Identifying, Tailoring, and Suggesting Form Template Method Refactoring Opportunities with Program Dependence Graph

Keisuke Hotta, Yoshiki Higo, Shinji Kusumoto
Graduate School of Information and Science Technology, Osaka University, Japan
1-5, Yamadaoka, Suita, Osaka, 565-0871, Japan
{*k-hotta, higo, kusumoto*}*@ist.osaka-u.ac.jp*

***Abstract*—Many research efforts have been performed on removing code clones. Especially, it is highly expected that clone removal techniques by applying Form Template Method have high applicability because they can be applied to code clones that have some gaps. Consequently some researchers have proposed techniques to support refactoring with Form Template Method. However, previous research efforts still have some issues. In this paper, we propose a new technique with program dependence graph to resolve these issues. By using program dependence graph, we can handle trivial differences that are unrelated to behavior of a program. Consequently the proposed method can suggest more appropriate removal candidates than previously proposed techniques.**

***Keywords*-Code Clones, Refactoring, Form Template Method, Program Dependence Graph, Software Maintenance**

I. Introduction

Recently, code clones have received much attention. Code clones are identical or similar code fragments in source code. Code clones are generated by various reasons such as copy-and-paste operations. Recent studies have revealed that a portion of code clones has negative impacts on software maintenance [1], [2]. The reason is that, if we modify a code fragment, it is necessary to check whether its cloned code fragments need the same modifications or not. Consequently, many research efforts have been performed on detecting code clones and removing them by applying some refactorings [3], [4].

Some researchers have proposed code clone removal techniques by applying *Form Template Method* [5]. *Form Template Method* is one of the refactoring patterns proposed by Fowler et al. [6]. *Form Template Method* uses *Template Method* that is one of the design patterns proposed by Gamma et al. [7]. In *Template Method*, developers write an outline of the process into the base class and implement the details of the process in the derived classes. In order to apply this pattern to similar methods that have a common base class, duplicated code fragments between the methods are pulled up to the base class, and non-duplicated code fragments remain in its derived classes. As a result, code clones in the similar methods are merged into the base class. Comparing to other clone removal techniques, these techniques are suitable to handle differences between target methods. However, previously proposed techniques remain some issues that they cannot suggest code clones as removal candidates if they have the following differences even if they have no impact on the behavior of the program.

- Different order of code fragments.
- Different implementation styles (such as for- and while-loops).

In this paper, we propose a new technique to support applications of *Form Template Method* with program dependence graph to resolve these issues. The proposed approach supports identifying the places to be refactored and applying the refactoring in refactoring activities proposed by Mens et al. [8].

II. Motivation

A. Related Work

Recently, many researchers have proposed techniques assisting refactorings [8]. Fowler et al. said that duplication of source code is a “bad smell” on software maintenance [6], and many research efforts have been performed on removing code clones by applying refactorings in this basis [3], [4], [9], [10]. However, it has been debated in recent years whether code clones really have negative impacts on software maintenance or not [1], [2], [11]. At present, there is no consensus of this issue because the results of these studies vary according to research methods or target software systems.

Many research efforts have been performed on search based software engineering in recent years [12]. One of the research areas in it is search based refactoring [13], [14]. Some researchers have pointed out that the final design after refactoring can be affected with the order of refactoring activities. Therefore, some techniques are proposed to optimize refactoring schedules to maximize the benefits and minimize the efforts of refactorings [15], [16].

The majority of clone removal techniques is based on *Extract Method* or *Pull-Up Method* refactorings, and there are few techniques based on *Form Template Method* refactoring. Juillerat et al. proposed a method to automatically apply *Form Template Method* to a pair of similar methods with abstract syntax tree [5]. Their method can show source code after the application of the pattern, and the execution time and memory space required to the calculation are not so high. However, their method cannot handle trivial differences that do not have any impacts on the behavior of programs, such as differences between user-defined identifiers, different order of code fragments, and different implementation styles (such as the difference of for- and while-loops).

B. Aim of This Study

There are some techniques that assist developers in applying *Form Template Method* refactorings [5]. However, they remain some issues that these techniques cannot handle trivial differences that have no impact on the behavior of the program such as the order of the program statements and differences of for- and while-loops. In the example shown in Figure 1, there is a difference of the order of code fragments, and there is also a difference of the implementation style of loop statements. However, these differences do not influence the meanings of the program. The only meaningful difference is the ways of calculations of variable *points*. Nevertheless the previous methods regard these trivial differences as gaps between the two methods and suggest only four lines as duplicate statements in the two methods. In this study, we aim to improve these issues by using program dependence graph, and we will suggest 11 lines except the calculation of *points* as duplicate statements.

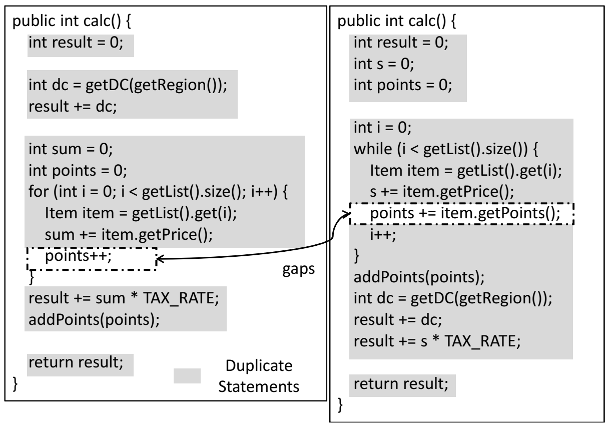
In addition, users need to specify target methods for using the previous techniques. This approach is useful for actual modifications in source code associated with refactoring activities. However, this approach cannot reduce efforts for identifying opportunities on which users want to apply refactorings. Because we think users have to pay many efforts to identify refactoring candidates, we aim to support both detection of refactoring candidates and actual modifications of source code. The proposed method detects refactoring candidates automatically and suggests all the candidates to users. Consequently, the proposed method can suggest refactoring candidates of which users are not aware. The proposed method also suggests code fragments to be merged in each of refactoring candidates, which makes it possible to reduce efforts required to modify source code to apply *Form Template Method* refactorings.

Note that the proposed method aim not to suggest candidates that should be refactored but to suggest candidates that can be refactored. The reason is that there is no strict and generic standard to judge whether code clones should be removed, and to judge whether *Form Template Method* should be used to remove code clones. Accordingly, we leave such decisions to users whether they need to apply refactorings on each candidate that our method suggests.

III. Preliminaries

A. Form Template Method

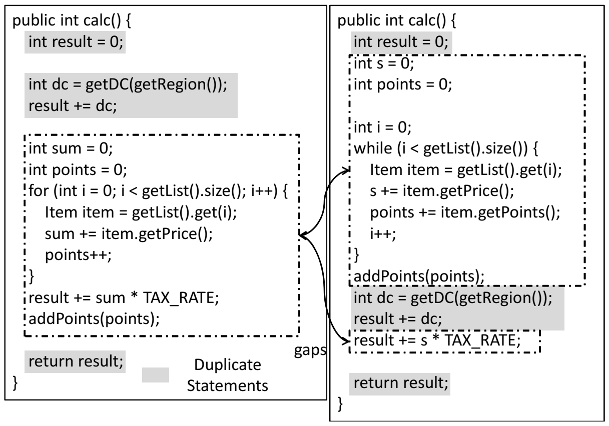
Form Template Method is one of the refactoring patterns proposed by Fowler et al. [6]. In this refactoring pattern, developers write an outline of the process in a base class, and the base class delegates implementations of the details to each of its derived classes. This pattern can be applied to remove code clones by pulling up code clones into the base class as a common process.



```
public int calc() {
  int result = 0;

  int dc = getDC(getRegion());
  result += dc;

  int sum = 0;
  int points = 0;
  for (int i = 0; i < getList().size(); i++) {
    Item item = getList().get(i);
    sum += item.getPrice();
    points++;
  }
  result += sum * TAX_RATE;
  addPoints(points);

  return result;
}
```

```
public int calc() {
  int result = 0;
  int s = 0;
  int points = 0;

  int i = 0;
  while (i < getList().size()) {
    Item item = getList().get(i);
    s += item.getPrice();
    points += item.getPoints();
    i++;
  }
  addPoints(points);
  int dc = getDC(getRegion());
  result += dc;
  result += s * TAX_RATE;

  return result;
}
```

(a) The Proposed Method



```
public int calc() {
  int result = 0;

  int dc = getDC(getRegion());
  result += dc;

  int sum = 0;
  int points = 0;
  for (int i = 0; i < getList().size(); i++) {
    Item item = getList().get(i);
    sum += item.getPrice();
    points++;
  }
  result += sum * TAX_RATE;
  addPoints(points);

  return result;
}
```

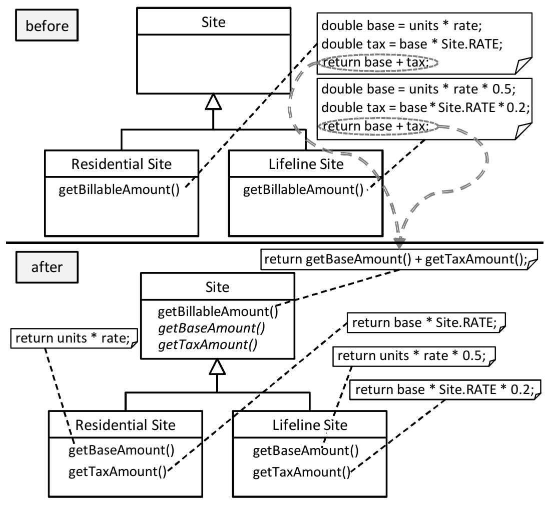
```
public int calc() {
  int result = 0;
  int s = 0;
  int points = 0;

  int i = 0;
  while (i < getList().size()) {
    Item item = getList().get(i);
    s += item.getPrice();
    points += item.getPoints();
    i++;
  }
  addPoints(points);
  int dc = getDC(getRegion());
  result += dc;
  result += s * TAX_RATE;

  return result;
}
```

(b) The Method Proposed by Juillerat et al.[5]

Figure 1. A Motivating Example

Figure 2 shows an example of application of *Form Template Method* [6]. There are two classes that have the same base class, *Site*, and these two classes have the methods that are similar to each other, *getBillableAmount*. By applying *Form Template Method* to these methods, the common code fragments are pulled up into the base class, and the unique code fragments in each method are extracted as new methods, *getBaseAmount* and *getTaxAmount*. We call the new method written in the base class as *template method*. By this transformation, code clones in the methods are merged into the template method, and the unique code fragments in each method are handled well by creating new methods without changing the behavior of the program.

In this paper, we call common code fragments that should be pulled up to base classes as a **common process**, and



```
double base = units * rate;
double tax = base * Site.RATE;
return base + tax;
```

```
double base = units * rate * 0.5;
double tax = base * Site.RATE * 0.2;
return base + tax;
```

```
return getBaseAmount() + getTaxAmount();
return units * rate;
return base * Site.RATE;
return units * rate * 0.5;
return base * Site.RATE * 0.2;
```

Figure 2. An Example of the Application of *Form Template Method*

unique code fragments that should remain in each of derived classes as a **unique process** instead.

B. Program Dependence Graph

Program Dependence Graph [17] (in short, **PDG**) is one of the directed graphs representing dependence between elements of program (such as statements and conditional predicates). Dependence in PDG are classified into the following two categories.

Data Dependence: There is data dependence from statement s to statement t, if a value is assigned to variable x in s, and t references x without changing the value of x.

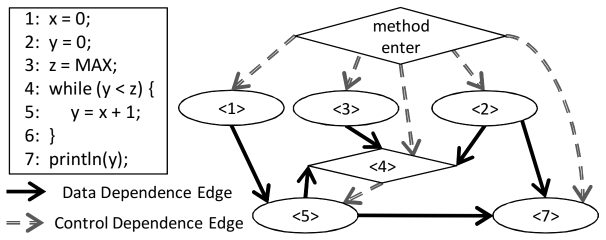
Control Dependence: There is control dependence from statement s to statement t, if s is a conditional predicate, and it directly determines whether t is executed or not.

Figure 3 shows an example of PDG. In this example, there is data dependence from the 2nd, 3rd, and 5th lines to the 4th line because variables y and z are referenced in the 4th line. There is control dependence from the 4th line to the 5th line because the conditional predicate in the 4th line directly controls the execution of the 5th line. In addition, there is a node labeled "*method enter*" that means the enter node of the method. In general, PDG contains a method enter node, and, in this case, there is control dependence from the enter node to all nodes that are directly contained by the method.

Moreover, we tailor PDGs to trace state changes of objects. The state of an object changes when the values of its fields change [18]. Figure 4 shows the tailoring. By applying this tailoring to PDGs, we can get the order of operations to objects from PDG. This enables us to apply refactorings with the operation order of objects preserved.

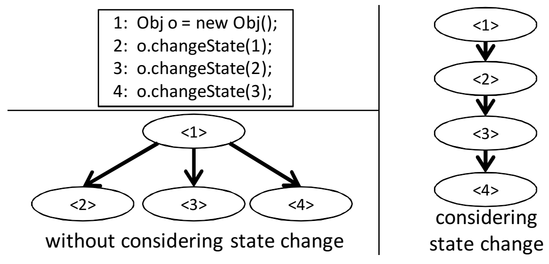
C. Code Clones Detection by Program Dependence Graph

There are many techniques to detect code clones automatically. They can be categorized by its data structures



```
1:  x = 0;
2:  y = 0;
3:  z = MAX;
4:  while (y < z) {
5:      y = x + 1;
6:  }
7:  println(y);
```

Figure 3. An Example of PDG



```
1:  Obj o = new Obj();
2:  o.changeState(1);
3:  o.changeState(2);
4:  o.changeState(3);
```

Figure 4. PDG Considering State Changes of Objects

[19]. PDG-based detection is one of the categories, and some researchers have proposed PDG-based detection techniques [20], [21], [22]. This technique detects isomorphic subgraphs on PDGs as code clones. PDG-based detection can detect code clones containing different order of statements, and trivial differences that do not have any impacts on the behavior of the program such as a difference between for- and while-loops.

In this paper, we use Scorpio [22] to detect code clones. Scorpio is one of the PDG-based detection tools, and it was developed by our research member. Scorpio can detect code clones that have differences of user-defined identifiers.

IV. Proposed Method

A. Inputs and Outputs

The proposed method takes the source code of the target software systems as its input. Then, the proposed method detects all the candidates that can be refactored with *Form Template Method* and suggests them to users. Note that we regard a pair of methods as a candidate of refactoring likewise the previous method. For each of refactoring candidates, the proposed method also suggests program statements that can be merged into the base class as the common process, and program statements that should be remain in the derived classes as the unique process. Additionally, for the unique process, the proposed method suggests the following two information.

- Sets of program statements that should be extracted as a single method.
- Pairwise relationships of new methods created in the pair of the derived classes between the couple of the

methods. Note that the new methods in a pairwise relationship can be extracted as methods whose signatures are the same as each other.

B. Definitions

In this subsection, we describe definitions of terms referenced in the following explanations.

A directed graph G is represented as $G = (f, V, E)$, where V is a set of nodes, E is a set of edges, and f is a map from edges to ordered pairs of nodes ($f : E \to V \times V$). In this paper, we write the set of nodes in G as V_G, the set of edges in G as E_G, and the map between edges and ordered pairs of nodes in G as f_G respectively.

A PDG of a program is one of the directed graphs. Given a PDG $G = (f, V, E)$, a node of G corresponds to an element of the program, and an edge of G corresponds to a dependence between two elements. Note that we build a PDG in each of methods, therefore every method has a corresponding PDG.

As described above, there are two types of dependence in PDGs.

Definition *4.1 (Dependence between Elements):* We write data dependence as *data*, and control dependence as *control*. We define *type* as a map from edges to the types of dependence that the edges represent ($type : E \to EdgeType$), where $EdgeType = \{data, control\}$. In addition, a data edge has information about the variable that the edge represents. We define $var(e_d)$ as the represented variable by a data edge e_d.

In the next, we define a tail of an edge $e \in E_G$ as $tail(e)$ and a head of e as $head(e)$. The definitions are as follows.

Definition *4.2 ($tail(e)$,$head(e)$):* We define $tail(e)$ as the first element of $f_G(e)$, and $head(e)$ as the last element of $f_G(e)$. In other words, $tail(e) := u$ and $head(e) := v$, where $f_G(e) = (u, v)$.

Herein, we define sets of edges $BackwardEdges(v)$ and $ForwardEdges(v)$ for $v \in V_G$. $BackwardEdges(v)$ is a set of edges whose head is v defined in the formula (1), and $ForwardEdges(v)$ is a set of edges whose tail is v defined in the formula (2).

Definition *4.3 ($BackwardEdges(v)$,$ForwardEdges(v)$):*

$$BackwardEdges(v) := \{e \in E_G \mid head(e) = v\} \quad (1)$$
$$ForwardEdges(v) := \{e \in E_G \mid tail(e) = v\} \quad (2)$$

Next, we describe definitions about code clones. We use Scorpio [22] to detect code clones. We can get isomorphic subgraphs in two PDGs given as its input data with Scorpio. We define a set of ordered pairs of isomorphic subgraphs detected with the technique as $ClonePairs(G_1, G_2)$, where G_1 and G_2 are PDGs given as its input data. The definition is as follows.

Definition *4.4 ($ClonePairs(G_1, G_2)$ and a clone pair):* We define $ClonePairs(G_1, G_2)$ with the formula (3), and we call every element of $ClonePairs(G_1, G_2)$ “clone pair”.

$$ClonePairs(G_1, G_2) := \{(G'_1, G'_2) \mid G'_1 \subset G_1 \wedge G'_2 \subset G_2 \wedge G'_1 \cong G'_2\} \quad (3)$$

where, G_1 and G_2 are PDGs given as input data, $G' \subset G$ indicates G' is a subgraph of G, and $G' \cong G''$ indicates G' and G'' are isomorphic subgraphs to each other.

We also define duplicate relationships on nodes of PDGs as follows.

Definition *4.5 (Duplication of nodes):* We define the two nodes $v_1 \in V_{G_1}$ and $v_2 \in V_{G_2}$ are duplicated to each other if they satisfy the formula (4). We represent $v_1 \sim v_2$ if v_1 and v_2 are duplicated to each other.

$$\exists (G'_1, G'_2) \in ClonePairs(G_1, G_2) \; [v_1 \in V_{G'_1} \wedge v_2 \in V_{G'_2} \wedge \varphi(v_1) = v_2] \quad (4)$$

where, G_1 and G_2 are PDGs, and φ indicates the isomorphism between G'_1 and G'_2 ($G'_1 \cong G'_2$).

C. Processing Flow

The processing flow of the proposed method is shown below.

STEP1: Create PDGs, and detect code clones in them.
STEP2: Identify pairs of methods that can be refactored by *Form Template Method*.
STEP3: Detect a common process and a unique process for each of method pairs.

In STEP1, we use the existing techniques. In the following subsections, we describe STEP2 and STEP3 in detail.

D. Identification of Refactoring Candidates

In this step, we detect pairs of methods that can be refactored by *Form Template Method* refactoring pattern with code clones detected by Scorpio. We regard a pair of methods as a refactoring candidate if it satisfies the following two requirements.

Requirement A: *Form Template Method* can be applied to the method pair.

Requirement B: There is at least one clone pair between the method pair.

We describe these requirements in detail in the following subsections.

1) Requirement A (Applicability of the Pattern): Form Template Method can be applied to methods whose owner classes have the same base class. Accordingly, owner classes of each of two methods in a refactoring candidate must have the same base class in the proposed method. In addition, we cannot apply *Form Template Method* to methods that are defined in the same class. Therefore, refactoring candidates

Algorithm 1 Removing Redundant Clone Pairs

```
Require: $ClonePairs(G_{m_1}, G_{m_2})$
Ensure: $ClonePairs(G_{m_1}, G_{m_2})$ after repaired
 1: for all $(G'_{m_1}, G'_{m_2}) \in ClonePairs(G_{m_1}, G_{m_2})$ do
 2:     for all $(G''_{m_1}, G''_{m_2}) \in ClonePairs(G_{m_1}, G_{m_2})$ do
 3:         if $\exists v_1 \in G'_{m_1}[v_1 \in G''_{m_1}]$ then
 4:             if $|G'_{m_1}| < |G''_{m_1}|$ then
 5:                 $ClonePairs(G_{m_1}, G_{m_2}) \overset{-}{\leftarrow} (G'_{m_1}, G'_{m_2})$
 6:             else
 7:                 $ClonePairs(G_{m_1}, G_{m_2}) \overset{-}{\leftarrow} (G''_{m_1}, G''_{m_2})$
 8:             end if
 9:         end if
10:         if $\exists v_2 \in G'_{m_2}[v_2 \in G''_{m_2}]$ then
11:             if $|G'_{m_2}| < |G''_{m_2}|$ then
12:                 $ClonePairs(G_{m_1}, G_{m_2}) \overset{-}{\leftarrow} (G'_{m_1}, G'_{m_2})$
13:             else
14:                 $ClonePairs(G_{m_1}, G_{m_2}) \overset{-}{\leftarrow} (G''_{m_1}, G''_{m_2})$
15:             end if
16:         end if
17:     end for
18: end for
```

in the proposed method must satisfy the following two requirements.

Requirement A-1: The two methods in the candidates are defined in different classes.

Requirement A-2: The owner classes of the two methods have the same base class.

2) Requirement B (Presence of Clone Pairs): If there is no clone pair between a method pair (m_1, m_2), $ClonePairs(G_{m_1}, G_{m_2})$ is empty, where G_{m_1} and G_{m_2} are PDGs of method m_1 and m_2. Therefore, we can check whether there is at least one clone pair by checking whether $ClonePairs(G_{m_1}, G_{m_2})$ is empty or not. In other words, the method pair m_1 and m_2 in the proposed method must satisfy the formula (5).

$$ClonePairs(G_{m_1}, G_{m_2}) \neq \emptyset \tag{5}$$

E. Tailoring Refactoring Candidates

For each of method pairs, we detect a common process, and a unique process in STEP3.

This step consists of the following three sub-steps.

STEP3-A: Detect a common process and a unique process in a given method pair.

STEP3-B: Detect sets of program statements, each of which should be extracted as a single method for the unique process.

STEP3-C: Detect pairwise relations of sets of program statements detected in STEP3-B in the method pair.

1) STEP3-A: Detection of Common and Unique Processes: In this sub-step, the proposed method detects a common process and a unique process in a given method pair. In the proposed method, we regard code clones in the method pair as the common process. We define $CommonNodes(G_{m_{1(2)}})$ as a set of nodes in $G_{m_{1(2)}}$ whose statements form the common process. The formula (6) represents the definition, where $G_{m_{1(2)}}$ indicates the PDG of method $m_{1(2)}$.

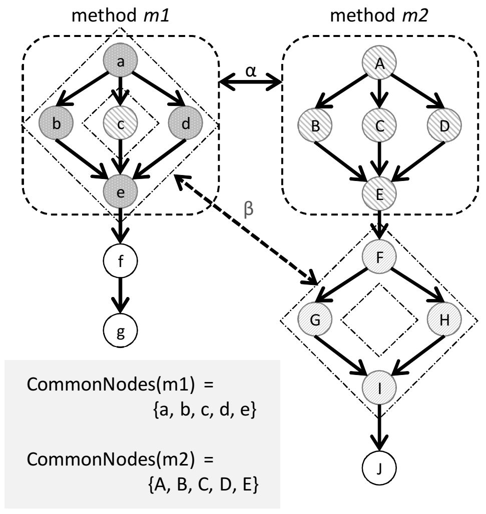

method m1
method m2
α
β
a
b
c
d
e
f
g
A
B
C
D
E
F
G
H
I
J
CommonNodes(m1) = {a, b, c, d, e}
CommonNodes(m2) = {A, B, C, D, E}

Figure 5. An Example of the Redundant Clone Pairs

$$CommonNodes(G_{m_{1(2)}}) := \{v \in V_{G_{m_{1(2)}}} \mid \exists w \in V_{G_{m_{2(1)}}}[v \sim w]\} \tag{6}$$

However, a node in $G_{m_{1(2)}}$ can be duplicated between two or more nodes in $G_{m_{2(1)}}$. In other words, the formula (7) can be satisfied in some cases, considering the two clone pairs $(G'_{m_1}, G'_{m_2}), (G''_{m_1}, G''_{m_2}) \in ClonePairs(G_{m_1}, G_{m_2})$.

$$\exists v \in V_{G'_{m_{1(2)}}}[v \in V_{G''_{m_{1(2)}}}] \tag{7}$$

In this case, we cannot merge all the nodes that are duplicate to other nodes in the other method. We remove some clone pairs from $ClonePairs(G_{m_1}, G_{m_2})$ to resolve this problem. Algorithm 1 shows the algorithm for removing clone pairs. Note that $|R|$ means the number of elements in a set R and $R \overset{-}{\leftarrow} r$ means the process to remove an element r from R.

By applying this algorithm, we can ensure that there is at most one duplicate node in the other method for all nodes in method m_1 and m_2. Nodes should be pulled up into the base class if they are contained in $CommonNodes(G_{m_{1(2)}})$ after this processing.

Figure 5 shows an example that contains redundant clone pairs. There are two clone pairs labeled α and β. The clone pair α consists of $(\{a, b, c, d, e\}, \{A, B, C, D, E\})$, and the clone pair β consists of $(\{a, b, d, e\}, \{F, G, H, I\})$. In this case, the algorithm selects α as the remaining clone pair, and remove β from $ClonePairs(G_{m_1}, G_{m_2})$ because the number of elements of α is larger than β's one. As a result, the common code fragments that the proposed method detects in this method pair ($CommonNodes(G_{m_{1(2)}})$) are $\{a, b, c, d, e\}$ and $\{A, B, C, D, E\}$ respectively.

On the other hand, the proposed method regards that program statements form a unique process in a given method pair if they are not included in the common process. We define $DiffNodes(G_{m_{1(2)}})$ as a set of nodes whose owner statements are not included in the common process. Formula (8) shows the definition of $DiffNodes(G_{m_{1(2)}})$.

$$DiffNodes(G_{m_{1(2)}}) := \{v \in V_{G_{m_{1(2)}}} \mid v \notin CommonNodes(G_{m_{1(2)}})\} \quad (8)$$

Note that nodes contained in $DiffNodes(G_{m_{1(2)}})$ need to remain in the class that has method $m_{1(2)}$.

2) STEP3-B: Detection of Statements Extracted as a Single Method: In this sub-step, the proposed method detects sets of statements that can be extracted as a single method in the unique process.

For applying *Form Template Method* refactorings, nodes remaining in derived classes have to be extracted as new methods. Therefore, we have to detect sets of code fragments included in $DiffNodes(G_{m_{1(2)}})$, each of which can be extracted as a single method.

In the proposed method, we regard nodes included in $DiffNodes(G_{m_{1(2)}})$ as a set that should be extracted as a single method, if there is at least one path that does not include nodes in $CommonNodes(G_{m_{1(2)}})$ for any pairs of the nodes in it. In other words, we regard a set of nodes $S_{m_{1(2)}}$ that is a subset of $V_{G_{m_{1(2)}}}$ as a set of nodes that should be extracted as a single method, if there is at least one path that satisfies the formula (9) for any two nodes $v_1, v_n (v_1 \neq v_n)$ in $S_{m_{1(2)}}$.

$$\forall i \in \{1 \dots n\}[v_i \in DiffNodes(G_{m_{1(2)}})] \quad (9)$$

We call a set of nodes that should be extracted as a single method an **Extract Node Set** (in short. **ENS**). In the example shown in Figure 6, we can find two ENSs. One consists of $\{d, g\}$ and the other consists of $\{b, c, h, k, l\}$. As shown in this example, each of methods in refactoring candidates can contain multiple ENSs. We define $DiffNodeSets(G_{m_{1(2)}})$ as a family of ENSs in method $m_{1(2)}$. Any node in $DiffNodes(G_{m_{1(2)}})$ must be included in a ENS in $DiffNodeSets(G_{m_{1(2)}})$ (formula (10)).

$$\forall v \in DiffNodes(G_{m_{1(2)}}) \, \exists S \in DiffNodeSets(G_{m_{1(2)}}) [v \in S] \quad (10)$$

3) STEP3-C: Detection of Pairwise Relationships of New Methods: In this sub-step, we detect pairwise relationships of ENSs in a given method pair. In other words, assuming that $\rightleftharpoons$ indicates the pairwise relationships and $S_{m_{1(2)}}$ is an ENS of method $m_{1(2)}$, for each

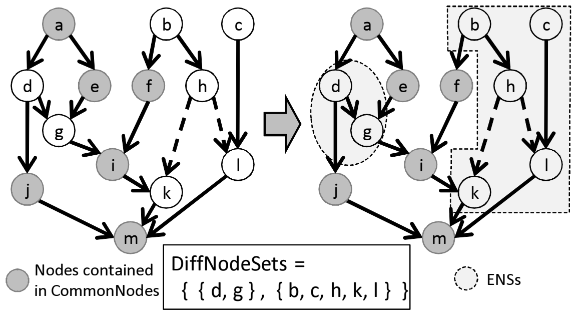

a
b
c
d
e
f
h
g
i
j
k
l
m
Nodes contained in CommonNodes
DiffNodeSets = { { d, g }, { b, c, h, k, l } }
ENSs

Figure 6. An Example of the Detection of ENSs

of $S_{m_{1(2)}} \in DiffNodeSets(G_{m_{1(2)}})$ we detect whether $S_{m_{2(1)}} \in DiffNodeSets(G_{m_{2(1)}})$ satisfies $S_{m_1} \rightleftharpoons S_{m_2}$ exists or not. Note that $S_{m_1} \rightleftharpoons S_{m_2}$ indicates that S_{m_1} and S_{m_2} can be extracted as methods whose signatures are the same as each other. If an ENS S has no correspondent in the other method, we have to make an empty method whose signature is the same as S in the derived class that does not have S.

We regard a pair of ENSs S_{m_1} and S_{m_2} as $S_{m_1} \rightleftharpoons S_{m_2}$ if they satisfy the following two requirements.

Requirement 3C-1: The types of return values of S_{m_1} and S_{m_2} are the same as each other.

Requirement 3C-2: The conditions to call the new methods created by extracting S_{m_1} and S_{m_2} are the same as each other.

Requirement of the Return Value*:* First, we define $OutputDataEdges(G, S)$ as a set of data edges whose tails are included in S and whose heads are not included in S, where G is a PDG and S is an ENS of G. The definition is shown in the formula (11).

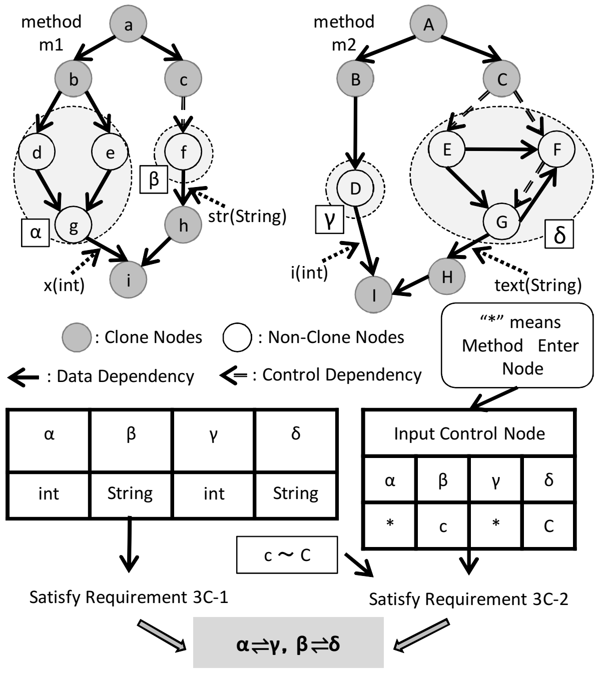
$$OutputDataEdges(G, S) := \{e \in E_G \mid tail(e) \in S \wedge head(e) \notin S \wedge type(e) = data\} \quad (11)$$

Now we can define a set of output variables of S by using this definition. We define $OutputVariables(G, S)$ in the formula (12).

$$OutputVariables(G, S) := \{p \mid \exists e \in OutputDataEdges(G, S)[p = var(e)]\} \quad (12)$$

We can judge whether ENSs S_{m_1} and S_{m_2} have the same types of return values by comparing $OutputVariables(G_{m_1}, S_{m_1})$ and $OutputVariables(G_{m_2}, S_{m_2})$.

Requirement of the Conditions for Call*:* Methods created by extracting ENSs S_{m_1} and S_{m_2} are called in the same conditions if the control dependence into S_{m_1} and S_{m_2} are the same as each other. Herein, we describe definitions to check this requirement.



α	β	γ	δ
int	String	int	String

Input Control Node			
α	β	γ	δ
*	c	*	C

Figure 7. The Detection of the Pairwise Relationships of ENSs

First, we define $\mathit{InputControlEdges}(G, S)$ as a set of control edges whose heads are included in S and whose tails are not included S, where G is a PDG and S is an ENS of G. The definition is shown in the formula (13).

$$\mathit{InputControlEdges}(G, S) := \{e \in E_G \mid (\mathit{tail}(e) \notin S) \wedge (\mathit{head}(e) \in S) \wedge (\mathit{type}(e) = \mathit{control})\} \tag{13}$$

In addition, we define $\mathit{InputControlNodes}(G, S)$ as a set of nodes that are tails of edges in $\mathit{InputControlEdges}(G, S)$ in the formula (14).

$$\mathit{InputControlNodes}(G, S) := \{v \in V_G \mid \exists e_c \in \mathit{InputControlEdges}(G, S)[v = \mathit{tail}(e_c)]\} \tag{14}$$

With these definitions, we define S_{m_1} and S_{m_2} are called in the same conditions if they satisfy the following two formulae (15) and (16).

$$|\mathit{InputControlNodes}(G_{m_1}, S_{m_1})| = |\mathit{InputControlNodes}(G_{m_2}, S_{m_2})| \tag{15}$$

$$\begin{aligned}&\forall v_1 \in \mathit{InputControlNodes}(G_{m_1}, S_{m_1})\\ &\exists v_2 \in \mathit{InputControlNodes}(G_{m_2}, S_{m_2})\\ &[(v_1 \sim v_2) \vee\\ &(\exists S'_1 \in \mathit{DiffNodeSets}(G_{m_1}) \exists S'_2 \in \mathit{DiffNodeSets}(G_{m_2})\\ &\qquad [S'_1 \rightleftharpoons S'_2 \wedge n_1 \in S'_1 \wedge n_2 \in S'_2])]\end{aligned} \tag{16}$$

An Example of Pairwise Relationships of ENSs*:* Figure 7 shows an example of the detection of pairwise relationships of ENSs between two methods. In this example, we can get two pairs of ENSs, $\alpha \rightleftharpoons \gamma$ and $\beta \rightleftharpoons \delta$.

V. Implementation

We have implemented the proposed method as a tool named **Creios** (*Clone Removal Expediter by Identifying Opportunities with Scorpio*) in Java. Creios can handle software systems written in Java, because Scorpio, the clone detection tool used in Creios, can handle only Java. However, the proposed method can be applied to other programming languages if PDGs are built.

In this section, we describe the functions of Creios.

A. Supporting Detection of Refactoring Candidates

Creios suggests all the candidates that can be refactored with *Form Template Method*. Therefore, the function that supports users to select suitable candidates for refactoring are required especially in the cases that there are a large number of candidates. Creios has a filtering function with some metrics to support users to select candidates (e.g. similarities between two methods and the number of new methods created by the refactoring).

B. Supporting Modifications of Source Code

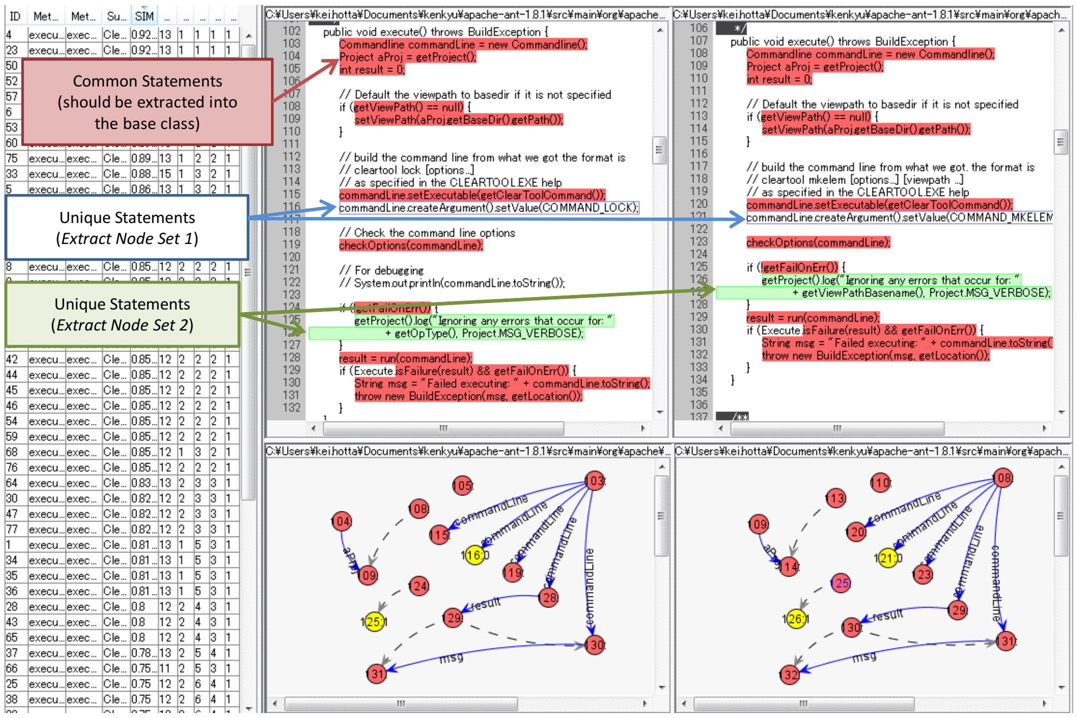
Creios shows the output information of the proposed method with GUI. Figure 8 shows a snapshot of Creios's output. For all the candidates, Creios shows PDGs and its source code for every candidate. In the source code view, Creios highlights duplicate statements in red, and draws color rectangles around the statements for each of ENSs. Note that the ENSs that are in the pairwise relationships are surrounded with the same color rectangles.

Creios does not have the function that modifies the actual source code automatically. Accordingly, users need to modify source code by themselves to apply *Form Template Method* refactoring.

VI. Experiment

In order to evaluate the proposed method, we conducted an experiment on two open source software systems.Table I shows the target software systems, the number of detected candidates, and elapsed time to execute Creios.

Figure 9 shows a refactoring candidate in Ant detected by Creios and the result of the refactoring. In this example, there is a base class, *ClearCase*, and there are two derived classes, *CCCheckout* and *CCCheckin*.There are also similar methods in the derived classes, *checkOption*. By applying *Form Template Method* to this target, duplicate statements are pulled up into in the method *checkOption* defined in the base class and new methods *checkOther* are created to implement the unique statements in each derived class. Note that there is a difference of the order of code fragments in



ID	Met...	Met...	Su...	SIM	...	...	...	...	...
4	execu...	exec...	Cle...	0.92...	13	1	1	1	1
23	execu...	exec...	Cle...	0.92...	13	1	1	1	1
50									
52									
57									
6									
53									
60									
75	execu...	exec...	Cle...	0.89...	13	1	2	2	1
33	execu...	exec...	Cle...	0.88...	15	1	3	2	1
5	execu...	exec...	Cle...	0.86...	13	1	3	2	1
8	execu...	exec...	Cle...	0.85...	12	2	2	2	1
42	execu...	exec...	Cle...	0.85...	12	2	2	2	1
44	execu...	exec...	Cle...	0.85...	12	2	2	2	1
45	execu...	exec...	Cle...	0.85...	12	2	2	2	1
46	execu...	exec...	Cle...	0.85...	12	2	2	2	1
54	execu...	exec...	Cle...	0.85...	12	2	2	2	1
59	execu...	exec...	Cle...	0.85...	12	2	2	2	1
68	execu...	exec...	Cle...	0.85...	12	1	3	2	1
76	execu...	exec...	Cle...	0.85...	12	2	2	2	1
64	execu...	exec...	Cle...	0.83...	13	2	3	3	1
30	execu...	exec...	Cle...	0.82...	12	2	3	3	1
47	execu...	exec...	Cle...	0.82...	12	2	3	3	1
77	execu...	exec...	Cle...	0.82...	12	2	3	3	1
1	execu...	exec...	Cle...	0.81 ...	13	1	5	3	1
34	execu...	exec...	Cle...	0.81 ...	13	1	5	3	1
35	execu...	exec...	Cle...	0.81 ...	13	1	5	3	1
36	execu...	exec...	Cle...	0.81 ...	13	1	5	3	1
28	execu...	exec...	Cle...	0.8	12	2	4	3	1
43	execu...	exec...	Cle...	0.8	12	2	4	3	1
65	execu...	exec...	Cle...	0.8	12	2	4	3	1
37	execu...	exec...	Cle...	0.78...	13	2	5	4	1
66	execu...	exec...	Cle...	0.75...	11	2	5	3	1
25	execu...	exec...	Cle...	0.75	12	2	6	4	1
38	execu...	exec...	Cle...	0.75	12	2	6	4	1

C:¥Users¥kei.hotta¥Documents¥kenkyu¥apache-ant-1.8.1¥src¥main¥org¥apache...

```
    public void execute() throws BuildException {
        Commandline commandLine = new Commandline();
        Project aProj = getProject();
        int result = 0;

        // Default the viewpath to basedir if it is not specified
        if (getViewPath() == null) {
            setViewPath(aProj.getBaseDir().getPath());
        }

        // build the command line from what we got the format is
        // cleartool lock [options...]
        // as specified in the CLEARTOOL.EXE help
        commandLine.setExecutable(getClearToolCommand());
        commandLine.createArgument().setValue(COMMAND_LOCK);

        // Check the command line options
        checkOptions(commandLine);

        // For debugging
        // System.out.println(commandLine.toString());

        if (!getFailOnErr()) {
            getProject().log("Ignoring any errors that occur for: "
                    + getOpType(), Project.MSG_VERBOSE);
        }
        result = run(commandLine);
        if (Execute.isFailure(result) && getFailOnErr()) {
            String msg = "Failed executing: " + commandLine.toString();
            throw new BuildException(msg, getLocation());
        }
```

C:¥Users¥kei.hotta¥Documents¥kenkyu¥apache-ant-1.8.1¥src¥main¥org¥apach...

```
     */
    public void execute() throws BuildException {
        Commandline commandLine = new Commandline();
        Project aProj = getProject();
        int result = 0;

        // Default the viewpath to basedir if it is not specified
        if (getViewPath() == null) {
            setViewPath(aProj.getBaseDir().getPath());
        }

        // build the command line from what we got. the format is
        // cleartool mkelem [options...] [viewpath ...]
        // as specified in the CLEARTOOL.EXE help
        commandLine.setExecutable(getClearToolCommand());
        commandLine.createArgument().setValue(COMMAND_MKELEM

        checkOptions(commandLine);

        if (!getFailOnErr()) {
            getProject().log("Ignoring any errors that occur for: "
                    + getViewPathBasename(), Project.MSG_VERBOSE);
        }
        result = run(commandLine);
        if (Execute.isFailure(result) && getFailOnErr()) {
            String msg = "Failed executing: " + commandLine.toString(
            throw new BuildException(msg, getLocation());
        }
    }


     /**
```

Figure 8. A Snapshot of the Result of *Creios*

Table I
TARGET SOFTWARE SYSTEMS

Name	In Short	LOC	# of Files	# of Candidates	Time[s]	Environment
Apache-Ant	Ant	212,401	829	226	237	CPU: Xeon 2.67GHz(4 core)CRAM: 4GB
Apache-Synapse	Synapse	58,418	383	45	95	

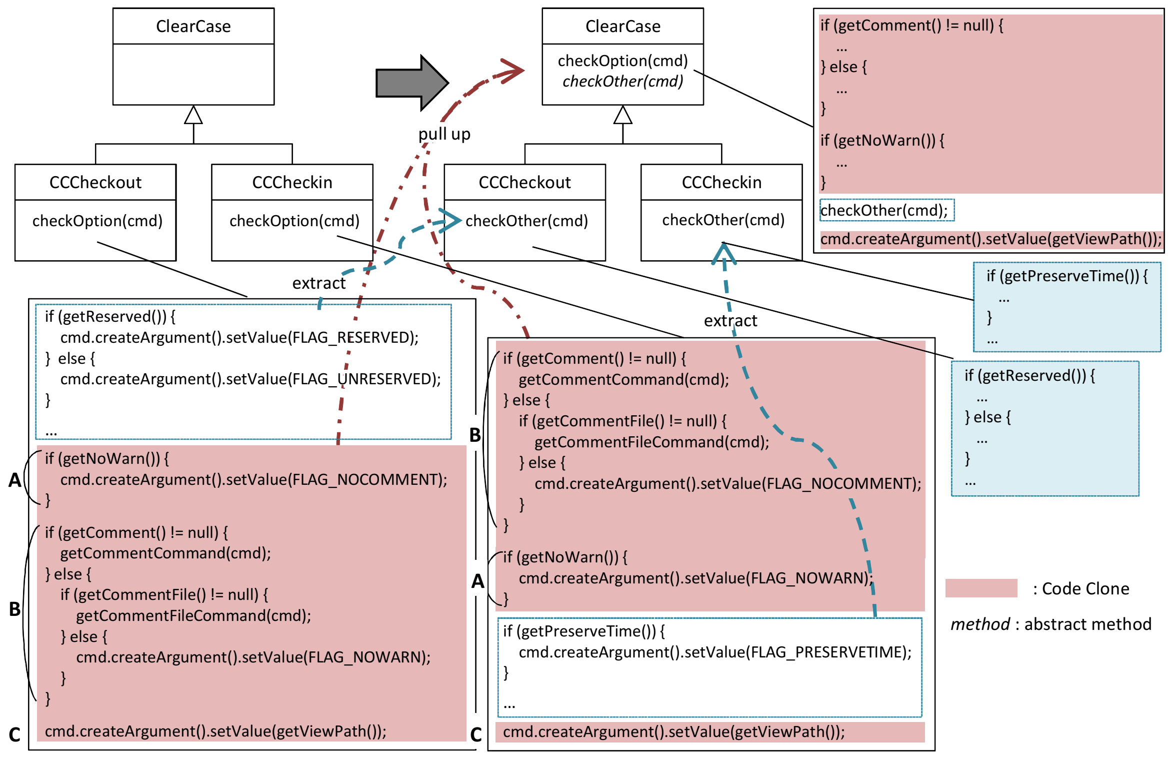
code clones: in *CCCheckout* the code fragments labeled A, B, and C are executed in this order, however in *CCCheckin* the order of code fragments is B-A-C. Therefore, this example is an instance that the previous techniques cannot detect.

Next, we show the comparison result to the previous method [5] in Table II. The column "Our method > Juillerat et al.'s method" indicates the number of candidates that the proposed method can handle trivial differences that the previous method cannot handle. The proposed method can recognize more duplicate statements than the previous method in these candidates, therefore we can say these candidates shows superiority of the proposed method compared to the previous method. We cannot found candidates that have differences of implementation style (e.g. for- and while-loops) in this experiment, but we confirmed that the proposed method can handle these differences with samples that we made.

In addition, we applied *Form Template Method* refactoring to all the 45 candidates that the proposed method had suggested in Synapse in order to confirm the adequacy and the efficiency of the proposed method as a technique to support refactorings. In this experiment, we successfully refactored all the 45 candidates detected with Creios in Synapse, and confirmed that the behavior of the program is preserved by using test suites attached to the software system. Additionally, we measured the time needed to each of the refactorings. Figure 10 shows the box-plots of the time needed to apply refactorings. Because Creios suggests all the candidates that can be refactored at a time, we run Creios at once and apply refactorings using the output. The time to execute Creios to Synapse is 95 seconds as shown in Table I. As a result, we could apply refactorings in few minutes in average nevertheless we are unfamiliar with the software.

VII. DISCUSSION

As we described in section VI, we applied *Form Template Method* refactoring to 45 candidates detected in Synapse. In some cases, we had to make some modifications that Creios did not indicate, or we had to make some adjustments to the output of Creios to apply the pattern. Table III shows the modifications or adjustments needed to apply refactorings, and the number of candidates that needed them. The term



```
if (getReserved()) {
   cmd.createArgument().setValue(FLAG_RESERVED);
} else {
   cmd.createArgument().setValue(FLAG_UNRESERVED);
}
...
if (getNoWarn()) {
   cmd.createArgument().setValue(FLAG_NOCOMMENT);
}
if (getComment() != null) {
   getCommentCommand(cmd);
} else {
   if (getCommentFile() != null) {
      getCommentFileCommand(cmd);
   } else {
      cmd.createArgument().setValue(FLAG_NOWARN);
   }
}
cmd.createArgument().setValue(getViewPath());
```

```
if (getComment() != null) {
   getCommentCommand(cmd);
} else {
   if (getCommentFile() != null) {
      getCommentFileCommand(cmd);
   } else {
      cmd.createArgument().setValue(FLAG_NOCOMMENT);
   }
}
if (getNoWarn()) {
   cmd.createArgument().setValue(FLAG_NOWARN);
}
if (getPreserveTime()) {
   cmd.createArgument().setValue(FLAG_PRESERVETIME);
}
...
cmd.createArgument().setValue(getViewPath());
```

```
if (getComment() != null) {
   ...
} else {
   ...
}
if (getNoWarn()) {
   ...
}
checkOther(cmd);
cmd.createArgument().setValue(getViewPath());
```

```
if (getPreserveTime()) {
   ...
}
...
```

```
if (getReserved()) {
   ...
} else {
   ...
}
...
```

Figure 9. An Example of Application of *Form Template Method* with Proposed Method

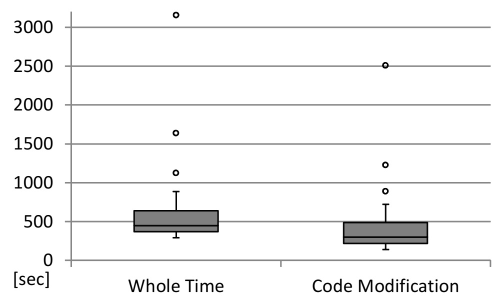


Figure 10. The Box-plots of the Time to Apply Refactorings

"modify ENS" means the cases in which we had to modify ENSs or their pairwise relationships between two methods that Creios suggests, the term "move methods into base class or change their visibility" means the cases in which some methods defined in derived classes are used in common processes and we had to move those methods into the base class and/or change their visibilities, and the term "replace field references to calls of getter methods" indicates the cases in which some fields are used in duplicate statements and they are not visible from the base class and we had to replace references of these fields to calls of getter methods of them.

The proposed method does not consider the visibility of methods and fields in the source code. Therefore, it is possible that code fragments that should be pulled up into template method call methods or reference fields that are not accessible from the base class. In such cases, we need additional modifications on the source code to apply *Form Template Method*. We can apply the pattern to such candidates by changing the visibility of methods and fields. However, it is not desirable that code clone removal requires increasing the visibility of methods or fields, because such

Table II
THE RESULT OF THE COMPARISON EXPERIMENTS

	Ant	Synapse
# of candidates	226	45
Our method > Juillerat et al.'s method	14	3
-Differences of order of code	10	1
-Differences of variable names	4	2
-Differences of implementation styles	0	0
Same result	202	42

Table III
THE CANDIDATES THAT NEED SOME MODIFICATIONS FOR CREIOS'S OUTPUTS

# of candidates that need no modifications	29
# of candidates that need some modifications	16
modify ENS	12
move methods into base class and/or change their visibilities	4
replace field references to calls of getter methods	2

changes could cause vulnerability [23]. For fields, if fields have getters and setters, we can resolve this problem by using them.

VIII. Conclusion

In this paper, we proposed a new technique to assist developers to apply *Form Template Method* refactorings on code clones. It detects refactoring candidates automatically and suggests all the candidates that can be applied *Form Template Method*. It uses PDG as its data structure, which enables us to handle trivial differences that have no impact on the meaning of the program.

As future work, we are going to make it possible to handle not only pairs of methods but also sets of them. In addition, we are going to improve our method for assuring behavior preservation, and implement a function that suggests the source code after the application of *Form Template Method*.

Acknowledgment

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 Grant-in-Aid for Exploratory Research (23650014) 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.

References

[1] N. Göde and R. Koschke, "Frequency and risks of changes to clones," *33rd International Conference on Software Engineering*, pp. 311–320, May 2011.

[2] N. Bettenburg, W. Shang, W. M. Ibrahim, B. Adams, Y. Zou, and A. E. Hassan, "An empirical study on inconsistent changes to code clones at the release level," *Science of Computer Programming in Press*, 2011.

[3] R. Fanta and V. Rajlich, "Removing clones from the code," *Journal of Software Maintenance*, pp. 223–243, 1999.

[4] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, "Aries: Refactoring support environment based on code clone analysis," *the International Conference on Software Engineering and Applications*, pp. 222–229, 2004.

[5] N. Juillerat and B. Hirsbrunner, "Toward an implementation of the "form template method refactoring"," *7th IEEE International Working Conference on Source Code Analysis and Manipulation*, pp. 81–90, 2007.

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

[7] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[8] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Transactions on Software Engineering*, vol. 30, no. 2, pp. 126–139, Feb. 2004.

[9] N. Göde, "Clone removal: Fact or fiction?" in *Proc. of the 4th International Workshop on Software Clones*, 2010.

[10] S. Schulze and M. Kuhlemann, "Advanced analysis for code clone removal," in *Proc. of the 11th Workshop on Software Reengineering*, 2009.

[11] C. J. Kapser and M. W. Godfrey, ""cloning considered harmful" considered harmful: Patterns of cloning in software," *Empirical Software Enginieering*, 2008.

[12] M. Harman, "The current state and future of search based software engineering," *Future of Software Engineering*, 2007.

[13] O. Seng, J. Stammel, and D. Burkhart, "Search-based determination of refactorings for improving the class structure of object-oriented systems," in *Proc. of the 8th Annual Conference on Genetic and Evolutionary Computation*, July 2006, pp. 1909–1916.

[14] M. O'Keeffe and M. O. Cinnéide, "Search-based refactoring for software maintenance," *Journal of Systems and Software*, vol. 81, no. 4, pp. 502–516, Apr. 2008.

[15] S. Lee, G. Bae, H. S. Chae, D.-H. Bae, and Y. R. Kwon, "Automated scheduling for clone-based refactoring using a competent ga," *Software: Practice and Experience*, 2010.

[16] M. F. Zibran and C. K. Roy, "A constraint programming approach to conflict-aware optimal scheduling of prioritized code cloen refactoring," in *Proc. of the 11th International Working Conference on Source Code Analysis and Manipulation*, 2011.

[17] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, pp. 319–349, 1987.

[18] N. Tsantalis and A. Chatzigeorgiou, "Identification of extract method refactoring opportunities for the decomposition of methods," *Journal of Systems and Software*, vol. 84, no. 10, pp. 1757–1782, Oct. 2011.

[19] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," *Technical Report No. 2007-541, Queen's University*, 2007.

[20] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in *8th International Symposium on Static Analysis*, 2001, pp. 40–56.

[21] J. Krinke, "Identifying similar code with program dependence graphs," *In Proc. the 8th Working conference on Reverse Engineering*, pp. 301–309, Oct. 2001.

[22] Y. Higo and S. Kusumoto, "Code clone detection on specialized pdgs with heuristics," *15th European Conference on Software Maintenance and Reengineering*, pp. 75–84, 2011.

[23] K. Maruyama and T. Omori, "A security-aware refactoring tool for java programs," *Proc. of the 4th Workshop on Refactoring Tools*, pp. 22–28, 2011.