtain a reasonable threshold, the distribution of Number of Method Parameters (NOMP) is analyzed based on 324155 methods of 41747 classes in 100 open source Java projects in the course of research. Most of these projects are from SourceForge, GitHub and other open source software sharing sites, and the rest are from some well-known Java open source frameworks and open source projects. The total lines of source code are 3630906, excluding comments. Of the 100 projects, 9 projects have more than 100,000 lines of source code, 60 projects have 10,000 to 100,000 lines, and 31 projects have less than 10,000 lines. Most of these projects are of high quality, developed by experienced developers.

The total of 324155 methods are preliminarily screened and analyzed, and the result shows that the maximum value of NOMP is 52 from the method addMember() in class com.mvnforum.admin. MemberXML in mvnForum project, an open source BBS project based on Java EE. Result also shows that 5 methods have NOMP greater than 50, constituting 0.001542%, 4 of which have 51 parameters and 1 of which has 52 parameters. The number of methods is reduced to 324150 by excluding this small proportion in order to facilitate statistical analysis and mapping; thus, the remaining methods with $NOMP \leq 30$ are considered for further analysis. Part of the statistical results is listed in Table 2. The statistic result of NOMP is presented in the histogram in Figure 4 by analyzing these 324150 methods.

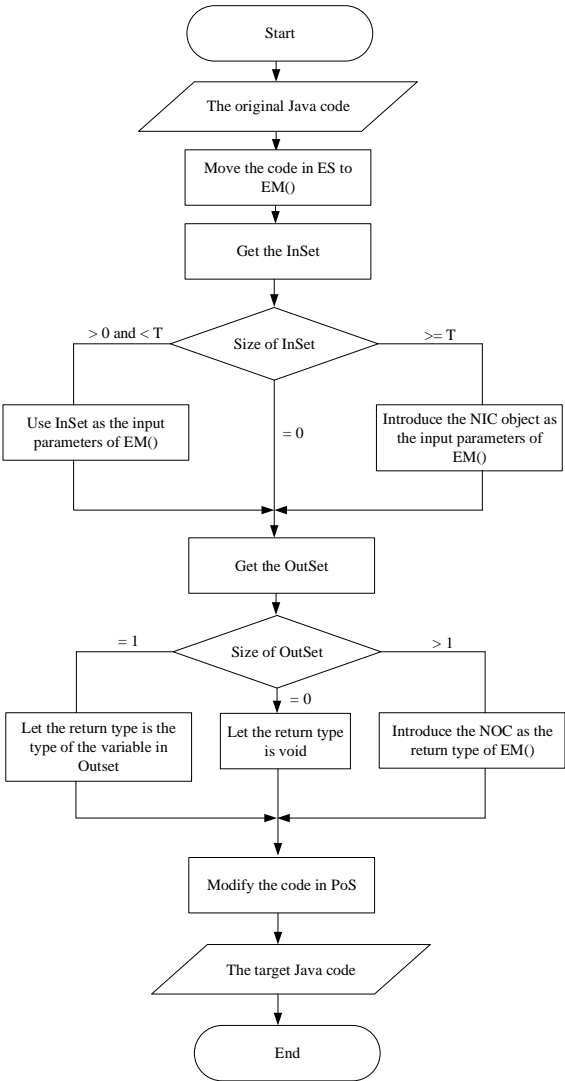


Figure 3. The flowchart of code extraction in CAACSER

Table 2. Statistical results of NOMP (partial data)

NOMP	Number of methods	Proportion	Cumulative proportion
0	134664	0.415437298	0.415437298
1	121598	0.375128798	0.790566096
2	38514	0.118815363	0.909381459
3	16256	0.050149622	0.959531081
4	6924	0.021360481	0.980891563
5	3202	0.009878143	0.990769705
6	1447	0.004463983	0.995233688
7	685	0.002113219	0.997346907
8	308	0.000950177	0.998297085
9	227	0.000700293	0.998997378
10	110	0.000339349	0.999336727
28	6	1.85099E-05	0.999990745
30	3	9.25497E-06	1

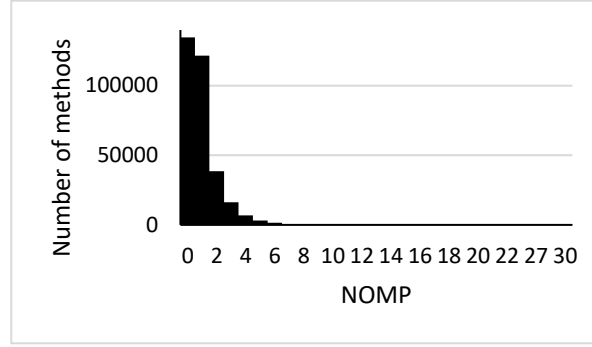


Figure 4. Statistical results histogram of NOMP

Figure 4 shows that a NOMP of 0 and 1 take up 134664 and 121598 methods respectively, accounting for 41.54% and 37.51% of the total 324150 methods. As NOMP increases, the number of methods decreases rapidly, presenting a long-tailed distribution. It is fitted by the Weibull distribution with a shape parameter $\alpha = 0.5844$ and a scale parameter $\beta = 0.3824$. The corresponding probability density function (PDF) and cumulative distribution function (CDF) are as follows.

$$f_w(x) = P(X = x) = \frac{\alpha}{\beta} \left(\frac{x}{\beta}\right)^{\alpha-1} e^{-(x/\beta)^\alpha} = 1.5283 \left(\frac{x}{0.3824}\right)^{-0.4156} e^{-(x/0.3824)^{0.5844}} \quad (1)$$

$$F_w(x) = P(X \leq x) = 1 - e^{-(x/\beta)^\alpha} = 1 - e^{-(x/0.3824)^{0.5844}} \quad (2)$$

The statistical results are analyzed below.

- (1) If the significance level α is set at 0.05, the confidence level is 95%. For minimum cumulative proportion greater than 95%, the cumulative probability P is 95.95% when $X \leq 3$, that is, 95.95% of methods have 3 or less parameters. This is compared to the value derived from CDF $F_w(x)$. If $F_w(x) = 0.95$, $x = 2.4997$, the ceiling of which is 3. This means that NOMP=3 satisfies the significance level of 0.05 consistent with the statistical analysis (empirical value).
- (2) If the significance level α is set at 0.01, the confidence level is 99%. For minimum cumulative proportion greater than 99%, the cumulative probability P is 99.08% when $X \leq 5$, that is, 99.08% of methods have 5 or less parameters. This is compared to the value derived from CDF $F_w(x)$. If $F_w(x) = 0.99$, $x = 5.2172$, the ceiling of which is 6. This means that NOMP=6 satisfies the significance level of 0.01, which is slightly greater than the statistical analysis (empirical value).

If the number of parameters in a method exceeds a certain threshold, parameters can be reduced by introducing a parameter class encapsulating the parameters into an object. However, this results in a large number of Getter and Setter methods used in the newly generated EM(), which to some extent affects the readability of the code. In this study, in order to keep the code structure clear and complete after extraction and refactoring, the threshold chosen is relatively large, i.e., a NOMP set at 6 as the threshold with a significance level of 0.01. That means NIC will be introduced in encapsulation only when the number of parameters reaches 6 or more. The threshold selected in this study is less than the default of PMD long parameter list threshold 10 and Checkstyle threshold 7. The threshold chosen instead has a good statistical significance, as 99.08% of methods have less than 6 parameters; in other words, less than 1% of methods have 6 or more parameters.

2.4. Implementation of the CAACSER Method

The pseudo code of the CAACSER algorithm is shown in Table 3.

Lines 1-9 of the pseudo-code are used to declare the data set used in the algorithm. Line 11 is used to move the code to be extracted into a new method EM(). Lines 12-13 are used to calculate the input set InSet and output set OutSet respectively. Lines 15-17 are the branch for $\text{size}(\text{InSet}) < 6$, in which the parameter(s) in the input set is/are taken into input list EM(). Lines 18-26 are the branch for $\text{Size}(\text{InSet}) \geq 6$, in which a New Input Class NIC is created to encapsulate all inputs for the input of EM(). Lines 28-29 are for $\text{Size}(\text{OutSet}) = 0$ where EM() returns void. Lines 30-32 are for $\text{Size}(\text{OutSet}) = 1$ where EM() returns a variable in the same type as the one in output set, and Lines 33-42 are for $\text{Size}(\text{OutSet}) > 1$ where a new output parameter class NOC is created and used as the return type to encapsulate outputs. Finally, Line 43 saves the code after refactoring.

Table 3. Pseudocode of CAACSER

Line No.	Algorithm pseudocode
	Input: The original code before refactoring, the start line number and end line number of the extraction code.
	Output: The new code after refactoring.
1	declare a Set named <i>CASet</i> which contains all fields of the class
2	declare a Set named <i>MPSet</i> which contains parameters of the method to be refactored
3	declare a Set named <i>PrSDefSet</i> which contains variables declared in PrS
4	declare a Set named <i>ESDefSet</i> which contains variables declared in ES
5	declare a Set named <i>ESUseSet</i> which contains variables used in ES
6	declare a Set named <i>PoSUseSet</i> which contains variables used in PoS
7	declare a Set named <i>InSet</i>
8	declare a Set named <i>OutSet</i>
9	declare a new MethodDeclaration node named <i>EM()</i>
10	
11	move ES to <i>EM()</i>
12	assign $ESUseSet \cap (CASet \cup MPSet \cup PrSDefSet)$ to <i>InSet</i>
13	assign $PoSUseSet \cap (ESDefSet \cup ESUseSet)$ to <i>OutSet</i>
14	
15	if $Size(InSet) < 6$
16	set all variables stored in <i>InSet</i> as <i>EM()</i> 's parameters
17	add a MethodInvocation node for <i>EM()</i> at the extraction position
18	else if $Size(InSet) \geq 6$
19	create an inner class named NIC
20	set all variables stored in <i>InSet</i> as NIC's fields
21	generate Getters and Setters for NIC's fields in NIC
22	declare a variable named <i>inClass</i> of class NIC before the extraction position
23	set <i>inClass</i> 's fields
24	set the <i>inClass</i> as <i>EM()</i> 's parameter
25	add a MethodInvocation node for <i>EM()</i> at the extraction position
26	end if
27	
28	if $Size(OutSet) = 0$
29	set void as <i>EM()</i> 's return type
30	else if $Size(OutSet) = 1$
31	set the type of variable stored in <i>OutSet</i> as <i>EM()</i> 's return type
32	invoke <i>EM()</i> at the extraction position and declare a variable to store the return value with the type of variable stored in <i>OutSet</i>
33	else if $Size(OutSet) > 1$
34	create an inner class named NOC
35	set all variables stored in <i>OutSet</i> as NOC's fields
36	generate Getters and Setters for NOC's fields in NOC
37	declare a variable named <i>outClass</i> of class NOC after the extraction position
38	invoke Setters of <i>outClass</i> and set values to <i>outClass</i> 's fields
39	set NOC as <i>EM()</i> 's return type
40	invoke <i>EM()</i> at the extraction position and set its return value to <i>outClass</i>
41	invoke Getters of <i>outClass</i> and set return values to variables stored in <i>outSet</i> after invocation of <i>EM()</i>
42	end if
43	save the code after refactoring

The algorithm shown in Table 3 converts the source code into an abstract syntax tree to process whose nodes are used as the unit for operation [16]. The algorithm can be applied in other complex refactoring as an independent utility class with good reusability.

CAACSER is more powerful and rigorous than the built-in Extract Method in Eclipse. Users are involved in the code extraction process during CAACSER implementation, including the naming of the new method and of the input and output classes. Therefore, an extraction wizard is provided to guide users in code extraction in three steps as shown in Figure 5. The first step (Figure 5-a) is method settings, including setting the name and modifier of the extracted method and giving the preview of its signature. The second step (Figure 5-b) is input settings, used for the naming of the input class encapsulated from multiple inputs under users' requirements. The algorithm by default encapsulates if the number of inputs is greater than or equal to 6. The third step (Figure 5-c) is output settings, used for naming of the output class, and the encapsulation process by default is automatic if more than one output exists. The code extraction wizard interface as shown in Figure 5 is practical, user-friendly, and simple to use, enabling users to customize code extraction and related parameters.

(a) The interface of method settings

(b) The interface of input parameters' settings

(c) The interface of output parameters' settings

Figure 5. The wizard interfaces of code fragment extraction

3. Experiment and Result Analysis

In order to verify the correctness of the CAACSER method, a series of code segments are selected or designed for white-box testing. There are three possible parameter sets for both input and output in the CAACSER method, which results in 9 cases of combinations. Moreover, in the case of code segments unable to be extracted, such as code segments containing return or break statements, the correctness of CAACSER is checked using manually selected code segments. For each of these 10 cases, five test code segments are selected mostly from the 100 open source projects used in section 2.3, and for each test, three cases are designed to determine whether the behavior of the code after refactoring has been changed. The main purposes of this experiment are as follows.

- (1) To verify whether CAACSER deals with different situations correctly, such as whether code segments meeting certain conditions can be correctly extracted, or whether code not meeting certain conditions can be prompted with error messages.
- (2) To verify whether errors are introduced into code after refactoring of using CAACSER, regression testing is carried out to judge whether the system behavior has changed after code extraction.

The experimental results are listed in Table. 4.

Table 4. The testing results of CAACSER

Test Cases	Number of code segments for testing	Number of correct extractions	Total number of test cases	Number of test cases with consistent results before and after refactoring
InSet(0) ∧ OutSet(0)	5	5	15	15
InSet(0) ∧ OutSet(1)	5	5	15	15
InSet(0) ∧ OutSet(>1)	5	5	15	15
InSet(1-5) ∧ OutSet(0)	5	5	15	15
InSet(1-5) ∧ OutSet(1)	5	5	15	15
InSet(1-5) ∧ OutSet(>1)	5	5	15	15
InSet(>=6) ∧ OutSet(0)	5	5	15	15
InSet(>=6) ∧ OutSet(1)	5	5	15	15
InSet(>=6) ∧ OutSet(>1)	5	5	15	15
CanNotExtract	5	0	15	--

In the column of Test Cases shown in Table 4, InSet(0), InSet (1-5), InSet(≥ 6) indicate the number of inputs as 0, 1 to 5, and greater than and equal to 6, respectively. OutSet(0), OutSet(1) and OutSet(>1) indicate the number of outputs as 0, 1, and greater than 1, respectively. CanNotExtract is used to indicate that extraction cannot be performed by CAACSER, i.e., the part in bold shown in Figure 6.

```
int i = 1, j = 1;
if (i >= 0) {
//The bold part is the code segment to be extracted while
CAACSER is not applicable.
    if (j >= 0)
        return false;
    else
        i++;
}
System.out.println(i);
return true;
```

Figure 6. A code segment sample of CanNotExtract

```
.....
// If we are in a tag then let's do it....
if (gotTag) {
    // Get the tag name (action)
    firstQuote = line.indexOf("'", tagLoc);
    lastQuote = line.indexOf("'", firstQuote + 1);
    tag = line.substring(firstQuote + 1, lastQuote);
    // Get nameSpace (optional)
    .....
}
```

Figure 7. Code segment for testing (before refactoring)

As can be seen from Table 4, CAACSER deals with various combinations of conditions effectively performing code extraction correctly for code segments meeting corresponding conditions, while ensuring the consistency of results and system behavior before and after refactoring.

An example shown in Figure 7 is a test code segment from method applyTemplate() in org.cofax.WysiwygTemplate class in the Cofax project, a web-based text and multimedia management system. This method is a typical Long Method of 395 lines with the potential to be optimized and refactored. Code segments in bold are to be extracted corresponding to the source code in Lines 197-199. As the function of this segment is to obtain the tag name and is relatively independent, it is extracted as a separate method to improve comprehensibility and maintainability.

```
.....
// If we are in a tag then let's do it....
if (gotTag) {
    // Get the tag name (action)
    OutPutClass outClass = getTagName(lastQuote, tag, line, tagLoc, firstQuote);
    tag = outClass.getTag();
    line = outClass.getLine();
    // Get nameSpace (optional)
    .....
//New method after extracting
public OutPutClass getTagName(int lastQuote, String tag, String line, int tagLoc, int firstQuote) {
    OutPutClass outClass = new OutPutClass();
    firstQuote = line.indexOf("'", tagLoc);
    lastQuote = line.indexOf("'", firstQuote + 1);
    tag = line.substring(firstQuote + 1, lastQuote);
    outClass.setTag(tag);
    outClass.setLine(line);
    return outClass;
}
//New output class after extracting
class OutPutClass {
    private String tag;
    private String line;
    public void setTag(String tag) {
        this.tag = tag;
    }
    public String getTag() {
        return tag;
    }
    public void setLine(String line) {
        this.line = line;
    }
    public String getLine() {
        return line;
    }
}
```

Figure 8. Code segment for testing (after refactoring)

The code segment in Figure 7 refactored by CAACSER is shown in Figure 8. As the code to be extracted contains 5 inputs and 2 outputs, OutPutClass is introduced to encapsulate these 2 outputs. The code refactored in Figure 8 consists of three parts: the modified code after extracting, a new extraction method and a new output class, respectively.

In addition, CAACSER is compared with the Extract Method built in Eclipse by designing a test code refactored by both. The comparison is based on the same code segments selected manually, and then CAACSER and Eclipse are used respectively for code extraction. Table 5 shows the test code, and Table 6 shows the comparison results of CAACSER and the Extract Method built in Eclipse.

Table 5. The test code for extracting by CAACSER and Eclipse

Line No.	Source Code
1	public void foo() {
2	int a = 0, b = 1, c = 2, d = 3, e = 4, f = 5;
3	for (int i = 2; i <= 10; i++) {
4	System.out.println("Start");
5	a++;
6	b += 2;
7	c = i;
8	d = a + b;
9	i++;
10	f = a * c;
11	System.out.println(a + b + c + d + i + f);
12	}
13	}

Table 6. The results comparison of code extraction by CAACSER and Eclipse(Refactor)

No. of line(s) for extracting	Description	CAACSER	Eclipse
4	Method to be extracted with no input and no output	Direct extraction	Direct extraction
5-6	Method to be extracted with less than 6 inputs	Direct extraction, inputs turned to method parameter list	Direct extraction, inputs turned to method parameter list
11	Method to be extracted with 6 or more inputs	Encapsulates inputs to an input object for the new method	Direct extraction, inputs turned to method parameter list
6	Method to be extracted with 1 or less output	Method returns a variable in the same type of the output or void if no output	Method returns a variable in the same type of the output or void if no output
6-7	Method to be extracted with more than 1 output	Encapsulates outputs as an output object for the new method	Unable to process

Table 6 presents the differences between these two code extraction methods. Methods with six or more inputs are seen with bad smell of long parameters list, which needs to be removed by CAACSER. CAACSER encapsulates multiple inputs into an input object to carry parameters into the method, then uses the Getter and Setter methods to obtain and set parameters, which shortens the length of parameters list and removes the bad smell of long parameters list. In addition, in the case of the extracted method with multiple outputs, the Extract Method in Eclipse terminates code extraction with errors while CAACSER packages outputs into an output object similar to the input object, and uses the Getter and Setter methods to obtain and set the output parameters. Therefore, CAACSER is more flexible than the Extract Method in complex scenarios, especially for methods with multiple inputs and outputs.

4. Conclusions

In this paper, we present a context-aware automatic code segment extraction and refactoring method, CAACSER. The Three-Segment Analysis (TSA) in code analysis takes into account the context of the code to be extracted and evaluates the size of the input and output sets obtained to determine whether to introduce input and output classes in refactoring. CAACSER provides a complete set of wizard-style interfaces to guide users to refactor code using the Extract Method without changing the behavior of the original code, and it is also useful in semi-automatic code extraction with step-by-step guidance for developers.

CAACSER still has limitations that need to be improved in follow-up work. First, if the source code contains keywords such as return, break and continue, CAACSER cannot perform the extraction; second, CAACSER cannot deal with comments;

third, CAACSER only considers variables used in the extracted code without distinguishing read and write operations. Some variables encapsulated in the input class may remain unchanged in output class as they are read-only in the code extracted or are only written in the context. Therefore, follow-up work will focus on the improvement of code extraction for more complex refactoring scenarios.

Acknowledgements

This work was supported by the Scientific Research Foundation of Education Department of Hunan Province (No. 16C1201).

References

1. F. Martin, K. Beck, "Refactoring: improving the design of existing code", Addison-Wesley, 1999.
2. S. H. Kannangara, W. M. J. I. Wijayanayake, "An empirical evaluation of impact of refactoring on internal and external measures of code quality", *International Journal of Software Engineering & Applications*, vol. 6, no.1, pp. 51–67, 2015
3. N. Kumari, A. Saha, "Effect of refactoring on software quality". *International Journal of Computer Science & Information Technology*, vol. 4, no. 5, pp. 37-46, 2014.
4. D. Cedrim, L. Sousa, A. Garcia, R. Gheyi, "Does refactoring improve software structural quality? A longitudinal study of 25 projects", in *Proc. 30th Brazilian Symposium on Software Engineering (SBES '16)*, Maringa, Brazil, 2016, pp. 73-82.
5. M. Kaya, J.W. Fawcett, "Identification of extract method refactoring opportunities through analysis of variable declarations and uses", *International Journal of Software Engineering & Knowledge Engineering*, vol. 27, no. 1, pp. 49-69, 2017.
6. S. Charalampidou, A. Ampatzoglou, A. Chatzigeorgiou, A. Gkortzis, P. Avgeriou, "Identifying extract method refactoring opportunities based on functional relevance", *IEEE Transactions on Software Engineering*, vol. 43, no. 10, pp. 954-974, 2017
7. J.A. Dallal, "Predicting move method refactoring opportunities in Object-Oriented code", *Information & Software Technology*, vol. 92, pp. 105-120, 2017.
8. R. Terra, M. T. Valente, S. Miranda, V. Sales, "JMove: A novel heuristic and tool to detect move method refactoring opportunities", *Journal of Systems & Software*, vol. 138, pp. 19-36, 2018.
9. M. Weiser, "Program slicing", in *Proc. 5th International Conf. on Software Engineering (ICSE '81)*, San Diego, USA, pp. 439-449, 1981.
10. M. Weiser, "Program slicing", *IEEE Transactions on Software Engineering*, vol. 10, no. 4, pp. 352-357, 1984.
11. N. Tsantalis, A. Chatzigeorgiou, "Identification of extract method refactoring opportunities for the decomposition of methods", *Journal of Systems and Software*, vol. 84, no. 10, pp. 1757-1782, 2011.
12. T. Sharma, "Identifying extract-method refactoring candidates automatically", in *Proc. 5th Workshop on Refactoring Tools (WRT '12)*, Rapperswil, Switzerland, 2012, pp. 50-53.
13. K. Maruyama, "Automated method-extraction refactoring by using block-based slicing", in *ACM Symposium on Software Reusability: putting software reuse in context (SSR '01)*, Toronto, Canada, 2001, pp. 31-40.
14. A. Chandra, A. Singhal, A. Bansal, "A study of program slicing techniques for software development approaches", in *IEEE 2015 1st International Conference on Next Generation Computing Technologies (NGCT)*, Dehradun, India, 2016, pp. 622-627.
15. I. Mastroeni, D. Zanardini, "Abstract program slicing: an abstract interpretation-based approach to program slicing", *ACM Transactions on Computational Logic*, vol. 18, no. 1, pp. 1-54, 2017.
16. T. Kuhn, Eye Media GmbH, O. Thomann, IBM Ottawa Lab, "Abstract Syntax Tree", 2006-11-20, http://www.eclipse.org/articles/Article-JavaCodeManipulation_AST/index.html.

Wei Liu received his PhD degree in Computer Application Technology from the Central South University of China. He is an associate professor and senior engineer at the School of Informatics, Hunan University of Chinese Medicine, China. His research interests include software engineering, data mining and medical informatics. He has published over 20 papers in related fields.

Xindi Huang received her MSc. degree in Communication Engineering from the University of Manchester, UK. She is an engineer and assistant lecturer at the School of Informatics, Hunan University of Chinese Medicine, China. Her research interests include data mining, medical informatics and software engineering.

Zhigang Hu received his PhD degree in Mechanical Manufacturing and Theory from the Central South University of China. He is a professor and doctoral supervisor at the School of Software, Central South University, China. His research interests include software engineering, parallel computing and cloud computing. He has published over 200 papers in related fields.