2009 16th Asia-Pacific Software Engineering Conference

Identifying Fragments to Be Extracted from Long Methods

Limei Yang*, Hui Liu*‡ and Zhendong Niu*†
*School of Computer Science and Technology
Beijing Institute of Technology, Beijing 100081, China
{eicoyang,liuhui2005}@gmail.com,zniu@bit.edu.cn
†Corresponding author
‡Key Laboratory of High Confidence Software Technologies (Peking University)
Ministry of Education, Beijing 100871, China

***Abstract*—Long and complex methods are hard to read or maintain, and thus usually treated as bad smells, known as *Long Method*. On the contrary, short and well-named methods are much easier to read, maintain, and extend. In order to divide long methods into short ones, refactoring *Extract Method* was proposed and has been widely used. However, extracting methods manually is time consuming and error prone. Though existing refactoring tools can automatically extract a selected fragment from its inclosing method, which fragment within a long method should be extracted has to be determined manually. In order to facilitate the decision-making, we propose an approach to recommend fragments within long methods for extraction. The approach is implemented as a prototype, called *AutoMeD*. With the tool, we evaluate the approach on a non-trivial open source project. The evaluation results suggest that refactoring cost of long methods can be reduced by nearly 40%. The main contribution of this paper is an approach to recommending fragments within long methods to be extracted, as well as an initial evaluation of the approach.**

***Keywords*-Refactoring, Long Method, Extract Method, Bad Smells.**

I. Introduction

Software refactoring is to change software internal structures to facilitate further maintenance and evolution, while keeping external behaviors unaffected. Refactoring is widely accepted as an effective means to improve software quality, especially maintainability, extensibility and reusability.

Bad Smell is a key concept in software refactoring. As illustrated in [1], bad smells are designs in source code screaming for refactoring. In other words, bad smells are badly designed source code that should be implemented in other ways to facilitate further maintenance and evolution.

As the proposers of *Bad Smell*, Fowler and Beck [1] proposed and discussed 22 kinds of Bad Smells. Among them, *Long Method* is one of the most common bad smells. Long and complex methods are hard to read or maintain. The main problems with long methods include:

- Hard to read. According to the theory of Miller [2], human beings usually can only handle 5-9 things at one time. Consequently, if a long method contains dozens of variables, hundreds of statements, and dozens of branches, it is hard to discover its intent (function), let alone how it fulfils the intent.
- Hard to reuse. A long method usually fulfils more than one responsibilities. Though these responsibilities are required in many scenarios, it is hard to find scenarios where all of the responsibilities are required. As a result, the method has few chances to be reused.

As a solution to *Long Method*, refactoring rule *Extract Method* was proposed [1]. *Extract Method* is to extract relatively independent and cohesive fragments from long methods as new, short, and reusable methods. As a result, the system is composed of short and well-named methods which are easy to read and reuse. The process of the refactoring involves the following two steps:

- Identifying fragments to be extracted. In order to divide a long method, we should identify which fragments should (and can) be extracted as a new method. The key point is to find out a cohesive fragment which has appropriate length and low coupling with other parts of the method.
- Extracting selected fragments. Once fragments to be extracted are identified, the extraction begins. However, the extraction is not easy. Variables accessed but not declared by the fragment should be passed in as input parameters; variables modified by the fragment and accessed by following fragments should be passed out as output (return) parameters.

Resolving *Long Method* manually with *Extract Method* is time consuming and error-prone. In order to facilitate the refactoring, some refactoring tools are proposed. However, these tools focus on the second step only (extracting selected fragments), and the first step (identifying fragments to be extracted) has to be done manually. For example, the *Eclipse* fulfils the refactoring *Extract Method*, but it can be applied only if fragments to be extracted are selected manually. However, the manual identification is not easy: Software engineers have to analyze the whole method, divide the method into cohesive fragments, calculate the coupling among the fragments, and decide which fragment should be extracted according to their responsibilities (intents) , cohesion, and independency.

In order to facilitate the first step of *Extract Method*

1530-1362/09 $26.00 © 2009 IEEE
DOI 10.1109/APSEC.2009.20

refactoring, we propose an approach to automating the identification of extractable fragments in long method. We also implement the approach as an Eclipse plug-in *AutoMeD*, and evaluate it on a non-trivial open source project. The evaluation results suggest that a reduction up to 40% in refactoring cost could be achieved with the identification tool.

The rest of the paper is organized as follows: Section II presents an illustrating example used throughout the paper.Section III proposes an approach to identify fragments to be extracted. Section IV presents and discusses evaluations. Section V discusses related work, and Section VI gives a conclusion.

II. Illustrating Example

In order to demonstrate the approach to recommending fragments within *Long Method* to be extracted, we present here a motivating example. The source code is extracted from a class: *Diff*, the class is a difference algorithm which is an implementation of the longest common subsequences algorithm for Java. The following method (method *traverseSequences* of class *Diff*, downloaded from http://www.incava.org/projects/java/javadiff/) will be used throughout the paper as an illustrating example.

```
1 protected void traverseSequences()
2 {
3   Integer[] matches =
        getLongestCommonSubsequences();
4
5   int lastA = a.length - 1;
6   int lastB = b.length - 1;
7   int bi = 0;
8   int ai;
9
10  int lastMatch = matches.length - 1;
11
12  for (ai = 0; ai <= lastMatch; ++ai) {
13      Integer bLine = matches[ai];
14
15      if (bLine == null) {
16          onANotB(ai, bi);
17      }
18      else {
19          while (bi < bLine.intValue()) {
20              onBNotA(ai, bi++);
21          }
22
23          onMatch(ai, bi++);
24      }
25  }
26
27  boolean calledFinishA = false;
28  boolean calledFinishB = false;
29
30  while (ai <= lastA || bi <= lastB) {
31
32      // last A?
33      if (ai == lastA + 1 && bi <= lastB) {
34          if (!calledFinishA && callFinishedA())
                {
35              finishedA(lastA);
36              calledFinishA = true;
37          }
38          else {
39              while (bi <= lastB) {
40                  onBNotA(ai, bi++);
41              }
42          }
43      }
44
45      // last B?
46      if (bi == lastB + 1 && ai <= lastA) {
47          if (!calledFinishB && callFinishedB())
                {
48              finishedB(lastB);
49              calledFinishB = true;
50          }
51          else {
52              while (ai <= lastA) {
53                  onANotB(ai++, bi);
54              }
55          }
56      }
57
58      if (ai <= lastA) {
59          onANotB(ai++, bi);
60      }
61
62      if (bi <= lastB) {
63          onBNotA(ai, bi++);
64      }
65  }
66 }
```

III. Approach

A. Overview

In this section, the overview of the proposed approach is presented, and the details of each step are illustrated in the following sections.

The flow chart of the approach is presented in Fig. 1. In the first step (*Division*), long methods are divided into sequential fragments according to their structures, e.g., blank lines, iterations, and branches. Since program structures are hierarchical, the fragments are also hierarchical too, i.e., one fragment may contain sub-fragments. In the second step, basic fragments are composed to form bigger composite fragments. In the third step, variables accessed (read or changed) in each fragment are collected. The variables are necessary for the computation of coupling between fragments. Some variable declarations are adjusted to reduce coupling among different fragments while software behaviors are unaffected. Coupling among fragments are computed in the fifth step. Finally, the approach sort the fragments and recommend an *Extractable Fragment* as refactoring candidate according to the coupling and other information of the fragments.

The following section illustrate the approach in detail, each section for a step.

B. Breaking Long Methods into Fragments

Statements within a method are divided into two categories: basic statements and complex control structures. Complex control structures include: *ForStatement*, *IfStatement*, *WhileStatement*, *TryStatement*, *DoStatement* and *SwitchStatement*. All of these structures are defined in

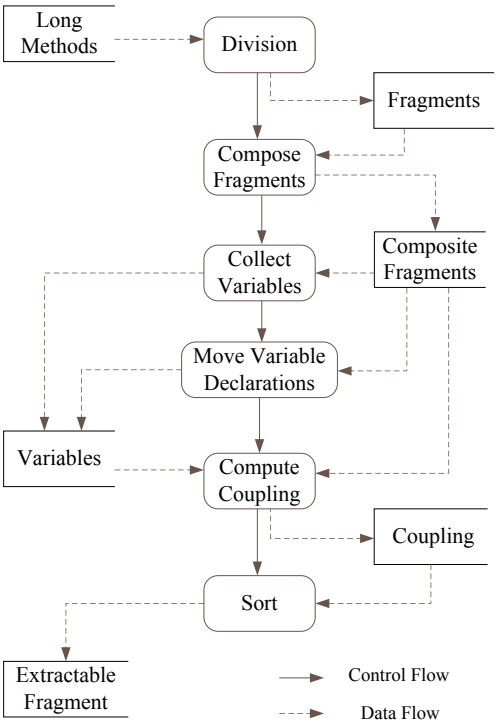

Figure 1. Flow Chart of The Approach

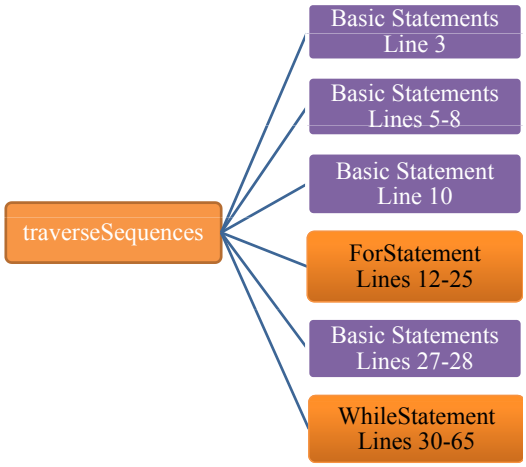

Figure 2. Top Level Division of Method *traverseSequences*

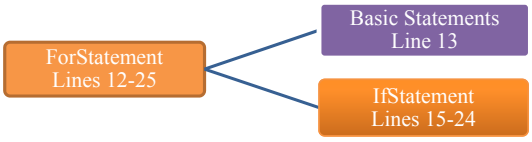

Figure 3. Further Division of *ForStatement*

Eclipse Java Development Tools (JDT) [3]), and *JDT* would recognize these structures automatically.

Basic statements are further divided into groups: Two basic statements belong to the same group if they are not separated by any complex control structure or blank lines which are widely used to separate relatively independent fragments. In other words, basic statements are divided by complex control structures and blank lines.

For the illustrating example, basic statements are divided into 4 groups (fragments) as shown in Fig. 2:

- Fragment 1: Line 3;
- Fragment 2: Lines 5-8;
- Fragment 3: Line 10;
- Fragment 4: Lines 27-28.

Fragment 1, *Fragment 2* and *Fragment 3* are separated by blank lines, while *Fragment 3* and *Fragment 4* are separated by a *ForStatement*.

For complex control structures, the division is recursive. Each control structure, e.g., a *ForStatement*, should be treated as a single fragment in the top level as shown in Fig. 2 (*ForStatement Lines 12-25*). At the same time, the statements within the control structure deserve further division. A complex control structure may contain a long sequence of statements, or even nest other complex control structures. In this case, the structure should be further divided into fine-grained fragments suitable to be extracted as a short but complete method. The division is the same as that of long methods.

For example, for the *ForStatement* in the illustrating example:

```
12 for (ai = 0; ai <= lastMatch; ++ai) {
13          Integer bLine = matches[ai];
14
15          if (bLine == null) {
16              onANotB(ai, bi);
17          }
18          else {
19              while (bi < bLine.intValue()) {
20                  onBNotA(ai, bi++);
21              }
22
23              onMatch(ai, bi++);
24          }
25 }
```

the inner-fragment (lines 13-24) should be further divided. It can be divided into a *Basic Statements* (Line 13) and a *IfStatement* (lines 15-24). The resulting structure is presented in Fig. 3. Note that the *IfStatement* in Fig. 3 should also be further divided until all leaf nodes are Basic Statements.

C. Composition of Fragments

As discussed in the Section I, a good method should be well-sized. But the fragments got by division do not necessarily meet the requirement. It is impossible to extract a single statement as a new method because the cost would

overtake the benefit. In some cases, sequential small fragments which are consecutive sibling nodes (fragments) in the fragment hierarchy (as shown in Fig. 2) should be composed to form well-size fragments as candidates for *Extract Method* refactoring. Any combination of consecutive sibling nodes are acceptable if and only if their length is suitable. For example, if the minimal and maximal lengths of a good method are 6 and 15, respectively, only these combinations fall into the range are acceptable.

D. Collection of Variables

For each fragment, we collect variables modified by it as V_w, and variables accessed but not modified by it as V_r. All variables appear before the fragments are collected as V_b, and variables appear after the fragment are collected as V_a.

If the fragment is extracted as a new method, and called in the original place, variables which appear before the fragment and accessed by the fragment (no matter read or write) should be passed in as input parameters. Those modified by the fragment and accessed by following fragments should be returned as output parameters. The detailed formulas are presented as follows:

$$P_{in} = V_b \cap (V_w \cup V_r) \quad (1)$$

$$P_{out} = V_a \cap (V_w) \quad (2)$$

E. Move of Variable Declarations

For some old programming languages, e.g., *C of C89 standard* [4], all variables should be declared in the very beginning of methods. But it is not true anymore for modern programming languages, e.g., *C++*, *c#*, and *Java*. As a result, it is possible to move declarations of variables while keeping program behaviors unaffected.

In the illustrating example, the variables $lastA$ and $lastB$ are defined in line 5 and 6, respectively. But until line 30, they have not be accessed by any statement. Consequently, they may be moved to the front of the *WhileStatement* in line 30.

Declaring variables as late as possible helps to reduce coupling between the original method and the new method extracted from the original one. Consequently, we move variable declarations whenever possible to reduce coupling.

F. Computation of Coupling

In previous sections, we get some well-sized fragments ready for extraction. In order to extract the most independent fragments, we should calculate the coupling between the fragments to be extracted and its inclosing method.

In this paper, we mainly focus on data coupling. Consequently, we calculate the coupling between the original method and the new method (result of *Extract Method* refactoring) by counting how many parameters are needed by the new method. The detailed formula is presented as follows:

$$Coupling(f) = |p_{in}(f)| + |p_{out}(f)| \quad (3)$$

where $|p_{in}(f)|$ and $|p_{out}(f)|$ are the amounts of the input parameters and output parameters of the new method if fragment f is extracted from its inclosing method.

G. Recommending Candidate Fragments

$Coupling(f)$ indicates the cost of extracting fragment f as a new method. But the cost is not the only factor in method extraction. Otherwise, nothing would be extracted (in this case, no cost would be paid). What we do care is the balance between benefits and cost.

The benefits of *Extract Method* refactoring are the reduction in the length (and complexity) of long methods. The reduction in length approximates the length of the extracted fragment. As a result, the benefit of extracting fragment f can be represented as:

$$Benefit(f) = |f| \quad (4)$$

Where $|f|$ is the length of fragment f.

The ratio of benefit/cost is:

$$R(f) = \frac{Benefit(f)}{Coupling(f)} = \frac{|f|}{(|p_{in}(f)| + |p_{out}(f)|)} \quad (5)$$

All candidate fragments are sorted according to their benefit/cost ratios in a descending order. And the one with highest ratio will be recommended.

For the illustrating example, our approach identifies the following fragment to be extracted as a new method:

```
——Fragment To be Extracted——
R(f)=3
In Parameters: [matches, bi, ai]
Out Parameters: [bi]

Integer bLine=matches[ai];                          13
                                                    14
if (bLine == null) {                                15
  onANotB(ai,bi);                                   16
}                                                   17
 else {                                             18
  while (bi < bLine.intValue()) {                   19
    onBNotA(ai,bi++);                               20
  }                                                 21
                                                    22
  onMatch(ai,bi++);                                 23
}                                                   24
```

The identified fragment is a composite fragment: Composed of three sub-fragments of the *ForStatement* (lines 12-25) as presented in Fig. 3. If the fragment is extracted as method, and called in the original place, three variables should be passed in as input parameter ($matches$, bi, and ai), and one variable should be returned (bi). The benefit/cost ratio is 3.

IV. Evaluation

In this section, we present the evaluation of the proposed approach . We describe the objective, subjects, and process first. And then we present and discuss the results of the evaluation.

A. Objective

The objective of the evaluation is to investigate the following questions:

1) Accuracy of the approach. Is the identification accurate? How often does the identification is accepted by software engineers?
2) Impact on refactoring cost. Does this approach help to ease the resolution of long methods? If yes, to what extent the refactoring cost is reduced?
3) Impact on software quality. Does the approach lead to higher or lower quality systems? If yes, to what extent software quality is affected by the approach?

B. Subject

For evaluation, we applied the proposed approach to an open source project: *ThoutReader* [5]. *ThoutReader* is a multi-document help system allowing users to browse, search, bookmark, and append documentation. All documents are stored in XML (Extensible Markup Language) files. *ThoutReader* is platform independent because it was implemented in Java.

We downloaded the source code of version 1.8.0. from *Sourceforge* [6]. The project is non-trivial, and contains nearly 20,000 lines of source code, defining 269 classes.

C. Process

To investigate whether $AutoMeD$ helps to reduce refactoring cost or improve software quality (the second and third investigated questions), we assigned two software engineers to resolve long methods in the same project *ThoutReader* independently. One of them carried out refactoring with $AutoMeD$ while the other identified manually fragments to be extracted. In previous experiments, the two people showed approximately equivalent ability in refactoring, including efficiency and effect. As a result, the difference in refactoring cost and the quality of resulting systems would suggest the impact of $AutoMeD$ on refactoring cost and software quality.

To investigate the accuracy of $AutoMeD$, we should know which fragments were recommended by $AutoMeD$ as candidates for *Extract Method*, and which fragments the software engineer did extract. Via comparing the recommended fragments with eventually extracted fragments, we can find out the accuracy of the proposed approach. Since no undeterminate algorithm is used in $AutoMeD$, $AutoMeD$ recommends the same fragment for the same method whenever the $AutoMeD$ is applied. Consequently, we do not have to record the recommendations because they may recur whenever they are needed. We do not record the fragments extracted by engineers either because this information can be inferred by comparing the original version with the resulting version. As a result, the engineer did not record anything during the evaluation, which made the evaluation more reliable. Otherwise, we have to tell him that he is carrying out an evaluation and must do some external work unnecessary in real practice, which may be a threat to the validity.

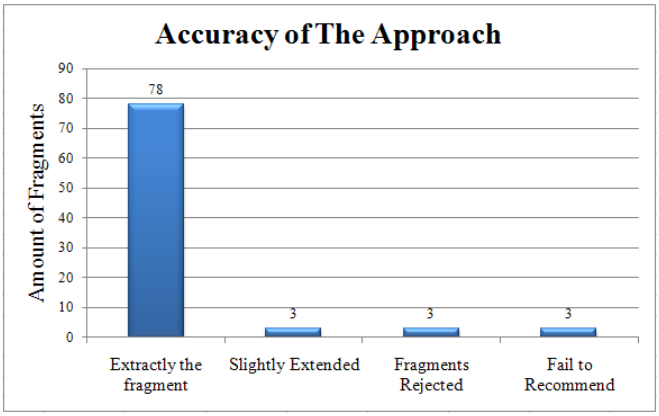

Figure 4. Accuracy of The Approach

D. Results

Results of the evaluation is discussed concerning the three investigated questions proposed in Section IV-A.

1) Impact on Software Quality: Metrics of the original system and resulting systems are presented in Table I. The first column presents categories of different metrics which may be influenced by *Extract Method*. The second column is metrics of the original system. The third and fourth columns present metrics of resulting systems of *Extract Method* refactoring without and with $AutoMeD$, respectively.

From the table, we observe that refactoring *Extract Method* did improve software quality. Complex of methods (*McCabe per Method*) was reduced while coupling (*Efferent Coupling and Afferent Coupling*) was kept unchanged.

We also observe that metrics of the two resulting systems (column 2 and column3) are close, suggesting that identifying fragments to be extracted with $AutoMeD$ has little influence on the quality of resulting systems compared to manual identification.

2) Accuracy of The Approach: The accuracy of the approach is presented in Fig. 4. From Fig. 4, we observe that 78 recommended fragments were accepted without any adjustment, 3 recommended fragments were slightly extended, 3 recommended fragments were rejected, and $AutoMeD$ failed to recommend fragments for three methods.

78 out of 84 fragments recommended by $AutoMeD$ were accepted without any adjustment. The ratio is up to 92.86%, suggesting a high accuracy of the approach.

3 out of 84 recommended fragments (3.57%) were slightly extended before extraction. A type example is method *getConfigProperties* of class *OSoftConfig*. $AutoMeD$ recommended all statements within a *TryStatement*, but the software engineer extracted its associated *Catch* at the same

Table I
METRICS OF RESULTING SYSTEMS

	Original System	Resulting System (without AutoMeD)	Resulting System (with AutoMeD)
Number of Methods	484	541	540
Number of Static Method	45	67	66
Method Lines of Code per Method	6.892	6.215	5.919
Lack of Cohesion of Methods	0.244	0.253	0.261
Weighted Methods per Class	15.818	17.045	17.03
McCabe per Method	1.974	1.85	1.855
Nested Block Depth	1.495	1.507	1.508
Afferent Coupling	4.875	4.875	4.875
Number of Parameters	0.758	0.877	0.85
Efferent Coupling	3.875	3.875	3.875

time. $AutoMeD$ failed to recommend the *Catch* because of predefined threshold *MaxLength*. Fragments larger than *MaxLength* would be ignored. *MaxLength* is used to guarantee that new methods would not be treated as *Long Methods*. However, length of methods is not the only criterion on deciding a method is too long or not (in fact there is no widely accepted criterion on the decision at all). As a result, the threshold *MaxLength* may prevent suitable fragments to be recommended.

3 out of 84 recommended fragments (3.57%) were rejected by the software engineer. $AutoMeD$ sorts fragments according to their benefit/cost ratios, i.e.,

$$\frac{|f|}{(|p_{in}(f)| + |p_{out}(f)|)}$$

as discussed in Section III-G. But in fact, other information, e.g., semantics and responsibilities of fragments, also influences the decision on how to decompose long methods. However, semantics and responsibilities of fragments are hard to formalize, and thus are not taken into account in the proposed approach. As a result, some times (3.57% in the evaluation), recommended fragments may be rejected.

3 out of 87 (84+3) times (3.45%), $AutoMeD$ failed to recommend any fragment. For the three methods, $AutoMeD$ failed because there is no fragments with suitable size to be extracted. Method *loadClasses* of class *BrowserLauncher* is a good example. A long sequence of basic statements are not separated by any blank line as expected. As a result, $AutoMeD$ treated them as a single large fragment instead of dividing them into smaller ones. In conclusion, the assumption that independent fragments should be separated by blank lines or control structures is a drawback of the approach though the assumption holds for most of the time (held in 96.55% cases in the evaluation).

In conclusion, in 92.86% cases, recommended fragments were accepted without any adjustment, suggesting a high accuracy of the approach; in 3.57% cases, recommend fragments were accepted with slightly adjustment; in 3.57% cases, recommended fragments are rejected; in 3.45% cases, $AutoMeD$ failed to recommend any fragment though software engineers did extract some.

3) Impact on Refactoring Cost: As discussed in Section I, resolution of long methods is composed of two steps: Identification of fragments to be extracted, and extraction of selected fragments. Since the last step is automated by existing refactoring tools, e.g., *Eclipse*, the refactoring cost is mainly composed of the cost of the first step and recursive testing after refactoring.

Sine $AutoMeD$ would identify fragments to be extracted, and the accuracy is higher than 90%, we expect it to reduce dramatically the cost of the first step. The evaluation results suggest that the reduction is up to 40% (6 hours vs. 10 hours).

E. Threats to Validity

Resolution of long methods is subjective. Since there is no commonly accepted criterions on what kind of methods are *Long Method*, one may treat a method as *Long Method* while other do not. There is no commonly accepted laws on deciding how to divide long methods, either. As as result, different people may extract different fragments from the same method.

The subjectivity in resolution of long methods is a threat to the validity of our evaluation because results of two different people are compared in the evaluation.It is possible that the difference in the two people, instead of the tool proposed in this paper, leads to the difference in their results. In order to minimize the threat, we evaluated candidate engineers in previous refactoring practice, and selected two whose efficiency and effect were close.

V. RELATED WORK

Long Method as a typical kind of bad smells, has been investigated for a long time. Fowler et al. [1] made a detailed analysis on *Long Method*, explaining why long methods are hard to read or maintain, how to refactoring long methods with refactoring rule *Extract Method*, and what the benefit

and cost of refactoring are. Refactoring rule *Extract Method* was also motivated and explained by Wake [7].

A lot of work has been done on how to extract selected fragments from long methods. As a result, most of the mainstream IDEs provide automatic support for *Extract Method*. For example, both Microsoft's *Visual Studio* [8] and *Eclipse* [9] fulfil the refactoring.

However, works on identifying which fragments should be extracted from long methods are rarely mentioned. Tsantalis and Chatzigeorgiou [10] applied the technology of program slicing [11] on methods to identify refactoring opportunities for *Extract Method*. Program slicing usually extract all computation on one variable as a slice. As a result, the approach [10] based on program slicing usually identifies computation on one variable as extractable fragments. One of the problems is that it is hard to guarantee that slice extraction would not change the external behaviors of the program. On the contrary, our approach extract uninterrupted statements, and thus it is easier to guarantee the equivalence of the external behaviors of the programs before and after refactoring. Another problem with their approach is that the decomposition is not responsibility oriented. It is variable oriented, instead. Consequently, the slice may seem independent and extractable (the new method needs few, or even no parameters), but its fulfills no meaningful and complete responsibility, and thus hard to be accept as a method.

VI. Conclusion

Long methods are hard to read, maintain, or reuse, and thus are usually treated as bad smells. Though existing refactoring tools have automate the extraction of selected fragments, they fail to automate the identification of fragments to be extracted. The identification is time consuming and error-prone if done manually. In this paper, we propose an approach to facilitate the identification, by recommending and sorting candidate fragments. We also implement a prototype of the approach and evaluate it with an open source project. The evaluation results suggest that the approach may reduce refactoring cost by 40%. The main contribution includes an approach to facilitate the identification of fragments to be extracted from long methods, a prototype implementation, and an initial evaluation.

As discussed in Section IV-D2, semantics and responsibilities of fragments may also influence the decomposition of long methods besides coupling among fragments. Consequently, we plan to investigate how to integrate this information into the proposed approach to improve its accuracy.

Acknowledgements

We would like to express our gratitude to all those who helped us during the writing of this paper. Grateful acknowledgement is made to the students in the laboratory, they gave us many help during the paper preparation. In addition, the authors deeply appreciate the anonymous reviewers for their valuable comments and suggestions.

The work is funded by the National Natural Science Foundation of China No.60773152, the National Grand Fundamental Research 973 Program of China No.2005CB321805, the National High-Tech Research and Development Plan of China No.2007AA01Z127 and 2007AA010301.

References

[1] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison Wesley Professional, 1999.

[2] G. Miller, "The magical number seven, plus or minus two: Some limits on our capacity for processing information," *Psychological Review*, vol. 63, pp. 8–97, 1956.

[3] "http://help.eclipse.org/help33/topic/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/statement.html."

[4] *Rationale for International Standard Programming Languages C*, C99 Standard, Rev. 5.10, 2003.

[5] OSoft, "Thout Reader, http://sourceforge.net/projects/thout/."

[6] OSoft, "Thout Reader 1.8.0 Source release, http://sourceforge.net/project/showfiles.php?group_id=101817&package_id=113253&release_id=329516."

[7] W. C. Wake, *Refactoring Workbook*. Addison Wesley, August 2003.

[8] "http://msdn.microsoft.com/en-us/library/0s21cwxk(vs.80).aspx."

[9] R. Enns, "Refactoring in Eclipse," Department of Computer Science, University of Manitoba, Winnipeg, Manitoba, Canada, Tech. Rep., February 25, 2004.

[10] N. Tsantalis and A. Chatzigeorgiou, "Identification of extract method refactoring opportunities," March 2009, pp. 119–128.

[11] M. Weiser, "Program slicing," *IEEE Transactions on Software Engineering*, vol. 10, no. 4, pp. 352–357, 1984.