How to Extract Differences from Similar Programs? A Cohesion Metric Approach

Akira Goto[1], Norihiro Yoshida[2], Masakazu Ioka[1], Eunjong Choi[1], Katsuro Inoue[1]
[1]Graduate School of Information Science and Technology, Osaka University, Japan
{a-gotoh, m-ioka, ejchoi, inoue}@ist.osaka-u.ac.jp
[2]Graduate School of Information Science, Nara Institute of Science and Technology, Japan
yoshida@is.naist.jp

***Abstract*—Merging similar programs is a promising solution to improve the maintainability of source code. Before merging programs, any syntactic difference has to be extracted as a new method. However, it is difficult for a developer to identify and extract differences from programs appropriately because he/she has to consider not only syntactic and semantic correctness but also the modularity of the programs after merging. In this paper, we propose a slice-based cohesion metrics approach to suggesting the extractions of differences from similar Java methods. This approach identifies syntactic differences from two methods, and then suggests sets of cohesive regions including those differences. The case study shows that the proposed approach can suggest refactorings that not only merge two methods but also increase the cohesiveness.**

I. Introduction

A method in source code often has one or more similar methods to it in the source code[1][2]. Merging similar methods is one of refactoring activities to improve the maintainability of source code[3].

Fig. 1 illustrates merging of similar methods. In merging similar methods, developers decompose them into multiple new methods, each of which represents the common or the different parts of the similar methods. "Form template method" refactoring is a repeated solution to merge similar methods[3]. During the decomposition of similar methods, developers have to consider the cohesiveness of new methods extracted from the similar methods (see step(2) in Fig. 1) because cohesiveness is a main factor of the maintainability of source code [4].

Refactoring support to recommend code transformations is needed for developers who merge similar methods because it is difficult to identify and extract cohesive fragments simultaneously as new methods from similar methods. A method extraction in one of similar methods should be simultaneously performed with the corresponding method extraction in the other similar method.

So far, several approaches have been proposed on refactoring of similar methods. Those existing approaches [5], [6], [7] provide the locations of similar methods, or the identification of the differences between them, however, and little consideration for the cohesiveness of new methods extracted from similar methods. Therefore, it is difficult for developers to identify code transformations to improve the maintainability of the source code.

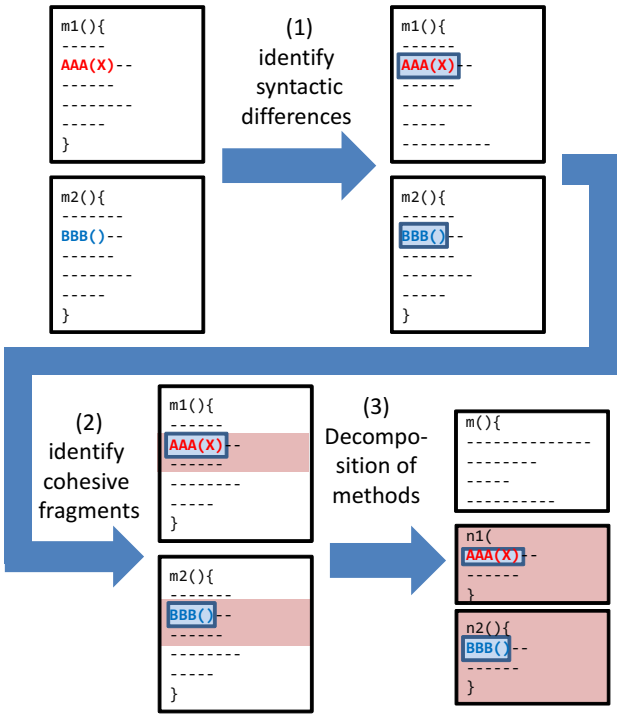

Fig. 1. Decomposition of similar methods

In this paper, we propose a cohesion metric approach to recommending and ranking method extractions for merging similar methods (Fig.2). This approach aimed to help developers who transform similar methods into cohesive methods desired for understanding and maintaining source code. At first, the proposed approach accepts a pair of similar methods, and then detects syntactic differences between them. After that, "Extract method (EM)" candidate sets are identified, each of which extracts cohesive methods including all of the syntactic differences. Finally, EM candidate sets are ranked according to cohesiveness.

Also, we have developed a tool for the identification and cohesion-based ranking of method extractions for merging similar methods (see Fig.9), and then conducted a case study of a pair of similar methods in an open source software system Ant 1.7.0. The result indicate the proposed approach is promissing to suggest refactorings that not only preserve

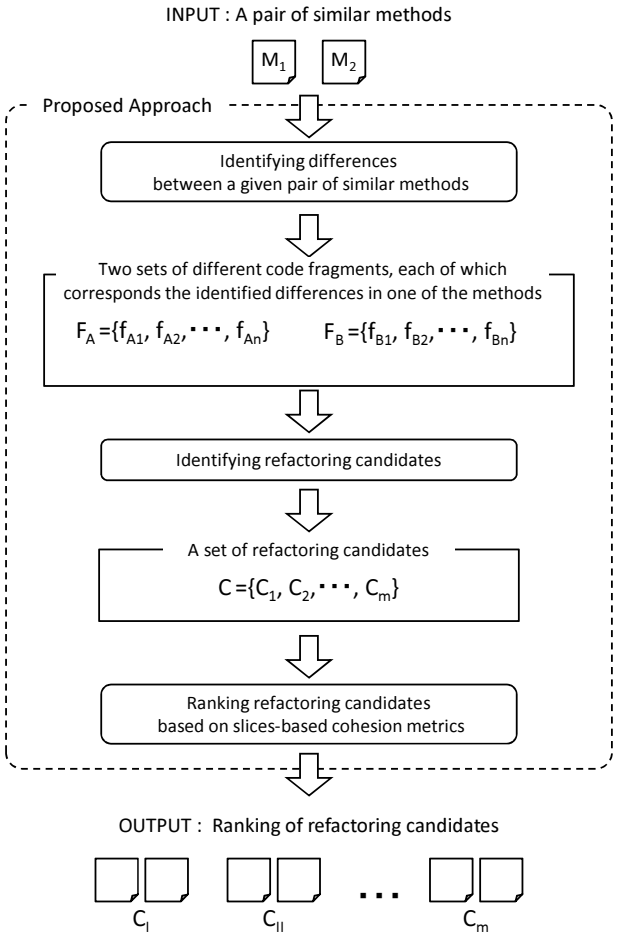

Fig. 2. Overview of the proposed approach

external behavior but also increase the cohesiveness of related source code.

The contributions of this study are as follows:

- To help developers who merge similar methods, we propose an approach to differentiating a given pair of methods in source code, and ranking differences according to each of cohesion metrics for code fragments.
- As a case study, we applied the proposed approach into a pair of similar methods in an open source software system. The result shows that the proposed approach increased two out of the three slice-based cohesion metrics.

II. Identifying Extract Method candidates

In this section, we propose an approach to identifying "Extract Method"(EM) candidates for merging similar methods.

The identification of "Extract Method" candidates is comprised of the following three steps:

1) Build Abstract Syntax Trees (ASTs) of a given pair of similar methods
2) Detect differences from the ASTs
3) Identify cohesive code fragments as EM candidates, each of which includes at least one detected difference

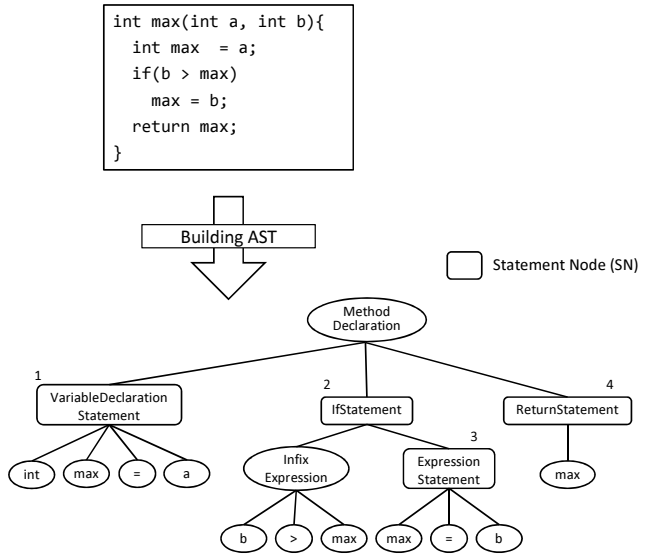

Fig. 3. An example of building an AST in this paper (Each number is assigned to *SNs* in the same order as breadth-first traversal starting with a root)

A code fragment is denoted as 5-tuples (*ID*, *SL*, *SC*, *EL*, *EC*) where *ID* denotes unique number of a given source file, *SL* denotes start line, *SC* denotes start column, *EL* denotes end line, and *EC* denotes end column.

A. Building ASTs

We build a pair of ASTs corresponding to a given pair of similar methods (M_A, M_B) using Eclipse JDT[1]. Fig.3 illustrates an example of a part of an AST. Please note that we denote a node that represents a statement in source code as Statement Node (*SN*), and each number is assigned to *SNs* in the same order as breadth-first traversal starting with a root.

B. AST Differencing

The AST differecing in the proposed approach performs statement-level differentiation because it is aimed at finding code fragments for 'Extract method' refactoring. The result of token-level differentiation [5] sometimes includes small-scale differences that are less-attractive for method extraction.

Step 1: Node-level differencing of a pair of ASTs

The differentiation between a pair of ASTs as shown in Algorithm 1. In the case of the comparison between two nodes that represent block statements including multiple child nodes, The *DPMatching(N_A,N_B)* function is invoked for the determination of correspondence between child nodes in the two nodes. The detemination is performed by dynamic programming-based approximate string matching [8].

Step 2: Identifying subtrees including node-level differences

This step identifies subtrees, each of which includes at least one node-level difference determined in Step 1, and derives the most minimum (deepest) subtree including each node-level difference. The traversal starts with each δ in the all of the

[1]http://www.eclipse.org/jdt/index.php

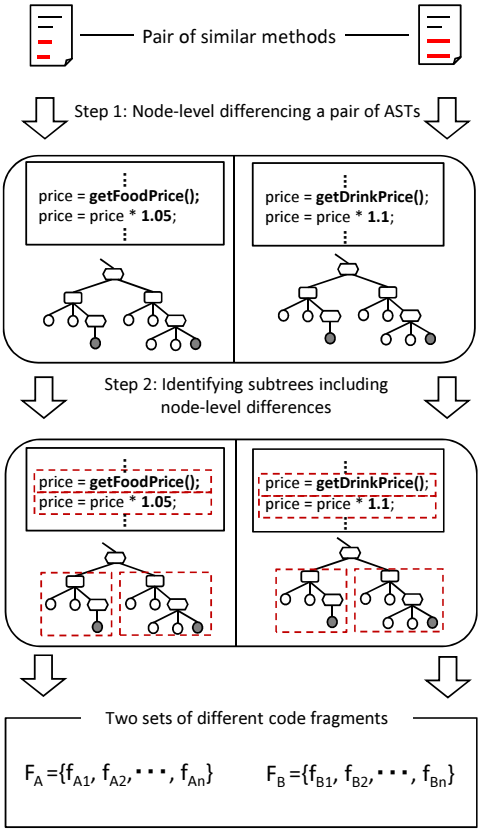

Fig. 4. Detecting syntactical-diffrences between a pair of similar methods

Algorithm 1 $comparison(N_A, N_B)$

input:N_A and N_B are node of ASTs of M_A and M_B
output: Sets of different nodes Δ_A and Δ_B

if $N_A.label \neq N_B.label$ **then**
  $\Delta_A \leftarrow N_A$
  $\Delta_B \leftarrow N_B$
else
  if $N_A.label == Block$ **then**
    $DPMatching(N_A, N_B)$
  else
    if $N_A.childNum == N_B.childNum$ **then**
      for $i = 0$ **to** $i = N_A.childNum$ **do**
        $comparison(N_A.child[i], N_B.child[i])$
      end for
    else
      $\Delta_A \leftarrow N_A$
      $\Delta_B \leftarrow N_B$
    end if
  end if
end if

node-level differences identified in Step 1, and then finds the nearest parent node that represents a Statement Node (*SN*). When the root of a subtree is the nearest parent node, this subtree is identified as the subtree including δ. Hereafter, $s_{Ai} \in$

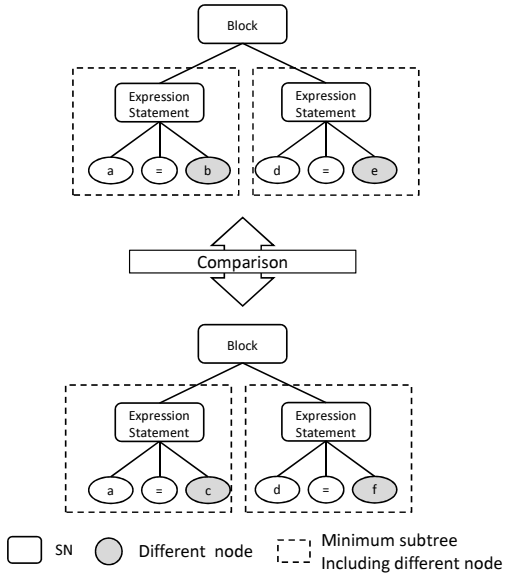

Fig. 5. Detecting different statements

S_A and $s_{Bi} \in S_B$ denote two subtrees including differences $\delta_{Ai} \in \Delta_A$ and $\delta_{Bi} \in \Delta_B$ respectively, where Δ_A and Δ_B denote the two sets of node-level differences between the M_A and M_B.

C. Detecting code fragments for Extract method refactoring

We explain definitions of preconditions of the Extract method refactoring (Definition 3.2) and the Extract method candidates for merging similar methods (Definition 3.3).

Definition 3.2 (Preconditions of the Extract method refactoring) Murphy-hill et al. defined three preconditions of the Extract method refactoring [9].

1) **Condition 1** Within the selection, there must be no assignments to variables that might be used later in the flow of execution. For Java, this can be relaxed to allow assignment to one variable, the value of which can be returned from the new method.
2) **Condition 2** Within the selection, there must be no conditional returns. In other words, the code in the selection must either always return, or always flow beginning to end.
3) **Condition 3** Within the selection, there must be no branches to code outside of the selection. For Java, this means no break or continue statements, unless the selection also contains their corresponding targets.

Definition 3.3 (The Extract method candidates for merging similar methods (EM candidates)) EM candidates are defined as a pair of two sets of code fragments $C = (E_A, E_B)$, E_A and E_B are two sets of code fragments involved in the pair of similar methods (M_A, M_B). Each code fragment in the E_A and E_B must satisfy the following four conditions:

1) **Condition A** Each code fragment in the E_A and the E_B can be refactored (i.e., for every code fragment in the

E_A and the E_B satisfies the Conditions 1, 2, and 3 in Definition 3.2)

2) **Condition B** After each code fragment in the E_A and the E_B is extracted for refactoring, the old method remaining token sequences of M_1 and M_2 are exactly same.
3) **Condition C** Token sequences of any code fragments $e_{Ai}, e_{Aj} \in E_A (i \neq j)$ are not overlapped, and also token sequences of any code fragments $e_{Bi}, e_{Bj} \in E_B (i \neq j)$ are not overlapped.
4) **Condition D** For all $f_{Ai} \in F_A$, the e_{Aj} exists in E_A that include f_{Ai}, and also for all $f_{Bi} \in F_B$, the e_{Bj} exists in E_B that include f_{Bi}.

Fig.6 shows an example of an EM candidate. In this figure, the differences between the M_A and M_B are emphasized by the bold font. The E_A and E_B are highlighted with colors.

We explain the steps of enumerating all of EM candidates from a given pair of methods. The enumeration starts with the two sets of subtrees S_A and S_B derived from the M_A and M_B respectively, and then finds the all of EM candidates that satisfy the four conditions in Definition 3.3.

Let us assume that a set of subtree *ES* initialized with $\{s_1\}$, a first element of a set of subtree S_A (i.e. a subtree that corresponds to the code fragment cf). Also, the maximum and minimum numbers of root (*SNs*) of each subtree in *ES* is defined as i_{max} and i_{min}, respectively. The detailed steps of the enumeration are as follows:

1) **Step C.1:** In this step, a subtree is added to a set *ES* when the root of the subtree is *SN* and satisfies the following conditions: (1) An assigned number of the *SN* is $i_{max}+1$ or $i_{min}-1$ (2)the *SN* and a root of a subtree in the *ES* are sibilng nodes. This step is illustrated in the left part of the Fig.7.
2) **Step C.2:** This step is illustrated in the right part of the Fig.7. The traversal starts with the root of each subtree in the *ES*. When the nearest parent node that represents a *SN* is found, all of the subtrees in *ES* are removed. A subtree having *SN* as root is added to *ES*, and then the enumeration goes back to the Step C.1.
   On the other hand, when the nearest parent node that represents a *SN* is not found until reaching the root, this step is over.

III. Ranking EM Candidates

Developers need much effort to choose refactoring candidates if a large number of refactoring candidates are detected. Therefore, in our approach, detected refactoring candidates are ranked based on slice-based cohesion metrics. In general, cohesion is a measure which expresses, in order for each of the constituent parts within a module to realise a specific feature, the extent to which they work together [4]. The cohesion takes a high value for a module which implements a single feature, and conversely, a low value if it implements multiple features which are not related.

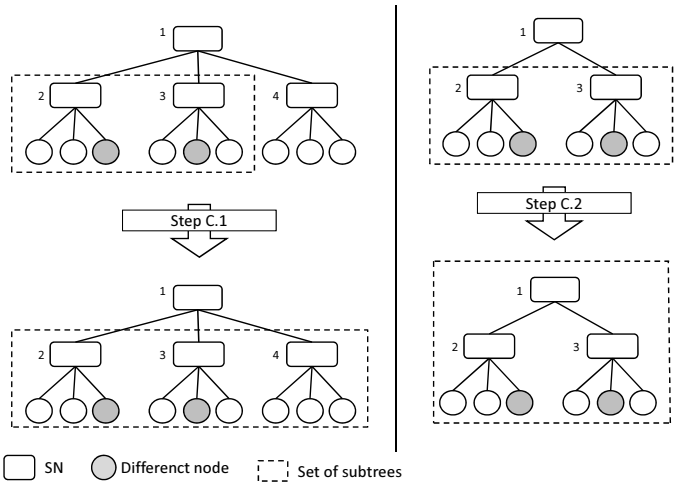

Fig. 7. Steps of extending nodes

We consider that each extracted method should have high cohesion. Therefore, slice-based cohesion metrics are used for ranking EM candidates in our approach.

A. Slice-Based Cohesion Metrics

At first, we explain Program Dependence Graph (PDG) and program slicing. PDG is a directed graph consisted of the vertices representing statements in a program and the edges representing dependences between the statements [10]. Program slicing is a technique to extract a set of statements related to a slicing criterion using PDG. A slicing criterion is tuple of a statement and a variable.

Weiser proposed five metrics based on program slicing to measure program cohesion [11]. Furthermore, Ott et al. showed the effectiveness of *Tightness*, *Coverage*, and *Overlap* in the metrics proposed by Weiser [12]. These three slice-based metrics are used with a little modification for our approach.

Slice-based cohesion metrics used in proposed approach are defined as follows: Let *M* be a method, *len(M)* is the number of statements in the *M*, V_i is a set of argument variables in the *M*, V_o is a set of output variables in the *M*, V is a union of V_i and V_o, FSL_x is a program slice that is calculated based on variable *x* using forward slicing, BSL_x is a program slice that is calculated based on variable *x* using backward slicing, and SL_{int} is an intersection of $FSL_x, x \in V_i$ and $BSL_x, x \in V_o$.

$$FTightness(M) = \frac{|SL_{int}|}{len(M)}$$

$$FCoverage(M) = \frac{1}{|V|}\left(\sum_{x \in V_i} \frac{|FSL_x|}{len(M)} + \sum_{x \in V_o} \frac{|BSL_x|}{len(M)}\right)$$

$$FOverlap(M) = \frac{1}{|V|}\left(\sum_{x \in V_i} \frac{|SL_{int}|}{|FSL_x|} + \sum_{x \in V_o} \frac{|SL_{int}|}{|BSL_x|}\right)$$

Fig.8 shows an example of computing of slice-based cohesion metrics. In this figure, a vertical bar represents that a statement is involved in a program slice, and $FTightness = 0.500$, $FCoverage = 0.722$ and $FOverlap = 0.750$.

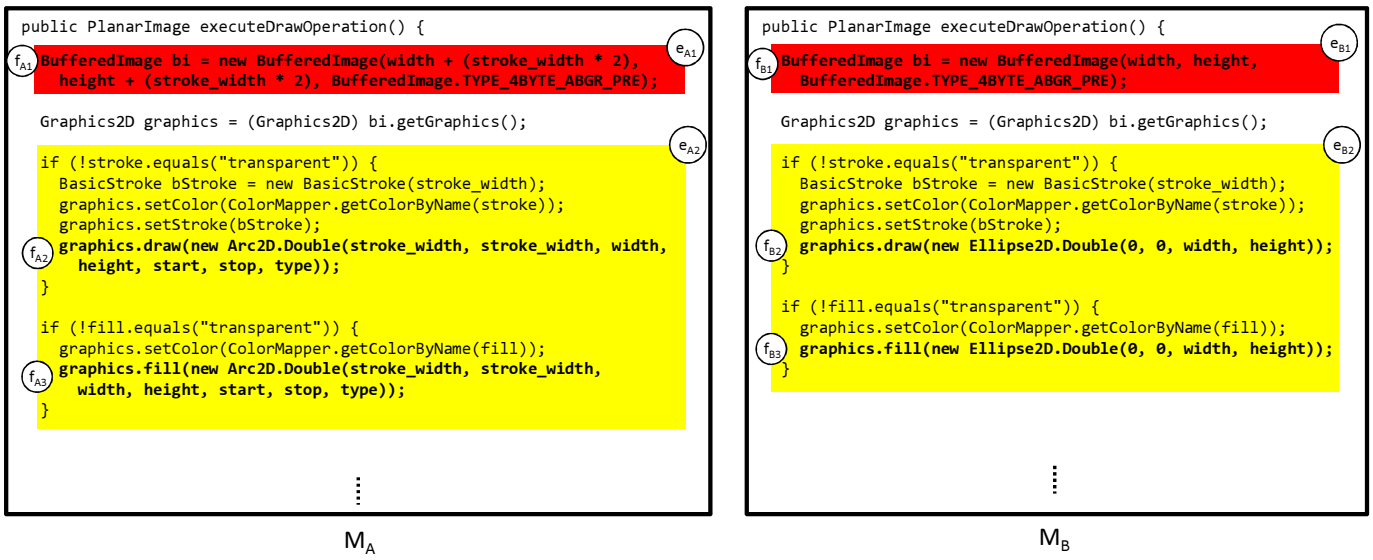

Fig. 6. An example of an EM candidate

		FSLa	FSLb	BSLres	SLint
1	int permutation(int a, int b) {				
2	int i;			|	
3	int res = 1;			|	
4	for (i = 0; i < b; i++) {		|	|	
5	res = res * a;	|	|	|	|
6	a = a – 1;	|	|	|	|
7	}				
8	return res;	|	|	|	|
9	}				

Fig. 8. Computing of slice-based metrics

B. Computing Cohesion for EM candidates

Cohesion of an EM candidate $C = (E_A, E_B)$ is defined as an average value of cohesion metrics of code fragments $e_{Ai} \in E_A$ and $e_{Bi} \in E_B$. Its cohesion is computed using slice-based metrics explained above. Detected EM candidates are ranked according to each metric respectively. As a result, three types of rankings are generated according to *FTightness*, *FCoverage*, and *FOverlap*.

IV. Implementation

The proposed approach is implemented as an Eclipse plugin. Building ASTs and checking whether a code fragment can be extracted as a method are implemented using Eclipse JDT. PDGs of a pair of similar methods are built by source code analysis tool, MASU [13]. A screenshot of implemented tool is shown in Fig.9. In this figure, a pair of similar methods are shown in both the left part and the right part. One of tabs in the upper part corresponds to an EM candidate. Each code fragment highlighted by colors can be extracted as a primitive method. A metric for ranking EM candidates can be chosen by the lower part buttons.

The similar methods in Fig.9 realize the rendering of a diagram. The red part realizes the allocation of a buffer, the green part realizes the rendering of the outline of the diagram, and the yellow part realizes the painting of the diagram. Each of the red, green, and yellow parts has high cohesion because each code fragment corresponds to a single functionality.

V. Case Study

As a case study, we applied the implemented tool into a pair of methods in an open source software Ant [2]. The names of the the two methods are the same *executeDrawOperation()* (see the methods in Fig. 9). Each of the two methods belong to the classes *Arc* and *Ellipse* in Ant, respectively.

The case study was aimed at confirming the changes of the slice-based cohesion metrics *Tightness*, *Coverage*, and *Overlap* [12] before and after "Form Template Method" refactoring using the implmented tool.

The implmented tool derived 34 sets of extract method(EM) candidates from Ant. We selected each top 10 sets of EM candidates (TOP10) from *FTightness*, *FCoverage*, and *FOverlap* rankings, respectively. As a result, the TOP10 from *FTightness* ranking is the same as *FCoverage* ranking.

And then we calculated the slice-based cohesion metrics *Tightness*, *Coverage*, and *Overlap* before and after refactoring using the implemented tool. Each refactoring extracted EM candidates in one of the TOP10s manually. The first author performed all of the refactorings.

Before refactoring, each of cohesion metrics *Tightness*, *Coverage*, and *Overlap* was completely the same value between the two methods *executeDrawOperation()* in the classes *Arc* and *Ellipse*. The cohesion metrics *Tightness*, *Coverage*, and *Overlap* were 0.4, 0,4 and 1.0, respectively.

[2]http://ant.apache.org/

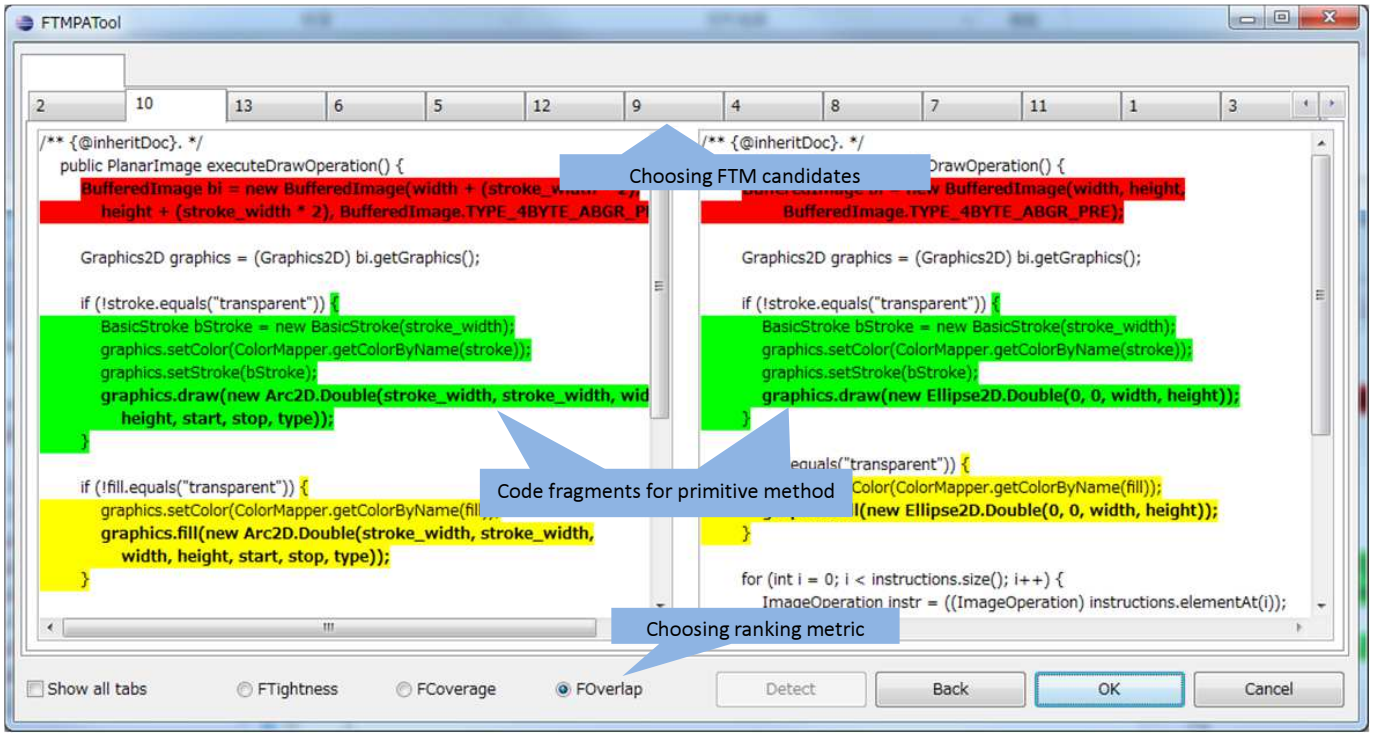

Fig. 9. A screenshot of the implemented tool

TABLE I
INCREASE RATE OF SLICE-BASED COHESION METRICS OF METHODS BETWEEN BEFORE AND AFTER REFACTORING

	Tightness			Coverage			Overlap		
	average	maximum	minimum	average	maximum	minimum	average	maximum	minimum
FTightness	22%	54%	5%	22%	54%	5%	0%	0%	0%
FCoverage	17%	25%	5%	17%	25%	5%	0%	0%	0%
FOverlap	22%	54%	5%	22%	54%	5%	0%	0%	0%

A. Result

Table I shows the increase rate of the cohesion metrics *Tightness*, *Coverage*, and *Overlap* of the methods (i.e., two *executeDrawOperation()* methods in *Arc* and *Ellipse* classes) between before and after refactoring. The leftmost column indicates that what kind of cohesion metrics was selected for ranking EM candidates from *FTightness*, *FCoverage*, and *FOverlap*. The "average", the "maximum" and the "minimum" in the table indicate the average, the maximum, and the minimum cohesion metric between each TOP 10, respectively.

In any case of TOP10 according to *FTightness*, *FCoverage*, and *FOverlap*, each of *Tightness* and *Coverage* always increased by extracting all of EM candidates in the set. Firstly,in the cases of *FTightness* and *FOverlap*, the average, the maximum, and the minimum increases of *Tightness*, *Coverage* were 22%, 54%, and 5%, respectively. Secondly, in the case of *FConverage*, average, maximum, and minimum increases of *Tightness*, *Coverage* were 17%, 25%, and 5%, respectively. Finally, in any case of *FTightness*, *FConverage* and *FOverlap*, *Overlap* metric was always unchanged from 1.0 after any refactoring in each TOP10.

For behavioral preservation, we confirmed that the results of all JUnit test suites in Ant are unaltered before and after all of the refactorings.

B. Discussion

According the result of the case study, the proposed approach with a TOP10 based on each of *FTightness*, *FCoverage*, and *FOverlap* can increase slice-based cohesion metrics *Tightness*, *Coverage*. Especially, the TOP10s based on *FTightness* and *FCoverage* lead higher increase rate. *Overlap* was unchanged in any case because the value was 1.0 before each refactoring.

Using JUnit test suites in Ant, we confirmed that the proposed approach is promissing to support refactoring that preserves the external behavior of the programs.

In the case study, we applied the proposed approach into only a pair of programs. The further case studies are needed to generalize the discussion.

VI. RELATED WORK

Juillerat et al. proposed an approach to automating Form Template Method refactoring (FTM) [5]. In this approach,

differences between a pair of similar methods are detected using ASTs, and then a code fragment corresponding to a subtree including the difference between a pair of similar methods is extracted as a primitive method. However, unlike the approach proposed in this paper, such kind of automatically determined code fragments do not necessarily have a single functionality. In addition, developers are able to select a candidate of FTM in the approach that we have proposed.

Hotta et al. proposed an approach that detecting FTM candidates [6]. In our approach, FTM candidates are ranked by slice-based cohesion metrics. By contrast, in the Hotta's approach, FTM candidates are only detected and presented to developers. Therefore, our approach is more effective when a large number of FTM candidates are detected.

Wang et al. proposed an approach that automatically divides a method into meaningful blocks based on data flows, control flows and syntax information [14]. In our approach, cohesion of a code fragment is measured using slice-based cohesion metrics. This is based on data dependence and control dependence of PDG. Using syntax information, we can improve the ranking algorithm in our approach.

VII. Summary

In this paper, we proposed a cohesion metric approach to recommending and ranking method extractions for merging similar methods. This approach aimed to help developers who transform similar methods into cohesive methods desired for understanding and maintaining source code. Also, we have developed a tool for the identification and cohesion-based ranking of method extractions for merging similar methods. As a case study, we applied the actual pair of similar methods in Ant 1.7.0. The result indicates that the proposed approach can suggest refactorings that not only preserve external behavior but also increase two (i.e., *Tightness* and *Coverage*) out of the three slice-based cohesion metrics.

As future work, we should conduct larger case studies to show the usefulness of the proposed approach. As a case study, we plan to ask developers to vote refactoring suggested by the proposed approach, and confirm the relationship between the voting result and the cohesion metrics-based ranking derived by the proposed approach. Also, we are interested in knowing how close the recommendations of the proposed approach are to refactoring instances during actual software development.

Acknowledgement

We are grateful to the anonymous reviewers for their useful comments.

References

[1] J. Mayrand, C. Leblanc, and E. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics," in *Proc. of ICSM*, 1996, pp. 244–253.

[2] Y. Sasaki, T. Yamamoto, Y. Hayase, and K. Inoue, "Finding file clones in freebsd ports collection," in *Proc. of MSR*, 2010, pp. 102–105.

[3] M. Fowler, *Refactoring:Improving the Design of Existing Code*. Addison Wesley, 1999.

[4] W. P. Stevens, G. J. Myers, and L. L. constatine, "Structured design," *IBM Systems Journal*, vol. 13, no. 2, pp. 115–139, 1974.

[5] N. Juillerat and B. Hirsbrunner, "Toward an implementation of the "form template method" refactoring," in *Proc. of SCAM*, 2007, pp. 81–90.

[6] K. Hotta, Y. Higo, and S. Kusumoto, "Identifying, tailoring, and suggesting form template method refactoring opportunities with program dependence graph," in *Proc. of CSMR*, 2012, pp. 53–62.

[7] Z. Xing, Y. Xue, and S. Jarzabek, "Clonedifferentiator: Analyzing clones by differentiation," in *Proc of ASE 2011*, 2011, pp. 576–579.

[8] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*, 2nd ed. Addison Wesley, 2011.

[9] E. Murphy-hill and A. P. Black, "Breaking the barriers to successful refactoring: observations and tools for extract method," in *Proc. of ICSE*, 2008, pp. 421–430.

[10] J. Ferrante, K. J. Ottenstein, and J. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, 1987.

[11] M. Weiser, "Program slicing," in *Proc. of ICSE*, 1981, pp. 439–449.

[12] L. M. Ott and J. J. Thuss, "Slice based metrics for estimating cohesion," in *Proc. of METRICS*, 1993, pp. 71–81.

[13] Y. Higo, A. Saitoh, G. Yamada, T. Miyake, S. Kusumoto, and K. Inoue, "A pluggable tool for measuring software metrics from source code," in *Proc. of IWSM-MENSURA*, 2011, pp. 3–12.

[14] X. Wang, L. Pollock, and K. V. Shanker, "Automatic segmentation of method code into meaningful blocks to improve readability," in *Proc. of WCRE*, 2011, pp. 35–44.