IEICE TRANS. INF. & SYST., VOL.E101–D, NO.7 JULY 2018

PAPER Special Section on Knowledge-Based Software Engineering

Refactoring Opportunity Identification Methodology for Removing Long Method Smells and Improving Code Analyzability

Panita MEANANEATRA[†a)], ***Member***, **Songsakdi RONGVIRIYAPANISH**[†], ***and*** **Taweesup APIWATTANAPONG**[††], ***Nonmembers***

SUMMARY An important step for improving software analyzability is applying refactorings during the maintenance phase to remove bad smells, especially the long method bad smell. Long method bad smell occurs most frequently and is a root cause of other bad smells. However, no research has proposed an approach to repeating refactoring identification, suggestion, and application until all long method bad smells have been removed completely without reducing software analyzability. This paper proposes an effective approach to identifying refactoring opportunities and suggesting an effective refactoring set for complete removal of long method bad smell without reducing code analyzability. This approach, called the long method remover or LMR, uses refactoring enabling conditions based on program analysis and code metrics to identify four refactoring techniques and uses a technique embedded in JDeodorant to identify extract method. For effective refactoring set suggestion, LMR uses two criteria: code analyzability level and the number of statements impacted by the refactorings. LMR also uses side effect analysis to ensure behavior preservation. To evaluate LMR, we apply it to the core package of a real world java application. Our evaluation criteria are 1) the preservation of code functionality, 2) the removal rate of long method characteristics, and 3) the improvement on analyzability. The result showed that the methods that apply suggested refactoring sets can completely remove long method bad smell, still have behavior preservation, and have not decreased analyzability. It is concluded that LMR meets the objectives in almost all classes. We also discussed the issues we found during evaluation as lesson learned.

key words: *code analyzability, long method bad smell, refactoring, software engineering, software maintenance*

1. Introduction

In any software development life cycle, the maintenance cost accounts for around 40–70% of the total cost [1]. Software often needs to be maintained in order to add new functionalities and fix remaining faults [2]. Ad-hoc maintenance produces legacy software that is too large and complex [3]. For developers, this legacy software is hard to understand and modify, which makes faults fixing more difficult. In a faults fixing activity, if code is easy to understand, the cause of faults can be spotted quickly. This paper argues that such code has high analyzability, which is an important sub-characteristic of maintainability (ISO/IEC9126, 2003). The main root cause of low analyzability in software is its poor structure, such as duplicated code, large class, or long method. These are collectively known as bad smells [4].

Bad smell removal makes software easier to understand and modify [3], [5], increases software analyzability, and decreases the time needed to analyze faults. Although there are several types of bad smell, this paper addresses only the long method bad smell since it occurs most frequently [6] and causes other bad smells, such as large class [7]. This bad smell makes code difficult to understand and reuse [8], [9], and thus lessening analyzability of the code. The long method bad smell exhibits four characteristics: 1) too many statements inside a method, 2) too many unnecessary local variables, 3) too many parameters, and 4) excessively complex conditions [4]. One of the well-known techniques for removing bad smells is refactoring.

1.1 Motivation

Refactoring is a technique for improving the structure of software without changing its behavior. Most low quality programs have bad smells. Removing bad smells improves several characteristics, especially analyzability. Fowler [4] explained informally situations where refactorings could be applied and proposed a set of refactoring techniques used for removing each kind of bad smell. Following these guidelines, a developer still has to decide which refactorings to apply at which code location. Currently, most of the existing tools only help a developer apply the refactoring technique. They do not help the developer making those decisions.

In case where several refactorings are applicable, in order to select appropriate refactoring to apply, a developer needs to compare the impact of each refactoring on the analyzability improvement and then selects the best one. These tasks require a lot of time and effort. Therefore, an automatic refactoring suggestion tool becomes crucial. It helps developers recognize the appropriate refactorings, identifies the code locations where the refactoring techniques should be applied, quantifies the analyzability improvement resulting from applying the refactoring and finally suggest a set of refactoring techniques to eliminate completely long method bad smell without reducing code analyzability.

For removing long method bad smells, Fowler [4] proposed six refactoring techniques: 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. However, the existing research works have proposed approaches to identifying

Manuscript received November 8, 2017.
Manuscript revised March 14, 2018.
Manuscript publicized April 26, 2018.

[†]The authors are with Computer science department, Thammasat University, 178–5990 Thailand.
[††]The author is with National Science and Technology Development Agency, Thailand.
a) E-mail: panita.meananeatra@nectec.or.th

DOI: 10.1587/transinf.2017KBP0026

onlytwo refactoring techniques: the extract method [6], [7], [10], [11] and replace temp with query [6], [10]. No research has proposed an approach to identifying replace temp with query for a temp variable inside for loop, or while loop and the other four refactorings for removing long method bad smells. Moreover, when several refactoring techniques are possible, no approach proposed to consider the impacts on code analyzability to select the refactoring.

In our opinion, effective refactoring suggestion approach should suggest refactorings that remove bad smells, increase code analyzability, and minimize the number of statements impacted by refactoring. No research has proposed an approach to suggesting and applying refactorings until long method bad smells are completely removed from the code without reducing code analyzability and minimizing the impacts on the code.

1.2 Related Works

There are six refactoring techniques that can, collectively remove long method bad smells [4]. However, existing research on refactoring opportunity identification uses only two techniques.

The first technique "replace temp with query", Pienlert and Muenchaisri [6] presented an approach for detecting and locating bad smells based on software metrics. They defined the following detection techniques for six bad smells including long method. Each detection technique was described in the following format: definition, motivation, strategy, metrics, specification of outliers, and application of refactoring. After applying refactoring following their approach, they showed that by using their maintainability metrics, the maintainability of the refactored code improved. The disadvantage of their approach was that it could only identify elements to be refactored at the class or method levels. It could not suggest elements at the statement level. Kataoka et al. [14] defined a set of conditions for identifying refactoring techniques using program invariants: when particular invariants hold at a program point, a specific refactoring is applicable. Since most programs lack explicit invariants, an invariant detection tool called Daikon was used to infer the required invariants. In summary, two research studies [6], [14] proposed an approach to identifying temp variable using information from an abstract syntax tree representation. However, their research did not concern a temp variable which is inside for loop, or while loop. Moreover, their research did not consider code analyzability to select appropriate refactorings.

The second technique "extract method", Maruyama [10] proposed a mechanism that extracted lines of code, to a new method from an existing method using block based slicing. Calling and called methods are unconsidered in his approach. Side effects may occur in the new method, which consequently violate the condition of behavior preservation. Liu and Niu [11] proposed an approach to recommending fragments within long methods for extraction using blank line slicing. The disadvantage is that it analyzes statements but does not cover all cases of long method bad smell (such as variables and parameters) and the variable relationships are unconsidered, so the extracted method was incomplete. Tsantalis et al. used the union of static slices for extracting the complete computation of a given variable declared inside a method [7]. Their approach also proposed a set of rules that preserved the code behavior and prevented code duplication. They could find code in which statements were affected by applying extract method refactoring and could improve the quality of the code with the removal of code duplication. In summary, these research works focused on identifying the code fragments to be extracted method by using code slicing approaches. Our approach used the technique embedded in JDeodorant tool [13], developed by Tsantalis et al. to identify extract method refactoring. However, some research [7], [13] used a single metric and suggested only one appropriate refactoring; therefore, they may not be able to remove long method bad smells completely.

1.3 Our Contribution

In this paper, we propose an effective approach to identifying appropriate refactorings for long method bad smell removal, while increasing code analyzability and minimizing the number of statements impacted by the refactorings. Our approach will also help developers decrease the number of refactorings and lower the risk of incurring side effects. The contributions of our approach are as follows:

- Defining the refactoring enabling conditions in the form of rules in logic programming for four refactoring techniques: replace temp with query, introduce parameter object, preserve whole object, and decompose conditional using program analysis and software metrics. We complemented the existing research works [6], [7], [10], [11] with our refactoring enabling conditions to cover the five refactoring techniques for removing long method bad smell and make an automatic refactoring suggestion for long method removal possible.
- Proposing an algorithm for identifying statements impacted by the four refactoring techniques using program slicing technique and quantifying the analyzability level of code. The number of statements impacted by a refactoring and analyzability level will be used as criteria for refactoring selection.
- Proposing the steps of refactorings suggestion using two criteria: code analyzability level and the number of statements impacted by the refactorings

This paper is organized as follows. Section 2 describes our approach for refactoring identification. Section 3 discusses our evaluation and Sect. 4 presents our conclusions.

2. Methodology

2.1 The Main Concept

The long method bad smell exhibits four long method characteristics, and Fowler [4] proposed five refactorings for removing all four, as summarized in Table 1. To automate the identification of refactoring opportunities, we formalize the situations of long method characteristics into refactoring enabling conditions. In this work, we defined refactoring enabling conditions for four refactoring techniques: replace temp with query (RTWQ), introduce parameter object (IPO), preserve whole object (PWO), and decompose conditional (DC). We also used JDeodorant to find extract method (EM) refactoring opportunities. A refactoring enabling condition is composed of arithmetic and logical expressions based on code structure and code metrics such as the control flow path of a method and the number of parameters. When the refactoring enabling condition of a refactoring technique is met, its related refactoring is applicable.

Our research problem is "How to identify appropriate refactorings and code locations that should be applied to remove the long method bad smells effectively?" and involves three challenges: 1) identifying of the information required to understand the code structure and means of obtaining that information; 2) calculating the refactorings necessary to remove all traits of long method bad smells in an effective way, and 3) determining to which code elements and code locations the selected refactorings should be applied.

The first challenge is identifying the information required to understand the code structure and the means of obtaining that information. To support refactoring enabling condition, we need several program analysis techniques: control flow, data flow, control dependence, data dependence, point-to analysis, and program slicing. These techniques use five representations: an abstract syntax tree, a control flow graph, a data flow graph, a class dependence graph, and point-to graph. We analyzed the informal condition for selecting four refactoring techniques: RTWQ, IPO, PWO, and DC, as defined by Martin Fowler.

For example, to detect a local variable for "replace temp with query" refactoring or to detect a parameter list for "introduce parameter object" refactoring, we used an abstract syntax tree as the representation for program analysis. All informal conditions for enabling four refactorings and the representations used for checking the refactoring enabling conditions of each refactoring are shown in Table 2.

The second challenge is calculating the refactorings necessary to remove all characteristics of long method bad smells in an effective way. An effective approach to identifying and suggesting appropriate refactoring requires two core steps: 1) identify applicable refactorings, and 2) suggest a set of effective refactorings. In the first step, we define conditions to identify all applicable refactorings for removing long method characteristics. In our previous study [15], we proposed a set of refactoring filtering conditions based on software metrics, and defined them in terms of elements in data flow and control flow graphs. For example, in the refactoring filtering condition "replace temp with query", a temporary variable becomes a candidate for being replaced by a query method if it is assigned and then used at least once along a data flow path.

These refactoring filtering conditions are defined as rules enabling an application of refactoring for removing long method bad smells. In another work [16], we combined logic meta programming, software metrics, control flow and data flow analysis, to improve refactoring filtering conditions. In this paper, we further extended the previous method by including control and data dependence analysis as well as pointer analysis. We call the enhanced refactoring filtering conditions as the refactoring enabling conditions. We present the refactoring enabling conditions formally in Fig. 1.

In the second step, we consider all applicable refactorings according to two criteria: code analyzability and the number of statements impacted by refactorings. Our approach uses code analyzability model based on statistical method and identify statements impacted by refactorings. We suggest an effective refactoring set candidate that can remove the bad smells, enhance code analyzability and minimize statements impacted by refactorings.

Table 1 Summary of relationship between long method characteristics and refactoring solutions.

Long method characteristics	Solution	Refactoring	Situation of refactoring	Brief of refactoring application	Element
1) too many statements inside a method	Find parts of the method that seem to go nicely together and make a new method	Extract method	You have a code fragment that can be grouped together.	Turn the fragment into a method whose name explains the purpose of the method	Statements
2) too many unnecessary local variables	To eliminate the temps	Replace temp with query	Using a temp variable to hold the result of an expression	Extract the expression into a method. Replace all references to the temp with the expression. The new method can then be used in other methods.	A local variable
3) too many parameters	Long lists of parameters can be slimmed down	Introduce parameter object	You have a group of parameters that naturally go together.	Replace them with an object	parameters
		Preserve whole object	You are getting several values from an object and passing these values as parameters in a method call.	Send the whole object instead	
4) too complex conditions	To deal with conditional expressions (conditional and loops)	Decompose conditional	You have a complicated conditional (if-then-else) statement.	Extract methods from the condition, then part, and else parts.	a boolean condition expression

Table 2 The refactoring techniques and the models for program analysis.

Refactoring	Informal condition	Representation
Replace temp with query	1) A candidate variable for RTWQ is a part of statements other than loop or selection statements, and must be only of primitive type. It must be assigned at least once in such paths, but not more than once, and must be used at least once in computation or predication. 2) A candidate variable for RTWQ is a part of a selection statement, must be only of primitive type, and must be assigned at least once in such paths, but not more than once, and must be used in computation or predication. 3) A candidate variable for RTWQ is a part of a loop statement, must be only of primitive type, and must not depend on other variables. It must be assigned at least once in such paths, with an expression that has no side effects. This candidate variable must be used at least once in computation or predication within the path where it is assigned.	abstract syntax tree, control flow graph, data flow graph, point-to graph, class dependence graph
Introduce parameter object	1) A candidate set of parameters for IPO is a subset of parameters having a size greater than one and used together in invocation methods. 2) A candidate set of parameters for IPO is a subset of parameters having size greater than one and which is used at least once in a computation statement. 3) A candidate set of parameters for IPO is a subset of parameters having size greater than one and used at least once in predicate statements.	abstract syntax tree, control flow graph, data flow graph,
Preserve whole object	A set of parameters for PWO must belong to the same type. Each variable of the set must be assigned once and be used in computation or predication at least once.	abstract syntax tree, control flow graph, data flow graph,
Decompose conditional	A condition for DC must be a part of a loop or selection statement; it must have more than one sub-expression on a branch.	abstract syntax tree, control flow graph,

The refactoring enabling condition of "Replace temp with query": v_1 is a candidate variable to apply the refactoring "RTWQ". v_1 must satisfy one of the following rules:

Rule1: $\exists v_1 \in DU_{mc_1} \cdot t(v) \epsilon PT \cdot size(CFGPATH_{mc_1}) > 0 | \forall dfgpath_{v_1} \in DFGPATH_{v_1} | du(v_1)_{dfgpath_{v1}} = 1 \wedge (cu(v_1)_{dfgpath_{v1}} > 0 \vee du(v_1)_{dfgpath_{v1}} > 0)$

Rule2: $\exists v_1 \in DU_{mc_1} \cdot t(v) \epsilon PT \cdot statement_{v_1} \in SELECTSTATEMENT_{mc_1} \cdot isStatementinCFGPath(statement_{v_1}, CFGPATH_{mc_1}) \cdot size(CFGPATH_{mc_1}) > 0 | \forall dfgpath_{v_1} \in DFGPATH_{v_1} | du(v_1)_{dfgpath_{v1}} = 1 \wedge (cu(v_1)_{dfgpath_{v1}} > 0 \vee pu(v_1)_{dfgpath_{v1}} > 0)$

Rule3: $\exists v_1 \in DU_{mc_1} \cdot t(v) \epsilon PT \cdot statement_{v_1} \in LOOPSTATEMENT_{mc_1} \cdot isStatementinCFGPath(statement_{v_1}, CFGPATH_{mc_1}) \cdot size(CFGPATH_{mc_1}) > 0 | \forall dfgpath_{v_1} \in DFGPATH_{v_1} | du(v_1)_{dfgpath_{v1}} = 1 \wedge (cu(v_1)_{dfgpath_{v1}} > 0 \vee pu(v_1)_{dfgpath_{v1}} > 0) \wedge assignVarAtMostOnceInPathWithExpression(statement_{v_1}, CFGPATH_{mc_1}, Exp_1) \wedge noSideEffect(v_1, Exp_1)$

The refactoring enabling condition of "Introduce parameter object": $params$ is a candidate set of parameters to apply the refactoring technique "IPO". $params$ must satisfy one of the following rules:

Rule1: $\exists mc_1 \in M_c \; \exists params \subseteq PARAM_{mc_1} | size(params) \geq 2 \wedge \exists cm \in CM_{mc_1} \; \forall param \in params \; \exists \arg \in ARGUMENT_{mc} | isArgumentOf(arg, param) = true$

Rule2: $\exists mc_1 \in M_c \; \exists params \subseteq PARAM_{mc_1} | size(params) \geq 2 \wedge cu(params) \geq 1$

Rule3: $\exists mc_1 \in M_c \; \exists params \subseteq PARAM_{mc_1} | size(params) \geq 2 \wedge pu(params) \geq 1$

The refactoring enabling condition of "Preserve whole object": $params$ is a candidate set of parameters to apply the refactoring technique "PWO". $params$ must satisfy the following rule:

Rule: $\exists mc_1 \in M_c \; \exists params \subseteq PARAM_{mc_1} \exists mi \exists args \in ARGUMENT_{mi} | isMethodInvocation(mi, mc_1) \; params \subseteq PARAM_{mc_1} \wedge size(params) \geq 2 \wedge size(args) == size(params) \wedge (\forall \arg \in args \; \exists param \in params \; \exists o_1, o_2 \in O | isArgumentOf(arg, param) \wedge (derivedFrom(arg, o_1) \wedge derivedFrom(arg, o_2) \wedge o_1 == o_2)$

The refactoring enabling condition of "Decompose conditional": con_1 is a candidate boolean condition to apply the refactoring "DC". C1 must satisfy the following rule:

Rule: $\exists \in CON_{mc_1} | size\left(LO_{con1_{mc_1}}\right) \geq 2$

Fig. 1 The refactoring enabling condition of four refactoring techniques.

The third challenge is determining the code elements and code locations where the selected refactorings should be applied. We use program slicing to analyze the code structure and use class dependence graphs to identify statements affected by each selected refactoring.

2.2 The Steps of the Proposed Approach

Our approach, called the long method remover or LMR, is a semi-automated approach to removing long method bad smells that requires some manual actions and decisions by from a developer. The approach includes eleven steps, outlined in Fig. 2, all but two of which can be automated. The steps that require intervention of the developer are steps 6 "does the developer reject all refactoring candidates?" and 10 "does the developer accept this solution?" The proposed approach starts when the developer inputs a piece of code with long method bad smells and an analyzability model. The approach focuses on the code at the method level and considers each method in a class individually.

To demonstrate our approach, we used an example written in Java with long method smell from Fowler [4]. This method "getFlowBetween" of the class Account is shown in Fig. 3. Code in each line is designated with a line identifier as shown on the left edge of the figure.

2.2.1 Transform Code to an Abstract Syntax Tree and Four Graphs

To calculate the metrics, LMR transforms code with long

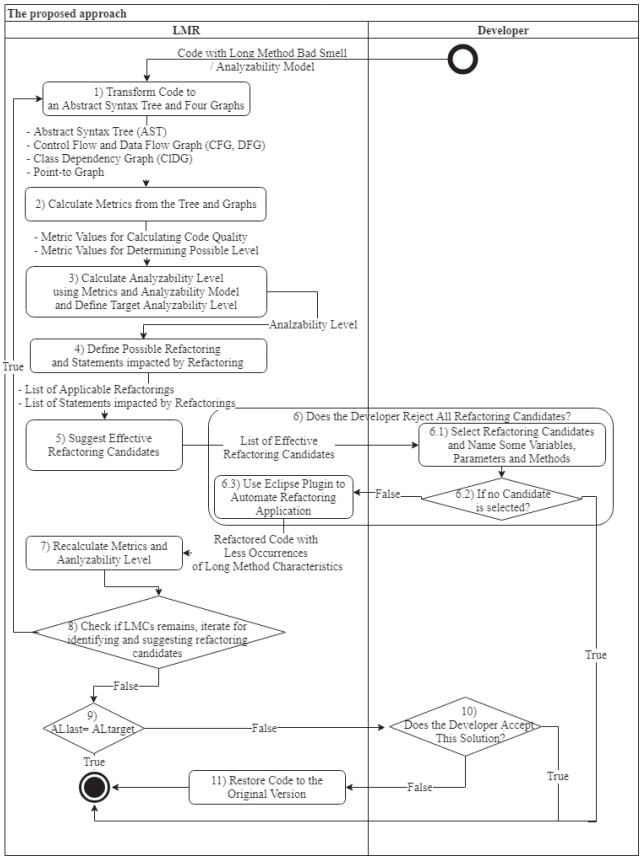

Fig. 2 Steps of the proposed approach.

```
   public class Account{
     private Vector _enties = new Vector();

S0   double getFlowBetween(Date start, Date end){
S1     double result=0;
S2     Enumeration e = _enties.element();
S3       while(e.hasMoreElements()){
S4         Entry each = (Entry)e.nextElement();
S5         Date date = each.getDate();
S6         if(date.equals(start)||date.equals(end)||date.after(start)||date.after(end)){
S7             result +=each.getValue();
           }
         }
S8       return result;
     }
   }
```

Fig. 3 Code method "getFlowBetween" of a class account.

method bad smells into an abstract syntax tree and four graph representations: control flow graph, data flow graph, class dependence graph, and point-to graph.

After transforming the code to an abstract syntax tree, we can retrieve code information such as class name, modifier of class, attributes of class, and so on. We also obtain other information, such as the number of parameters, required for computing the analyzability level.

Next, we analyze the code in the form of a control flow graph, data flow graph, and class dependence graph. The control flow graph represents the flow of control from one instruction to another [21]. The data flow graph represents the flow of data from definitions to usages [21]. From these two graphs, we calculate the metrics used in the refactoring enabling conditions, such as the number of the definition uses of a variable and the computation uses of parameters. The class dependences graph represents the statement and variable dependence inside methods. It contains statements as nodes and two edge types: control dependence edge and data dependence edge [21]. We use this graph for program slicing, which identifies the statements to which refactorings will be applied.

Finally, the point-to graph shows the objects in the heap and the variables that point to them. We use this graph for checking whether a method invocation has a side effect by applying the purity analysis technique [22]. If a side effect is detected, the refactorings cannot be applied.

For our example, we transform code in Fig. 3 to an abstract syntax tree and four graphs. The control flow graph contains eight nodes (the number of executed statements in the code) and 10 edges, e.g., edge s3->s4 is the true branch of the while statement. Due to the lack of space, from here on, we neither show the graphs nor go into details; we will only provide the results of each step.

2.2.2 Calculate Metrics from Abstract Syntax Tree and Graphs

The LMR calculates two metric groups: 1) metrics for computing the code analyzability level, and 2) metrics used in refactoring enabling conditions. The outputs from this step are metric values of these two groups for a version of code.

The first group of metrics depends on the analyzability model provided as input. In this case, we use the analyzability model from previous work [17]. This analyzability model requires three metrics for calculating the analyzability level: 1) method lines of code, 2) the number of parameters in the method, and 3) the nested block depth. Methods line of code is the number of non-blank and non-comment lines inside each method body.

To use refactoring enabling conditions, LMR requires five metrics: (1) the definition use of a variable, du(v), (2) the computation use of a variable, cu(v), (3) the predicate use of a variable, pu(v), (4) the computation use of parameters, cu(params), and (5) the predicate use of a group of parameters, pu(params). du(v) counts the number of assignments of variable v in data flow graph paths. cu(v) counts the number of computations of variable v in data flow graph paths. pu(v) counts the number of predicate uses of variable v in data flow graph paths. cu(params) counts the number of computation statements that use all of the parameters in the list "params". The pu(params) is calculated by a formula which counts the number of the predicate statements that use all of the parameters in the list "params".

2.2.3 Calculate the Analyzability Level Using Metrics and Analyzability Model and Define the Target Analyzability Level

Assume that the analyzability model provide as input spec-

ifies the following formulas Eq. (1)–Eq. (5) for calculating the analyzability level. With the values of metrics used for calculating code analyzability, we can derive the analyzability level of the code by applying these formulas. The result can be one of the three levels: poor (1); fair (2), and good (3). Before identifying refactoring, LMR calculates the analyzability level of the original code, defined as $AL_{original}$. In addition, LMR sets the target analyzability level, defined as AL_{target}, to "good" as the default value.

$$P(Y \leq 1) = \frac{1}{1 + e^{-cp_1 - (C_1 PAR) - (C_2 LMOC) - (C_3 NBD)}} \quad (1)$$

$$P(Y \leq 2) = \frac{1}{1 + e^{-cp_2 - (C_1 PAR) - (C_2 LMOC) - (C_3 NBD)}} \quad (2)$$

$$P(Y \leq 3) = 1 \quad (3)$$

$$P(Y = 2) = P(Y \leq 2) - P(Y \leq 1) \quad (4)$$

$$P(Y = 3) = 1 - P(Y \leq 2) \quad (5)$$

Where

MLOC stands for method lines of code.
PAR stands for the number of parameters in method.
NBD stands for the nested block depth.

Also assume that the analyzability model from our previous work [17] specifies the values of $cutpoint_1$–cp_1 and $cutpoint_2$–cp_2 as −10.051 and −1.170, respectively and the values of C_1, C_2, and C_3 as (−1.675), (−0.032) and (0.51), respectively. According to the analyzability model, the probability values of each analyzability level are calculated and then compared. The highest probability computed determines the analyzability level of the code.

For our example, using the aforementioned analyzability model, LMR calculates the analyzability levels of method "getFlowBetween" by the applying the metric values in the formula, PAR = 2, MLOC = 9, NBD = 2. The calculated analyzability level is three because this is just a tiny example to illustrate our approach.

2.2.4 Determine Possible Refactorings and Statements Impacted by Refactorings

LMR uses JDeodorant to identify extract method opportunities and uses the refactoring enabling conditions to identify opportunities for applying the other four refactorings. This step is divided into two sub-steps.

(1) Determine all possible refactorings

To determine all possible refactorings in a long method, LMR checks refactoring enabling conditions with the metric values obtained from step 2 of our approach see Sect. 2.2.2. When a condition of the refactoring enabling conditions (see Table 3) of a refactoring is met, LMR records that refactoring in a list of applicable refactoring techniques and long method characteristics.

(2) Determine statements impacted by the refactorings

LMR identifies statements impacted by each refactoring opportunity as follows

1) The statements impacted by replace temp with query are the assignment statement of that temporary variable, which will be extracted to create a query methodand the statements for which the temporary variable is used. To identify these statements, we use our algorithm from our previous work [20].

2) The statements impacted by introduce parameter object are the statements containing the parameters that will be replaced by the parameter object.

3) The statements impacted by preserve whole object are the statements containing the parameters that will be replaced by the whole object.

4) The statements impacted by decompose conditional are the statements containing the conditions that will be replaced by the condition method.

5) The statements impacted by extract method are the statements of the code which will be extracted for creating an extracted method. These statements are identified by the technique proposed by Tsantalis and Chatzigeorgiou [7].

As an example, we illustrate only the refactoring enabling condition of refactoring "introduce parameter object". This refactoring enabling condition has three rules (as shown in Fig. 1). Rule 3 is met because the number of parameter is two and the predicate use of parameters is one. In other word, this method has two parameters, and they are used together in one statement. Since the condition requires that only one rule is met, this example method satisfies the condition of refactoring "introduce parameter object". In summary, LMR identifies two refactoring opportunities: IPO (start, end) and DC (if). Then LMR identifies and counts statements impacted by two applicable refactorings. The statement impacted by IPO (start, end) is S0 (in Fig. 3) and the statements impacted by DC (if) is S6. Both the number of statements impacted by IPO (start, end) and the number of statements impacted by DC (if) are one.

Table 3 Refactoring identification tool comparisons.

Refactoring techniques/Tools	LMR Plugin	JDeodorant	IntelliJ IDEA
Replace temp with query	✓	×	✓
Introduce parameter object	✓	×	×
Preserve whole object	✓	×	×
Decompose conditional	✓	×	×
Extract method	Uses JDeodorant	✓	×

Note:
✓-- can identify a refactoring technique.
× -- cannot identify a refactoring technique.

2.2.5 Suggest Effective Refactoring Candidates

The objective of our approach is to suggest refactoring candidates that remove long method bad smell, increase code analyzability, and have the least impact on the code in terms of the number of statements impacted by the refactorings. LMR considers the list of applicable refactoring using two factors: analyzability level and the number of statements im-

pacted by the refactorings. To suggest effective refactoring candidates, LMR executes three steps and these steps are described as follows

(1) Check conflicting pairs of refactorings opportunities

First, our approach checks whether any two refactoring opportunities cannot be applied successively in any order, i.e., one refactoring conflicts with each other. A conflict occurs if a refactoring affects the same statements as the other one. LMR checks conflicts using the following Boolean predicate.

$$isConflicting_{ROi,j} = (ImpactStatement_{ROi} \cap ImpactStatement_{ROj}) \neq \varnothing$$

$$where\ i,j \in [i,\ldots,n_2]$$

where as;

- *RO : stands for a refactoring opportunity.*
- $ImpactStatement_{ROi}$ *: stands for a set of impacted statements of* RO_i.

isConflicting is true if the intersection of two sets of affected statements does not equal an empty set.

For our example, LMR checks conflicts in the two refactoring opportunities identified in the previous step. As a result, both refactoring opportunities do conflict because they affect statement S6, thus these two refactoring opportunities cannot be applied successively in any order.

(2) Identify refactorings sets

LMR enumerates refactoring sets using the graph theory. A refactoring set is a set of refactoring opportunities where each member does not conflict with one another. First, LMR builds an undirected graph where each refactoring opportunity is a vertex and each edge connects two refactoring opportunities that do not conflict, i.e., $isConflicting_{ROi,j}$ is false. Based on this graph, finding a refactoring set corresponds to finding the set of vertices in a maximal clique (Let G be a graph. A clique in G is a subgraph in which every two nodes are connected by an edge [21]) because, by definition of a clique, each and every refactoring opportunity in the subgraph of our graph representation does not conflict with one another. Therefore, to numerate all refactoring sets, LMR finds all maximal cliques in the graph and identifies the set of vertices in each maximal clique as a refactoring set.

For our example, the number of vertices and edges are two and zero; so the number of maximal cliques for this method is two. It implies that there are two refactoring sets, each contains one refactoring opportunity.

(3) Select an effective refactorings sets

Finally, LMR selects an effective refactoring set. An effective refactoring set is a set of refactorings which, when applied, results in code with the highest code analyzability level. If there is a tie, LMR uses the lowest of the number of statements impacted by refactoring as tie-breakers. However, because no refactorings are applied at this stage, LMR cannot determine the actual analyzability level. Therefore, LMR needs to predict analyzability level. To do so, LMR needs to predict new values of the three metrics used.

$$EffectiveRefactoringSet = \exists j \in [1..n_1]\ RS_j \in RS \mid \forall i \in [1..n_1] \wedge i \neq j\ (PredictAL_{RS_j} > PredictAL_{RS_i}) \vee \Big((PredictAL_{RS_j} == PredictAL_{RS_i}) \wedge (NSIR_{RS_j} \leq NSIR_{RS_i})\Big)$$

Where as;

- *EffectiveRefactoringSet:* stands for an effective refactoring set.
- RS_k *: stands for set of refactoring k.*
- $PredictMetrics_{RS_k}$ *: stands for predicted metric values of code after applying refactoring set "*RS_k*".*

$$= <MLOC_{RS_k}, PAR_{RS_k}, NBD_{RS_k}>$$

- $MLOC_{RS_k}$ *: stands for predicted metric values of method's lines of code (MLOC) of code after applying refactoring set "*RS_k*".*
- PAR_{RS_k} *: stands for predicted metric values of the number of parameters (PAR) of code after applying refactoring set "*RS_k*".*
- NBD_{RS_k} *: stands for predicted metric values of the nested block depth (NBD) of code after applying refactoring set "*RS_k*".*
- $PredictAL_{RS_k}$ *: stands for predicted analyzability level (AL) of code after applying refactoring set "*RS_k*".*

$$= AnalyzabilityLevel(PredictMetrics_{RS_k})$$

- $AnalyzabilityLevel(PredictMetrics_{RS_k})$: *a function for* calculating *predicted analyzability level of code after applying refactoring set "* RS_k *" using* $PredictMetrics_{RS_k}$ *and code analyzability model.*
- $NSIR_{RS_k}$ *: stands for the number of statements impacted by refactoring set "*RS_k*", defined in section* 2.2.4.

Fig. 4 The effective refactoring selection algorithm.

For brevity, we only demonstrate how to predict these three metrics after applying a replace temp with query refactoring. This refactoring removes the assignment statement of a temporary variable and creates a new query method. Applying a replace temp with query refactoring may remove a nested block because the whole block is moved to the new method. Therefore, LMR predicts that the new value of method lines of code after applying RTWQ will be reduced, while the value of the nested block depth may or may not be reduced. However, the value of the number of parameters in the method will remain the same.

Since a refactoring set may comprise several refactoring opportunities, LMR must predict the three metrics, the analyzability level, and the number of statements impacted as if all the refactorings in the set are applied. After predicting the values for of all refactoring sets, LMR selects an effective refactoring set with the effective refactoring selection algorithm. The algorithm is shown in Fig. 4.

For our example, RS_i is refactoring opportunity "IPO (start, end)". we predict three metric values: the number of parameters in method becomes one, because LMR applies refactoring "introduce parameter object", the method lines of code is still nine, and the nested block depth is still two. The predicted analyzability level is three, while the number of statements impacted refactoring is two. In RS_j as refactoring opportunity "DC (if)", we predict three metric values likewise the previous values and the number of statements impacted refactoring is one. After using effective refactoring selection algorithm, LMR suggests that RS_j or refactoring opportunity "DC (if)" because the analyzability level of them is the same but the number of statements impacted

refactoring of RS_j is less than the number of statements impacted refactoring of RS_i.

2.2.6 The Steps for the Decision to Break LMR

In step 6, if the developer rejects all the refactorings suggested in step 5, the proposed approach is terminated. Otherwise, the developer chooses one refactoring set and applies it, using the Eclipse plugin. In step 7, LMR recalculates the metrics and code analyzability, as in steps 2 and 3 of the proposed approach. After applying all of the suggested refactorings in the set, LMR recalculates the analyzability level.

In steps 8-11, LMR checks whether any long method characteristic persists by checking the refactoring enabling conditions and using JDeodorant. If a long method characteristic remains, LMR repeats steps 1-7. When all long method characteristics have been removed, LMR compares AL_{last} with AL_{target}. If AL_{last} is equal to AL_{target}, the LMR outputs the refactored code. Otherwise, the developer is asked to accept or reject this solution. In the case of rejection, LMR restores the code to its original version.

In our example, LMR applies refactoring opportunity "DC (if)" at the first iteration. Then LMR recalculates metrics and the AL; the analyzability level of refactored code after the first iteration, AL_1 is good. In step 8, LMR found a long method characteristic. Since LMR repeats steps 1-7, LMR applies refactoring opportunity "IPO (start, end)" at the second iteration. Then LMR recalculates metrics and the AL; the analyzability level of refactored code after the second iteration, AL_2 is good. In steps 8, LMR does not find any long method characteristic. Therefore, AL_{last} is AL_2 and the value of AL_{last} is also good. After that. AL_{last} is compared to AL_{target}. Since AL_{last} is equal to AL_{target}, LMR shows the refactored code that removes all long method characteristics and the same AL as the original code. In summary, LMR applies refactoring to candidates in only two iterations. AL_{last} equals to $AL_{original}$ and AL_{last} is good.

We explain here the reason why an infinite loop cannot occur in steps 1-8. We note that each refactoring technique relates to only one code element, as shown in Table 1, and does not introduce any element that can cause other long method characteristics. To prove that an infinite loop cannot occur in the algorithm proposed by our approach, we give the definitions and prove the conjectures, lemmas and theorems as follows.

Definition: For a long method being proceeded,
1. Let n_1 and n_1' be the numbers of statements applicable for extract method in the iterations i and i + 1 respectively.
2. Let n_2 and n_2' be the numbers of local variables applicable for "replace temp with query" in the iterations i and i + 1 respectively.
3. Let n_3 and n_3' be the numbers of parameter groups applicable for "introduce parameter object" or "preserve whole object" in the iterations i and i + 1 respectively.
4. Let n_4 and n_4' be the numbers of boolean conditions applicable for "decompose conditional" in the iterations i and i + 1 respectively.
5. Let n_{total} and n_{total}' be the total number of all long method characteristics in the iteration i and i + 1, respectively. Formally, $n_{total} = n_1 + n_2 + n_3 + n_4$ and $n_{total}' = n_1' + n_2' + n_3' + n_4'$.

Conjecture 1: After applying refactoring technique "extract method", the number of long method characteristics "too many statements" in the code of a long method decreases and no long method characteristic is added. It implies that $n_1' < n_1 \wedge n_2' \leq n_2 \wedge n_3' \leq n_3 \wedge n_4' \leq n_4$.

Proof: Removing a long method characteristic "too many statements" by applying "extract method" will move some local variables or some statements from the long method to a newly introduced method which does not cause additional three other long method characteristics. Thus, n_1' is strictly less than n_1, while the new values of other variables are equal to or less than their pervious values.

Conjecture 2. After applying refactoring technique "replace temp with query", the number of long method characteristics "too many unnecessary local variables" in the code of a long method decreases and no long method characteristic is added. It implies that $n_1' \leq n_1 \wedge n_2' < n_2 \wedge n_3' \leq n_3 \wedge n_4' \leq n_4$.

Proof: Removing "too many unnecessary local variables" by applying "replace temp with query" will remove some unnecessary local variables or some statements and introduce a new method which does not cause three other long method characteristics. Thus, n_2' is strictly less than n_2, while the new values of other variables are equal to or less than their pervious values.

Conjecture 3: After applying refactoring technique "introduce parameter object" or "preserve whole object", the number of long method characteristics "too many parameters" in the code of a long method decreases and no long method characteristic is added. It implies that $n_1' \leq n_1 \wedge n_2' \leq n_2 \wedge n_3' < n_3 \wedge n_4' \leq n_4$.

Proof: Removing "too many parameters" by applying "introduce parameter object" will remove some parameters and introduce a new class which does not cause three other long method characteristics. For applying "preserve whole object" will remove some parameters and use whole object. It does not cause three other long method characteristics. Thus, n_3' is strictly less than n_3, while the new values of other variables are equal to or less than their pervious values.

Conjecture 4: After applying refactoring technique "decompose conditional", the number of long method characteristics "too complex conditions" in the code of a long method decreases and no long method characteristic is added. It implies that $n_1' \leq n_1 \wedge n_2' \leq n_2 \wedge n_3' \leq n_3 \wedge n_4' < n_4$.

Proof: Removing "too complex conditions" by applying "decompose conditional" will remove some complex con-

ditions and introduce a new method which does not cause three other long method characteristics. Thus, n_4' is strictly less than n_4, while the new values of other variables are equal to or less than their pervious values.

Theorem 1: When applying the proposed approach, the number of long method characteristics in a long method strictly decreases from iteration to iteration: $n_{total}' < n_{total}$.

Proof: By conjecture 1 and definition 5:

$$n_1' < n_1 \wedge n_2' \leq n_2 \wedge n_3' \leq n_3 \wedge n_4' \leq n_4,$$
$$\text{hence } n_{total}' \leq n_{total}$$

By conjecture 2 and definition 5:

$$n_1' \leq n_1 \wedge n_2' < n_2 \wedge n_3' \leq n_3 \wedge n_4' \leq n_4,$$
$$\text{hence } n_{total}' \leq n_{total}$$

By conjecture 3 and definition 5:

$$n_1' \leq n_1 \wedge n_2' \leq n_2 \wedge n_3' < n_3 \wedge n_4' \leq n_4,$$
$$\text{hence } n_{total}' \leq n_{total}$$

By conjecture 4 and definition 5:

$$n_1' \leq n_1 \wedge n_2' \leq n_2 \wedge n_3' \leq n_3 \wedge n_4' < n_4,$$
$$\text{hence } n_{total}' \leq n_{total}$$

Theorem 2: An infinite loop cannot occur in the algorithm proposed by our approach.

Proof by contradiction: Suppose that an infinite loop occurs, it implies that a loop condition of our approach remains true infinitely. In other words, it means that after applying refactoring technique suggested by our approach from iteration to iteration, the number of long method characteristics will not decrease to zero. Using theorem 1, since the number of long method characteristics strictly decreases, it implies that the number of long method characteristics will decrease to zero at the end. Therefore by contradiction, we can conclude that an infinite loop cannot occur in the algorithm proposed by our approach.

3. Evaluation

Our research goal is to identify appropriate refactorings that remove long method bad smell, increase code analyzability and minimize the number of impacted statements. Therefore, to evaluate our approach, we implemented it into an Eclipse plugin, ran the plugin on a subject program and collected the results. The evaluation objectives are to assess: 1) the preservation of code functionality, 2) the bad smell removal rate, 3) the improvement on code analyzability. This research assesses the preservation of code functionality by running a test suite on the refactored code and determining whether all test cases pass. For removal rate, this research determines whether all existing long method characteristics are removed after applying refactorings and no new one is introduced. This research assesses the improvement on analyzability by comparing the analyzability level of the refactored code against the original one.

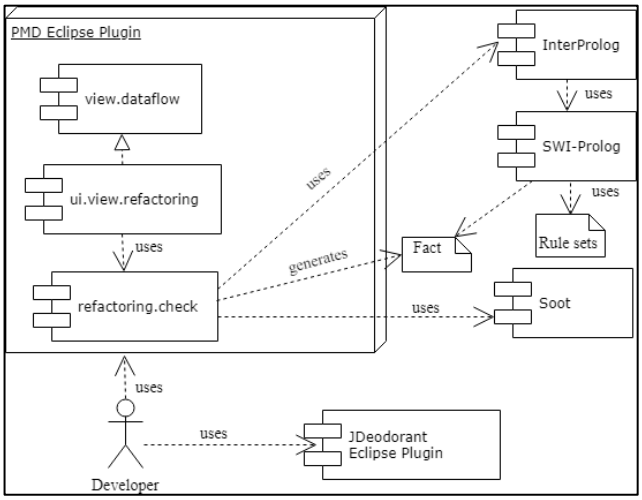

Fig. 5 LMR plugin architecture.

3.1 Tools

To automate refactoring opportunities selection and remove all long method bad smell, we need a tool which covers the suggestion for all five refactoring techniques: replace temp with query, introduce parameter object, preserve whole object, and decompose conditional, and extract method. As JDeodorant can identify opportunities for extract method, then we constructed an eclipse plugin that implemented our LMR approach and by itself can identify refactoring opportunities and their impacted statements for four remaining refactoring techniques. We call our eclipse plugin as LMR plugin. A developer can use our LMR plugin in cooperation with JDeodorant plugin to remove all long method bad smell.

The architecture of our LMR plugin is shown in Fig. 5. The LMR plugin was enhanced from PMD. We use PMD for creating class dependence graph which shows statement and variable dependences inside a method and for constructing facts about method which will be used with refactoring enabling conditionsfor identifying refactoring opportunities. A class dependence graph is created from control flow and data flow which are computed by PMD. Refactoring enabling conditions are defined as Prolog rule sets to identify opportunities for four refactoring techniques and suggest the most effective refactoring set.

Our plugin uses SWI-Prolog, which is an open source tool for running Prolog rules, and InterProlog, which is an open source API for communicating between Eclipse and SWI-Prolog. To identify a candidate temporary variable applicable for "replace temp with query" refactoring, we must guarantee that an assignment expression of that variable has no side effect. Our LMR uses Soot [22] to check for side effects. Soot is a static analysis tool that can check for side

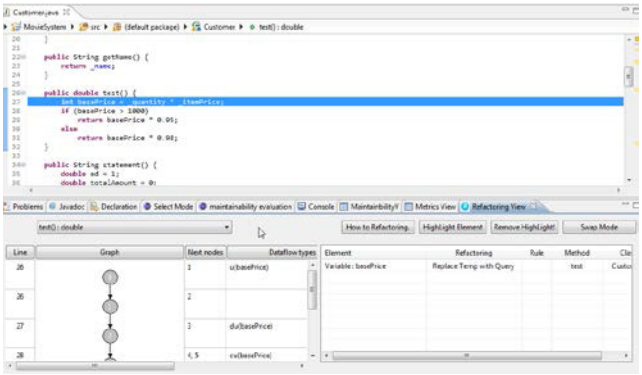

Fig. 6 LMR plugin identifies replace temp with query refactoring technique.

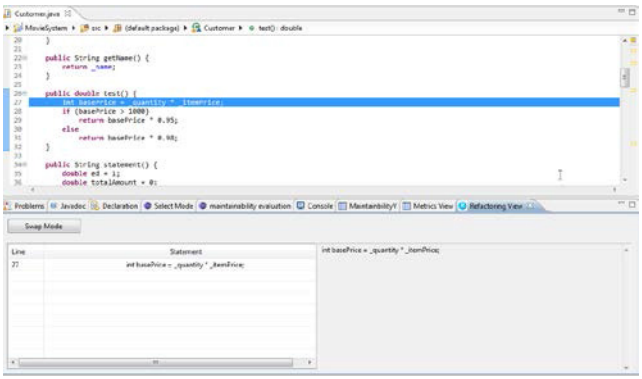

Fig. 7 LMR plugin identifies the statements impacted by replace temp with query refactoring techniques.

effects using point-to graph model and purity analysis [19]. Our plugin also identifies statements impacted by the replace temp with query refactoring by using the approach described in Ref. [20].

To facilitate refactoring application and long method bad smell removal, as shown in Fig. 6, the LMR plugin shows the list of refactoring opportunities in form of table and the control flow and data flow graphs of the code before refactoring. Our plugin highlights all impacted statements of refactoring opportunities as shown in Fig. 7, so that developer can use Eclipse to refactoring by applying the suggested refactoring technique to the highlighted statements.

We compare the capability of refactoring identification tools between LMR plugin and two other tools, as shown in Table 3. JDeodorant and IntelliJ IDEA identify refactoring opportunities only for extract method and replace temp with query, respectively, while the LMR plugin identifies refactoring opportunities for four refactorings except extract method. We compare the impacted statements by replace temp with query refactoring technique identified by our LMR plugin and IntelliJ IDEA. As a result, our tool can identify codes to form a query method for replacing a temporary variable in a long method covering more cases and more correctly than IntelliJ IDEA [23]. Our tool can identify temp variable inside loop statement and selection statement while IntelliJ IDEA cannot.

3.2 Subject

We selected our subject using three criteria: 1) being an open source software that is widely adopted and written in Java; 2) containing long method bad smells, and 3) availability of a test suite. This research selected JFreeChart, which is publicly used and is a popular subject for experiment. We chose its core package version 1.0.17, which has 5,665 lines of code and contains 20 classes and 552 methods.

After detecting long method characteristics in all classes and methods, we found long method bad smells in 13 classes. These classes altogether had 489 methods, 172 of which had long method characteristics. Next, we applied the proposed approach to the 172 methods to identify refactoring opportunities and their locations. LMR could identify opportunities for five refactoring techniques. We found that these methods could be divided into three types: 1) methods in which one refactoring technique was applicable at one location (T1); 2) methods in which one refactoring technique was applicable at a number of locations (T2), and 3) methods in which refactoring techniques could be applied at a number of locations (T3). Method examples of the three types are shown in Table 4.

From Table 4, method "ChartColor" of class "ChartColor" is classified type 1 since it had one refactoring "introduce parameter object" applicable to one location parameter group [r, g, b]. In Type 2, method "applyToXYAnnotation" of class "StandardChartTheme" had one refactoring "extract method" applicable to two locations local variables in blocks B1 and B2. In Type 3, method "createPieChart" of class "ChartFactory" has four refactoring candidates "introduce parameter object, replace temp with query, extract method, and decompose conditional" applicable at six locations.

In summary, there were 90, 25 and 57 methods in method types 1, 2 and 3, respectively. In Type 1, LMR found three long method characteristics: too many statements inside a method, too many unnecessary local variables, too many parameters, and too complex conditions. LMR then suggested three refactoring techniques: "replace temp with query", "introduce parameter object", and "extract method," covering 12 classes. The number of methods type 1 applicable for "replace temp with query", "introduce parameter object", "extract method," are 3, 73 and 11 respectively, while there is no method applicable for "preserve whole object and decompose conditional". LMR found two long method characteristics: too many unnecessary local variable and too many statements inside a method. LMR suggested three refactoring techniques: "replace temp with query" and "extract method" covering 8 classes. The number of methods type 2 applicable for "replace temp with query" and "extract method" are 9 and 16, respectively. In Type 3, LMR found four long method characteristics and suggested four refactoring techniques (4RT), three refactoring tech-

Table 4 Example of methods of three types.

Method Type	Method ID	Class	Method	Refactoring opportunities	Location
1	1	ChartColor	ChartColor (int r, int g, int b)	Introduce parameter object	- [r, g, b]
2	479	Standard	applyToXYASAnnotation (XYASAnnotation annotation)	Extract method	- xyta - B1 - xyta – B2
3	29	ChartFactory	createPieChart (String title, PieDataset dataset, PieDataset previousDataset, int percentDiffForMaxScale, boolean greenForIncrease, boolean legend, boolean tooltips, Locale locale, boolean subTitle, boolean showDifference)	Introduce parameter object	- [title, legend]
				Replace temp with query	- colorPerPercent - percentChange
				Extract method	- series - B1 - series - B3
				Decomposed conditional	- if(!greenForIncrease && newValue.doubleValue() > oldValue.doubleValue() || !greenForIncrease && newValue.doubleValue()< oldValue.doubleValue())

niques (3RT), and two refactoring techniques (2RT). The number of methods of 4RT, 3RT and 2RT are 2, 6 and 49, respectively.

3.3 The Experiment of Three Objectives

3.3.1 Evaluation of the Preservation of Code Functionality

Before performing the case study, the original code was used to rerun the test suite and the result was a pass. After a developer has applied the refactorings suggested by our approach for each method in the case study, the regression test suites will be re-run. If the refactored method passes this test suite, it implies that LMR approach preserves its behavior as specified by the test suite. Then, this refactored method in which there was no impact on code functionality was counted as a case. All methods of all types followed the same steps and the percentage of cases in which there was no impact on code functionality (NICF) formula was calculated using Eq. (6).

3.3.2 Evaluation of the Removal Rate of Long Method Characteristics

For each method in the case study, we detected long method characteristics of the refactored method and compared them with those of the original code. If the refactored method does not reintroduce long method characteristics previously found in the original code or introduce new long method characteristics, the refactored method was given the status true, and counted as a case in which all long method characteristics were removed. This implies that the removal rate criterion is satisfied. All methods of all types followed the same steps and the results were calculated using Eq. (7).

3.3.3 Evaluation of the Improvement on Analyzability

For each method in the case study, we compared the analyzability levels of the refactored code and original code. If the analyzability level of the refactored method was greater than or equal to that of the original method, the refactored method was counted as a case with no analyzability level degradation. This implies that the refactored method satisfied the impact on analyzability criterion. All methods of all types followed the same steps and the results were calculated using Eq. (8).

$$\textit{The percentage of cases with NICF for Type X} = \frac{\textit{Number of cases with NCIF for Type X}}{\textit{Total of cases in Type X}} \times 100 \quad (6)$$

$$\textit{The percentage of cases which remove all LMCs for Type X} = \frac{\textit{Number of cases which remove all LMCs for Type X}}{\textit{Total of cases in Type X}} \times 100 \quad (7)$$

$$\textit{The percentage of cases whose AL is not degraded for Type X} = \frac{\textit{Number of cases whose AL is not degraded for Type X}}{\textit{Total of cases in Type X}} \times 100 \quad (8)$$

where X is the type of method: type 1, type 2, type 3.

3.4 Result

LMR plugin executes three steps. In the first step, LMR plugin takes around one minute to transform JFreeChart code to an abstract syntax tree and four graphs for all classes and methods. In second step, LMR plugin takes around one second to identify refactoring opportunities after a developer selects a long method bad smell. In the last step, LMR plugin takes around one second to identify statements impacted by refactoring "replace temp with query" after a developer select that refactoring opportunity.

3.4.1 The Code Functionality and the Removal Rate

We found that all methods of three types passed all test

cases; so it implied that a hundred percent of methods after applying refactorings can preserve behavior. Furthermore, we found that all methods of three types removed long method bad smells; so it implied that is a hundred percent of long method bad smells can be removed for all three types.

3.4.2 The Improvement on Analyzability

In this case study, there is no method with analyzability level 1 or level 2 but the numbers of methods type 1, type 2, and type 3 with analyzability level 3 are 90, 25, 57 respectively. Analyzability level of these methods before and after applying refactorings remains the same. The percent of methods whose analyzability level remains unchanged for type T1, T2, and T3 is 100%. Therefore, we conclude that using our approach to remove long method characteristics does not alter the analyzability level of all methods.

3.5 Discussion

As shown in the results, for this subject program, our approach identifies and suggests that applying one of the three refactoring techniques: EM, IPO, and RTWQ is adequate to remove all the long method characteristics in methods of types 1 and 2. The results show that, in more complex cases such as in methods of type 3, our approach can remove all the long method characteristics by identifying and suggesting combinations of refactorings of all five techniques.

We found that, in all cases, our approach can preserve behavior and remove all long method bad smells. The analyzability level of all methods after applying refactorings suggested by our approach are unchanged. Therefore, our approach can completely remove long method bad smell and preserve their behavior without decreasing their analyzability. In type 3, our approach can suggested an effective set of refactoring techniques that remove several long method bad smell characteristics simultaneously.

3.6 Threats to Validity

The results in this experiment were subject to several threats. In this section, we discussed these limitations from two perspectives: internal validity and external validity.

3.6.1 Internal Validity

Our approach focused on analyzability improvement in code, but does not take into account other code qualities. We use code analyzability model from our previous work [17], which is based on three code metrics using logistic regression statistical method trained by the data from another with a Java open source project: JEdit. Since, JEdit and JFreeChart are similar for its size and architecture, we use the code analyzability model without adjusting cut point values.

If a developer uses other code analyzability models or other code maintainability models, she must validate models for its fitness. If these models are not suitable, she must adjust the models such as by changing constants or parameters. When the input analyzability model is changed, the metrics used in model may also change. A method should therefore be developed for predicting the values of these metrics. Because the predicted analyzability value is one of criteria for selecting an effective refactoring set.

In this experiment, we used the test suite of a subject program for evaluating code behavior preservation. JFreeChart has a test suite, which can be used to check whether main method can function correctly. In order to confirm that the methods after removing all long method bad smell characteristics, we suggest developer to check the quality of test suites related to all bad smell methods. Developer may use code coverage criteria i.e. statement or branch decision to assess quality of test suite.

3.6.2 External Validity

For our experiment, we selected a subject using three criteria: 1) it was an open source software that is widely adopted and written in Java; 2) it had long method bad smells, and 3) it had a test suite. Therefore, the proposed approach supports code in Java, but does not support generated code from automation tools.

For decreasing the number of parameter object, the conditions for identifying refactoring introduce parameter object should take into account the parameters of other classes. Moreover, our proposed approach excludes the architectural aspect.

4. Conclusion

This research proposed an effective approach for identifying and suggesting appropriate refactorings to completely remove the long method characteristics with minimal impact (in terms of the number of statements) and no reduction in code analyzability. The proposed approach, called the long method remover or LMR, focuses on code at the method level and considers each method in a class individually.

To identify refactoring opportunities and statements impacted by the refactoring techniques, LMR used refactoring enabling conditions and statements- impacted-by-the-refactoring algorithm for four refactoring techniques: "replace temp with query, introduce parameter object, preserve whole object, and decompose conditional" and used JDeodorant for refactoring technique "extract method". A refactoring enabling condition, written as a rule, is composed of arithmetic and logical expressions based on the code structure and code metrics from program analysis. This algorithm predicts impacted statements using program slicing. Moreover, LMR suggested a refactoring set using two criteria: the code analyzability level and the number of statements impacted by the refactoring techniques.

To evaluate our approach, we assessed three objectives: 1) the preservation of code functionality; 2) the removal rate, and 3) the improvement on code analyzability. We se-

lected as a subject the core package of JFreeChart, which has 20 classes and 552 methods. After investigating long method characteristics in all classes and methods, we found 172 methods with long method characteristics. These methods could be classified into three method types: 1) methods in which one refactoring technique was applicable at one location (T1); 2) methods in which one refactoring technique was applicable at more than one location (T2), and 3) methods in which refactoring techniques could be applied at more than one location (T3). The results showed that, for all of the methods, our approach can suggest refactoring sets that can complete remove long method bad smell, preserve behavior and maintain the code analyzability.

For future work, we plan to extend the evaluation as follows: 1) increase the number of cases of types 1 and 2 to cover all five refactoring techniques, and 2) evaluate the soundness of our approach by seeking judgments from experts. Lastly, we will implement a fully automatic tool for supporting the proposed approach.

References

[1] R. Pressman, Software Engineering: A Practitioner's Approach, 7th ed., McGraw-Hill, Inc., New York, NY, USA, 2010.
[2] B.S. Neto, V.T.d. Silva, M. Ribeiro, E. Costa, and C. Lucena, "Using jason to develop refactoring agents," Proc. Brazilian Conference on Intelligent Systems, pp.44–50, 2013.
[3] G. Singh and H. Singh, "Effect of software evolution on metrics and applicability of lehman's laws of software evolution," SIGSOFT Softw. Eng. Notes, vol.38, no.1, pp.1–7, 2013.
[4] M. Fowler and K. Beck, Refactoring: Improving the Design of Existing Code, Component software series, Addison-Wesley, 1999.
[5] T. Mens and T. Tourw´e, "A survey of software refactoring," IEEE Trans. Softw. Eng., vol.30, no.2, pp.126–139, 2004.
[6] T. Pienlert and P. Muenchaisri, "Bad-smell detection using object-oriented software metrics," Proc. Computer Science, Software Engineering, Information Technology, e-Business, and Applications (CSITeA04), 2004.
[7] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," IEEE Trans. Softw. Eng., vol.35, no.3, pp.347–367, 2009.
[8] G. Bavota, A. De Lucia, and R. Oliveto, "Identifying extract class refactoring opportunities using structural and semantic cohesion measures," Journal of System and Software, vol.84, no.3, pp.397–414, 2011.
[9] G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, and A.D. Lucia, "Methodbook: Recommending move method refactorings via relational topic models," IEEE Trans. Softw. Eng., vol.40, no.7, pp.671–694, 2014.
[10] K. Maruyama, "Automated method-extraction refactoring by using block-based slicing," Proc. 2001 Symposium on Software Reusability: Putting Software Reuse in Context, SSR '01, pp.31–40, 2001.
[11] L. Yang, H. Liu, and Z. Niu, "Identifying fragments to be extracted from long methods," Proc. 2009 16th Asia-Pacific Software Engineering Conference, APSEC '09, IEEE Computer Society, pp.43–49, 2009.
[12] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "Jdeodorant: Identification and removal of type-checking bad smells," Proc. 12th European Conference on Software Maintenance and Reengineering, pp.329–331, 2008.
[13] N. Tsantalis and A. Chatzigeorgiou, "Identification of extract method refactoring opportunities for the decomposition of methods," Journal of System and Software, vol.84, no.10, pp.1757–1782, 2011.
[14] Y. Kataoka, D. Notkin, M.D. Ernst, and W.G. Griswold, "Automated support for program refactoring using invariants," Proc. IEEE International Conference on Software Maintenance (ICSM'01), ICSM'01, IEEE Computer Society, pp.736–, 2001.
[15] P. Meananeatra, S. Rongviriyapanish, and T. Apiwattanapong, "Using software metrics to select refactoring for long method bad smell," Proc. 8th Electrical Engineering/ Electronics, Computer, Telecommunications and Information Technology (ECTI) Association of Thailand, pp.492–495, 2011.
[16] P. Meananeatra, S. Rongviriyapanish, and T. Apiwattanapong, "Identifying refactoring through formal model based on data flow graph," Proc. 2011 Malaysian Conference in Software Engineering, pp.113–118, 2011.
[17] P. Meananeatra, E. Rattanaleadnusorn, S. Rongviriyapanish, T. Kitcharoensup, T. Wisuttikul, and B. Charoendouysil, "Modeling code analyzability at method level in j2ee applications," Proc. 20th Asia-Pacific Software Engineering Conference (APSEC'13), IEEE Computer Society, pp.61–66, 2013.
[18] J. Ferrante, K.J. Ottenstein, and J.D. Warren, "The program dependence graph and its use in optimization," ACM Transactions on Program Language System, vol.9, no.3, pp.319–349, 1987.
[19] A. Rountev, "Precise identification of side-effect-free methods in java," Proc. 20th IEEE International Conference on Software Maintenance, pp.82–91, 2004.
[20] S. Rongviriyapanish, N. Karunlanchakorn, and P. Meananeatra, "Automatic code locations identification for replacing temporary variable with query method," Proc. 12th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON), pp.1–6, 2015.
[21] M. Sipser, Introduction to the Theory of Computation, 1st ed, International Thomson Publishing, 1996.
[22] P. Lam, E. Bodden, L. Hendren, and T.U. Darmstadt, "The soot framework for java program analysis: a retrospective," Proc. Cetus Users and Compiler Infrastrueture Workshop, 2011.
[23] S. Rongviriyapanish, N. Karunlanchakorn, and P. Meananeatra, "Automatic code locations identification for replacing temporary variable with query method," ECTI Transactions on Computer and Information Technology (ECTI-CIT), vol.10, no.1, pp.88–96, 2015.

Panita Meananeatra is a Ph.D. Student at department of Computer Science, faculty of Science and Technology, Thammasat University, Thailand. She has been an assistant researcher, working 10 years, at national electronics and computer technology center (NECTEC). Her research interests Refactoring, Maintainability, Software Quality, Software Testing, Software Process, and Software Engineering.

Songsakdi Rongviriyapanish received the Ph.D. in Informatics (specialized in Software Architecture) from University Nancy 2, from Nancy in France. Since February 2001, he has been assistant professor and worked at the department of computer science, faculty of Science and Technology at Thammasat University and gives a lecture on Software Engineering, Object-oriented programming, Analysis and design, Component-based Development courses. His research interests are about software quality and metrics, software design and formal methods.

Taweesup Apiwattanapong received his Ph.D. in Computer Science at the College of Computing, Georgia Institute of Technology. He worked at the Information Technology Center, Siam Cement Public Co. Ltd. (1998-2001). He was a researcher at National Electronics and Computer Technology Center (NECTEC). Currently, he works at National Science and Technology Development Agency (NSTDA). His research interests are in software evolution.