Automatic code locations identification for replacing temporary variable with query method

Songsakdi Rongviriyapanish[1],
Nopphawit Karunlanchakorn[2], and **Panita Meananeatra**[3], Non-members

ABSTRACT

Automatic application of refactoring techniques can help developer save effort in removing bad smells from their code which improves software maintainability. To automatically remove long method bad smell, which is one of the most serious bad smells, we need an automatic application of six refactoring techniques. However, only one refactoring technique "Extract Method" can be automated. In this research, we propose an algorithm to identify code locations which will be extracted for creating a query method. We create a program dependence graph from the code with a long method bad smell and use a technique of program slicing to identify the code of a query method to replace a temporary variable. This is the most important step towards an automatic application of refactoring technique "replace temp with query". We conducted an experiment to compare the correctness of refactored code between using our algorithm and using IntelliJ IDEA that is a java integrated development environment (IDE) with a feature of automatic refactoring. The result confirmed that our proposed algorithm can identify codes to form a query method for replacing a temporary variable in a long method with higher accuracy than IntelliJ IDEA almost three times

Keywords: Refactoring Application, Program Dependence Graph, Bad Smell, Software Maintenance

1. INTRODUCTION

During developing an object-oriented program, developers often find evidences of bad smells in the code. However, they typically do not have time or are afraid to change the code to eliminate bad smells since changing code might affect the accuracy of other code sections. Bad smells remaining in the code may cause the problem in application development such as decreased software maintainability.

Samples of bad smell often appeared are code duplication, large class and long method. In particular, long method bad smell, which is a method with many statements, too many temporal variables or a long list of parameters, often causes other subsequent bad smells and has a significant impact on software maintainability [1].

Martin Fowler [2] proposed six refactoring techniques: extract method, replace temp with query, introduce parameter object, preserve whole object, decompose conditional and replace method with method object for removing long method bad smell. So far, only the technique of extract method can be automatically applied for removing long method bad smell [3]. No algorithm has been proposed to automatically remove long method with the other five techniques. Consequently, at present we cannot automatically eliminate long method bad smell with existing research work. In this paper, we propose an algorithm for automatic replacement of a temporary variable with a query method. This makes our approach to automatic removing long method bad smell unique comparing to the existing methods.

The technique of replace temp with query is to eliminate unnecessary defined temporary variable from the code by replacing it with a query method. The five steps in the process of replacing temporary variables with query methods are as follows:

1. Identifying a temporary variable suitable to be replaced with query method.

2. Identifying locations of codes which will be extracted for creating a query method.

3. Creating a query method.

4. Locating temporary variable to be replaced by the query method.

5. Replacing temporary variable with the query method.

In this research, we propose an algorithm to perform step 2. We use our previous research works [4, 5] to identify suitable temporary variables to be replaced with query method. Section 2 explains the background concepts required to understand our algorithm such as the technique of replace temp with

Manuscript received on August 15, 2015 ; revised on March 27, 2016.

Final manuscript received on April 5, 2016.

[1,2]The authors are with Computer Science Department, Thammasat University, Phathumthani, Thailand, E-mail: rongviri@cs.tu.ac.th and 5409520029@student.cs.tu.ac.th

[3]The author is with Standard and Testing Development Laboratory, National Electronics and Computer Technology Center,Phathumthani, Thailand., E-mail:panita.meananeatra@nectec.or.th

query, program dependence graph and program slicing technique used in our algorithm. Section 3 presents some research works related to our research. Section 4 presents our algorithm to identify codes which will be extracted and formed as query method. Section 5 shows the experiment result of applying our algorithm to different cases. We compare the correctness of refactoring using our algorithm with the one obtained using IntelliJ IDEA - a Java IDE from JetBrains1 which offers a possibility to apply replace temp with query technique.

2. BACKGROUND

This section explains the detail of refactoring technique of replace temp with query and reviews program dependence graph (PDG) and program slicing technique which will be used in our algorithm.

2.1 Replace Temp with Query

From the introduction section, the refactoring technique which helps to remove unnecessary temporary variables in a long method is replace temp with query. This refactoring technique is selected when a temporary variable is assigned once in an expression (except 'if' block and 'switch' block) and be used in computation or predicate. To apply replace temp with query, there are four steps: 1) identify temporary variable, 2) identify locations of code which will be extracted for creating a query method, 3) identify occurrences of temporary variable to be replaced with query method and 4) refactor code by applying this refactoring.

2.2 Program Dependence Graph

Program dependence graph or PDG was proposed by Ferrante et al. for representation of control and data flow between operations of a procedure [6]. PDG is a graph whose node represents a statement or predicate expression and has two types of edge: control dependence edge and data dependence edge. Control dependence edge represents a statement sequence which depends on a control condition. For example, there is one control dependence edge linking a control statement, such as 'if-else', 'while', 'for', with each statement inside the block of that control statement. Data dependence edge represents flow of data dependence between statements.

In this paper, our method uses PDG to analyze program structure.

2.3 Program Slicing

A slice of program, developed by Weiser [7], is a program point P and a set of program variable V of all statements and predicates in the program that may affect the values in V at P. Weiser used the CFG and DFG analysis for slicing. [8, 9] proposed techniques using PDG for slicing. There are three techniques of program slicing: backward slicing, forward slicing and chop slicing. First, the backward slice at program point p is the program subset that may affect p and compute backward reachability in the PDG from node p. Second, the forward slice at program point p is the program subset that may be affected by p and computed forward reachability in the PDG from node p. Lastly, the chop between program point p and q is the program subset that may be affected by p and that may affect q and computed between point p and q to identify all paths between p and q.

```
public double getPrice(){
    double basePrice = _quantity*_itemPrice;
        if(basePrice>1000)
            return basePrice*0.95;
        else
            return basePrice*0.98;
}
```

Fig.1: *Sample Code for Demonstrating Forward Slicing.*

For a selected statement, forward slicing is to analyze on which statements it has an impact. While backward slicing is to analyze from which statements a selected statement is affected. Chop slicing makes a union of the output from forward and backward slicing.

To understand how to do forward slicing, we use a sample code in Fig. 1 and demonstrate forward slicing of this statement: "double basePrice = _quantity*_itemPrice". The output set of the statements from forward slicing contains four statements:

{double basePrice =_quantity*_itemPrice;,
if (basePrice>1000), return basePrice*0.95;,
return basePrice*0.98; }

To understand how to do backward slicing, we use an example code and perform backward slicing on this statement: "return basePrice*0.98 ;". The output set of the statements from backward slicing is

{double basePrice = _quantity*_itemPrice;,
if (basePrice>1000), return basePrice*0.98;}

since the value of the selected statement "basePrice*0.98" depends on the statements in the output set.

To do chop slicing, we select two statements, one for doing forward slicing and another one for backward slicing. For example, the output set of chop slicing of forward slicing "double basePrice = _quantity*_itemPrice;" and backward slicing "return basePrice*0.98;" is a union set of output sets of forward slicing and backward slicing that is

{double basePrice=_quantity*_itemPrice;,
If (basePrice>1000), return basePrice*0.95;,
return basePrice*0.98;}

In this paper, we use backward slicing and forward slicing techniques to identify code locations which will be extracted to form a new query method.

3. RELATED WORKS

Steps of refactoring application using replace temp with query technique are similar to the steps of extract method technique but focusing on eliminating unnecessary temporary variables.

Maruyama proposes a mechanism that separates new methods from existing methods of object-oriented programs by using block-based slicing [10]. This mechanism indicates only variable of interest in the code. However, this mechanism does not consider calling and called methods. A side effect may occur in a new method which, consequently, violates condition of behaviour preservation.

Yang, Lui and Niu proposed an approach to recommend code fragments to be extracted from the body of a long method [11]. Their approach does not consider a side effect of a code fragment to be extracted. In some cases, a statement with a side effect should be extracted and sometimes it should not.

Tsantails and Chatzigeorgiou proposed an approach that automatically identified extract method refactoring opportunities [3, 12]. This approach uses the union of static slice for extracting the complete computation of a given variable declared inside a method. Their approach also proposes a set of rules that preserves the code behaviour after slice extraction and prevents the code duplication. Based on their approach, they implemented a tool named JDeodorant which claimed to be able to remove long method bad smell but they can automatically apply only one refactoring technique named extract method. This is not enough to completely remove long method bad smells which required other refactoring techniques such as replace temporary variable with a query method to be applied.

In this paper, our method uses PDG for analyzing program structure and program slicing for identifying code location.

4. METHODOLOGY

Given a temporary variable suitable to be replaced with query in a long method, our proposed algorithm will identify the statements to be extracted and formed as a query method by using program slicing technique. We analyze the relationship between the statements by using three program slicing techniques in combination such as forward slicing, backward slicing, and chop slicing.

4.1 Steps of the Proposed Methodology

We use a sample code with a long method bad smell as an input to our algorithm. A temporary variable to be replaced with query method is identified and selected. We obtain a result set which contains all statements forming a query method. Our algorithm for identifying locations of code which will be extracted to form a query method has four steps.

1. Create program dependence graph from code
   This purpose of this step is to create a program dependence graph (PDG) from code which contains our temporary variable to be replaced with query method.
2. Add assignment statements which use the temporary variable to be replaced to the result set
   This purpose of this step is to add an assignment statement of the temporary variable to be replaced to the result set. In addition, we perform forward traversal in the data flow graph starting from the node of the assignment statement. During forward traversal, we add every visited statement node to the result set except the statement satisfying one of these conditions:
   a) statement which uses that temporary variable in a computation, called Computation Use statement (CU statement) but does not use it in a definition statement, called Definition Use statement (DU statement).
   b) statement which returns a value of that temporary variable.
   c) statement which uses that temporary variable in a boolean expression, called Predicate Use statement (PU statement).
3. Include preceding statements of the assignment statements obtained in step 2 to the result set
   Since we cannot include only assignment statements in the result set, we must also include their preceding statements. For each statement in the result set, we perform backward traversal in the control flow graph. During backward traversal, we add every visited statement during backward traversal to the result set.
4. Include statements which initialize variables used by the statements in the result set.

In the last step, we must include all statements that initialize the variables used by the statements in the result set. Therefore, we initialize a list with all statements in the result set which contains at least one temporary variable but no statement in the result set initializes its value. For each statement node in the newly created list, we perform backward traversal from that node by following data dependence edge and add every visited statement node to that list. Finally, we add all elements in the list to the result set and repeat step 4 until no statement node in the result set satisfies the condition for being added to the list.

4.2 A Case Study

To illustrate our algorithm, we will show how to apply it to identify the codes of query method which will be used for replacing a temporary variable "renterPoints" in a long method "statement( )". This

example is based on an example in the book of Martin Fowler [2]. The temporary variable "renterPoints" is shown as a yellow rectangle in Fig. 2. In step 1, we create a program dependence graph of long method "statement ( )" as shown in Fig 2. At the right hand side of the figure marked by CDE, we show the control dependence edges and control flow graph of method "statement( )" while at the left hand side marked by DDE, we show the data dependence edges and data flow graph of the method.

Step 2, we find all assignment statements of "renterPoints" by forward slicing following data dependence edge. At the beginning, we add the assignment statement of the temporary variable "renterPoints" to the result set. Therefore, the result set is {int renterPoints =0;}. Due to limited space, we indicate the line numbers of code as elements of the result set instead of writing the complete statements. As the line number of the assignment statement is 2 so the result set is now {2}. We add all visited statements obtained by performing forward traversal from statement number 2 and following data dependence edges. As there are two data dependence edges linking: node 2 to nodes 9 and 10 shown as green edges in Fig. 3, therefore we add nodes 9 and 10 to the result set and it becomes {2, 9, 10}.

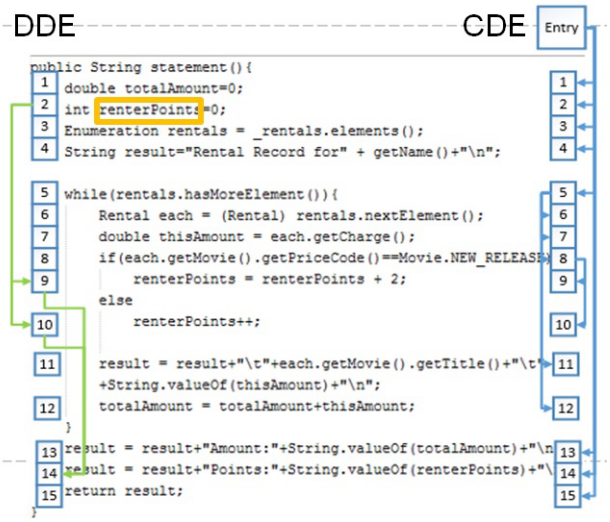

Fig.2: *Program Dependence Graph of the long method "statement" .*

No condition for removing statements after forward traversal which is mentioned in step 2 is satisfied, so the result set remains unchanged. Next, we repeat step 2 with nodes 9 and 10 shown as green edge in Fig. 4. Only a new node 14 is likely to be added but as it satisfies the condition a) of step 2 since it is a DU statement for a variable "result" and a CU statement for the variable "renterPoints", therefore we do not add it to the result set. The result set still remains {2, 9, 10}.

Step 3, we include not only assignments in the result set but also their preceding statements. For example, if we include "if statement" we should also include "while statement" since "if statement" is under the scope of "while statement". Therefore, we perform backward traversal from the statement node 2 by following control dependence edges. There is a control dependence edge from node 2 back to the Entry node which will be ignored. The result set still remains {2, 9, 10}. We continue the step with the statement node 9 which has a control dependence edge back to statement node 8 (shown as black edge in Fig. 5) so the result set becomes {2, 9, 10, 8}. We repeat with all elements and we get node 5 (shown as black edge in Fig. 5) added in the result set. In brief, we obtain {2, 9, 10, 8, 5} as the result set.

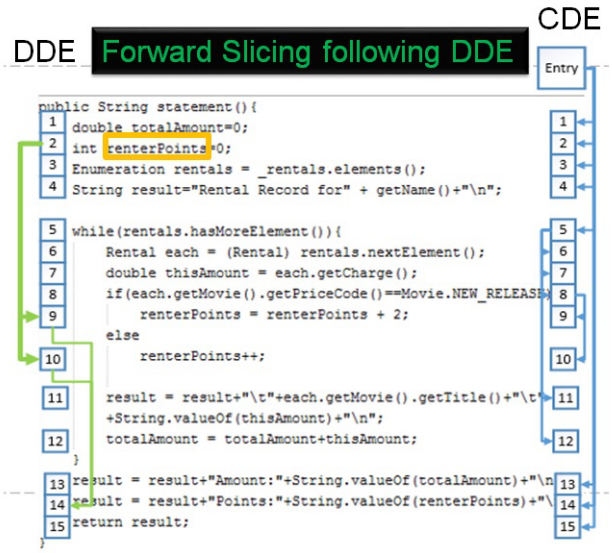

Fig.3: *Forward Slicing following DDE of node 2.*

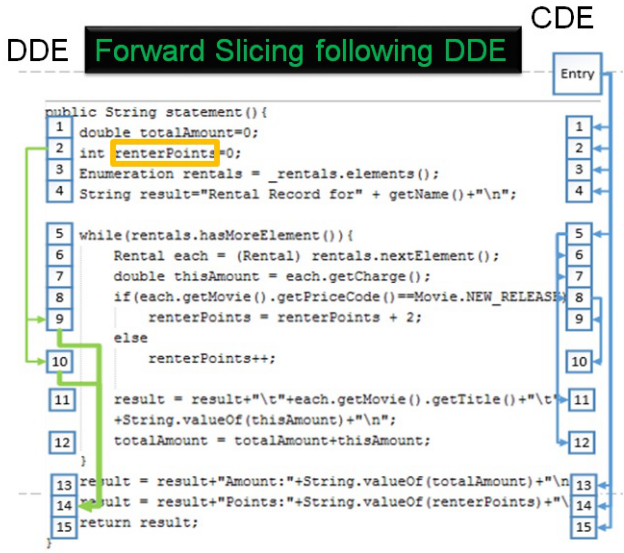

Fig.4: *Forward Slicing following DDE of nodes 9, 10.*

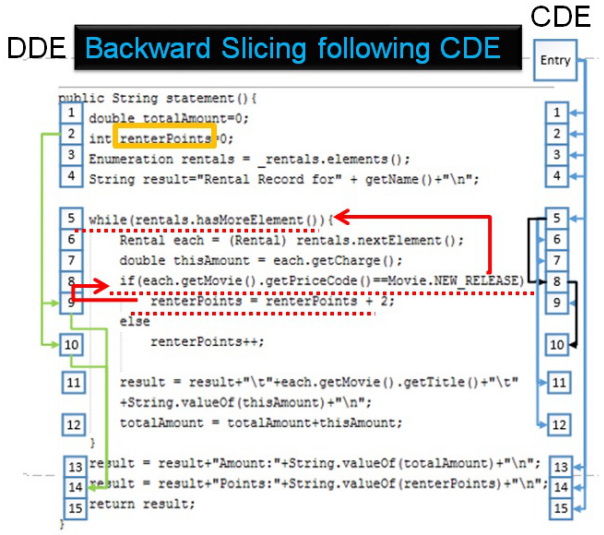

Fig.5: *Backward Slicing following DDE of nodes 10, 8.*

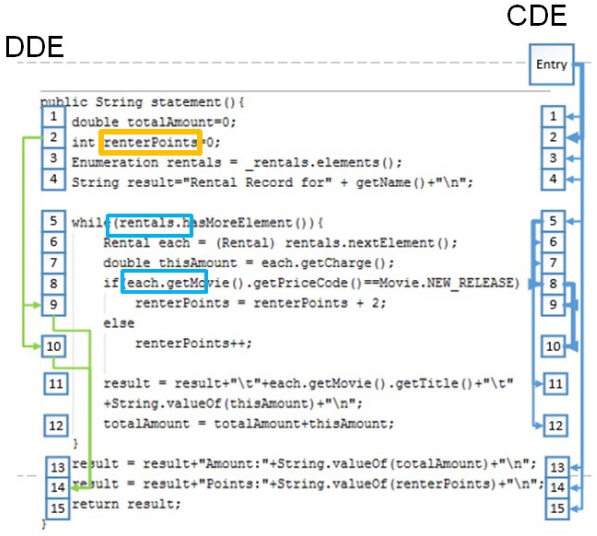

Fig.6: *The Temporary Variables "rentals" and "each".*

Step 4, we must include all statements that initialize the variables used by the statements in the result set. In our example, we consider all statements in the result set {2, 9, 10, 8, 5}, we create a new list with statement nodes 5 and 8 since the statement nodes 5 and 8 contain the temporary variables "rentals" and "each" respectively shown as blue rectangles in Fig. 6 and no statement in the result set initializes value of these variables. We start performing backward traversal from statement node 5. There is a data dependence edge from statement node 5 back to statement node 3 shown as yellow edge in Fig.7, so we add statement node 3 to the list: {5, 8, 3}. We continue the step with statement node 8 and add statement node 6 shown as purple edge in Fig. 8 to the list: {5, 8, 3, 6} We add all elements of the second result set to the result set and obtain {2, 3, 5, 6, 8, 9, 10} as the result set. We repeat the step 4 but no statement satisfies the condition required for step 4, therefore our algorithm ends and returns the result as {2, 3, 5, 6, 8, 9, 10}.

We extract statements with the number indicated by the result set to form body of the query method as shown in Fig. 9. We create a new query method named "getRenterPoint( )" with the statements identified by our algorithm and we replace the temporary variable "renterPoints" with this query method. In the body of this query method, we generate the number of statement node appeared in PDG to show which statements we extracted to from the body of method.

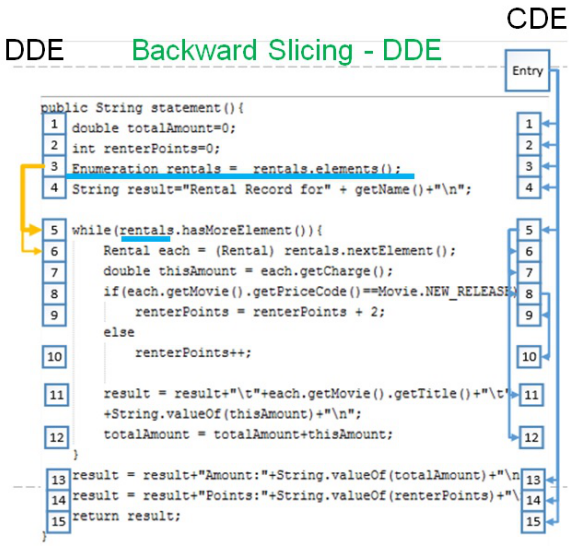

Fig.7: *Backward Slicing following DDE of node 5.*

5. EXPERIMENTATION

We perform an experiment to evaluate if our algorithm can correctly identify temporary variable to be replaced with query method in different cases and can identify the correct set of codes to be extracted and formed as a query method. We use 8 java code samples each of which contains a temporary variable to be replaced with query method. These samples are from different sources such as refactoring book [2], research papers [4, 5, 10, 13] and web sites that teach refactoring techniques [14]. Each sample has a code before and after refactoring was applied which allows us to evaluate if our algorithm can correctly identify the codes to be extracted and formed as query method. The java code samples can be classified into eight cases as shown in Table 1: case 1 where in a long method there exists a temporary variable declared as a substitution of a read-only expression,

case 2 where in a long method there exists a temporary variable declared as a substitution of a no side

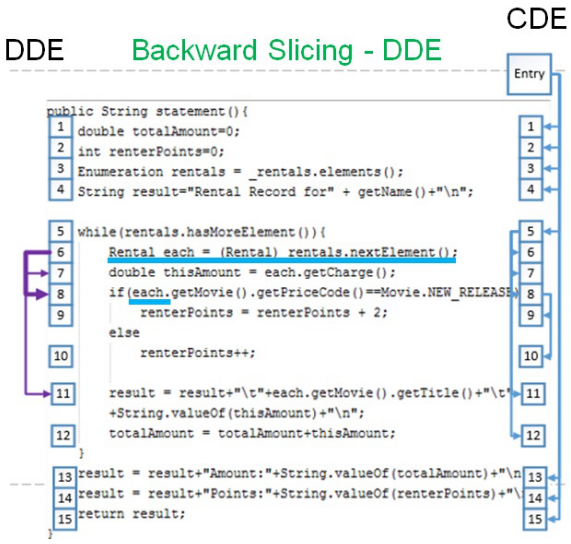

Fig.8: *Backward Slicing following DDE of node 8.*

```
public String statement(){
    double totalAmount=0;
    Enumeration rentals = _rentals.elements();
    String result="Rental Record for" +getName()+"\n";
    while(rentals.hasMoreElement()){
        Rental each = (Rental) rentals.nextElement();
        double thisAmount = each.getCharge();
        result = result+"\t"+each.getMovie().getTitle()+"\t"
        +String.valueOf(thisAmount)+"\n";
        totalAmount = totalAmount+thisAmount;
    }
    result = result+"Amount:"+String.valueOf(totalAmount)+"\n";
    result = result+"Points:"+String.valueOf(renterPoints)+"\n";
    return result;                          replaced by getRenterPoint()
}
```

```
public int getRenterPoint(){
    int renterPoints=0;
    Enumeration rentals = _rentals.elements();
    while(rentals.hasMoreElement()){
        Rental each = (Rental) rentals.nextElement();
        if(each.getMovie().getPriceCode()==Movie.NEW_RELEASE)
            renterPoints = renterPoints + 2;
        else
            renterPoints++;
    }
    return renterPoints;
}
```

Fig.9: *Code Extracted by Our Algorithm to Form a Query Method.*

effect method invocation,

case 3 where in a long method there exists a temporary variable declared as a constant whose value is defined by selection statement such as 'if-else',

case 4 where in a long method there exists a temporary variable used in a switch statement,

case 5 where in a long method there exists a temporary variable used as an accumulative variable in a for loop and used outside for loop.

case 6 where in a long method there exists a temporary variable used as an accumulative variable in a 'while' loop and used outside 'while' loop,

case 7 where in a long method there exists a temporary variable used for a calculation in a while loop with an 'if-statement' nested inside the 'while' loop and used outside 'while' loop,

case 8 where in a long method there exists a temporary variable declared in term of method invocations.

We compare performance of our algorithm with IntelliJ IDEA that is a Java integrated development environment (IDE) with a feature of automatic refactoring. IntelliJ IDEA claimed to be able to automatically identify and refactor code with "replace temp with query" technique. For each case, we apply our algorithm to determine if our algorithm can identify temporary variable to be replaced with a query method as well as a set of codes to be extracted and formed as a query method. We perform the same process using IntelliJ IDEA.

Table 1: *Reference of cases used in experiment .*

Case No.	Reference	Fig.
1	[2] temporary variable: basePrice	1
2	[4,5] temporary variable: partnershipId	10
3	[2] temporary variable: discountFactor	1
4	[2] temporary variable: thisAmount	11
5	[13] temporary variable: totalAge	12
6	[2] temporary variables: frequentRenterPoints, totalAmount	13
7	[10] temporary variables: renterPoints, totalAmount	14
8	[14] temporary variable: aString	15

```
public void processIncomingMessage(EbmsRequest request,EbmsResponse response) throws
MessageServiceHandlerException {
...
String partnershipId = findPartnershipId(ebxmlRequestMessage);

    if (partnershipId == null && ! (messageType.equalsIgnoreCase(MessageClassifier.ACTION_PING) ||
(messageType.equalsIgnoreCase(MessageClassifier.ACTION_PONG) && this.isPartner(ebxmlRequestMessage)))) {

        // unauthorized user
        // generate error msg (sync reply)
        EbmsProcessor.core.log
            .error("Unauthorized message, no partnership is found");
        ebxmlResponseMessage = processUnauthorizedMessage(
            ebxmlRequestMessage, false);
        response.setMessage(ebxmlResponseMessage);
        return;
    }
...
}
```

Fig.10: *Sample Code of Case 2: Existing Method Invocation.*

5.1 Correctness of Identifying Temporary Variable to be Replaced

We evaluate the performance of our algorithm and IntelliJ IDEA in identifying a temporary variable to be replaced with a query method in terms of whether

```
public String Statement() {
  double totalAmount = 0;
  int frequentRenterPoints = 0;
  ...
  while ( rentals.MoveNext()) {
     double thisAmount = 0;
     Rental each = (Rental) rentals.Current;
     switch(each.Movie.PriceCode) {
         case PriceCodes.Regular:
               thisAmount += 2;
               if (each.DayRented > 2) {   thisAmount += (each.DayRented - 2) * 1.5; }
               break;
          case PriceCodes.newRelease:
                thisAmount += each.DayRented * 3;
               break;
          case PriceCodes.Childrens:
                thisAmount += 1.5;
               if (each.DayRented > 3) { thisAmount += (each.DayRented - 3) * 1.5; }
               break;
     }
     frequentRenterPoints ++;
     ...
  }
  ...
  return result;
}
```

Fig.11: *Sample Code of Case 4: Switch Statement.*

```
public String computeReport(){
     Stirng result = "";
     double totalAge = 0;
     double totalSalary = 0;
     for(int i =0; i< people.length; i++) {
         totalAge += people[i].age;
         totalSalary += people[i].salary;
     }
     double avgAge = (totalAge / people.length);
     result += "average age:" + avgAge;
     result += "total salary" + totalSalary;
}
```

Fig.12: *Sample Code of Case 5: For Loop.*

it can identify codes to be extracted and formed as a query method. The correctness of identifying temporary variable to be replaced obtained by running our proposed algorithm and IntelliJ IDEA with eight cases are shown in column "Correctness of identifying a temporary variable to be replaced" in Table 2.

For example, case 7 of the experiment is an example illustrated in section IV. Set of the extracted statements for this case is {2, 3, 5, 6, 8, 9, 10}. Our proposed algorithm can correctly identify codes to be extracted shown as a symbol ✓ in the column "Correctness of identifying a temporary variable to be replaced" in Table 2 while IntelliJ IDEA cannot identify them but shows an error message that is "cannot per-

```
public String statement () {
   double  totalAmount  = 0;
   int frequentRenterPoints = 0;
   Enumeration rentals = _rentals.elements();
  String result = "Rental Record for " + getName() + "\n";
       while (rentals.hasMoreElement()) {
            Rental each = (Rental) rentals.nextElement();
            frequentRenterPoints += each.getFrequentRenterPoints ();
             result = result +"\t" + each.getMovie().getTitle()+ "\t"
                       +String.valueOf(each.getCharge()) + "\n";
            totalAmount = totalAmount + each.getCharge();
       }
      result = result +"Amount owed is " + String.valueOf(totalAmount) + "\n";
      result = result +"You earned " + String.valueOf(frequentRenterPoints )
               + "frequentRenterPoints";
      return result;
}
```

Fig.13: *Sample Code of Case 6: While Loop.*

```
public String statement () {
   double  totalAmount  = 0;
   int renterPoints = 0;
   Enumeration rentals = _rentals.elements();
  String result = "Rental Record for " + getName() + "\n";
       while (rentals.hasMoreElement()) {
            Rental each = (Rental) rentals.nextElement();
            double thisAmount = each.getCharge();
             if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE)
                  renterPoints = renterPoints +2;
              else
                 renterPoints ++;
             result = result +"\t" + each.getMovie().getTitle()
                       + "\t" +String.valueOf(thisAmount) + "\n";
            totalAmount = totalAmount + thisAmount;
       }
      result = result +"Amount :" + String.valueOf(totalAmount) + "\n";
     result = result +"Points : " + String.valueOf(renterPoints);
      return result;
}
```

Fig.14: *Sample Code of Case 7: While Loop and If Statement Nested in While Loop.*

```
public void method() {
    String str ="str";
    String aString = returnString().concat(str);
    System.out.println(aString);
}
```

Fig.15: *Sample Code of Case 8: Assignment Statement with Method Invocations.*

form refactoring, variable "totalAmount" is accessed for writing". Therefore, ✘[b] is marked in the column "Correctness of identifying a temporary variable to be replaced" in Table 2.

The result of the experiment shows that our proposed algorithm can identify temporary variable to be replaced for all cases while IntelliJ IDEA can identify correctly only three cases (case 1, 2 and 8). In some cases (cases 4, 5, 6, 7) where it cannot identify the temporary variable, Intellij IDEA shows error message but in some cases (case 3) it does not show any error message such as in the case of selection statement.

5.2 Correctness of Codes to be Extracted and Formed as a Query Method

For each case, we compare set of code locations identified by our algorithm with code after refactoring provided by the source where we collect the code sample. If the set of codes corresponds completely to the codes after refactoring, we consider the correctness of algorithm is 100% otherwise we consider it as incorrect. For 8 cases, we compare the codes to be extracted which are suggested by our algorithm with the result of refactoring using IntelliJ IDEA. The results are shown in the column "Correctness of Codes to be extracted and formed as a query method" in Table 2. Our algorithm can correctly identify codes to form a query method in all cases while IntelliJ IDEA can do it correctly only in three cases (case1, 2 and 8).

Table 2: *The Comparison of Correctness Percentage of Proposed Algorithm and IntelliJ IDEA.*

Case	Correctness of identifying a temporary variable to be replaced		Correctness of Codes to be extracted and formed as a query method	
	Proposed Algorithm	IntelliJ IDEA	Proposed Algorithm	IntelliJ IDEA
1	✓	✓	100%	100%
2	✓	✓	100%	100%
3	✓	✘[a]	100%	Incorrect
4	✓	✘[b]	100%	Incorrect
5	✓	✘[b]	100%	Incorrect
6	✓	✘[b]	100%	Incorrect
7	✓	✘[b]	100%	Incorrect
8	✓	✓	100%	100%

Note:
✓−− can identify temporary variable to be replaced.
✘−− cannot identify temporary variable to be replaced.
[a]−− cannot identify temporary variable and does not show error message.
[b]- cannot identify temporary variable but shows error message.

In summary, based on the experiment result, we can conclude that our proposed algorithm can identify codes of query method to replace temporary variable more accurately than IntelliJ IDEA almost three times (the ratio between the correctness percentage of our proposed algorithm and correctness percentage of IntelliJ IDEA - 8:3 = 2.67 ≈ 3). In some cases where there exist temporary variables inside a statement such as loop statement or selection statement, our approach can identify temporary variables to be replaced and codes of query method to replace temporary variable more accurately than using IntelliJ IDEA.

6. CONCLUSION

This paper proposes an algorithm to identify locations of codes for creating query method and focuses on the first step required for automatic application of refactoring technique "replace temp with query".

Our proposed algorithm uses program dependence graph and two slicing techniques: forward slicing by following data dependence edge and backward slicing by following control dependence edge.

We conducted an experiment with eight cases of java code to compare the correctness percentage between our algorithm and IntelliJ IDEA. The cases cover all statement types such as selection with 'if-else' or 'switch', loop with 'for' or 'while'. The experiment result shows that our proposed algorithm can identify codes to form a query method for replacing a temporary variable in a long method more correctly than IntelliJ IDEA almost three times. While IntelliJ IDEA cannot refactor the code with the refactoring technique "replace temp with query" in cases of code with selection or loop statements.

For our future work, we plan to develop an algorithm for the remaining steps so that we can completely automate the application of refactoring "replace temp with query method". We also plan to evaluate the performance of our approach in terms of processing time and apply our algorithm with more cases. Moreover, we plan to implement our algorithm for this refactoring as an Eclipse plug-in.

References

[1] H. Liu, et al., "Facilitating software refactoring with appropriate resolution order of bad smells," in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering2009*, ACM: Amsterdam, Netherlands, pp. 265-268, 2009.

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

[3] N. Tsantalis and A. Chatzigeorgiou, "Identification of extract method refactoring opportunities for the decomposition of methods," *J. Syst. Softw.*, 84(10): p. 1757-1782, 2011.

[4] P. Meananeatra, S. Rongviriyapanish and T. Apiwattanapong, "Using software metrics to select refactoring for long method bad smell," *Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON), 2011 8th International Conference on*, Khon Kaen, pp. 492-495, 2011.

[5] P. Meananeatra, S. Rongviriyapanish, and T. Apiwattanapong, "Identifying refactoring through formal model based on data flow graph," in *Software Engineering (MySEC), 2011 5th Malaysian Conference in.*, Johor Bahru, pp. 113-118, 2011.

[6] J. Ferrante, K.J. Ottenstein and J.D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, Vol.9,(3): pp. 319-349, 1987,

[7] M. Weiser, "Program slicing," in *Proceedings of the 5th international conference on Software engineering, IEEE Press: San Diego*, California, USA, pp. 439-449, 1981.

[8] A. Ouni, M. Kessentini and H. Sahraoui, "Search-Based Refactoring Using Recorded Code Changes," *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*, Genova, pp.221-230, 2013.

[9] G. Villavicencio, "A New Software Maintenance Scenario Based on Refactoring Techniques," *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, Szeged, pp. 341-346, 2012,.

[10] K. Maruyama, "Automated method-extraction refactoring by using block-based slicing," in

Proceedings of the 2001 symposium on Software reusability: putting software reuse in context2001, ACM:, Toronto, Ontario, Canada, Vol.26, pp. 31-40, 2001.

[11] L. Yang, H. Liu and Z. Niu, "Identifying Fragments to be Extracted from Long Methods," *2009 16th Asia-Pacific Software Engineering Conference*, Penang, pp. 43-49, 2009.

[12] N. Tsantalis and A. Chatzigeorgiou, "Identification of Extract Method Refactoring Opportunities," *Software Maintenance and Reengineering, 2009. CSMR '09. 13th European Conference on*, Kaiserslautern, pp. 119-128, 2009.

[13] R. Ettinger, Formal Program Re-design or (more specifically) Automating "Replace Temp with Query," *Programming Tools Group. Computing Laboratory*, University of Oxford,VASTT/ASTReNet Slicing Day Workshop, November 8th, 2004.

[14] http://www.jetbrains.com/idea/webhelp/replace-temp-with-query.html.

Songsakdi Rongviriyapanish He obtained his Ph.D. in Informatics (specialized in Software Architecture) from University Nancy 2, from Nancy in France. Since February 2001, he has 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, Practices and Patterns in Object-Oriented Programming courses. His research interests are about software quality and metrics, software design and formal methods.

Nopphawit Karunlanchakorn He is a master student at department of Computer Science, faculty of Science and Technology, Thammasat University, Thailand. He is a developer at Thomson Reuters Thailand.

Panita Meananeatra She 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.