Using Software Metrics to Select Refactoring for Long Method Bad Smell

Panita Meananeatra and Songsakdi Rongviriyapanish
Computer Science Department, Thummasat University, Pathumthani, Thailand
email: panita.meananeatra@nectec.or.th,rongviri@cs.tu.ac.th
Taweesup Apiwattanapong
Software Engineering Laboratory, National Electronics and Computer Technology Center, Pathumthani, Thailand email: taweesup.apiwattanapong@nectec.or.th

***Abstract*—Refactoring is a technique for improving software structure without changing its behavior which can be used to remove bad smells and increases software maintainability. But only few approaches have been proposed to address the identification of appropriate refactorings. Specifically, our research proposes a method to select refactoring based on software metrics which are defined in terms of data flow and control flow graphs. The method consist of 4 steps: 1) calculate metrics, 2) find candidate refactoring by using refactoring filtering condition (RFC), 3) apply a suite of candidate refactorings and compute maintainability, and 4) identify the refactoring that gives the highest maintainability. We demonstrate out approach by giving an example of removing a long method bad smell in a customer class in a movie rental system. Our approach proves to be able to suggest an appropriate set of refactoring techniques such as extract method, replace temp with query, and decompose condition, to solve the long method bad smell.**

***Index Terms*—Refactoring Identification, Software Metrics, Long Method, Bad Smell, Software Maintainability**

I. Introduction

Softwares often need to be changed for different reasons such as requirement change, technology change, environment change, etc). Developers must modify software continuously. Sometimes, modified software loses good structure and leads to decreased maintainability. One of the techniques that improve maintainability is refactoring.

Refactoring is a technique for improving software structure without changing its behavior. Most of the bad quality programs have bad smells which, if removed, increases many quality characteristics especially its maintainability. The process of removing bad smells can be divided into four major steps: (1) detecting the locations of bad smells where the fragment of code should be refactored, (2) identifying the refactorings that should be applied, (3) applying the appropriate refactorings and (4) assessing the refactoring impact on the quality of software [1].

Most of research works focuses on bad smell detection [1,3,5,6] but, only few approaches have been proposed to address the problem of identifying the appropriate refactorings and the program elements (class, method, variable, etc.) to which they should be applied [1,2,7]. Such an approach will be useful especially for a novice developer who is not sure of which refactoring should be used and to which program elements it should be applied.

Long Method Bad Smell occurs when a method is too long or has many variables, parameters or conditions. The main problems with long methods include hard to read and hard to reuse [4]. The refactoring book [8] proposes 6 refactorings for Long Method: 1) extract Method, 2) replace temp with query, 3) introduce parameter object, 4) preserve whole object, 5) decompose conditional and 6) replace method with method object. Since there are refactorings to choose, the novice programmer may not know which refactorings to use.

Our research aims to answer the research problem How to select refactoring for resolving long method bad smell and improve maintainability?. Our approach introduces refactoring selection conditions, which help programmers in refactoring identification. These conditions are defined in terms of software metrics based on control-flow graphs and data flow graphs, e.g., definition use, predicate use of the method.

This paper is organized as follows. Section 2 introduces the related work. Section 3 describes steps of our approach. Section 4 presents examples and results of experiments. Finally, section 5 discusses conclusions and future work.

II. Related Work

Pienlert and Muenchaisri [1] presented an approach for detecting and locating bad smells based on software metrics. They defined detection techniques for six bad smells: Feature Envy, Large Class, Lazy Class, Long Method, Long Parameter Lists and switch statement. Each detection technique was described in the following format: definition, motivation, strategy, metrics, and specification of outliers and application of refactoring. This work could only identify elements to be refactored at the class and method levels. It could not suggest elements at the statement level, i.e., inside a method body.

JDeodorant [2] is a tool which can identify the type-checking bad smells suggested by the authors of each example and also applied the refactoring Replace Conditional with Polymorphism or Replace Type Code with State/Strategy. The paper is limited to the application of refactoring for switch statements and if-else statements.

Meananeatra and Muenchaisri [3] presented a technique to find thresholds of metrics for detecting bad smells and non-bad smells in java code. First, bad smells predicting models are constructed using regression analysis on 10 programs for training. After that, thresholds for using these metrics to detect bad smell and non-bad smell java code was defined and validated by the previous 10 training programs and 2 test data sets respectively. This research focused only on a bad smell detection which applied by threshold of metrics, but did not suggest how to select refactorings and where to apply them.

Yang, Liu and Niu [4] proposed an approach to recommend fragments within long methods for extraction. This approach has six steps: (1) divide for fragments by types of statement, (2) compose fragments for combination of consecutive sibling fragment, (3) collect variables declared as parameter in or parameter out, (4) move variable declarations for reducing coupling, (5) compute coupling, and (6) sort benefit/cost ratios of all candidate fragments for recommending fragment. For evaluation, they focus on the accuracy of the approach, impact on refactoring cost and impact on software quality. This paper suggests blank lines for dividing fragment. It implies that if source code has no blank lines, it should not use this approach.

In conclusion, the existing works could not suggest elements at the statement (level inside a method body) and did not suggest how to select refactorings and where to apply them. Therefore, our research focuses on defining metrics based on data flow and control flow for identifying refactoring inside a method body and suggesting program element to apply.

III. METHODOLOGY

This section explains the steps of our approach and demonstrates the approach using an example. This approach focuses on long method bad smell and consists of four steps: 1) calculate metrics, 2) find candidate refactorings by using refactoring filtering condition, 3) apply candidate refactorings (one at a time) and compute maintainability, and 4) identify the refactoring that gives the highest maintainability metric values.

First, our approach calculates two sets of metrics: one for comparing the maintainability before and after refactoring, and the other for finding candidate refactorings in the next step. Second, the approach filters out inapplicable refactorings by evaluating refactoring filtering conditions (RFC) using the metrics mentioned earlier. Then, applicable refactorings are applied one at a time, and the maintainability metrics are computed for each case. Finally, the refactoring with the highest maintainability metric values is chosen. The following sections explain each step in more details.

To demonstrate our approach, we use statement method in Customer class of a movie rental system. This system is a small system from the refactoring book and has three classes: Movie, Rental, and Customer. The method has a long method bad smell (as shown in Fig.1).

A. Calculate Metrics

This step consists of computing metrics values from source code. Our approach uses three well-known maintainability

```
1  class Customer {

13 public String statement () {
14 double  totalAmount  = 0;
15 int frequentRenterPoints = 0;
16 Enumeration rentals = _rentals.elements();
17 String result =Rental Record for  + getName()+ \n;
18  while (rentals.hasMoreElement()) {
19    double thisAmount = 0;
20    Rental each = (Rental) rentals.nextElement();
      //determine amounts for each line
21    switch (each.getMovie().getPriceCode())  {
22      case Movie.REGULAR:
23        thisAmount += 2;
24        if (each.getDaysRented() > 2)
25        thisAmount += (each.getDaysRented()- 2)*1.5;
26      break;
27      case Movie.NEW_RELEASE:
28        thisAmount += each.getDaysRented()*3;
29      break;
30      case Movie.CHILDRENS:
31        thisAmount += 1.5;
32        if (each.getDaysRented() > 3)
33        thisAmount += (each.getDaysRented()- 3)*1.5;
34      break;
35    }// end of switch statement  .
46 }
```

Fig. 1. A part of source code in ”statement” method.

TABLE I
MAINTAINABILITY METRICS

Name	Formula
MCX	Edge node+2
LOC	Number of statements in method
LCOM	(m a / v) / (m-1) m : the number of methods in the class. a : the number of methods in a class that access an instance variable. v : the number of instance variables.

metrics: Complexity of Method (MCX), Line of Code in Method (LOC), and Lack of cohesion of method (LCOM). Table I shows their formulas. For these metrics, lower metric values imply higher maintainability.

Refactoring filtering metrics (as shown in Table II) are based on control flow graph (CFG) and data flow graph (DFG). CFG [10] is a representation, using graph notation, of all paths that might be traversed through a program during its execution. Each node in the graph represents a statement. Directed edges are used to represent jumps in the control flow. DFG[10] is a graph which represents data dependencies between a number of operations. Data as variables which are classified as a definition or a use such as Definition use (de-uses) - defined variable, computation uses (c-uses)- computed variable, and predicate uses (p-uses)- decided variable.

For this example, the values of maintainability metrics are MCX = 8, LOC = 37, and LCOM = 0.33. The metrics values for filtering refactoring are shown in Table III. (The shaded

TABLE II
REFACTORING FILTERING METRICS

Metric Name	Description	Formula
#switch	number of Switch Statement	# Switch Statement in the Method
#edge	number of Edge	# Edge
#node	number of Node	# Node
def	defined of a variable	# defined of a variable
c-use	computed of a variable	# computed of a variable
p-use	decided of a variable	# decided of a variable
cc	complexity of condition	# Edge - #Node in the condition

TABLE III
METRICS VALUES OF STATEMENT METHOD IN CUSTOMER CLASS

No	TE	Element	Metrics				
			# switch	def	c-use	p-use	cc
1	V	totalAmount		1	1	0	
2	V	frequentRenterPoints		1	1	0	
3	V	rentals		0	0	1	
4	V	result		4	3	0	
5	V	thisAmount		5	6	0	
6	V	each		1	3	5	
7	S	Switch (in lines 21-35)	1				
8	D	if ($each.getDaysRented() > 2$)					0
9	D	if ($each.getDaysRented() > 3$)					0
10	D	if ((each.getMovie(). getPriceCode() $== Movie.NEW\_RELEASE$) && $each.getDaysRented() > 1$)					1

Note: Types of element (TE) are classified as
$V - variable$, $P - parameter$, $S - statement$, and $D - decision$.

TABLE IV
REFACTORING FILTERING CONDITION (RFC)

Refactoring	RFC
R1. Extract Method	# $switch > 0$
R2. Replace temp with query	a variable:((def= 1)and ($c - use > 0$ or $p - use > 0$)) == true
R3. Introduce parameter object	set of variables used together in many methods (set of variables : def = 1) == true
R4. Preserve whole object	set of variables belong to the same type (ex. Date, string). set of variables : ((def = 1) and ($c - use > 0$ or $p - use > 0$)) == true
R5. Decompose conditional	$(\#edge \# node) > 0$

rows will be explained in the next section.)

B. Filter Candidate Refactorings

In most cases, not all refactorings are able to resolve the bad smell; therefore, our approach needs to filter refactorings. For this purpose, our approach defines refactoring filtering conditions (RFC) as boolean expressions over a set of CFG/DFG-based metrics (Table IV).

TABLE V
THE MAINTAINABILITY METRICS VALUES BEFORE AND AFTER EACH APPLIED CANDIDATE

Maintainability Metrics	Before Appiled Candidate	After Appiled Candidate			
		C1	C2	C3	C4
MCX	8	3	3	7	8
LOC	37	24	19	32	37
LCOM	0.33	0.5	0.37	0.37	0.5

In our example, the conditions in Table IV are instantiated for all elements in Table III and then evaluated. Consider the local variable totalAmount (the first row in Table III), it has one definition and one c-use. Therefore, this temp meets the RFC for R2. After all condition instances are evaluated, four instances pass: C1) R1 with switch statement (lines 21-35), C2) R2 with temp as totalAmount, C3) R2 with temp as frequentRenterPoints, and C4) R5 with if statement (lines 37).

C. Apply Candidate Refactoring

In this step, we apply candidate refactorings the one at a time and compute maintainability after applied. An applying example of four candidate refactorings can explain. For C1, switch statement fragment is moved to a new method (as shown in Fig. 2). After that, the values of maintainability metrics are computed. These values are MCX = 3, LOC = 24, and LCOM = 0.5.

```
21   getCharge(); // line 21-35 is removed.

}

   getCharge(){
        // line 21-35 is moved into this method.
   }
```

Fig. 2. Source code after C1 is applied

Analogously, we applied the other candidates one at a time and calculate the values of maintainability metrics. These values are shown in Table V.

D. Identify Refactoring Which Give the Height Maintainability

Our approach identifies the refactoring using rules. Rule 1: our approach selects the candidate with the lowest metric value for every metric. Rule 2: the approach selects the candidate with the lowest metric value for the greater number of metrics. Rule 3: if there is a tie, the approach randomly picks one.

In our example, Table V shows maintainability metrics values. C2 has the lowest value for every metric (Rule1); therefore, our approach identifies C2) replace temp with query for temp frequentRenterPoints to resolve the bad smell.

IV. EXPERIMENT

We perform an experiment to evaluate the effectiveness of our approach. The experiment compares the maintainability metric values after the refactorings selected by our

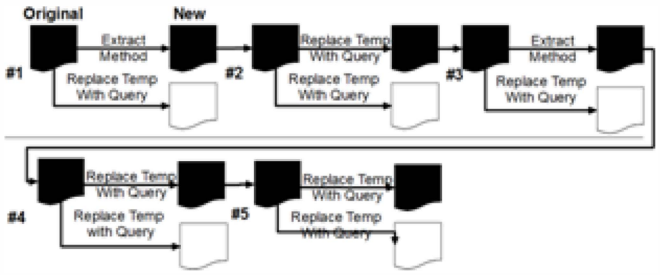

Fig. 3. Cases for Approach Evaluation

Maintainability Metrics	#1				#2				#3			
	old	book	ours	diff	old	book	ours	diff	old	book	ours	diff
MCX	8	3	3	0	3	3	2	1	3	2	2	0
LOC	37	26	19	17	36	22	17	5	21	18	16	2
LCOM	0.33	0.5	0.37	0.13	0.5	0.5	0.37	0.13	0.33	0.33	0.37	-0.04
Maintainability Metrics	#4				#5							
	old	book	ours	diff	old	book	ours	diff				
MCX	2	2	2	0	2	2	2	0				
LOC	18	16	16	0	16	13	13	0				
LCOM	0.33	0.37	0.37	0	0.37	0.4	0.4	0				

Fig. 4. Maintainability Metrics Values of Each Evaluation

approach and Fowlers book are applied. The approach considers refactoring used in the book as a baseline. If the approachs maintainability values are equal or more than the books maintainability values, it implies that the approach is acceptable for these cases. We use simple code metric (SCM) Netbeans plug in for calculating maintainability metrics. This experiment uses the case studies from the Fowlers book. The book shows five cases to resolve a long method bad smell and suggests a refactoring for each case. These cases are shown in Fig. 3, and the black symbol denotes the source code from the book. For each case, we use our approach for selecting refactoring, and the white symbol denotes source code refactored by our approach. After that, we calculate maintainability metric values of the applied code and compare these values.

Fig. 4 shows maintainability metrics values of each evaluation. The values in Before column are the maintainability metrics values of original code. The values in Book column are the maintainability metrics values of code after applying refactoring selected by book approach. The values in Ours column are the maintainability metrics values of code after applying refactoring selected by our approach. Diff column represents the difference of the metrics values between book column and ours column. For example, book and ours in case 1 yield less MCX values than before, so it implies that these two approaches help increase understandability. However, diff of MCX in case 1 is 0, it implies that understandability of book and ours is the same.

Fig. 5 shows that, in most cases, the ratio of our approach is more than the ratio of the refactoring book approach. Therefore, it can be concluded that our approach is more effective than the book approach for these cases. The experiment also confirms that the refactorings (extract method, replace temp with query, and decompose condition) are applicable to solve the long method bad smell.

Maintainability Metrics	#1			#2			#3		
	% Change		our : book	% Change		ours : book	% Change		ours : book
	book	ours	Ratio	book	ours	Ratio	book	ours	Ratio
MCX	-166.67	-166.67	1	0	-50	∞	-50	-50	1
LOC	-42.31	-94.74	2.24	-63.64	-111.76	1.76	-16.67	-31.25	1.88
LCOM	34	10.81	0.32	0	35.14	∞	0	12	∞
Maintainability Metrics	#4			#5					
	% Change		ours : book	% Change		ours : book			
	book	ours	Ratio	book	ours	Ratio			
MCX	0	0	∞	0	0	∞			
LOC	-12.5	-12.5	1	-23.08	23.08	1			
LCOM	10.81	10.81	1	7.5	7.5	1			

Fig. 5. The Result of %Change of book, %Change of ours and ratio Ours:Book

V. CONCLUSION AND FUTURE WORKT

Our approach introduces refactoring filtering conditions, which help novice programmers to find candidate refactorings. These conditions are defined at the program element level. They help novice programmers to know which program element should be refactored. Furthermore, we proposed the rules which help to select the refactoring that, if applied to original source code, yields the highest maintainability. The experiment result (Table IV) implies that our approach can identify more effective refactorings than the refactoring books.

However, our example covers only two refactorings (extract method and replace temp with query). In our future work, we plan to conduct an additional experiment that covers six refactorings for resolving long method.

REFERENCES

[1] T. Pienlert and P. Muenchaisri, *Bad-Smell Detedction Using OO SW Metrics*, International Conference on Computer Science, Software Engineering, Information Technology, e-Business, and Applications (CSITeA, Cairo, Eypt, December,2004)

[2] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, *JDeodorant: Identification and Removal of Type-Checking Bad Smells* , European Conference on Software Maintenance and Refactoring, 2008.

[3] P. Meananeatra and P. Muenchaisri, *Threshold of Object-Oriented Software Metrics for Detecting Bad Smells code*, Chulalongkorn University, Department of Computer Engineering, Thailand, 2001.

[4] L. Yang, H. Liu and Z. Niu, *Identify Fragments to Be Extracted from Long Methods*, Asia-Pacific Software

[5] F. Simon, F.Steinbruckner, and C. Lewerntz, *Metrics Based Refactoring* , In Proceeding Fifth European Conference

[6] R. Marinescu, *Detecting Design Flaws via Metrics in Object Oriented System* , In Proceeding of 39th International

[7] M. Fokaefs, N. Tsantalis and A. Chatzigeorgiou , *JDeodorant: Identification and Removal of Feature Envy Bad Smells* , 23rd IEEE International Conference on Software Maintenance (ICSM 2007).

[8] M. Fowler, *Refactoring: Improving the Design of Existing Programs*, Addison-Wesley, 1999.

[9] M. J. Harrold, G. Rothermel and A. Orso, *Representation and Analysis of Software*, 2002.

[10] B. Beizer , *Software Testing Techniques*, Van Nostrand Reinhold, 1999