A Visualization Method of Program Dependency Graph for Identifying Extract Method Opportunity

Tomoko Kanemitsu[1] Yoshiki Higo[1] Shinji Kusumoto[1]
[1]Graduate School of Information Science and Technology, Osaka University
{t-kanemt,higo,kusumoto}@ist.osaka-u.ac.jp

ABSTRACT

Refactoring is important for efficient software maintenance. However, tools supports are highly required for refactoring because manual operations of refactoring are troublesome and error prone. This paper proposes a technique that suggests *Extract Method* candidates automatically. *Extract Method* refactoring is to create a new method from a code fragment in an existing method. Previous research efforts showed that the *Extract Method* refactoring is often performed prior to other refactorings, so that it is important to support *Extract Method* refactoring. Previous studies have proposed methods that suggest *Extract Method* candidates based on linage or complexity. However it is originally desirable to divide methods based on their functionalities. This paper uses the strength of data connection between sentences in the source code. We deem that strongly-connected data expresses a single function. This paper proposes a technique that suggests *Extract Method* candidates based on strongly-connected data.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Technique—*Object-oriented design methods*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Enhancement, Restructuring, reverse engineering, and reengineering*

General Terms

Design, Management

Keywords

Refactoring, Visualization, Program Dependency Graph

1. INTRODUCTION

Refactoring is a set of operations for improving internal structure of software without changing its external behavior. Refactoring can reduce the cost required for maintaining software in the future. Even after commencing operations of a software system, its source code is often changed because of various reasons such as fixing field bugs, adding new functionalities, and adaptive maintenance tasks. Eick et al. reported that such repetitive changes degrade maintainability of the source code, so that more cost is required for maintenance as time passes [1]. Refactoring is a promising technique to prevent the source code from being degraded.

However, manual refactoring includes cumbersome operations. It requires depthful knowledge of maintained software and enriched experiences of programming languages used in its source code. If refactoring is performed inappropriately, the maintainability of the source code is not improved, or it even injects new faults into the source code. Consequently, tool support is highly required to perform appropriate refactorings.

One of the most often-performed refactorings is *Extract Method* [7]. *Extract Method* consists of a set of operations for extracting a part of an existing method as a new method. *Extract Method* enhances the cohesion of the refactored method and reduces its size. Therefore, the refactored method requires less maintenance cost in the future.

There are some software metrics that can be used for supporting *Extract Method* refactoring. For example, lines of code (LOC) is one of the metrics to identify candidates for *Extract Method*. Fowler addresses that small methods are greater than large ones [2]. Small methods have better readability and reusability than large ones. Methods including complicated control flow are also targets of *Extract Method*. McCabe proposed a cyclomatic complexity, which represents how much control flow is complicated [4]. However, there is a basic principle that one method should process only one thing. Not all long or complicated methods are against the principle.

The present paper proposes a visualization method for identifying candidates of *Extract Method* refactoring. The visualization makes it easy to understand which parts of existing methods can and should be refactored. The contributions of the present paper are as follows:

- The proposed method defines strength of data dependency in Program Dependency Graph (in short, PDG). In the visualization, the strength is reflected as the distance between nodes. Such a visualization enables to suggest *Extract Method* refactoring in keeping with the important principle that one method should process only one thing.
- The proposed method has been implemented as a software tool. The tool supports refactorings by interactive communication with a user. A user can understand why every refactoring candidate is suggested by the tool, and determine whether she performs the suggested refactorings or not by herself.
- We conducted an empirical evaluation with 14 subjects. In the evaluation, the proposed tool was compared with an ex-

isting refactoring support tool, JDeodrant. The evaluation showed that the proposed visualization is intuitively and easy to understand, whereas it also revealed some flaws of the proposed method.

2. RELATED WORK

Komondoor and Horwitz proposed a technique identifying extractable regions in functions of C program [3]. The input to the technique is a control flow graph (in short, CFG) of a function and a set of nodes. The technique identifies extractable regions including all the specified nodes in fully-automatic process. Tsantalis and Chatzigeorgiou also proposed a technique for identifying extractable regions [9]. The input is a PDG of a method of Java program, and the output is multiple extractable regions. An extractable region is identified for every variable appeared in the PDG. The both methods use data dependency and control dependency in the input graph for identifying practical candidates for *Extract Method*. For example, their methods replicate conditional predicates in the extracted region as necessary. The aim of the both methods is to identify where *can* be extracted in existing functions or methods, and the aim of the proposed visualization is to identify where *can* and *should* be refactored. The proposed visualization has a complementary relationship with their methods.

Refactoring Annotation, which was proposed by Murphy-Hill and Black, tells users whether a specified region can be extracted as a new method [5]. They also proposed a visualization method, Box View, which visualizes the block hierarchy in a given method, so that it can be used for identifying where should be refactored. However, Box View cannot identify non-contiguous regions or regions across multiple blocks as refactoring candidates because it is just an abstract visualization of the block hierarchy. On the other hand, the proposed visualization can tell users where should be extracted even if the region is non-contiguous.

The proposed method is a kind of smell detector. There are may research efforts related to smell detectors. For example, Murphy-Hill and Black proposed a method for visualizing *Feature Envy* [6]. *Feature Envy* smell is a typical target of *Extract Method*. Simon et al. proposed a method identifying method level code smells [8]. They used cohesion metrics for identifying code smells. In this research, we use a metrics representing the strength of data connection, which is similar to cohesion.

3. PROGRAM DEPENDENCY GRAPH

A PDG is a directed graph representing the dependencies between program elements (statements and conditional predicates). A PDG node is a program element, and a PDG edge indicates a dependency between two nodes. There are two types of dependencies in a PDG, namely, *control dependency* and *data dependency*. When all of the following conditions are satisfied, a control dependency from statement s_1 to s_2 exists:

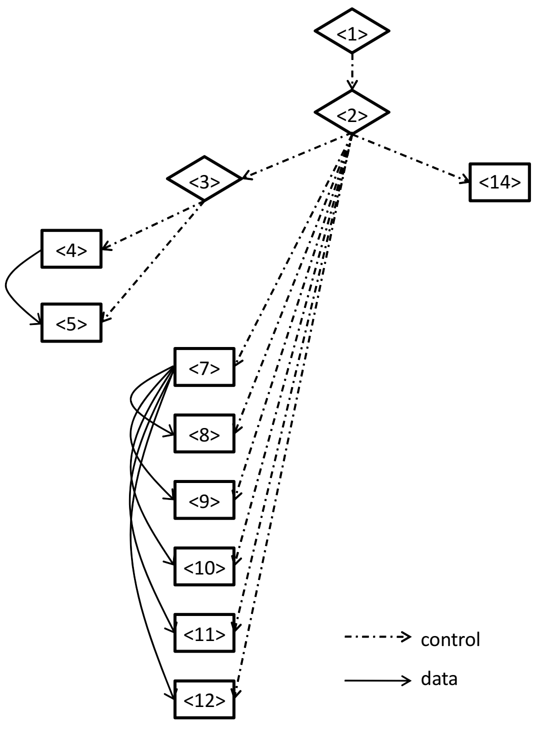
- s_1 is a conditional predicate, and
- the result of s_1 directly influences whether s_2 is executed.

When all the following conditions are satisfied, there is a data dependency from statement s_3 to s_4 via variable v:

- s_3 defines v, and
- s_4 references v, and
- there is at least one execution path from s_3 to s_4 without redefining v.

```
 1: String sample1(){
 2:   if(this.trueOrFalse()){
 3:     if(null == this.getPath()){
 4:       Project proj = this.getProject();
 5:       this.setPath(proj.getBaseDir());
 6:     }
 7:     StringBuilder text = new StringBuilder();
 8:     text.append("String A");
 9:     text.append("String B");
10:     text.append("String C");
11:     text.append("String D");
12:     return text.toString();
13:   }else{
14:     return "";
15:   }
16: }
```

(a) Original Source Code



(b) Generated PDG

Figure 1: Example of traditional PDG

Figure 1 is a simple example of source code and a PDG generated from it. Labels attached to the nodes mean the lines where their elements locate in the source code. The node labeled <1> is the enter node of the PDG. In this example, there are data dependencies between nodes using variables ("prog" or "text"), and there are conditional dependencies between the control predicates of the if-statements and their inner statements.

4. DEFINISIONS OF DISTANCES BETWEEN NODES IN PDGS

This section proposes a PDG visualization method for supporting *Extract Method*. The proposed method defines the distance between every pair of nodes having a data dependency. If the proposed method determines that the two nodes should not be separated by *Extract Method*, they are placed close to each other in the visualization. The remainder of this section describes three special situations where two node should not be separated. For each special situation, we describes **Situation**, **Example**, and **Reason**. **Situation** is a brief description of code situation, and **Example** shows an example code under the situation. **Reason** describes why the situation is special. Note that, variables that $v_i (i = 1,2,3)$ appear in the remainder of this section are defined by user input in the interactive visualization.

4.1 Atomic Data Dependency

[**Situation**]: A statement defines (initializes) a variable, and the variable is referenced in another statement just one time.

[**Example**]: Variable tmp in the following code is in this situation:

```
    final java.lang.Object tmp = objectA;
    objectA = objectB;
    objectB = tmp;
```

[**Reason**]: There is only a single edge of data dependency for such a variable. Temporary variables or constant definitions are in this context. Authors think that the defining statement and the referencing statement of such a variable should be in the same method. In the present paper, the relationship of such data dependency is called **Atomic Data Dependency** (in short, **ADD**). Two nodes having ADD are placed close to each other in the visualization. Assume that $distance_{ADD}$ represents the distance of two node having ADD, and it is defined as follows:

$$distance_{ADD} = v_1 \tag{1}$$

4.2 Spread Data Dependency

[**Situation**]: A statement defines a variable, and many other statements reference the variable.

[**Example**]: Variable tax_rate in the following code is in this situation.

```
float tax_rate = 0.05;
int taxA = articleA.getPrice() * tax_rate;
int taxB = articleB.getPrice() * tax_rate;
int taxC = articleC.getPrice() * tax_rate;
```

[**Reason**]:An often-referenced variable can be a core role for implementing a functionality. Thus, statements using such a variable should not be separated. The present paper calls the relationship of such data dependency **Spread Data Dependency** (in short, **SDD**). Nodes having SDD are placed close to one another in the visualization. We assume that $distance_{SDD}$ is the distance between nodes having SDD relationship. and it is defined as follows:

$$distance_{SDD} = \frac{v_2}{n} \tag{2}$$

n is the number of node referencing the variable. Consequently, the more nodes reference it, the closer they are.

4.3 Gathered Data Dependency

[**Situation**]: A statement references many variables defined in other statements.

[**Example**]: The last statement in the following code is in this situation.

```
int a = coefficients.getA();
int b = coefficients.getB();
int c = coefficients.getC();
Answer ans = QuadraticFormula.getAnswer(a,b,c);
```

[**Reason**]: A statement referencing many variables is a core role of a functionality, and the statements defining such variables prepares for execution of the core statement. Thus, they should not be separated because they are a part of the same functionality. In the present paper, the relationship of such data dependency is called **Gathered Data Dependency** (in short, **GDD**). Nodes having GDD are placed close to one another in the visualization. We assume that $distance_{GDD}$ is the distance between nodes having GDD relationship, and it is defined as follows:

$$distance_{GDD} = \frac{v_3}{m} \tag{3}$$

In the above formula, m is the number of nodes defining such variables. Consequently, the more nodes defining such variables there are, the closer they are placed.

If two nodes having 2 or more relationships in ADD, SDD, and GDD, the smallest distance is applied to the nodes.

5. IMPLEMENTATION

We have developed a prototype tool, ReAF (Refactoring Automated Finding), based on the proposed methods. The tool can handle the full grammar of Java language. The input to the tool is a set of source files forming a software system. The tool builds an intra-procedural PDG for every method in the input and it visualizes the built PDGs. A graph visualization framework, Jung[1], is used in the tool. Jung has a spring layout functionality, which automatically places nodes under the condition that given distances between the nodes are satisfied. At present, the proposed methods and the tool can handle only Java source code. However, it is not difficult to expand them to handle other programming languages.

Figure 2 is a snapshot of the tool. Target methods are shown with the class hierarchy format in the left side. If users select a method in this panel, then its source code and its PDG appear in the right side. Also, there are text fields for inputting values for $distance_{ADD}$, $distance_{SDD}$, and GDD in the upper side. Users can perform interactive analysis according to personal preference by changing the values.

A set of close-packed nodes is a candidate of *Extract Method*. If a user select a part of PDG by mouse dragging, then the corresponding source code is highlighted. If the selected part satisfying a following condition, the tool regards the part as an extractable code fragment and displays "OK". If not, the tool shows "NG" for telling the selected parts is difficult to be extracted.

[**Condition**]: The number of data dependency outgoing from the selected part is 1 or 0.

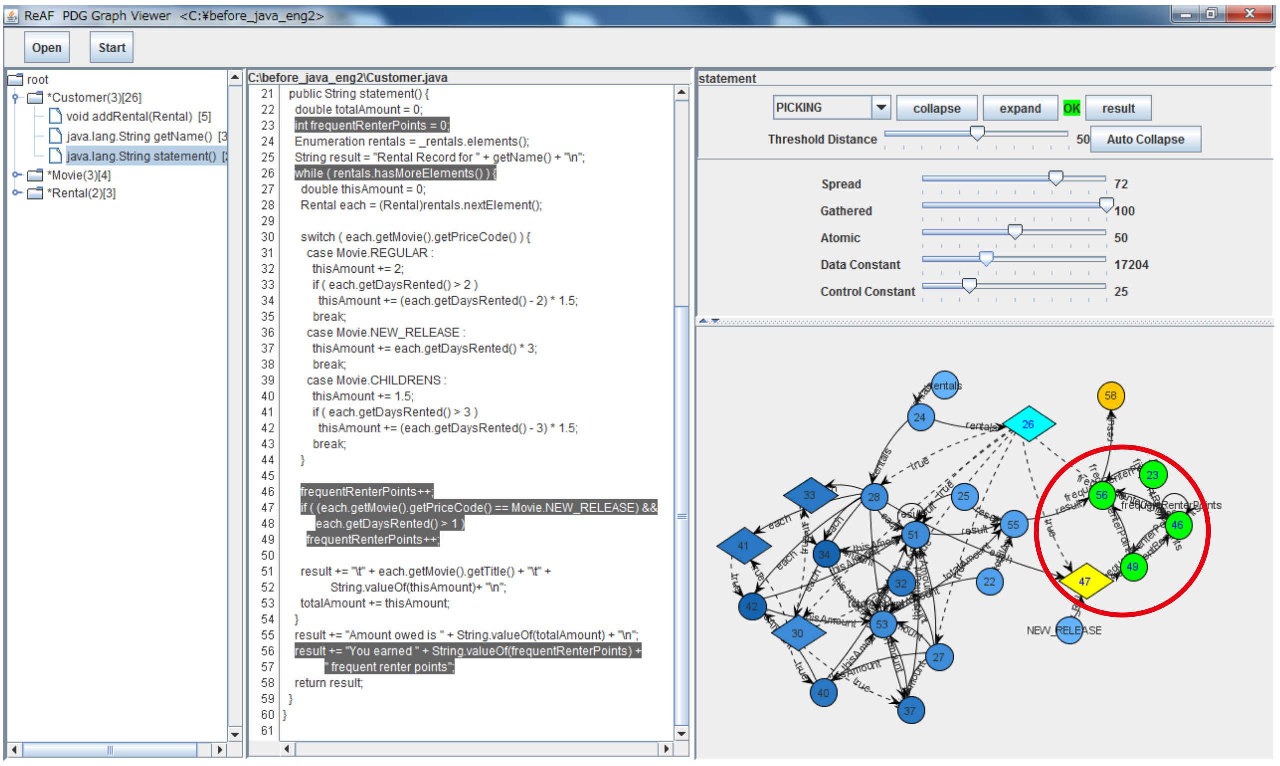
[**Reason**]: An outgoing data dependency means that data created in the selected part is referenced outside the part. If there is no outgoing edge from the selected part, the extracted method do not have to return any value. If the selected part has a single outgoing edge, it is easy to return the value to the new callee site. All we have to do is to add a return-statement. However, if there are 2 or more outgoing edges, further operations are required for extracting. Consequently, the tool uses this condition.

Figure 2 is an application on method *statement* in class *Customer* in Fowler's book [2]. In this application, we could identify instructions related to "rental point" as a target of *Extract Method*.

6. EVALUATION

We conducted an empirical study for evaluating the proposed method. In this study, 14 CS students evaluated ReAF. Seven of the students worked on the source code that they developed in the past. The other students worked on open source software that the author had downloaded in advance.

[1]http://jung.sourceforge.net/


ReAF PDG Graph Viewer <C:¥before_java_eng2>
Open
Start
root
*Customer(3)[26]
void addRental(Rental) [5]
java.lang.String getName() [3
java.lang.String statement() [
*Movie(3)[4]
*Rental(2)[3]
C:\before_java_eng2\Customer.java
statement
PICKING
collapse
expand
OK
result
Auto Collapse
NEW_RELEASE

Threshold Distance 50
Spread 72
Gathered 100
Atomic 50
Data Constant 17204
Control Constant 25

```
public String statement() {
  double totalAmount = 0;
  int frequentRenterPoints = 0;
  Enumeration rentals = _rentals.elements();
  String result = "Rental Record for " + getName() + "\n";
  while ( rentals.hasMoreElements() ) {
    double thisAmount = 0;
    Rental each = (Rental)rentals.nextElement();

    switch ( each.getMovie().getPriceCode() ) {
      case Movie.REGULAR :
        thisAmount += 2;
        if ( each.getDaysRented() > 2 )
          thisAmount += (each.getDaysRented() - 2) * 1.5;
        break;
      case Movie.NEW_RELEASE :
        thisAmount += each.getDaysRented() * 3;
        break;
      case Movie.CHILDRENS :
        thisAmount += 1.5;
        if ( each.getDaysRented() > 3 )
          thisAmount += (each.getDaysRented() - 3) * 1.5;
        break;
    }

    frequentRenterPoints++;
    if ( (each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
         each.getDaysRented() > 1 )
      frequentRenterPoints++;

    result += "\t" + each.getMovie().getTitle() + "\t" +
              String.valueOf(thisAmount)+ "\n";
    totalAmount += thisAmount;
  }
  result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
  result += "You earned " + String.valueOf(frequentRenterPoints) +
            " frequent renter points";
  return result;
 }
}
```

Figure 2: Application to the example in Fowler's Book. Green means that its node is included in the refactoring candidate, and yellow means that its node is the exit node of the PDG. Exit node must not be included in refactoring candidate.

6.1 Comparison Target

In this experiment, we also used an existing refactoring support tool, **JDeodrant**, which was developed by Tsantalis and Chatzigeorgiou [9]. JDeodrant identifies candidates for *Extract Method* by performing simple block slicing. JDeodrant is a plugin of Eclipse. If a user selects a target method, then JDeodrant shows a list of candidates that it identified. And if a user selects a candidate on the list, then the area of extracting target on the source code is highlighted. JDeodrant also has a functionality for comparing source code between before and after *Extract Method*. If a user considers that the candidate is appropriate as a *Extract Method* refactoring, she can push a button of automated source code modification.

6.2 Methodology

In this evaluation, every student identifies refactoring candidates from 3 methods. Students working on their own source code (GroupA) prepared refatoring targets by themselves and the authors prepared refactoring targets for students working on open source software (GroupB). In GroupA, every student works on different methods meanwhile all the students work on the same methods in GroupB.

Next, every student identifies *Extract Method* candidates from the target methods with the two tools. Halves of Group A and B firstly used ReAF, then they used JDeodrant[2]. The other halves firstly used JDeodrant, then they used ReAF. This is for avoiding bias of the order of tool applications. The students judged each of the candidates suggested by the tools as whether it is appropriate for *Extract Method* or not.

At last, every student answered a questionnaire about the refactorings with ReAF and JDeodrant. The questionnaire includes questions about the following aspects:

- Is tool's visualization helpful for identifying candidates?
- Is tool's operability is easy and intuitively?
- Do you want continuance usage for further refactorings on your own source code in the future?

Table 1: The number of methods that ReAF and JDeodrant can or cannot detect appropriate candidates

	ReAF	JDeodrant
including appropriate candidates	27	24
only inappropriate candidates	17	8
no candidate	0	12

6.3 Candidates Identification Result

Table 1 shows a quantitative result of the candidates identification. The row of "including appropriate candidates" means the number of methods where the tools could identify at least 1 appropriate candidate. We can see that ReAF identified appropriate candidates from more methods than JDeodrant. The row of "only inappropriate candidates" means the number of methods where the tools identified only inappropriate candidates. False positive of JDeodrant is less than ReAF. The row of "no candidate" means the number of methods where the tool identified no candidates.

[2]At the beginning of this evaluation, the authors trained students in how to use them.

Table 3: Result of the questionnaire about visualization capability, operability, and continuing usage of ReAF and JDeodrant

score	visualization		operability		continuing usage	
	ReAF	JDeodrant	ReAF	JDeodrant	ReAF	JDeodrant
4 (good)	4	4	1	5	2	4
3	9	10	3	6	7	6
2	1	0	9	3	4	3
1 (bad)	0	0	1	0	1	1
average	3.21	3.29	2.29	3.14	2.71	2.93

Table 4: Elapsed time to source code analysis by the tools and candidate identification by subjects

Time	analysis		identification		total	
(sec.)	ReAF	JDeodrant	ReAF	JDeodrant	ReAF	JDeodrant
max.	35.3	223.0	1,163	600.8	1,170	621.2
min.	0.02	0.01	20	12.4	23.7	16.0
ave.	8.2	35.6	305.1	142.8	313.4	178.4

Next, we investigated the methods where either of the tools could identify appropriate candidates. Table 2 shows the lines of the methods. For example the row of "ReAF" means the lines of methods where only ReAF could identify appropriate candidates. In the case of JDeodrant, the average lines of methods was 63.3, however a huge method (270LOC) was included in them. By removing it, the averge lines drops to 47.4. From this table, we can see that ReAF could identify appropriate candidates from large methods. One factor of the result is that: ReAF visualizes every statement in the source code as a node; the larger a method is, more node are created from it; if many nodes are included in a PDG, the density of the PDG become clearer; it is easy to select a dense part from the large PDG. However, in the case that there were so many nodes in a PDG (more than several hundred lines of code), it became difficult to identify with the proposed method.

6.4 User's Feedback

Table 3 shows the result of the questionnaire about the 3 items on ReAF and JDeodrant. The first item is *visualization capability*, which means how helpful tool's visualization is for identifying refactoring candidates. The second item is *operability*, which means the ease of operations for identifying refactoring candidates on the tools. The last item is *continuing usage*, which means the how much the subjects want to continue to use the tools on their own projects. From this table, on the evaluation of ReAF, the score of *visualization* is relatively high whereas *operability* is relatively low. The low operability was caused by usability problems on peripheral parts of the tool. The problem is not directly related to the proposed visualization. Consequently, it is not difficult to improve the operability of ReAF. Section 7 shows how we extended ReAF based on the result of this evaluation.

Table 2: The lines of methods where either of the tools could identify appropriate candidates

Tool	lines of methods		
	Average	Minimum	Maximum
ReAF	68.8	25	178
JDeodrant	63.3 (47.4)	16	270 (103)

On the other hand, JDeodrant marked high scores on both the *visualization* and *operability*. The reason why its operability was high is that JDeodrant is a plugin of Eclipse, so that subjects could use the tool on the wonted Eclipse screen. However, in many cases, JDeodrant identified many similar candidates from a method because there are often many similar slices in a method. In such a method, it was very burdensome to find appropriate candidates from a large list for subjects. While, subject were not suffered from such a problem on ReAF because ReAF displays a PDG itself.

6.5 Elapsed Time To Identification

Table 4 shows the elapsed time of source code analysis by the tools and candidates identification by subjects. The analysis time of ReAF is shorter than JDeodrant because JDeodrant calculates program slicings for every variable appeared in the methods. On the other hand, the identification time of ReAF is longer than JDeodrant. One of this factor is that the operability of ReAF was not as good as JDeodrant as shown in Subsection 6.4. However, in the case of JDeodrant, the identification requires several minutes because JDeodrant often identified so many candidates from a method.

6.6 Summary

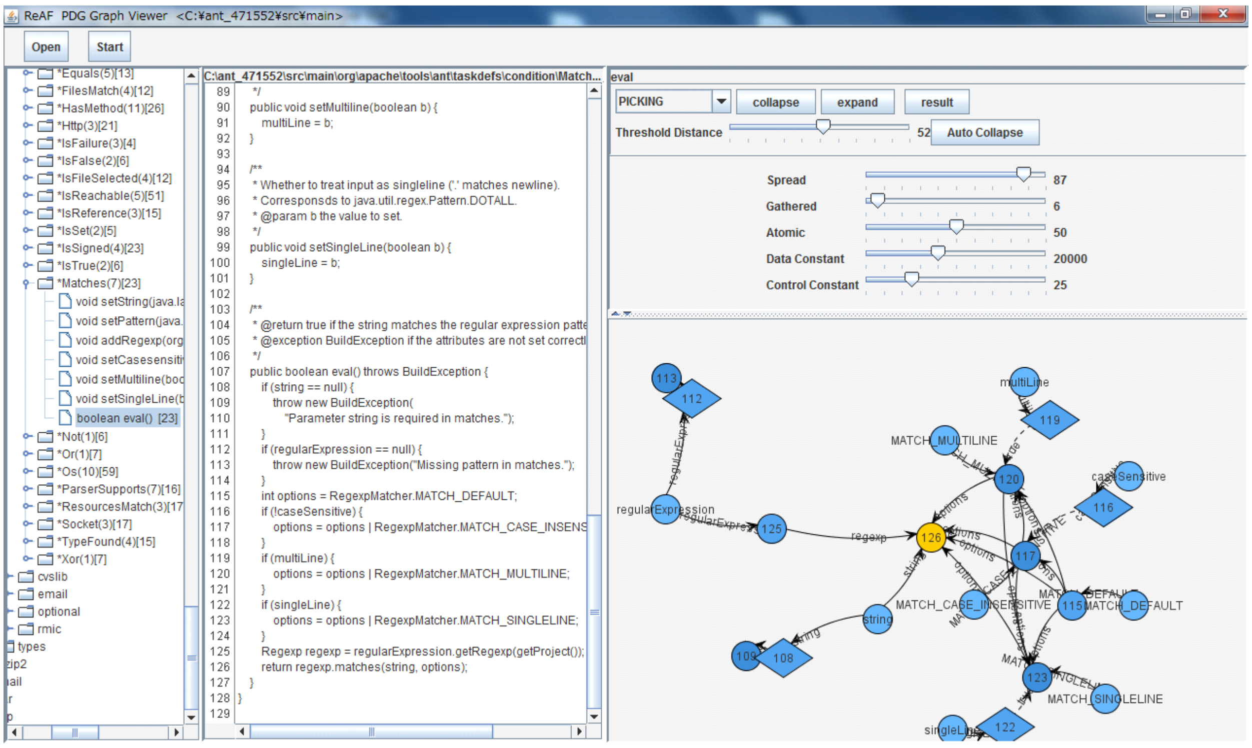
In this evaluation, there are several methods where either of ReAF and JDeodrant could identify appropriate candidates. Those happened because they have different definitions of code smells. In this experiment, we cannot conclude that either of them is superior to the other as a code smell detector. Consequently, we are going to expand the proposed method. This evaluation also revealed that ReAF has several problems on operability. Section 7 describes how we improved ReAF based on the result of this evaluation.

7. IMPROVEMENT BASED ON EVALUATION

We improved ReAF based on the result of the empirical study. The improvement includes the following 3 aspects:

- showing extraction result,
- merging multiple nodes into a single node,
- automatic recommendation.

The remainder of this section describes each of them in detail.


```
ReAF  PDG Graph Viewer  <C:¥ant_471552¥src¥main>
```

```
*Equals(5)[13]
*FilesMatch(4)[12]
*HasMethod(11)[26]
*Http(3)[21]
*IsFailure(3)[4]
*IsFalse(2)[6]
*IsFileSelected(4)[12]
*IsReachable(5)[51]
*IsReference(3)[15]
*IsSet(2)[5]
*IsSigned(4)[23]
*IsTrue(2)[6]
*Matches(7)[23]
    void setString(java.la
    void setPattern(java.
    void addRegexp(org
    void setCasesensitiv
    void setMultiline(boo
    void setSingleLine(b
    boolean eval()  [23]
*Not(1)[6]
*Or(1)[7]
*Os(10)[59]
*ParserSupports(7)[16]
*ResourcesMatch(3)[17
*Socket(3)[17]
*TypeFound(4)[15]
*Xor(1)[7]
cvslib
email
optional
rmic
types
zip2
ail
r
p
```

```
C:\ant_471552\src\main\org\apache\tools\ant\taskdefs\condition\Match...
     */
    public void setMultiline(boolean b) {
        multiLine = b;
    }

    /**
     * Whether to treat input as singleline ('.' matches newline).
     * Corresponsds to java.util.regex.Pattern.DOTALL.
     * @param b the value to set.
     */
    public void setSingleLine(boolean b) {
        singleLine = b;
    }

    /**
     * @return true if the string matches the regular expression patte
     * @exception BuildException if the attributes are not set correctl
     */
    public boolean eval() throws BuildException {
        if (string == null) {
            throw new BuildException(
                "Parameter string is required in matches.");
        }
        if (regularExpression == null) {
            throw new BuildException("Missing pattern in matches.");
        }
        int options = RegexpMatcher.MATCH_DEFAULT;
        if (!caseSensitive) {
            options = options | RegexpMatcher.MATCH_CASE_INSENS
        }
        if (multiLine) {
            options = options | RegexpMatcher.MATCH_MULTILINE;
        }
        if (singleLine) {
            options = options | RegexpMatcher.MATCH_SINGLELINE;
        }
        Regexp regexp = regularExpression.getRegexp(getProject());
        return regexp.matches(string, options);
    }
}
```

```
eval
PICKING
Threshold Distance  52
Spread  87
Gathered  6
Atomic  50
Data Constant  20000
Control Constant  25
```

Figure 3: A snapshot of ReAF in applying to a method *eval* in Ant 1.8.1

7.1 Showing Extraction Result

A big advantage of JDeoderant on the experimental study was that it had a functionality to show how the source code will be changed by the refactoring. Several subjects answered that such a functionality was very helpful to decide whether each refactoring should be performed or not. Therefore, we added such a functionality to ReAF. The improved ReAF has a button labeled as "Result". If a user presses the button, then ReAF shows the source code of the extracted code fragment.

7.2 Merging Multiple Nodes

In the case that there are so many nodes in a PDG, some subjects found it difficult to operate the PDG because screen size and performance. In order to reduce such a difficulty, we added a functionality to merge multiple nodes into a single node. If users identify a set of nodes that are not separated to different methods, then ReAF merges the set of nodes into a single node. The size of merged node is proportional to the number of the original nodes. This functionality reduces the number of nodes in PDGs, which is helpful to identify refactoring candidates more efficiently.

7.3 Automatic Recommendation

In the PDG visualization described in Section 4, users can freely select an any part of PDGs and then they perform refactoring. However, such selection sometimes needs a certain level of time, which is shown in Table 4. In order to more quickly identify refactoring candidates, automatic recommendation is indispensable. Therefore we added a functionality to recommend refactoring candidates.

The automatic recommendation on ReAF works in the following algorithm. The input of the algorithm is the threshold of edge length, k, and a PDG, P. The output is a merged node, which is a refactoring candidate.

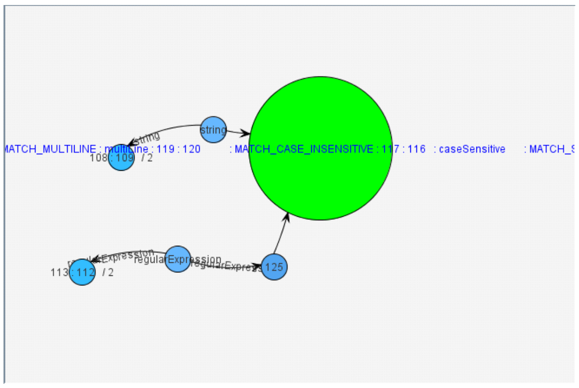

MATCH_MULTILINE : multiLine : 119 : 120
108
109
/2
string
string
: MATCH_CASE_INSENSITIVE : 117 : 116 : caseSensitive
: MATCH_S
113
112
/2
regularExpression
regularExpression
125

Figure 4: A snapshot of PDG with recommendation

1. get the shortest edge ,e, in P, and to go step.2,
2. if the length of e is shorter than k, then marge the two nodes at the ends of e, and go to step.3, otherwise exit the algorithm,
3. collect edges incoming to or outgoing from the merged node, and select the shortest edge as e, and go to step.2,

The output of the algorithm is the last merged node, which includes a set of nodes included in a code fragment that should be extracted as a new method. All a user has to do is just pushing "recommend " button. As described in Subsection 7.2, the size of the merged node is proposed to the number of original nodes. It is very easy to identify which is the merged node in the PDG.

```
  107: public boolean eval() throws BuildException {
  108:   if (string == null) {
  109:     throw new BuildException(
  110:         "Parameter string is required in matches.");
  111:   }
  112:   if (regularExpression == null) {
  113:     throw new BuildException(
            "Missing pattern in matches.");
  114:   }
++115:   int options = RegexpMatcher.MATCH_DEFAULT;
++116:   if (!caseSensitive) {
++117:     options = options |
                   RegexpMatcher.MATCH_CASE_INSENSITIVE;
++118:   }
++119:   if (multiLine) {
++120:     options = options |
                   RegexpMatcher.MATCH_MULTILINE;
++121:   }
++122:   if (singleLine) {
++123:     options = options |
                   RegexpMatcher.MATCH_SINGLELINE;
++124:   }
  125:   Regexp regexp =
              regularExpression.getRegexp(getProject());
  126:   return regexp.matches(string, options);
  127: }
```

(a) before refactoring (version 1.8.1)

```
107: public boolean eval() throws BuildException {
108:   if (string == null) {
109:     throw new BuildException(
110:         "Parameter string is required in matches.");
111:   }
112:   if (regularExpression == null) {
113:     throw new BuildException(
             "Missing pattern in matches.");
114:   }
115:   in options = RegexUtil.asOptions(
              caseSensitive, multiLine, singleLine);
116:   Regexp regexp =
              regularExpression.getRegexp(getProject());
117:   return regexp.matches(string, options);
118: }
```

(b) after refactoring (version 1.8.2)

```
 95: public static in asOptions(boolean caseSensitive,
                               boolean multiLine,
 96:                           boolean singleLine){
 97:   int options = RegexpMatcher.MATCH_DEFAULT;
 98:   if (!caseSensitive) {
 99:     options = options |
              RegexpMatcher.MATCH_CASE_INSENSITIVE;
100:   }
101:   if (multiLine) {
102:     options = options | RegexpMatcher.MATCH_MULTILINE;
103:   }
104:   if (singleLine) {
105:     options = options | RegexpMatcher.MATCH_SINGLELINE;
106:   }
107:   return options;
108: }
```

(c) extracted method (version 1.8.2)

Figure 5: An actual refactoring example in Ant

7.4 Application On OSS

We applied the improved ReAF to an open source software, Ant. Figure 3 shows a snapshot of ReAF applying to a method *eval* on class *ant.taskdefs.condition.Matches* in version 1.8.1. There are 22 nodes in this PDG. By using the functionality of automatic recommendation, the PDG visualization was changed to Figure 4. In Figure 4, a big green node was suggested as a new method. Figure 5 shows the source code of before and after the refactoring. The line beginning with ++ in Figure 5(a) forms a code fragment of the refactoring candidate. We can see that the automatic recommendation could identify a code fragment including instructions using variable *option*. We confirmed that this refactoring was actually performed between version 1.8.1 and 1.8.2. Figure 5(b) shows the source code after the refactoring, and Figure 5(c) shows the source code of the newly extracted method. Also, JDeadrant, which was a compared tool in Section 6 could not find this refactoring instance.

8. CONCLUSION

This paper presented a PDG visualization method for identifying opportunities of *Extract Method* refactoring and introduces an implementation of the proposed method, ReAF. We revealed the advantages and disadvantages of the proposed method by comparing it with an existing method on an empirical study. ReAF has been improved based the result of the empirical study. In the future, we are going to apply it to several open source software and evaluate its effectiveness and usefulness.

ACKNOWLEDGEMENT

The present research is being conducted as a part of the Stage Project, the Development of Next Generation IT Infrastructure, supported by the Ministry of Education, Culture, Sports, Science, and Technology of Japan. This study has been supported in part by Grants-in-Aid for Scientific Research (A) (21240002) and (C) (20500033) from the Japan Society for the Promotion of Science, and Grand-in-Aid for Young Scientists (B) (22700031) from Ministry of Education, Science, Sports and Culture.

9. REFERENCES

[1] S. G. Eick, T. L. Graves, A. F. Karr, and J. S. M. adn Audris Mockus. Does code decay? assessing the evidence from change management data. *Transactions on Software Engineering*, 27(1), Jan. 2001.

[2] M. Fowler. *Refactoring: improving the design of existing code*. Addison Wesley, 1999.

[3] R. Komondoor and S. Horwitz. Effective, Automatic Procedure Extraction. In *Proc. of the 11th International Workshop on Program Comprehension*, pages 33–42, May 2003.

[4] T. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, Dec. 1976.

[5] E. Murphy-Hill and A. P. Black. Breaking the Barriers to Successful Refactoring: Observations and Tools for Extract Method. In *Proc. of the 30th International Conference on Softwaer Engineering*, pages 421–430, May 2008.

[6] E. Murphy-Hill and A. P. Black. An Interactive Ambient Visualization for Code Smells. In *Proc. of the 5th International Symposium on Software Visualization*, pages 5–14, May 2010.

[7] E. Murphy-Hill, C. Parnin, and A. P. Black. How We Refactor, and How We Know It. In *Proc. of the 31st International Conference on Software Engineering*, pages 287–297, Oct. 2009.

[8] F. Simon, F. Steinbrückner, and C. Lewerentz. Metrics based refactoring. In *Proc. of the 5th European Conference on Software Maintenance and Reengineering*, pages 30–38, Mar. 2001.

[9] N. Tsantalis and A. Chatzigeorgiou. Identification of Extract Method Refactoring Opportunities. In *Proc. of the 13th European Conference on Software Maintenance and Reengineering*, pages 119–128, Mar. 2009.