Master Thesis

Title

A Refactoring Support for Form Template Method using PDG-based Code Clone Detection

Supervisor
Prof. Shinji KUSUMOTO

by
Keisuke HOTTA

February 7, 2012

Department of Computer Science
Graduate School of Information Science and Technology
Osaka University

Master Thesis

A Refactoring Support for Form Template Method using PDG-based Code Clone Detection

Keisuke HOTTA

Abstract

Recently, code clones have received much attention. Code clones are defined as source code fragments that are identical or similar to each other. Code clones are introduced into software systems by various reasons such as copy-and-paste operations. It is generally said that the existences of code clones make software maintenance more difficult. This is because if we modify a code fragment, it is necessary to check its correspondents whether they need the same modifications simultaneously or not. To avoid negative effects of code clones, it is effective to remove code clones with refactoring. Refactoring is a technique to transform one representation form of source code to another without changing the external behavior of the subject systems. By applying refactoring techniques to code clones, we can merge them spreading across multiple source files into a single module. However, we need much effort to apply manual refactorings to them. Also, applying manual refactorings is a complicated task, so that human related errors easily occur. Consequently, techniques or tools for supporting refactoring activities are required.

There are some techniques to remove code clones. Applying "Form Template Method" is one of the techniques and one of the refactoring patterns. Form Template Method focuses on similar methods whose owner classes have the same base class. In this refactoring pattern, developers write an outline of the process into the base class and implement the details of the process in the derived classes. By applying Form Template Method refactoring, code clones existing in similar methods are merged into the base class. One of the advantages of using this pattern is that we can handle differences between target methods.

Some researchers have proposed methods to support Form Template Method refactorings. However, they still have some issues. The issues are that they cannot handle trivial differences even though they have no impacts on the behavior of the program, and that they can support refactorings on only pairs of methods, which means they cannot support refactorings on groups consisting of three or more methods. This thesis proposes a new method for supporting Form Template Method applications to resolve all of these issues with program dependence graphs.

Keywords

Code Clones
Refactoring
Form Template Method
Program Dependence Graph
Software Maintenance

Contents

1 Introduction **1**

2 Preliminaries **3**

2.1 Program Dependence Graph 3

2.1.1 Generic Definition 3

2.1.2 PDGs in This Study 4

2.2 Code Clone 6

2.2.1 Definition 6

2.2.2 Causes of Creations 6

2.3 Code Clone Detection Methods 7

2.3.1 Categorization 7

2.3.2 Scorpio (PDG-based Code Clone Detection Tool) 11

2.4 Refactoring 12

2.4.1 Refactoring Activities 12

2.4.2 Behavior Preservation 13

2.4.3 Refactoring Patterns Used for Code Clone Removals 13

3 Related Work **19**

3.1 Techniques for Refactoring on Code Clones 19

3.2 Techniques for Refactoring on Code Clones with Form Template Method . . . 20

3.3 Techniques for Code Clone Managing 20

4 Motivation **23**

4.1 Issues of Previous Studies 23

4.1.1 Issue of Trivial Differences 23

4.1.2 Issue of Groups of Three or More Methods 24

4.2 Objective of This Study 26

5 Outline of the Proposed Method **27**

5.1 Inputs and Outputs 27

5.2 Processing Flow 28

5.3 Definitions 29

5.3.1 A Directed Graph 29

5.3.2 A PDG 30

5.3.3 Clone Pairs 31

6 **Supporting for Method Pairs** **33**
6.1 STEP-P1: Create PDGs 33
6.2 STEP-P2: Detect Code Clones 33
6.3 STEP-P3: Identify Method Pairs 34
6.4 STEP-P4: Detect Common and Unique Processes 35
6.5 STEP-P5: Detect Sets of Statements Extracted as a Single Method 37
6.5.1 Definition of the Extract Node Set 37
6.5.2 Parameters of ENSs 38
6.5.3 Output of ENSs 39
6.5.4 Conditions for Call 39
6.5.5 Requirements for ENSs to be Extracted as a Single Method 40
6.6 STEP-P6: Detect Pairwise Relationships 43
6.6.1 Requirement P6-1: Requirement the Type of the Return Value 44
6.6.2 Requirement P6-2: Requirement about Conditions for Call 44
6.6.3 An Example of Pairwise Relationships Detection 47

7 **Supporting for Method Groups** **48**
7.1 STEP-S7: Identify Method Groups 48
7.2 STEP-S8: Detect Common and Unique Processes 48
7.3 STEP-S9: Detect Relationships on ENSs 50

8 **Implementation** **51**
8.1 Overview 51
8.2 Functionalities for Method Pairs 52
8.3 Functionalities for Method Groups 57

9 **Evaluation** **59**
9.1 Evaluation of Supporting for Method Pairs 59
9.2 Evaluation of Supporting for Method Groups 61
9.3 Experiment with Subjects 61
9.3.1 Overview of the Experiment 61
9.3.2 Target Method Groups 61
9.3.3 Prodcedure of the Experiment 63
9.3.4 Result 64

10 Discussion **66**
10.1 PDG Creation 66
10.2 Detection of Common Statements 66
10.3 Candidates that Need to be Tailored 66
10.4 Detection of Method Groups 67
10.5 Threats to Validity of the Experiment with Subjects 68

11 Conclusion **69**

Acknowledgements **70**

References **71**

A Algorithms for Detecting Isomorphic Subgraphs **76**

List of Figures

1 An Example of PDG . . . 3
2 An Example of PDG with Execute Dependence Edges . . . 4
3 Data Dependence Considering State Changes of Objects . . . 5
4 An Example of Clone Pairs and Clone Sets . . . 6
5 An Example of ASTs . . . 9
6 A Code Clone with a Differnt Order of Statements . . . 10
7 An Example of Extract Class . . . 14
8 An Example of Extract SuperClass . . . 14
9 An Example of Extract Method . . . 15
10 An Example of Pull Up Method . . . 16
11 An Example of Refactorings with Form Template Method . . . 17
12 Motivating Example 1 . . . 23
13 Motivating Example 2 . . . 25
14 The Output of the Proposed Method . . . 27
15 A Directed Graph . . . 29
16 A PDG . . . 30
17 $ClonePairs(G_1, G_2)$ . . . 31
18 An example of Method Pairs Including Redundant Clone Pairs . . . 36
19 An example of the Detection of ENSs . . . 37
20 An Example of Inputs and Outputs of ENSs . . . 38
21 Behavior of Algorithm 2 . . . 41
22 An Instance of Segmentalization of Block Statements . . . 43
23 An Example of Wrong Pairwise Relationships Caused by not Considering Conditions for Call . . . 45
24 An Example of Pairwise Relationships . . . 46
25 An Example of Method Group . . . 49
26 A Whole Snapshot of Creios (for Method Pairs) . . . 51
27 A Snapshot of Source Code View . . . 52
28 A Snapshot of PDG View . . . 53
29 A Snapshot of Apposing View of Source Code View and PDG View . . . 54
30 An Example of Candidate Method Pair . . . 55
31 A Snapshot of Filtering View . . . 56
32 A Metrics Graph . . . 56
33 View of the Metrics Values . . . 57

34 A Whole Snapshot of Creios (for Method Groups) 57
35 An Example of Application of Form Template Method with the Proposed Method 59
36 The Box-Plot of the Time to Apply Form Template Method on Synapse 60
37 Candidate Method Groups 62

List of Tables

1 The Values of Metrics in the Method Pair of Figure 30 55
2 Target Software Systems 59
3 The Number of Detected Candidates and Elapsed Time on Method Pairs 60
4 The Number of Detected Candidates and Elapsed Time on Method Groups 61
5 The Features of Target Method Groups 63
6 Groups of Subjects 64
7 Elapsed Time to Finish Form Template Method Application 65
8 The Average Time 65
9 The Candidates that Need some Modifications for Creios's Outputs 67

List of Algorithms

1 Removing Redundant Clone Pairs 35
2 Division of an ENS 42
3 *detect*(v, S) 42
4 *parse*(S, R) 42
5 *ForwardSlice*(G_1, G_2, r_1, r_2, R_1, R_2) 76
6 *BackwardSlice*(G_1, G_2, r_1, r_2, R_1, R_2) 77

1 Introduction

Recently, code clones have received much attention and many research efforts have been performed on them [1, 2]. Code clones are defined as identical or similar code fragments to one another, and they are created by various reasons such as copy-and-paste operations. Because code cloning is easy and inexpensive, it can make software development faster and can enable "experimental" development. However, it has been pointed out that the presence of code clones has a negative impact on software maintenance because if we modify a code fragment, it is necessary to check its correspondents whether they need the same modifications or not. Therefore, various techniques and tools have been proposed to detect code clones automatically by many researchers [3–18].

Refactoring also has been studied intensively in recent years because it is highly expected that we can improve maintainability of software systems by applying refactorings. In Fowler's book, he defined refactoring as "*the process of changing a software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure*" [19]. It has been reported that the maintainability of software systems decays over time [20]. Refactoring is usable in such a case because we can prevent the decay of maintainability with suitable refactorings. However, applying refactorings requires much effort for maintainers, and it is quite difficult for maintainers to apply refactorings manually without introducing any human errors [21]. Because of these factors, techniques to assist refactoring activities are required, and indeed many techniques have been proposed in recent years [22].

To prevent the influence of code clones, it is effective to remove code clones by applying some refactorings. We can remove code clones by merging cloned code spreading across multiple source files into a single module with refactorings. Many research efforts have been performed on assisting code clone removal with refactorings. A majority of clone removal techniques are based on "Extract Method" refactoring pattern or "Pull-Up Method" refactoring pattern. However, these techniques have an issue that they cannot handle code clones with some gaps.

A clone removal technique with "Form Template Method" refactoring pattern can overcome this issue. Form Template Method uses Template Method pattern that is one of the design patterns proposed by Gamma et al. [23]. This pattern targets similar methods whose owner classes have the same base class. In this pattern, programmers write an outline of similar methods into the base class and implement detail processes in each derived class. By applying Form Template Method, code clones existing between similar methods are merged into the base class. One of the advantages of clone removal with this pattern is that we can apply this technique to methods having some gaps.

Some researchers have proposed techniques to support refactorings with Form Template

Method [24–26]. However, these techniques cannot support removing code clones if they include the following differences even if these differences have no impacts on the behavior of the program:

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

Moreover, the existing methods can handle only pairs of methods though Form Template Method refactoring can be applied to groups consisting of three or more similar methods.

This thesis proposes a new technique to support applying Form Template Method with program dependence graphs, which allows us to resolve the first issue. We also extend the proposed method to be able to handle groups of three or more similar methods.

The rest of this thesis is organized as follows: In Section 2, we introduce preliminaries related to this work. We describe related works in Section 3, then we explain our motivation in Section 4. Section 5 describes the outline of the proposed method, then we explain the proposed method in detail in Section 6 and Section 7. In Section 8, we describe the implementation of the proposed method. Section 9 reports the evaluation of the proposed method on open source software systems, and we discuss the result of the evaluation in Section 10. Finally, Section 11 summarizes this thesis and refers to the future work.

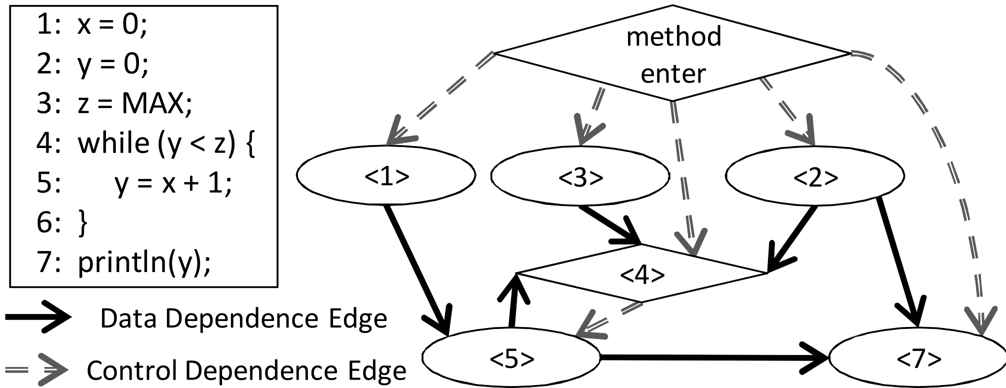


Figure 1: An Example of PDG

2 Preliminaries

2.1 Program Dependence Graph

2.1.1 Generic Definition

Program dependence graph (in short, **PDG**) is a directed graph that represents dependencies between the elements of the program [27,28]. A node in a PDG indicates an element of a program (such as a statement and a conditional predicate), and an edge in a PDG indicates a dependence between two elements. PDG is created based on flows of data and controls. Therefore, we get the same PDGs from two programs if their flows of data and controls are same, though the programming styles are not equal.

There are the following two types of dependencies in PDG.

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

Figure 1 shows an example of PDG. In this example, there are three data dependencies from the 2nd, 3rd, and 5th lines to the 4th line because variables y and z are referenced in the 4th line. On the other hand, there is a 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 with "*method enter*" that means the enter node of the method. In general, PDG contains a method enter node, and there are control dependencies from the enter node to all

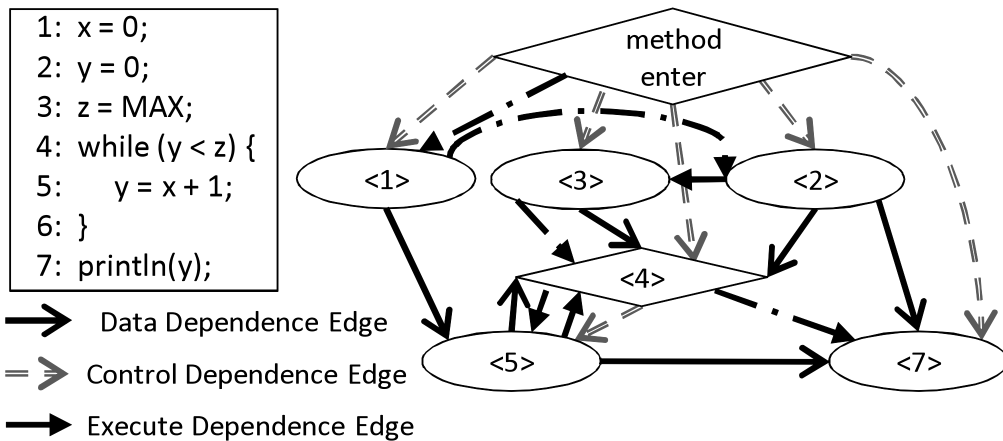


Figure 2: An Example of PDG with Execute Dependence Edges

nodes that are directly contained by the method. Note that we regard a node n as being directly contained by the method if s has no control dependencies from any other nodes in the PDG.

2.1.2 PDGs in This Study

PDGs used in this study is specialized for code clones detection and refactoring. The major differences of a traditional PDG and a specialized PDG are as follows:

- Having execute dependences and
- Tracing state changes of objects.

Execute Dependence

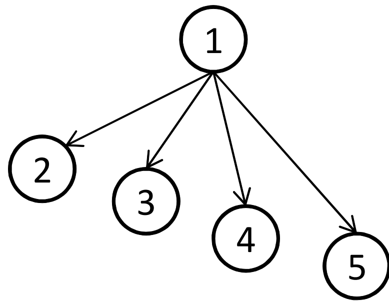
PDGs used in this study has an additional dependence called 'execute dependence'. The definition of execute dependence is as follow.

Execute Dependence: There is an execute dependence from element s to element t, if t can be executed in the next that s is executed.

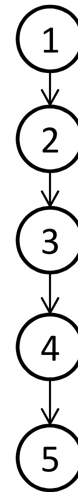
Figure 2 shows an example of PDG with execute dependence edges. We can detect more code clones with PDGs having execute dependence than with traditional PDGs. This is because the range of program slicing is expanded by introducing this dependence.

```
1: StringBuilder builder = new StringBuilder();
2: builder.append("A");
3: builder.append("B");
4: builder.append("C");
5: return builder.toString();
```

(a) Source Code



(b) Traditional



(c) Specialized

Figure 3: Data Dependence Considering State Changes of Objects

Tracing State Changes of Objects

In this study, we create data dependence edges with considering state changes of objects caused by method calls. Concretely, we regard that there is a data dependence from a method call statement s to other statement t, if the state of any objects is changed in s and t references the objects without redefining them.

Figure 3 compares a traditional PDG and a specialized PDG created from the same source code. In this figure, we omit control and execute dependences and the method enter node. In this example, the state of an object `builder` is changed in the 2nd, 3rd, and 4th lines by calling a method `append`. In the traditional PDG, all the elements that reference `builder` have data dependences from the 1st line. This is because the object `builder` does not re-defined or re-assigned until the end of the method. However, the specialized PDG used in this study considers state changes of objects. Therefore, we get the PDG shown in Figure 3 (c) from the source code.

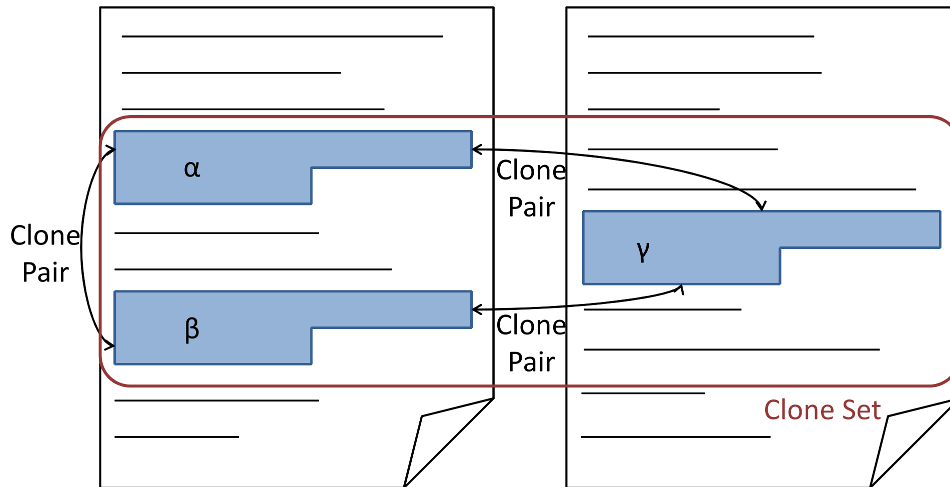


Figure 4: An Example of Clone Pairs and Clone Sets

Note that it is regarded that states of objects are changed by a method call if the values of any fields in the objects are changed by the method [29].

2.2 Code Clone

2.2.1 Definition

Code clone is defined as identical or similar code fragments in source code. As shown in Figure 4, we call a pair of code fragments α and β as a clone pair if α and β are similar. In addition, we call a set of code fragments S as a clone set if any pair of code fragments in S are clone pairs [30]. Note that there is neither a generic nor strict definition of code clone, therefore each clone detection method or tool has its own definition of code clone.

Code clones can be categorized into the following 3 types by the degree of their similarities [31].

Type-1: An exact copy except for white space and comments.

Type-2: Syntactically identical copy; only variable, type, or function identifiers were changed.

Type-3: A copy with further modifications in Type-2; statements were changed, added, or removed.

2.2.2 Causes of Creations

Code clones can be created or introduced by the following factors.

Copy-and-paste Operations

This is the most popular situation that code clones are created. The code reuse by copy-and-paste operations is a common practice in software development, because it is quite easy, and it enables us to make software development faster.

Stylized Processing

Processing used frequently (e.g. calculations of the income tax, insertions in queues, or access to data structures) may cause code duplication.

Lack of Suitable Functions

Programmers may have to write similar processes with similar algorithms if they use programming languages that do not have abstract data types or local variables.

Performance Improvement

Programmers can introduce code duplication intentionally to improve the performance of software systems in the case that the in-line expansion is not supported.

Automatically Generated Code

Code generation tools automatically create code based on stylized code. As a result, if we use code generation tools to handle similar processes, it may generate similar code fragments.

To Handle Multiple Platforms

Software systems that can handle multiple operation systems or CPUs tend to include many code clones in the processes handling each platform.

Accident

Different developers may write similar code accidentally. However, it is rare that the amount of similar code generated accidentally becomes high.

2.3 Code Clone Detection Methods

2.3.1 Categorization

There are many methods that detect code clones automatically, and there are also many code clones detectors implementing these methods. Code clones detectors can be loosely categorized

into the following categories by their detection units [1, 31].

Text-based Techniques

Text-based detection techniques detect code clones by comparing every line of code as a string. They detect multiple consecutive lines that match in specified threshold or more lines as code clones. The biggest advantage of this technique is that it can detect code clones quickly compared with other detection techniques. This technique requires no pre-processing on source code, which enables the fast detection. However, we cannot detect code clones including differences of coding styles (e.g. whether long lines are divided into multiple lines or not) with this technique.

The method proposed by Johnson [5] and the method proposed by Ducasse et al. [6] are instances of line-based clone detectors. In these methods, every line of code is compared after white space and tabs are removed. These methods are language-independent because they compare lines of code textually.

Token-based Techniques

In a token-based approach, source code is lexed/parsed/transformed to a sequence of tokens. This technique detects common subsequences of tokens as code clones. Compared to text-based approaches, a token-based approach is usually robuster against code changes such as formatting and spacing. Detection speed is inferior as compared with text-based techniques, meanwhile superior as Tree- or PDG-based approaches. This is because, in token-based approach, source code has to be transformed into intermediate representations such as AST and PDG.

CCFinder, a clone detector developed by Kamiya et al. [3], is one of the token-based detectors. *CCFinder* replaces user-defined identifiers by special tokens. By this pre-processing, it can detect code clones with different identifiers. In addition, it can handle multiple widely-used programming languages such as C/C++, Java, COBOL, and FORTRAN. Moreover, there is a major version up of *CCFinder* named *CCFinderX* [32]. In this version up, the detection algorithm is changed, and the detection speed is improved by multithreading.

CP-Miner is also a token-based detector. *CP-Miner* is developed by Li et al. [7]. Firstly, lexical and syntax analises are performed on source code. User-defined identifiers are replaced by special tokens as well as *CCFinder*. The major difference between *CP-Miner* and *CCFinder* is in detection algorithms. In *CP-Miner*, hash values are calculated from every statement, and then a frequent pattern mining algorithm is applied to detect code clones. Frequent patterns do not have to be consecutive, which means that *CP-Miner* can detect Type-3 clones.

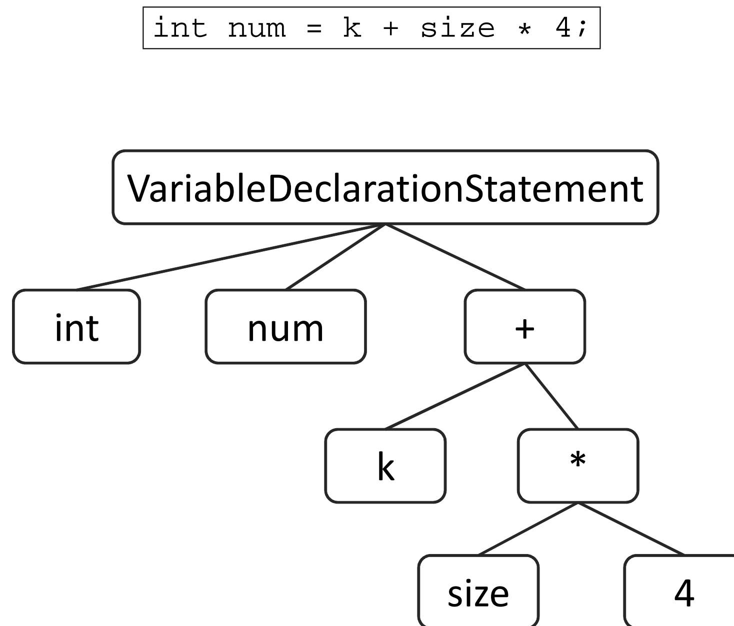


Figure 5: An Example of ASTs

Tree-based Techniques

In a Tree-based detection, a program is parsed to a parse tree or an abstract syntax tree (in short, AST) with a parser of the language in interest. An AST is one of the intermediate representations that capture the structure of source code. Figure 5 shows an example of ASTs. Common subtrees are regarded as code clones. This approach considers the structural information of source code, therefore tree-based detectors do not detect code clones ignoring the structure of source code such as code clones including a part of a method and a part of another method. However, a disadvantage of this approach compared with Text- and Token-based approaches is that it requires more detection costs because of the additional cost required to transform source code to parse trees or ASTs.

One of the pioneers of AST-based clone techniques is that of Baxter et al.'s *CloneDR* [4, 33]. *CloneDR* compares subtrees of ASTs by characterization metrics based on a hash function through tree matching, instead of comparing subtrees of ASTs directly. This processing allows *CloneDR* to detect code clones quickly from large software systems. It can handle a lot of programming languages. Moreover, it has a function to assist clone removal.

Koschke et al.'s method [8] and Jiang et al.'s method [9] are tree-based approaches as well as *CloneDR*. In Koschke et al.'s method, ASTs are compared with a suffix tree algorithm to have

```
  fp = lookaheadset + tokensetsize;
  for (I = lookaheas(state) ; I < k ; i++) {
%     fp1 = LA + i * tokensetsize;
%     fp2 = lookaheadset;
%     while (fp2 < fp3)
%           *fp2++ |= fp1++;
  }
```

(a) Code Fragment 1

```
  fp3 = base + tokensetsize;
  ...
  if (rp) {
      while ((j = *rp++) >= 0) {
           ...
#          fp1 = lookaheadset;
#          fp2 = LA + j * tokensetsize;
#          while (fp1 < fp3)
#                *fp1++ |= *fp2++;
      }
  }
```

(b) Code Fragment 2

Figure 6: A Code Clone with a Differnt Order of Statements

an increase of detection speed. On the other hand, Jiang et al. use a locality sensitive hashing algorithm to detect code clones. With the algorithm, Jiang et al.'s method can detect Type-3 code clones.

PDG-based Techniques

In a PDG-based approach, code clones are detected by comparing PDGs created from source code. Isomorphic subgraphs are regarded as code clones. PDGs require a semantic analysis for their creation, therefore this approach requires much cost than other detection techniques. However, this technique can detect code clones with additions/deletions/changes in statements or those with some differences that have no impact on the behavior of programs. This is because PDG-based techniques can consider the meanings of programs.

Figure 6 shows one of the code clones that include some differences that have no impact on the behavior of programs. Other techniques cannot detect these two code fragments as a code clone because there is a different order of statements.

One of the leading PDG-based clone detection approach is Komondoor and Horwitz's method [10]. Their method detects isomorphic subgraphs of PDGs with program slicing. They also propose an approach to group identified clones together while preserving the semantics of the original code for automatic procedure extraction to support software refactoring. Krinke's method [11], and Higo et al.'s method [12, 34] are also included in PDG-based techniques. Each detection method is optimized to reduce detection cost. Krinke sets a limit of the search range of PDGs with a threshold. By contrast, Higo et al. confine nodes to be base of subgraphs with some conditions. Moreover, Higo et al. introduce a new dependence named "execution dependence". That is, there is an execution dependence from a node A to another node B if the program element represented

by B may only be executed after the program element represented by A. By introducing this dependence, they succeeded to detect code clones that other PDG-based methods could not detect.

Other Detection Techniques

One of the detection techniques that can be categorized into this category is a metrics-based approach [13]. First, metrics-based detectors calculate metrics on every program module (such as files, classes, or methods), then detect code clones by comparing the coincidence or the similarity of these values.

Beside this, there are some file-based detection methods [14, 15]. This detection technique detects code clones by comparing every file instead of statements or tokens, which let it quick detections. However, this technique cannot find code clones that exist in a part of a file.

Moreover, incremental detection techniques are under intense studies [16–18]. In incremental detections, code clone detection results or their intermediate products persist by using databases, and it is used in the next code clone detection. By reusing previous revisions' analysis, it can reduce detection cost on the current revision substantially.

2.3.2 Scorpio (PDG-based Code Clone Detection Tool)

In this subsection, we describe a clone detector, Scorpio [34], used in this research.

Scorpio is one of the PDG-based clone detectors developed by Higo et al. [12]. Currently, Scorpio can handle software systems written in Java. The major features of Scorpio are as follows.

It can detect code clones with different user-defined variables

Scorpio replaces use-defined identifiers by special characters. Therefore, it can detect code clones having different user-defined variables.

It can detect Type-3 code clones and non-contiguous code clones

Scorpio can detect Type-3 code clones and non-contiguous code clones because it is a PDG-based clone detector.

It is robust for detecting contiguous code clones

One of the disadvantages of PDG-based clone detectors is that they cannot regard sequences of program elements as code clones if every element in the sequences has no dependence between other elements in the sequences. To improve this matter, Scorpio introduces execute dependence,

which enables it to expand the range of program slicing, so that the ability to detect contiguous code clones is improved.

It uses both of two graph search algorithms

There are two ways to search graphs, forward and backward slicing. Scorpio uses both of forward and backward slicing, which enlarges code clone detection result because there are similar subgraphs that cannot be detected by using only forward or backward slicing.

It confines nodes to be bases of slicing

To reduce detection costs, Scorpio limits slice points. Unnecessary slice points are identified and removed by this heuristic.

2.4 Refactoring

2.4.1 Refactoring Activities

Refactoring is a technique that improves internal structures of software systems without changing the external behavior of the programs [22]. We can prevent decay of maintainability of running software systems with suitable refactorings.

The refactoring process consists of several activities as follows [22]:

1. Identify places that should be refactored,
2. Determine which refactroing(s) should be applied to the places,
3. Guarantee that the behavior of the program is preserved by the selected refactoring(s),
4. Apply the refactoring(s),
5. Assess the effect of the refactoring(s) on quality of the software or the process and
6. Maintain the consistency between the refactored program and other software artifacts (e.g. documentation, design documents, requirements specification, tests).

Each of these activities can be supported by different tools, techniques or formalisms.

2.4.2 Behavior Preservation

The refactoring should not change the behavior of programs accoring to its definition.

The original definition of behavior preservation is suggested by Opdyke [35]. The definition states that, for the same set of input values, the resulting set of output values should be the same before and after the refactoring. However, requiring the preservation of input-output behavior is insufficient, since many other aspects of the behavior may be relevant as well. For example, in the case of *real-time software*, an essential ascect of the behavior is the execution time of certain operations. Thus, refactorings should preserve all the kinds of temporal constraints. For *embedded software*, memory constraints and power consumption are also important aspects. Consequently, we need a wider range of definitions of behavior that may or may not be preserved by a refactoring, depending on domain-specific or even user-specific concerns.

Another pragmatic way to guarantee the behavior preservation is using test suites. This means that if all the test suites are passed before and after refactorings, it is regarded that the refactorings do not affect the behavior of the program. If we have sufficient test suites, the fact that all the test suites still pass after the refactorings will be a good evidence that the behavior is preserved.

Another approach is to formally prove that refactorings preserve the full program semantics. We can prove the behavior preservation formally if a language with a simple and formally defined semantics is used in the target software systems. However, it is difficult to prove the behavior preservation for more complex languages such as C++. In such a case, we need to put some restrictions to prove the behavior preservation.

2.4.3 Refactoring Patterns Used for Code Clone Removals

Herein, we describe refactoring patterns proposed by Fowler used for clone removals [19, 36].

Extract Class/SuperClass

Extract Class indicates extracting a part of a class as a new class. If there is a large and/or complex class, the class requires much cost to be maintained. Extract Class is useful in such a case. If there is a class-level duplication, we can remove code clones by appllying Extract Class. Figure 7 shows an example of refactoring with Extract Class. In this case, there are duplicate fields `officeAreaCode` and `officeNumber`, and duplicate operation about them. By applying Extract Class to this example, duplicate fields and duplicate operation are extracted as a new class `TelephoneNumber`, and the classes `Person` and `Company` uses the class. By this modification, duplicate code is removed from the two classes.

If duplicate classes do not extend different base classes, Extract SuperClass may be a better

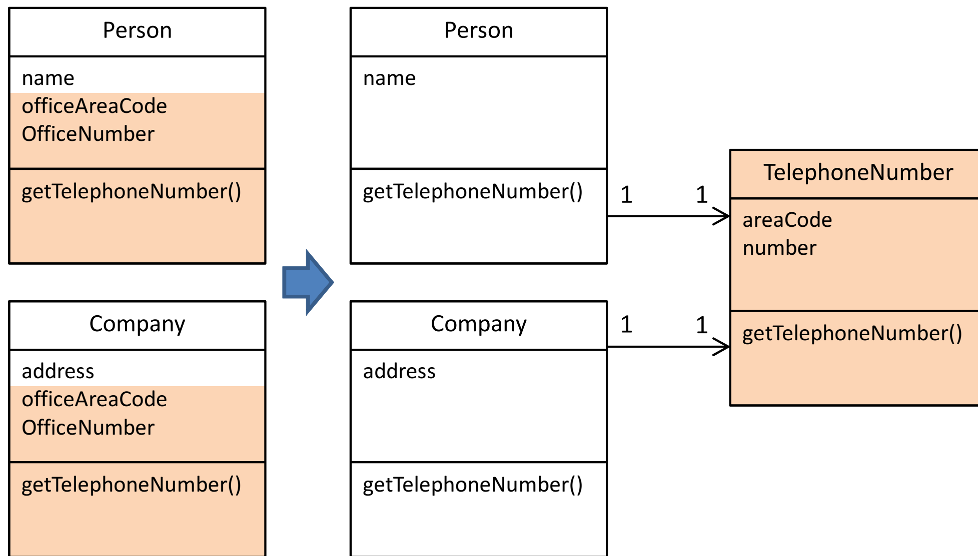


Figure 7: An Example of Extract Class

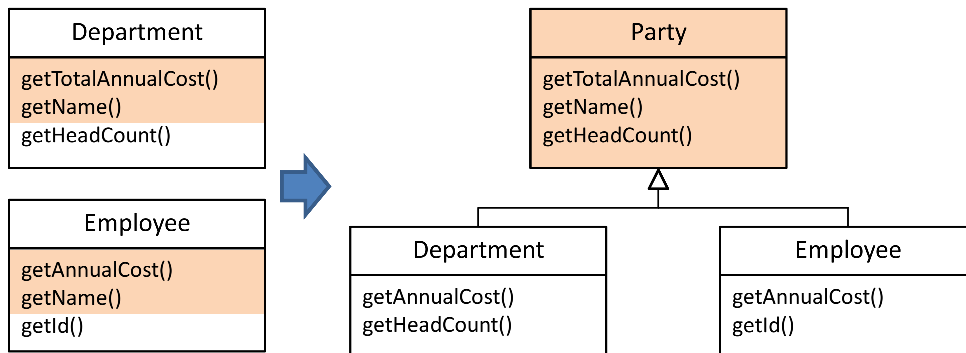


Figure 8: An Example of Extract SuperClass

solution for clone removal. Extract SuperClass is similar to Extract Class. The difference is that Extract SuperClass uses the inheritance; meanwhile Extract Class uses the delegation. In Extract SuperClass, duplication between two (or more) classes is extracted as a new class and all the original classes are changed to extend the new class. Figure 8 shows an example of the application of Extract SuperClass. In this example, a new class `Party` is created by extracting the duplication of two classes `Department` and `Emproyee`, then the two classes are changed to extend the class`Party`.

Extract Method

Extract Method indicates extracting a part of a method as a new method. This refactoring

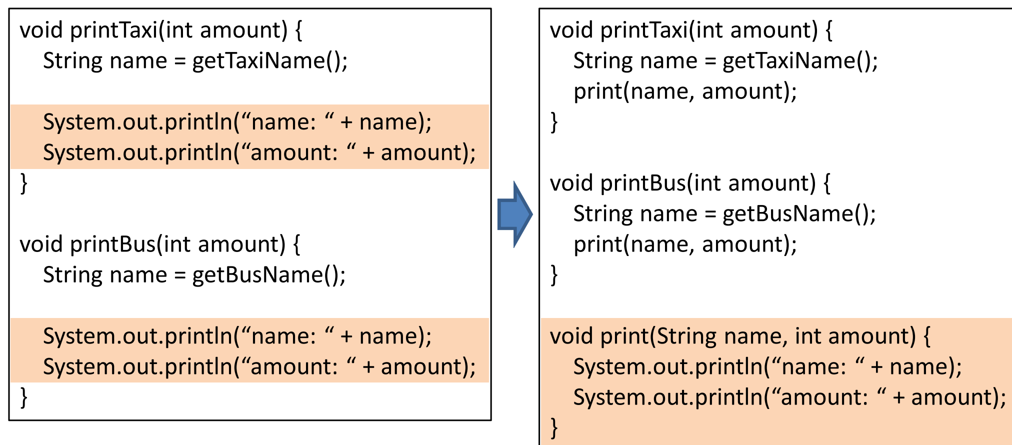


Figure 9: An Example of Extract Method

pattern is often used for improving reusability by segmentalizing too long and/or too complex methods into short and simple methods. We can remove code clones by extracting them as a new method and replace them by method call instructions for the method. Figure 9 shows an example of the application of Extract Method. In this example, there are same statements between two methods `printTaxi` and `printBus`. By applying Extract Method, these duplicate statements are extracted as a new method `print`, and the original statements are replaced by the method call. As a result, code clones between the two methods are merged into a single method. An advantage of this pattern as clone removal technique is that it can be applied if a part of a method contain code clones and the other part does not contain code clones. In addition, this pattern is capable of wide application because it does not use class hierarchies. Therefore, this pattern is useful in such a case that versatile processes that can be merged as a library are scattered across source code as code clones. However, this pattern introduces many methods if multiple clone fragments exist in a single method and there are some non-clone fragments between every two fragments.

Pull Up Method

Pull Up Method indicates pulling up identical methods existing in derived classes into their common base class as a new method. This pattern is effective if there are some methods that behave the same way in all the derived classes. By applying this pattern, duplicate methods are merged into a base class, which means that code clones existing in derived classes are removed. Figure 10 shows an example of the application of Pull Up Method. In this case, two duplicate methods `getName` in class `Salesman` and `Engineer` are pulled up into the same base class `Employee`. This pattern can be applied if and only if target methods are exactly same. Moreover, this pattern uses inheritance relationship of classes. Therefore, the range of application of this

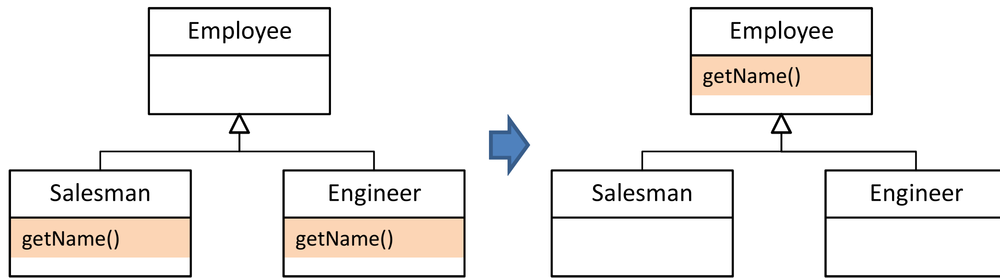


Figure 10: An Example of Pull Up Method

pattern is narrower than that of Extract Method refactoring pattern.

Parameterize Method

If there are similar methods in a single class, the duplication may be removed by Parameterize Method. Parameterize Method is used in a case that several methods do similar things but with different values contained in the method body. In this pattern, a new method that uses a parameter for different values is created.

Pull Up Constructor

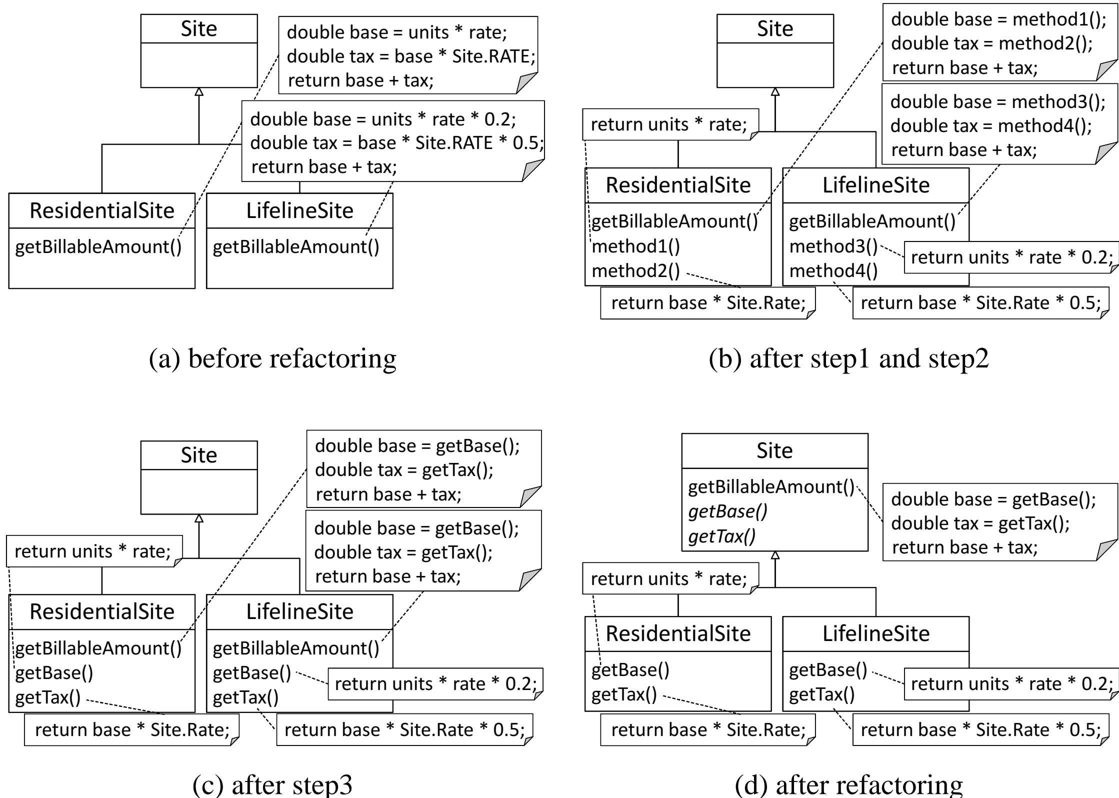
This pattern is very similar to the Pull Up Mehtod. The only difference is the target of this pattern is not a method but a constructor.

Form Template Method

Form Template Method refactroing pattern is a hybrid of Extract Method and Pull Up Method refactoring patterns. This pattern targets similar methods existing in derived classes that have a same base class. In this pattern, processes that are common in all the target methods are pulled up into the base class with Pull Up Method refactoring pattern. On the other hand, the processes that are not common in the target methods remain in each derived class. The remaining processes are unique in each derived classes. These unique processes are extracted as a new method with Extract Method refactoring pattern.

The steps for applying Form Template Method are as follows:

1. Detect common processes in all the target methods,
2. Extract unique processes as new methods with Extract Method refactoring pattern,
3. Rename methods to make correspondence of signatures. The targets of renaming are methods that created in 2. and called in the same point of the common processes and



(a) before refactoring

(b) after step1 and step2

(c) after step3

(d) after refactoring

Figure 11: An Example of Refactorings with Form Template Method

4. Pull up common processes as a new method in the base class with Pull Up Method refactoring pattern.

Figure 11 shows an example of refactorings with this pattern. 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*.

To apply Form Template Method to this target, at first, we have to distinguish the common and unique processes in the two methods. In this example, the differences of the two methods are in the calculation ways of variables *base* and *tax*.

Secondly, we extract each of the calculations of *base* and *tax* as new methods (shown in Figure 11(b)). We get 4 methods as a result of this step (currently, they are named as *method1*, *method2*, *method3*, and *method4*).

In the next step, we rename 4 new methods to make correspondence of signatures (shown in Figure 11(c)). In this example, *method1* in *ResidentialSite* and *method3* in *LifelineSite* are called as the first processing of the original methods. Also, *method2* and *method4* are called as the second processing of the original methods. Therefore, we rename *method1* and *method3* to make their

signatures correspondent. In this example, *method1* and *method3* are renamed as *getBaseAmount*. Similary, *method2* and *method4* are renamed as *getTaxAmount*.

Finally, we pull up the common processes as a new method. Note that we have to define *getBaseAmount* and *getTaxAmount* as abstract methods in the base class. Figure 11 shows the code that the refactoring has finished.

By applying this refactoring pattern to similar methods, code clones existing between these methods are merged into a base class. An advantage of clone removal with this pattern compared with *Pull Up Method* is that this pattern can be more widely than *Pull Up Method* because this pattern can be applied to methods that are not exactly same. Compared with clone removal with *Extract Method* refactoring pattern, the application range of this pattern is narrower. However, this pattern is effective in such a case that common processes are segmentalized by unique processes. This is because separated common processes can be merged as a single method with *Form Template Method* refactoring pattern, meanwhile each fragment of common processes is extracted as a method with *Extract Method* refactoring pattern.

In the rest of this thesis, we call a method created in base classes by pulling up the common processes as **template method**.

3 Related Work

3.1 Techniques for Refactoring on Code Clones

Fowler, a pioneer in the field of refactoring, mentioned that the "*number one in the stink parade is duplicate code*" [19]. He also presented some sets of operations for merging code clones. However, because it is quite difficult for maintainers to apply refactorings manually without introducing any human errors, many research efforts have been performed on refactoring assistance [22].

Higo et al. proposed a method for merging code clones [36]. Their method consists of 2 phases. The first phase is the quick detection of *refactoring-oriented code clones* from the source code. The second phase is the measurement of metrics indicating how the refactoring-oriented code clones should be merged. They implemented their method as a tool, ARIES. Using ARIES in the refactoring process, maintainers of the software system can readily know which and how code clones can be merged. They conducted a case study with ARIES, and they confirmed that ARIES performs the process successfully.

CLONEDR, which is an implementation of the AST-based technique, presents not only the locations of code clones but also forms of merged code fragments [4]. The forms help users understand what operations are required to merge code clones. However, the tool does not care about the positional relationship between code clones in the class hierarchy.

Bakazinska et al. proposed a refactoring method for the duplicate methods [37]. Their method provides the differences between code clones, which help users to determine whether code clones can be merged or not. Also, their method measures the coupling between a duplicated method and its surrounding code. In their method, code clones are removed by using two design pattern "Strategy" and "Template Method".

Cottrell et al. implemented a tool that visualizes the detailed correspondences between a pair of classes [38]. The classes are generalized to form an intermediate, AST-like structure that distinguishes between what is common and what is specific to each class. The specific instructions will influence the degree of relatively between the classes. The tool works after users identify 2 classes that should be merged.

Komondoor et al. proposed an algorithm for procedure extraction [39]. The inputs to the algorithm are (1) the CFG (control-flow graph) of a procedure and (2) a set of nodes in the CFG. The goal of the algorithm is to revise the CFG with the following conditions:

- The set of nodes that are extractable from the revised CFG;
- The revised CFG is semantically equivalent to the original CFG.

The implementation of this algorithm adopts heuristics for enhancing scalability. Although the

algorithm has a worst-case exponential time complexity, their experimental results indicated that it may work well in practice. However, the algorithm can be applied only to a single code clone. Different techniques are needed to determine how two or more code clones can be extracted as a single procedure with preserving semantics.

3.2 Techniques for Refactoring on Code Clones with Form Template Method

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 ASTs [24]. 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.

Masai et al. proposed a method to support refactorings with Form Template Method with ASTs likewise Juillerate et al. [25]. Their method consider the structural information of ASTs to detect unique processing, meanwhile Juillerat et al. compare ASTs with token sequences that are made from ASTs. Therefore, their method can extract code fragments that have some functionalities as unique processing. One of the differences between Masai et al.'s method and the proposed method is that their method suggests different sets of code fragments that should be merged in a specified method pair by expanding different part between methods consisting of the method pair. Also, they implemented a function to suggest suitable divisions between common- and different-part on the specified method pair to users with a cohesion metric COB [26].

3.3 Techniques for Code Clone Managing

At present, there is a huge body of work on empirical evidence on code clones, starting with Kim et al.'s report on clone genealogies [40]. They performed experiments on the repositories of open source software systems to investigate how code clones appear and disappear. The experimental results revealed the following points.

- Some code clones are short-lived. Merging (applying refactoring to) them does not improve the maintainability of the software systems.
- Most long-living code clones are not suited to be refactored because there is no abstraction function of the programming language that can handle them.

Kapser and Godfrey also suggested that, based on their experience, code clones are not always harmful [41]. They reported several situations where code duplication is a reasonable or even beneficial way to handle large-scale complex software systems. Also, Bettenburg et al. reported

that duplicate code does not have much a negative impact on software quality [42]. On the other hand, Monden et al. reported the opposite opinion, which is that the existence of code clones affects the quality of software systems [43]. They investigated the relation between software quality and code clones on the file unit. Their experiment selected a large scale legacy system, which was being operated in a public institution, as the target. The result showed that modules that included code clones were 40% lower quality than modules that did not include code clones. Moreover, they reported that the larger code clones a source file included, the lower quality it was. Lozano et al. investigated whether the presence of code clones was harmful or not [44], and they reported that methods including code clones tend to be more frequently modified than method including no code clone.

Krinke hypothesized that if code clones are less stable than non-cloned code, maintenance cost for code clones is greater than non-cloned code, and he conducted a case study in order to investigate whether the hypothesis is true or not [45]. The experimental result showed that non-cloned code was more *added*, *deleted*, and *modified* than cloned code. Consequently, he concluded that the presence of code clones did not necessarily make it more difficult to maintain source code.

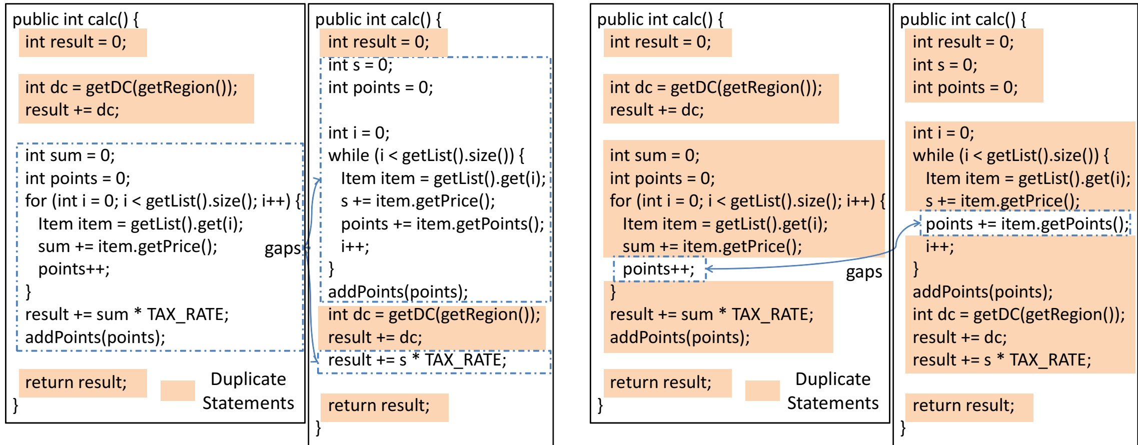
Göde et al. replicated Krinke's experiment [46]. Krinke's original experiment detected text-based code clones meanwhile their experiment detected token-based code cloens. The experimental result was the same as Krinke's one. Cloned-code is more stable than non-cloned code in the viewpoint of *added* and *deleted*. On the other hand, from *deleted* viewpoint, non-cloned code is more stable than cloned-code.

Also, Göde et al. conducted an empirical study on a clone evolution [47]. They performed a detailed tracking to detect when and how code clones had been changed. In their study, they traced clone evolutions and counted the number of changes on each clone genealogy. They manualy inspected the result in one of the target systems, and categorized all the modifications on clones into consistent or inconsistent. In addition, they carefully categorized inconsistent changes into intentional or unintentional. They reported that almost all code clones were never changed or only once during their lifetime, and only 3% of the modifications had high severity. Therefore, they concluded that many of clones does not cause additional change effort. They consisted that it is important to identify code clones with high threat potential for the effective code clone management.

Rahman et al. investigated the relationship between code clones and bugs. They analyzed 4 software systems written in C language with bug information stored in a bug management system, Bugzilla. They reported that only a small part of bugs located on code clones, and the presence of code clones did not dominate bug appearance [48].

Our research group also coudncted empirical studies to investigate the influence of code clones on software maintenance. We conducted an empirical study on 15 open source software systems with 4 clone detectors, and compare their modification frequency [49]. As a result, we found that code clones tend to be less frequently modified than non-cloned code. Consequently, we concluded that the presence of code clones did not necessarily have a negative impact on software maintenance. Moreover, we compared the experimental result of our investigation method with other 2 investigation methods [50]. We found that the result (whether the presence of code clones has a negative impact on software evolution or not) differs from every investigation method although the target software systems are same to one another.

At present, there is no consensus for the question whether the presence of code clones affects software maintenance or not. This is because the results of empirical studies vary according to research methods or target software systems. However, it can be said that removing all the code clones existing software systems is not effective because some researchers reported that code clones did not necessarily make it more difficult to maintain source code. Thus, it is important to focus on code clones that have a negative impact on software maintenance, or to remove them.



(a) The Method Proposed by Juillerat et al. [24]　　　(b) The Proposed Method

Figure 12: Motivating Example 1

4 Motivation

4.1 Issues of Previous Studies

As described in Section 3, there are some studies to support Form Template Method refactoring application. However, they still have some issues as follows.

- They cannot handle trivial differences that have no impact on the behavior of programs.
- They cannot handle groups of three or more similar methods in spite of that Form Template Method itself can be applied to them.

In the following subsections, we describe these issues in detail.

4.1.1 Issue of Trivial Differences

In previous studies, all the differences between target methods are regarded as unique processing even if some of them do not affect the meaning of programs. The following situations may be instances of the differences that do not affect the behavior of programs are as follows.

- The order of code statements is different in target methods. However, the behavior of the program is preserved even if we reorder the order of them.
- Iterations are implemented with for statements in a method of target methods, meanwhile they are implemented with while statements in another method of target methods. However, the meanings of the iterations are exactly same except the implementation styles.

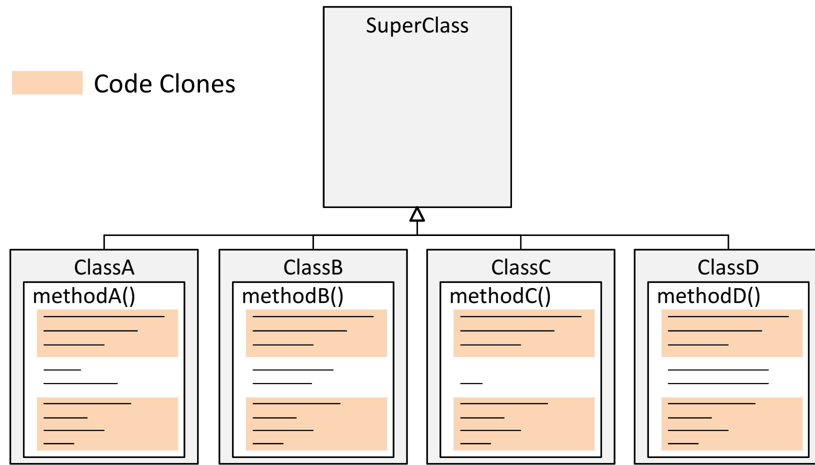
Figure 12 shows an example of our motivating example for this issue. In this example, there is a difference of the order of code statements, and there is also a difference of the implementation style of loop statements. However, these differences do not influence the meaning of the program. The only meaningful difference of these two methods is the ways of calculations of variable *points*. Nevertheless the methods described in previous studies regard these trivial differences as gaps between the two methods. Therefore, they can suggest only four lines as duplicate statements in the two methods (shown in Figure 12(a)). In this study, we aim to improve this issue by using PDGs, and we will suggest 11 lines except the calculations of variable *points* as duplicate statements (shown in Figure 12(b)).

4.1.2 Issue of Groups of Three or More Methods

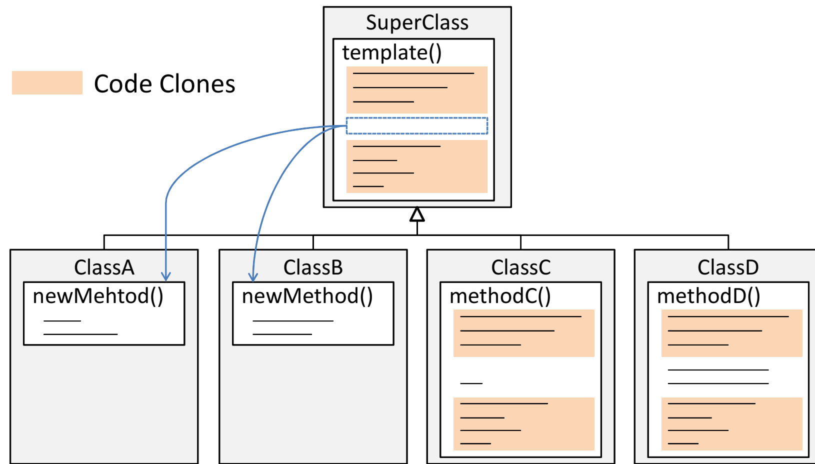
Form Template Method can be applied to a group of three or more similar methods. Nevertheless, the previous methods can handle only a pair of similar methods. Supporting Form Template Method application on only a pair of similar methods is not sufficient for clone removal. This is because code clones should remain after a refactoring with Form Template Method on a pair of methods if there are three or more similar methods.

In the example shown in Figure 13, there are four similar methods in four different classes, and these four classes have the same base class. If we apply Form Template Method refactoring on the pair of *method()* in *ClassA* and *method()* in *ClassB*, we get source code shown in Figure 13(b). As the figure shows, there are still code clones between *method()* in *ClassC* and *method()* in *ClassD* because we did not modify these two methods. Also, there are code clones between the template method and *method()* in *ClassC* and *ClassD*. Moreover, it is difficult to remove code clones from the source code of Figure 13(b) with Form Template Method refactoring. That is because a conflict of two template methods should occur if we apply Form Template Method on a pair of *method()* in *ClassC* and *method()* in *ClassD*. However, we can apply Form Template Method on all the four similar methods at a time. If we do so, we get the source code shown in Figure 13(c). As the figure shows, code clones are completely removed by the refactoring.

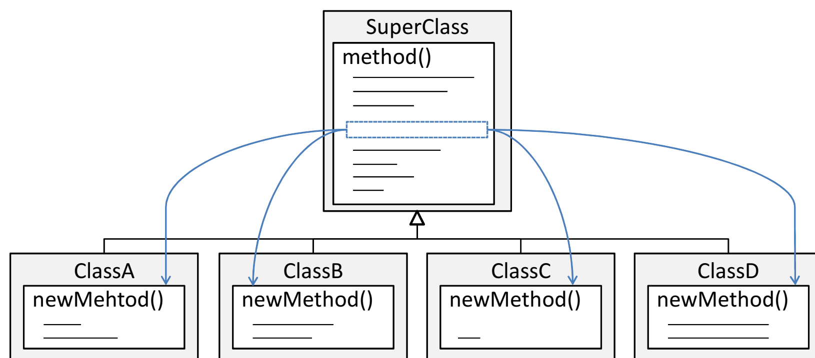
Moreover, some researchers reported that the quality of software systems after some refactorings is affected by the order of the refactorings [51,52]. In the case that refactorings on only a pair of methods are supported, the number of the candidates (pairs of methods) of Form Template Method refactorings is equal to the number of 2-combinations from a set of all the target methods. It is too difficult to detect the most appropriate order of refactorings from such a huge number of candidates. In the example of Figure 13, there are 6 pairs of methods that can be refactored with Form Template Method. However, it is difficult to decide which pair is most suitable to be refactored.



(a) before refactoring



(b) after refactoring on two methods



(c) after refactoring on all the four methods

Figure 13: Motivating Example 2

For these reasons, it is necessary to handle three or more methods at a time for effective clone removal with Form Template Method refactoring pattern. In this study, therefore, we expand proposed refactoring support technique on pairs of methods to be able to handle groups of three or more methods.

4.2 Objective of This Study

In this thesis, we propose a new refactoring support method with Form Template Method refactoring pattern. We aim to resolve the first issue of previous studies (described in Section 4.1.1), and we aim to resolve the second issue described in Section 4.1.2 by expanding the proposed method on pairs of methods to be able to handle groups of three or more methods.

Moreover, we aim to assist users in detecting refactoring candidates with Form Template Method. Users need to specify refactoring candidate (a pair of methods) for using the previous Form Template Method application assistance methods. The approach of previous studies is useful for actual modifications in source code associated with refactoring activities. However, it is not possible to reduce efforts required for identifying opportunities on which users want to apply Form Template Method refactorings in this approach. Because software systems become more large and more complex, it is difficult to comprehend structures of software systems appropriately. Hence, it is difficult to identify suitable clone removal candidates. This is the reason why we aim to support the detection of refactoring candidates.

To reduce efforts for identifying refactoring candidates, the proposed method detects refactoring candidates automatically, and suggests all the candidates to its users. Consequently, the proposed method can suggest refactoring candidates of which users are not aware. In addition, the proposed method also suggests common processing and unique processing in each of refactoring candidate to reduce efforts required for modifying source code to apply Form Template Method refactoring pattern.

Note that the proposed method aims to suggest candidates that **can** be refactored, not **should** be refactored. The reason is that there is not strict and generic standard to judge whether code clones should be removed. Also, there is not strict and generic standard to judge whether Form Template Method should be used to remove code clones. Accordingly, the proposed method leaves such decisions to its users whether they need to apply refactorings on each candidate that the proposed method suggests.

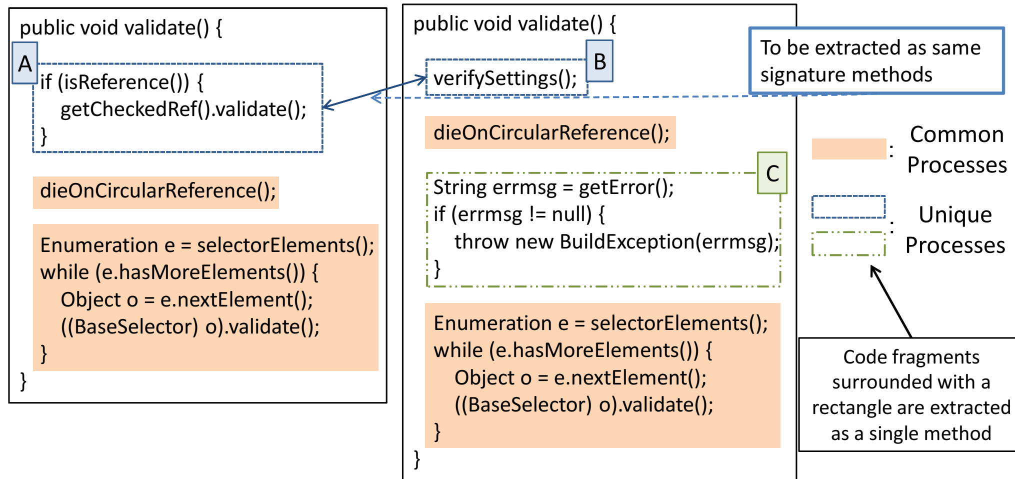


Figure 14: The Output of the Proposed Method

5 Outline of the Proposed Method

5.1 Inputs and Outputs

The proposed method takes source code of target software systems as its input. Then, the proposed method detects all the candidates of Form Template Method refactoring, and it suggests them to users. For each of the refactoring candidates, the proposed method suggests program statements that can be merged into the base class as the common processes, and program statements that should be remain in each derived class as the unique processes. Additionally, for the unique processes, the proposed method suggests the following two information.

- Sets of program statements that should be extracted as a single method.
- Relationships of new methods created by extracting the unique processes between the derived classes. This relationship means that the new methods under this relationship can be extracted as methods whose signatures are the same as each other.

Figure 14 shows the output information of the proposed method. In this example, there are two similar methods named *validate*, and the owner classes of these two methods have the same base class. The proposed method detects the common and unique processes between these methods. Herein, program statements highlighted with orange are the common processes that should be merged into the base class. Program statements that are not included in the common processes are regarded as the unique processes in each derived classes. For the unique processes, the proposed method detects sets of program statements that can be extracted as a single method. In this case, we get three sets of program statements (labeled with ‘A’, ‘B’ and ‘C’ in the figure). The proposed

method also detects relationships of new methods created by extracting the unique processes. In this example, the proposed method detect a relationship between 'A' and 'B', which means that the new methods created by extracting 'A' and 'B' should have the same signature to each other. Here, there is no correspondence of 'C'. In this case, we have to write an empty method that has the same signature of the method created from 'C' in the owner class of the left method.

5.2 Processing Flow

The processing of the proposed method can be separated into method-pairs version and method-groups version. The method-groups version is implemented as an extended version of method-pairs version.

The processing flow of the proposed method on pairs of methods is shown below.

STEP-P1: Analyze target source code, and create PDGs.

STEP-P2: Detect code clones with PDGs.

STEP-P3: Identify pairs of methods on which Form Template Method can be applied.

STEP-P4: Detect common processes and unique processes for each of method pairs.

STEP-P5: Detect sets of statements included in unique processes that should be extracted as a single method.

STEP-P6: Detect pairwise relationships between new methods created by extracting unique processes.

STEP-P7: Show all the analysis results.

The method-groups version uses the result of the method-pairs version. Therefore, processing steps from STEP-S1 to STEP-S6 are exactly identical to the processing steps from STEP-P1 to STEP-P6. The processing flow of the proposed method on method groups after STEP-S6 is shown below.

STEP-S7: Detect groups of methods on which Form Template Method can be applied with the information about pairs of methods.

STEP-S8: Detect common processes and unique processes for each of method groups.

STEP-S9: Detect relationships between new methods created by extracting unique processes.

STEP-S10: Show all the analysis results.

We describe each step in detail in Sections 6 and 7.

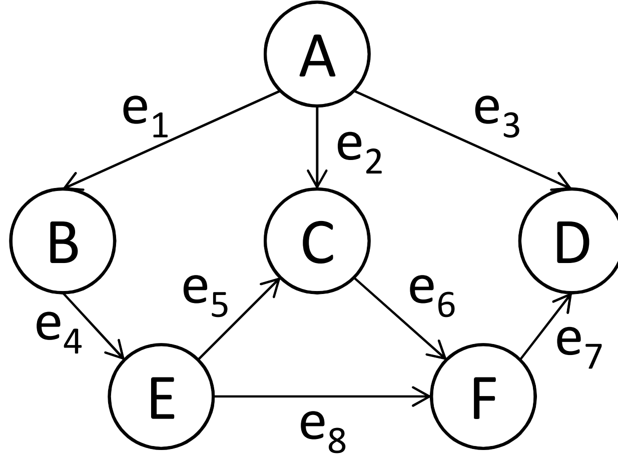


Figure 15: A Directed Graph

5.3 Definitions

Here, we describe definitions of terms referenced in the following explanations.

5.3.1 A Directed Graph

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 thesis, 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.

Figure 15 shows an example of directed graphs. Given that the graph of the figure is G, V_G, E_G, and f_G become as follows.

$$V_G = \{A, B, C, D, E, F\}$$
$$E_G = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8\}$$
$$f_G(e_1) = (A, B),\ f_G(e_2) = (A, C),\ f_G(e_3) = (A, D),\ f_G(e_4) = (B, E)$$
$$f_G(e_5) = (E, C),\ f_G(e_6) = (C, F),\ f_G(e_7) = (F, D),\ f_G(e_8) = (E, F)$$

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 5.1 ($tail(e)$, $head(e)$)**.** 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 $f_G(e) = (u, v)$.

For example, for an edge e_1 in the graph of Figure 15, $tail(e_1) = A$ and $head(e_1) = B$.

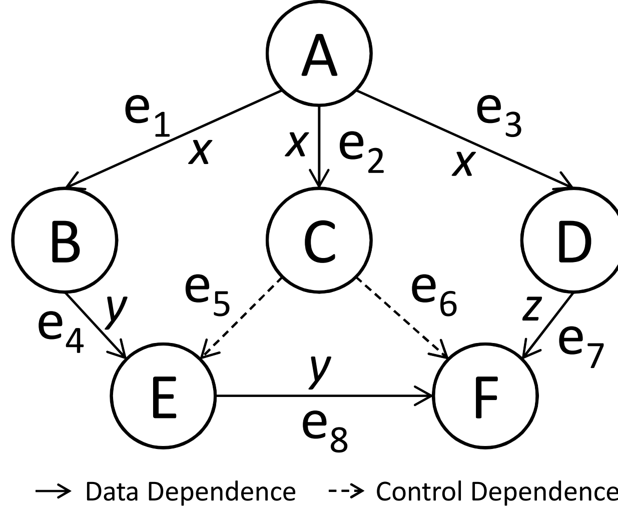


Figure 16: A PDG

In the next, 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 5.2 ($BackwardEdges(v)$, $ForwardEdges(v)$)**.**

$$BackwardEdges(v) := \{e \in E_G \mid head(e) = v\} \tag{1}$$

$$ForwardEdges(v) := \{e \in E_G \mid tail(e) = v\} \tag{2}$$

For a node C in the graph of Figure 15, $BackwardEdges(C)$ and $ForwardEdges(C)$ become as follows.

$$BackwardEdges(C) = \{e_2, e_5\}$$

$$ForwardEdges(C) = \{e_6\}$$

5.3.2 A PDG

A PDG is one of the directed graphs. Given a PDG $G = (f, V, E)$, a node of G corresponds to an element of programs, and an edge of G corresponds to a dependence between two elements. In this study, an element of programs indicates a statement of programs. 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 dependences in PDGs.

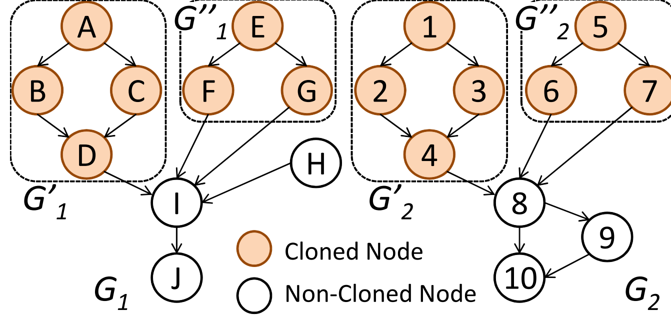


Figure 17: $ClonePairs(G_1, G_2)$

Definition 5.3 (Data Dependence and Control Dependence)**.** We write data dependences as *data*, and control dependences as *control*. We define *type* as a map from edges to the types of dependences that the edges represent ($type : E \rightarrow EdgeType$), where $EdgeType = \{data, control\}$. In addition, a data dependence edge has the information about the variable that the edge represents. We define $var(e_d)$ as the represented variable by a data dependence edge e_d.

In the PDG of Figure 16, *type* and *var* become as follows.

$$type(e_1) = data,\ type(e_2) = data,\ type(e_3) = data, type(e_4) = data$$
$$type(e_5) = control,\ type(e_6) = control,\ type(e_7) = data, type(e_8) = data$$
$$var(e_1) = x,\ var(e_2) = x,\ var(e_3) = x,\ var(e_4) = y,\ var(e_7) = z,\ var(e_8) = y$$

5.3.3 Clone Pairs

In the proposed method, code clones are detected with PDGs. PDG-based clone detectors regard isomorphic subgraphs of PDGs as code clones. Here, we define $ClonePairs(G_1, G_2)$ as a set of isomorphic subgraphs between PDGs G_1 and G_2.

Definition 5.4 ($ClonePairs(G_1, G_2) and a clone pair$)**.** We define $ClonePairs(G_1, G_2)$ in the formula (3), and we call every element of $ClonePairs(G_1, G_2)$ as a clone pair.

$$ClonePairs(G_1, G_2) := \{(G_1', G_2') \mid G_1' \subset G_1 \land G_2' \subset G_2 \land G_1' \cong G_2'\} \quad (3)$$

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.

In the example of Figure 17, there are two isomorphic subgraphs between G_1 and G_2. Therefore, $ClonePairs(G_1, G_2)$ become as follows.

$$ClonePairs(G_1, G_2) = \{(G_1', G_2'), (G_1'', G_2'')\}$$

where, $V_{G'_1} = \{A, B, C, D\}$, $V_{G'_2} = \{1, 2, 3, 4\}$, $V_{G''_1} = \{E, F, G\}$, and $V_{G''_2} = \{5, 6, 7\}$.

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

Definition 5.5 (Duplication of nodes)**.** The two nodes $v_1 \in V_{G_1}$ and $v_2 \in V_{G_2}$ are duplicated to each other if and only if they satisfy the formula (4). We represent $v_1 \sim v_2$ if v_1 and v_2 are duplicated to each other.

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

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

In the example of Figure 17, the binary relation $\sim$ becomes as follows.

$$\sim = \{(A, 1), (B, 2), (C, 3), (D, 4), (E, 5), (F, 6), (G, 7)\}$$

6 Supporting for Method Pairs

6.1 STEP-P1: Create PDGs

The proposed method internally uses an existing PDG-based clone detector, Scorpio [34], to detect code clones. In addition, Scorpio internally uses a source code analysis tool, MASU [53], to create PDGs. The first step of the proposed method is covered with MASU.

In PDGs created by MASU, a node corresponds to a statement of programs. Additionally, PDGs created by MASU have another dependence, "execution dependence", in addition of traditional two dependences, data and control dependences. Execution dependences indicate execution-next links.

Note that PDGs used in the proposed method need not to be always created by MASU. We can apply the proposed method on PDGs created by other tools if we can detect code clones on them with the way described in the next subsection.

6.2 STEP-P2: Detect Code Clones

As described above, we use Scorpio to detect code clones. Therefore, the second step of the proposed method is fully covered with Scorpio. Here, we describe the clone detection algorithm used in Scorpio briefly.

First, Scorpio calculates hash values for every node of PDGs. The hash values are calculated with information about the structure of the statement that every node represents. Scorpio replace variables' name or literals by their types, which enables to detect code clones with different variables' name or literals. Next, Scorpio classifies every node with its hash value. Nodes having the same hash value are classified as an equivalence class. Then, every pair (r_1, r_2) of nodes are selected from every equivalence class, and two isomorphic subgraphs that include r_1 and r_2 are identified. Both forward and backward slices are used to identify isomorphic subgraphs. Details of the slicing are described in Appendix A. Isomorphic subgraphs detected in this step is regarded as a clone pair. We set a minimal size of each isomorphic subgraph to 6 nodes to be detected as code clones. In the next step, Scorpio removes uninteresting clone pairs. The algorithm is that if a clone pair (s_1, s_2) is subsumed by another clone pair (s'_1, s'_2), it is removed from the set of clone pairs. Finally, clone sets are generated from clone pairs sharing the same isomorphic subgraphs.

Note that it is not necessary to detect code clones with this way to use the proposed method. The proposed method only needs $ClonePairs(G_{m_1}, G_{m_2})$ for any pair of methods (m_1, m_2) contained in the target programs, where G_{m_i} indicates a PDG of method m_i. The proposed method does not care how they are identified.

6.3 STEP-P3: Identify Method Pairs

In this step, we detect pairs of methods on which Form Template Method can be applied with the information about code clones detected by Scorpio. We regard a pair of methods as a refactoring candidate if it satisfies following requirements.

Requirement A: The two methods in the method pair are defined in different classes.

Requirement B: The owner classes of the two methods have the same base class.

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

We discuss these requirements in detail.

Requirement A

Form Template Method can not be applied on methods defined in the same class because it uses the inheritance relationships and the polymorphism. Thus, the method pair that the proposed method targets has to be defined in different classes.

Requirement B

Form Template Method targets similar methods whose owner classes have the same base class. It is possible that we apply Form Template Method on methods whose owner classes do not have the same base class. The way is that we insert a new class into class hierarchy and make the owner classes inheriting the new class. However, refactorings with this way may decay the quality of programs because two non-related classes are forced to be jointed in the class hierarchy. For this reason, the target method pairs are limited to having the same base class.

Requirement C

If there is no duplicate statement, Form Template Method can not be applied on such a method pairs because no statement is pulled up into the base class. Therefore, we make a requirement that there is at least one clone pair between the two methods of every target method pair.

Suppose that G_{m_1} and G_{m_2} are PDGs of methods m_1 and m_2. If there is no clone pair between a method pair (m_1, m_2), $ClonePairs(G_{m_1}, G_{m_2})$ is empty. Therefore, we can check whether there is at least one clone by checking whether $ClonePairs(G_{m_1}, G_{m_2})$ is empty or not. In other words, the method pair (m_1, m_2) must satisfy the formula (5).

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

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: $CopyOfClonePairs = \emptyset$
2: $CopyOfClonePairs \stackrel{+}{\leftarrow} ClonePairs(G_{m_1}, G_{m_2})$
3: **for all** $(G'_{m_1}, G'_{m_2}) \in CopyOfClonePairs$ **do**
4:     **for all** $(G''_{m_1}, G''_{m_2}) \in CopyOfClonePairs$ **do**
5:         **if** $(\exists v_1 \in G'_{m_1}[v_1 \in G''_{m_1}])\&(G'_{m_1} \neq G''_{m_1})$ **then**
6:             **if** $|G'_{m_1}| < |G''_{m_1}|$ **then**
7:                 $ClonePairs(G_{m_1}, G_{m_2}) \stackrel{-}{\leftarrow} (G'_{m_1}, G'_{m_2})$
8:             **else**
9:                 $ClonePairs(G_{m_1}, G_{m_2}) \stackrel{-}{\leftarrow} (G''_{m_1}, G''_{m_2})$
10:             **end if**
11:         **end if**
12:         **if** $(\exists v_2 \in G'_{m_2}[v_2 \in G''_{m_2}])\&(G'_{m_2} \neq G''_{m_2})$ **then**
13:             **if** $|G'_{m_2}| < |G''_{m_2}|$ **then**
14:                 $ClonePairs(G_{m_1}, G_{m_2}) \stackrel{-}{\leftarrow} (G'_{m_1}, G'_{m_2})$
15:             **else**
16:                 $ClonePairs(G_{m_1}, G_{m_2}) \stackrel{-}{\leftarrow} (G''_{m_1}, G''_{m_2})$
17:             **end if**
18:         **end if**
19:     **end for**
20: **end for**

6.4 STEP-P4: Detect Common and Unique Processes

In this step, the proposed method detects common and unique processes in each method pair. Suppose that a method pair of m_1 and m_2 is the given method pair, and $G_{m_{1(2)}}$ is the PDG of method $m_{1(2)}$.

The proposed method regards statements as common processes if and only if included in code clones existing between the two methods of the given method pair. We define $CommonNodes(G_{m_{1(2)}})$ as a set of nodes in $G_{m_{1(2)}}$ whose representing statements form common processes. The formula (6) represents the definition, where $G_{m_{1(2)}}$ indicates the PDG of method $m_{1(2)}$.

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

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.

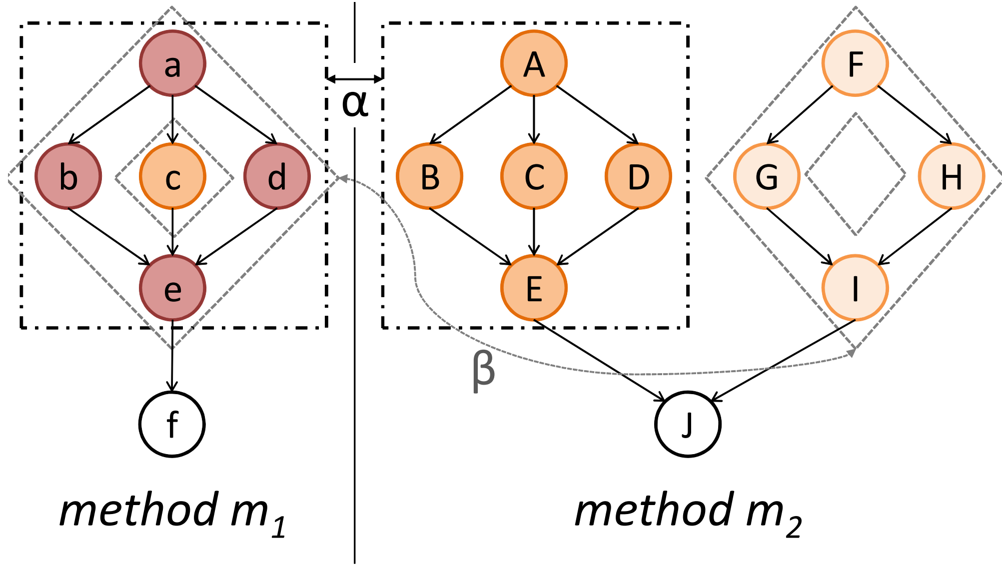


Figure 18: An example of Method Pairs Including Redundant Clone Pairs

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 $R \stackrel{-}{\leftarrow} r$ 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 18 shows an instance of method pairs that contain redundant clone pairs. There are two clone pairs; one is labeled with 'α', and another one is labeled with '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 removes β from $ClonePairs(G_{m_1}, G_{m_2})$ because the number of elements of α is larger than those of β. As a result, the common statements that the proposed method detects in this method pair (m_1, m_2) become as follows.

$$CommonNodes(G_{m_1}) = \{a, b, c, d, e\}$$
$$CommonNodes(G_{m_2}) = \{A, B, C, D, E\}$$

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

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

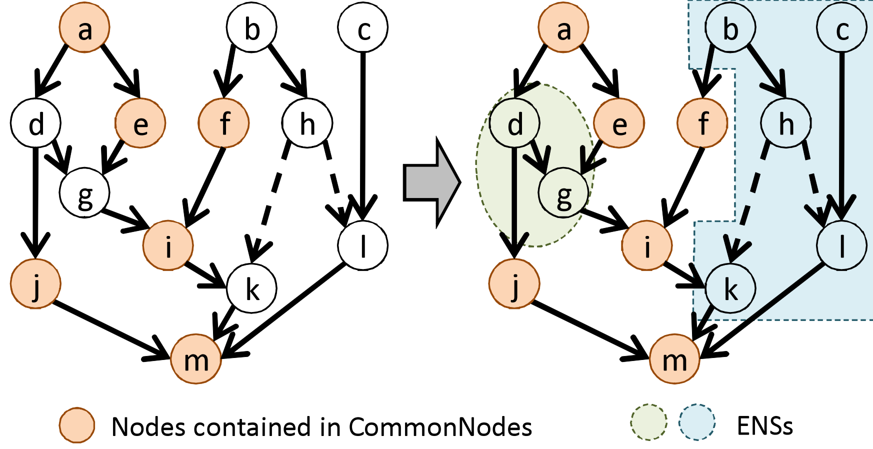


Figure 19: An example of the Detection of ENSs

In the method pair (m_1, m_2) shown in Figure 18, $\mathit{DiffNodes}(G_{m_{1(2)}})$ becomes as follows.

$$\begin{aligned} \mathit{DiffNodes}(G_{m_1}) &= \{f\} \\ \mathit{DiffNodes}(G_{m_2}) &= \{F, G, H, I, J\} \end{aligned}$$

6.5 STEP-P5: Detect Sets of Statements Extracted as a Single Method

In this step, the proposed method detects sets of statements that can be extracted as a single method in the unique processes. For applying Form Template Method refactorings, it is necessary that nodes remaining in derived classes are extracted as new methods. Therefore, we have to detect sets of program statements included in $\mathit{DiffNodes}(G_{m_{1(2)}})$, each of which can be extracted as a single method. In the reminder of this thesis, we call a set of nodes that should be extracted as a single method as an **Extract Node Set** (in short, **ENS**).

6.5.1 Definition of the Extract Node Set

In the proposed method, we regard nodes included $\mathit{DiffNodes}(G_{m_{1(2)}})$ as an ENS if there is at least one path that does not include nodes in $\mathit{CommonNodes}(G_{m_{1(2)}})$ for any pairs of the nodes in it ignoring directions of each edges. In other words, we regard a set of nodes $S_{m_{1(2)}} \subset V_{G_{m_{1(2)}}}$ as an ENS 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 $S_{m_{1(2)}}$.

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

In the example shown in Figure 19 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

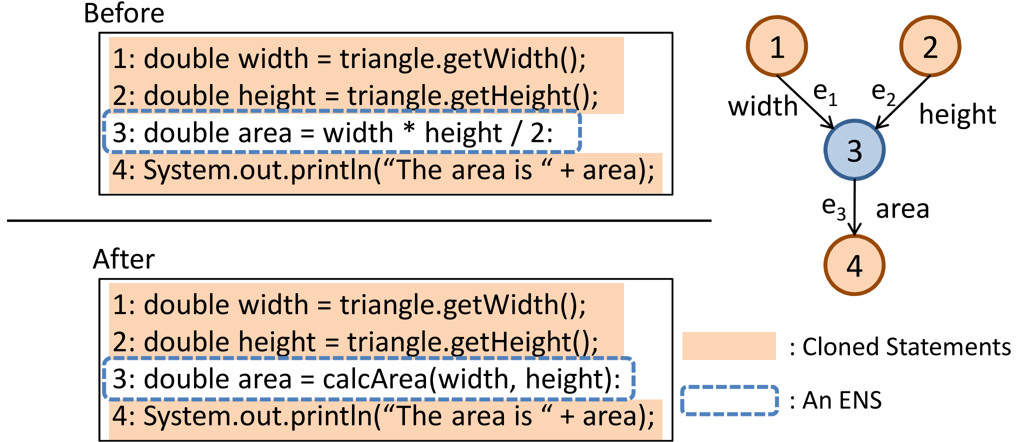


Figure 20: An Example of Inputs and Outputs of ENSs

contain multiple ENSs. We define $\mathit{DiffNodeSets}(G_{m_{1(2)}})$ as a family of ENSs in method $m_{1(2)}$. Suppose that m_1 indicates the name of the method shown in the figure, and G_{m_1} indicates its PDG, then, in the example of Figure 19, $\mathit{DiffNodeSets}(G_{m_1})$ becomes as follows.

$$\mathit{DiffNodeSets}(G_{m_1}) = \{\{d, g\}, \{b, c, h, k, l\}\}$$

Note that any node in $\mathit{DiffNodes}(G_{m_{1(2)}})$ must be included in a ENS in $\mathit{DiffNodeSets}(G_{m_{1(2)}})$ (forumla (10)).

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

6.5.2 Parameters of ENSs

Parameters of the method created by extracting an ENS S can be defined as variables represented by data dependence edges whose heads are included in S and whose tails are not included in S. Assume that G indicates a PDG, and S indicates an ENS of G. Under these assumptions, we define a set of data dependence edges whose tails are not included in S and whose heads are included in S as $\mathit{InputDataEdges}(G, S)$. Formula (11) shows the definition of $\mathit{InputDataEdges}(G, S)$.

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

Here, we define a set of variables of which parameters of the method created by extracting S consist as $\mathit{InputVariables}(S)$ in forumla (12).

$$\mathit{InputVariables}(G, S) := \{p \mid \exists e \in \mathit{InputDataEdges}(G, S)[\mathit{var}(e) = p]\} \tag{12}$$

In the example of Figure 20, there is an ENS consisting of the 3rd line. In this case, $InputDataEdges(G, S)$ and $InputVariables(G, S)$ become as follows

$$\begin{aligned} InputDataEdges(G, S) &= \{e_1, e_2\} \\ InputVariables(G, S) &= \{width, height\} \end{aligned}$$

Thus, a method created by extracting the ENS need 2 parameters, one is $width$, and the other is $height$.

6.5.3 Output of ENSs

Suppose that G indicates a PDG of a method, and S indicates an ENS of G. The output values of the method created by extracting S are defined as variables that are represented by data dependence edges whose heads are not included in S and whose tails are included in S.

First, we define $OutputDataEdges(G, S)$ as a set of data dependence edges whose tails are included in S and whose heads are not included in S. The definition is shown in formula (13).

$$OutputDataEdges(G, S) := \{e \in E_G \mid tail(e) \in S \wedge head(e) \notin S \wedge type(e) = data\} \quad (13)$$

Herein, we can define a set of output variables of S with this definition. We define it as $OutputVariables(G, S)$ in the formula (14).

$$OutputVariables(G, S) := \{p \mid \exists e \in OutputDataEdges(G, S)[p = var(e)]\} \quad (14)$$

In the example of Figure 20, $OutputDataEdges(G, S)$ and $OutputVariables(G, S)$ become as follows.

$$\begin{aligned} OutputDataEdges(G, S) &= \{e_3\} \\ OutputVariables(G, S) &= \{area\} \end{aligned}$$

Therefore, a method created from the ENS need to return a value of `double`.

6.5.4 Conditions for Call

The conditions to call methods created by extracting ENSs are represented by control dependence edges. For example, if there are control dependences from a conditional predicate of `if` statement to all the nodes included in an ENS S, a method created from S should be called in the case that the conditional predicate is satisfied.

First, we define $InputControlEdges(G, S)$ as a set of control dependence edges whose tails are not included in S and whose heads are included in S in the formula (15), where G is a PDG of a method and S is an ENS of G.

$$InputControlEdges(G, S) := \{e \in E_G \mid tail(e) \notin S \wedge head(e) \in S \wedge type(e) = control\} \quad (15)$$

In the next, we define nodes that have control dependences to nodes included in S as $\mathit{InputControlNodes}(G, S)$. The definition is shown in formula (16).

$$\mathit{InputControlNodes}(G, S) := \{v \in V_G \mid \exists e_c \in \mathit{InputControlEdges}(G, S)[v = \mathit{tail}(e_c)]\} \quad (16)$$

As described above, a PDG has a method enter node, and there is control dependence from the node to all the nodes that directly contained by the method. In addition, nodes contained in conditional blocks have control dependence from the conditional predicates of the blocks. In this case, there is no control dependence from the method enter node to nodes contained in conditional blocks because these nodes are not directly contained by the method. Therefore, all the nodes except the method enter node have at least and at most 1 control dependence from other nodes.

6.5.5 Requirements for ENSs to be Extracted as a Single Method

In some cases, we cannot extract each of ENSs as a single method. Concretely, we cannot extract an ENS S as a single method if it satisfies the following conditions.

- There are multiple return values in the method created from S.
- S includes a part of nodes in a block statement, and it also includes some nodes out of the block statement.

Multiple Return Values

It is necessary that an ENS S has at most one return value to be extracted as a method. Therefore, if there are two or more return values of S, it cannot be extracted as is.

To resolve this problem, we divide S into multiple ENSs satisfying the condition. Here, we describe the algorithm.

First, we define a set of nodes in S that are boundary end points of data dependences between S and out of S as $\mathit{BoundaryNodes}(G, S)$. Formula (17) is its definition.

$$\mathit{BoundaryNodes}(G, S) := \{v \in V_G \mid \exists e \in \mathit{OutputDataEdges}(G, S)[\mathit{tail}(e) = v]\} \quad (17)$$

Then, we divide S with the Algorithms 2, 3, and 4. Note that $R \xleftarrow{+} r$ means adding an element r into a set R. Besides, $\mathit{detect}(v, S)$ and $\mathit{parse}(S, R)$ indicates the following processing, respectively.

$\mathit{detect}(v, S)$: Return an ENS S' satisfying the condition that $\mathit{BoundaryNodes}(G, S') = \{v\}$ by dividing the original ENS S.

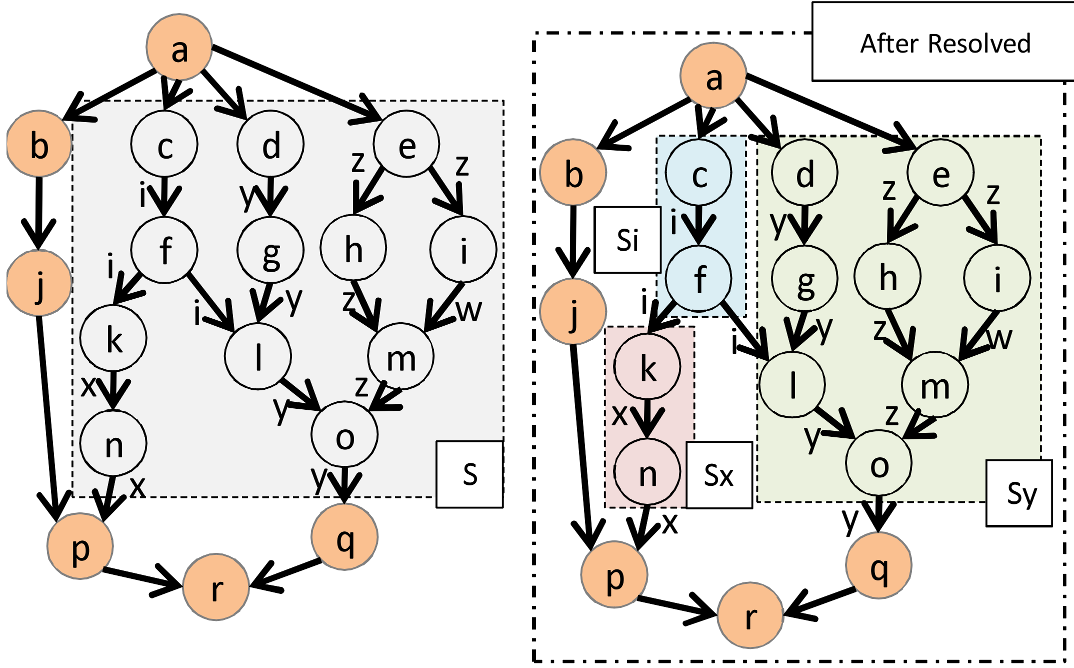


Figure 21: Behavior of Algorithm 2

$parse(S, R)$**:** Return a node set R' created by adding some nodes into the specified node set R. The added nodes must be reached from a node in R by tracing an edge in the reverse direction. Moreover, the addition of nodes must preserve the condition that $|BoundaryNodes(G, R)| = 1$.

Here, we describe the behavior of the algorithm with the example shown in Figure 21.

In the beginning, $BoundaryNodes(G, S) = \{n, o\}$. Here we consider the case that $detect(n, S)$ is called in the 3rd line in Algorithm 2.

In $detect(n, S)$, a node set R is initialized with n, then R is expanded by trace edges in the reverse direction. Obviously, $|BoundaryNodes(G, R)| = 1$.

In the next, $parse(S, R)$ is called. At first, we reach a node k and we need to judge whether it can be added into R or not. In this case, k has no dependences to nodes except n, therefore we judge that it can be included in R. In the next, we reach another node f. The node f has two dependences whose tail is f; one is to k, and the other is to another node i. Herein, the node i are not included in R. Consequently, if we add f into R, $BoundaryNodes(G, R)$ becomes $\{f, n\}$. Thus we judge that we cannot add f into R. $parse(S, R)$ stops here because there is no nodes that can be a candidate of expansion, and it returns $R' = \{k, n\}$.

Then the algorithm backs to the 3rd line in Algorithm 2. Here, $detect(o, R)$ is called, and it

Algorithm 2 Division of an ENS

Require: G, S

Ensure: $SeparetedNodeSets$

1: **while** $S \neq \emptyset$ **do**
2: **for all** $v \in BoundaryNodes(G, S)$ **do**
3: $SeparetedNodeSets \xleftarrow{+} detect(v, S)$
4: **end for**
5: **for all** $T \in SeparetedNodeSets$ **do**
6: **for all** $v' \in T$ **do**
7: $S \xleftarrow{-} v'$
8: **end for**
9: **end for**
10: **end while**

Algorithm 3 *detect*(v, S)

Require: v, S

Ensure: R

1: $R \leftarrow \{v\}$
2: **while** $|R| \neq |parse(S, R)|$ **do**
3: **for all** $v' \in parse(S, R)$ **do**
4: $R \xleftarrow{+} v'$
5: **end for**
6: **end while**

Algorithm 4 *parse*(S, R)

Require: S, R

Ensure: R'

1: $R' = R$
2: **for all** $v \in R$ **do**
3: **for all** $e \in BackwardEdges(v)$ **do**
4: **if** $tail(e) \in S \wedge tail(e) \notin R$ **then**
5: **if** $\forall e_d \in ForwardDataEdges(tail(e))[head(e_d) \in R]$ **then**
6: $R' \xleftarrow{+} tail(e)$
7: **end if**
8: **end if**
9: **end for**
10: **end for**

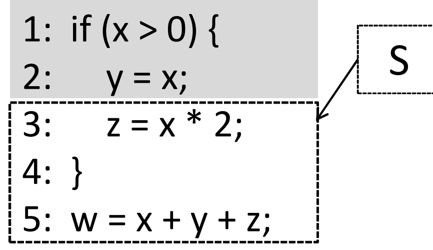


Figure 22: An Instance of Segmentalization of Block Statements

returns $R' = \{d, e, g, h, i, l, m, o\}$.

Then the algorithm goes to the 5th line in Algorithm 2. Here, nodes included in any element in $SeparetedNodeSets$ are removed from the original ENS S. In this case, $SeparetedNodeSets = \{\{k, n\}, \{d, e, g, h, i, l, m, o\}\}$, therefore S becomes $S = \{f, c\}$.

The algorithm repeats this process until $S \neq \emptyset$. Finally, we get 3 ENSs S_x, S_y, and S_i from the original ENS S, and all of them have to return only a single value x, y, and i, respectively.

Segmentalization of Block Statements

Suppose that $Nodes(b)$ indicates a set of nodes that are included in the given block statement b. If an ENS S satisfies all the following formulae (18) and (19), we cannot extract it as a single method.

$$\exists v \in Nodes(b)(v \in S) \land \exists u \in Nodes(b)(u \notin S) \tag{18}$$

$$\exists v \in S(v \in Nodes(b)) \land \exists u \in S(u \notin Nodes(b)) \tag{19}$$

Figure 22 shows an instance of ENSs that satisfy these formulae. As this figure shows, we cannot extract S as is. This is because one node in S is included in if statement, and the other node is not included in the statement. To resolve this problem, we restrict nodes in each of ENSs to be in the same block statement. By this restriction, the 3rd line and the 5th line in Figure 22 can be included in the same ENS. Therefore, we get 2 ENSs in this example, and each of them can be extracted as a single method.

6.6 STEP-P6: Detect Pairwise Relationships

In this 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 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 P6-1: The types of return values of S_{m_1} and S_{m_2} are the same as each other.

Requirement P6-2: The conditions to call the new methods created by extracting S_{m_1} and S_{m_2} are the same as each other.

We describe these requirements in detail in the following subsections. Herein, $EM_{S_{m_{1(2)}}}$ means the method created by extracting the ENS $S_{m_{1(2)}}$.

6.6.1 Requirement P6-1: Requirement the Type of the Return Value

To make $EM_{S_{m_1}}$ and $EM_{S_{m_2}}$ having the same signature, it is necessary that the types of return values of $EM_{S_{m_1}}$ and $EM_{S_{m_2}}$ are same to each other.

As described in 6.5.3, the return values of $EM_{S_{m_{1(2)}}}$ are defined as $OutputVariables(G_{m_{1(2)}}, S_{m_{1(2)}})$ (formula (14)). In addition, the number of elements in $OutputVariables(G_{m_{1(2)}}, S_{m_{1(2)}})$ is at most 1 because of the processing described in 6.5.5.

We define that $EM_{S_{m_1}}$ and $EM_{S_{m_2}}$ have the same type of the return value if they satisfy formula (20).

$$\begin{aligned}(|OutputVariables(G_{m_1}, S_{m_1})| = |OutputVariables(G_{m_2}, S_{m_2})|)\\ \wedge (\forall p \in OutputVariables(G_{m_1}, S_{m_1}) \exists q \in OutputVariables(G_{m_2}, S_{m_2})\\ [varType(p) = varType(q)]) \quad (20)\end{aligned}$$

Note that we do not consider parameters of $EM_{S_{m_1}}$ and $EM_{S_{m_2}}$ to detect the pairwise relationships. This is because we can make them having the same signature by adding non-used parameters in the case that the parameters of $EM_{S_{m_1}}$ and $EM_{S_{m_2}}$ are different. For example, suppose that $EM_{S_{m_1}}$ needs one parameter whose type is integer and $EM_{S_{m_2}}$ needs one parameter whose type is string. In this case, we can match the signatures of $EM_{S_{m_1}}$ and $EM_{S_{m_2}}$ by adding a string parameter in $EM_{S_{m_1}}$ and an integer parameter $EM_{S_{m_2}}$.

6.6.2 Requirement P6-2: Requirement about Conditions for Call

To extract $EM_{S_{m_1}}$ and $EM_{S_{m_2}}$ as same signature methods, it is necessary that $EM_{S_{m_1}}$ and $EM_{S_{m_2}}$ are called under the same conditions.

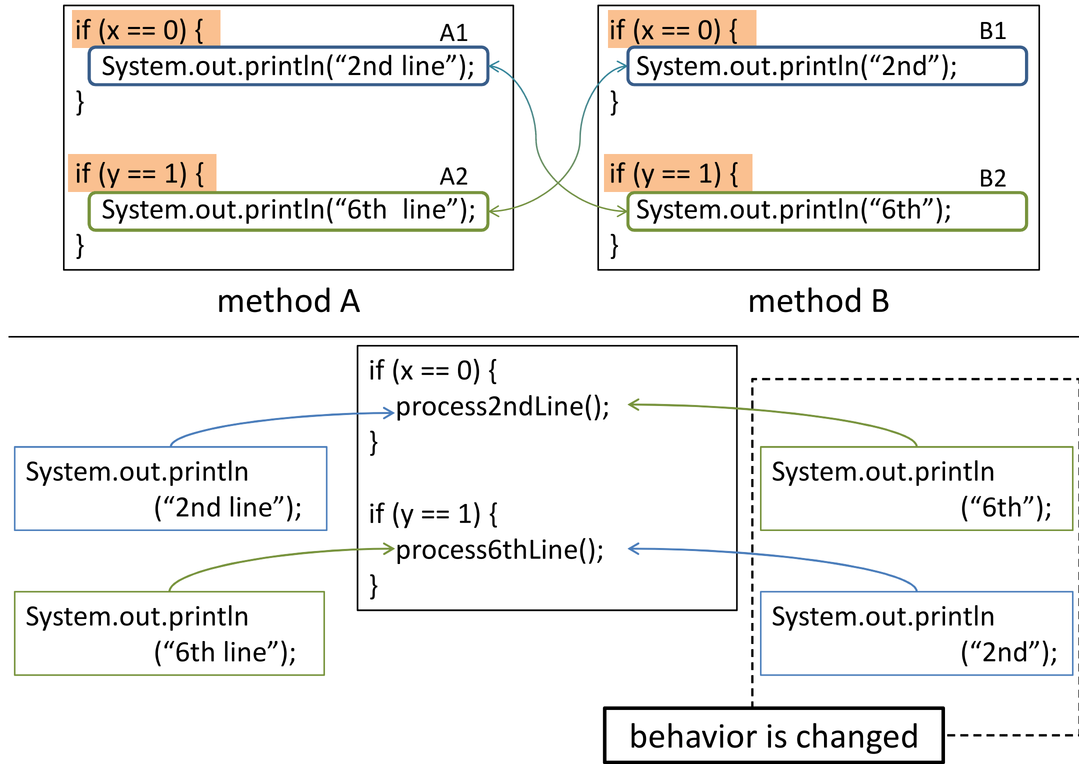


Figure 23: An Example of Wrong Pairwise Relationships Caused by not Considering Conditions for Call

Figure 23 shows an example of wrong correspondence of ENSs. This is caused by not considering the conditions for call of each ENSs. In this example, there are two ENSs $A1$ and $A2$ in methodA, and there are also two ENSs $B1$ and $B2$ in methodB. All of the ENSs are in if statements, which means that methods created by extracting these ENSs are called if the conditional predicates of the corresponding if statements are satisfied. However, the pairwise relationships shown in the figure do not consider the conditions, therefore the behavior of methodB is changed after the refactoring.

As described in 6.5.4, the conditions to call $EM_{S_{m_{1(2)}}}$ are represented by control dependence edges, and all the nodes always have 1 control dependence from other nodes. In addition, all the nodes in an ENS are contained by a single block statement or contained by their owner method directly by the process described in 6.5.5. Consequently, all the control dependence edges to S have the same tail node. In other words, the formula (21) is always satisfied for every ENS S.

$$|InputControlNodes(G, S)| = 1 \tag{21}$$

Here, we define ICN_S as the unique element in $InputControlNodes(G, S)$. We regard a pair of

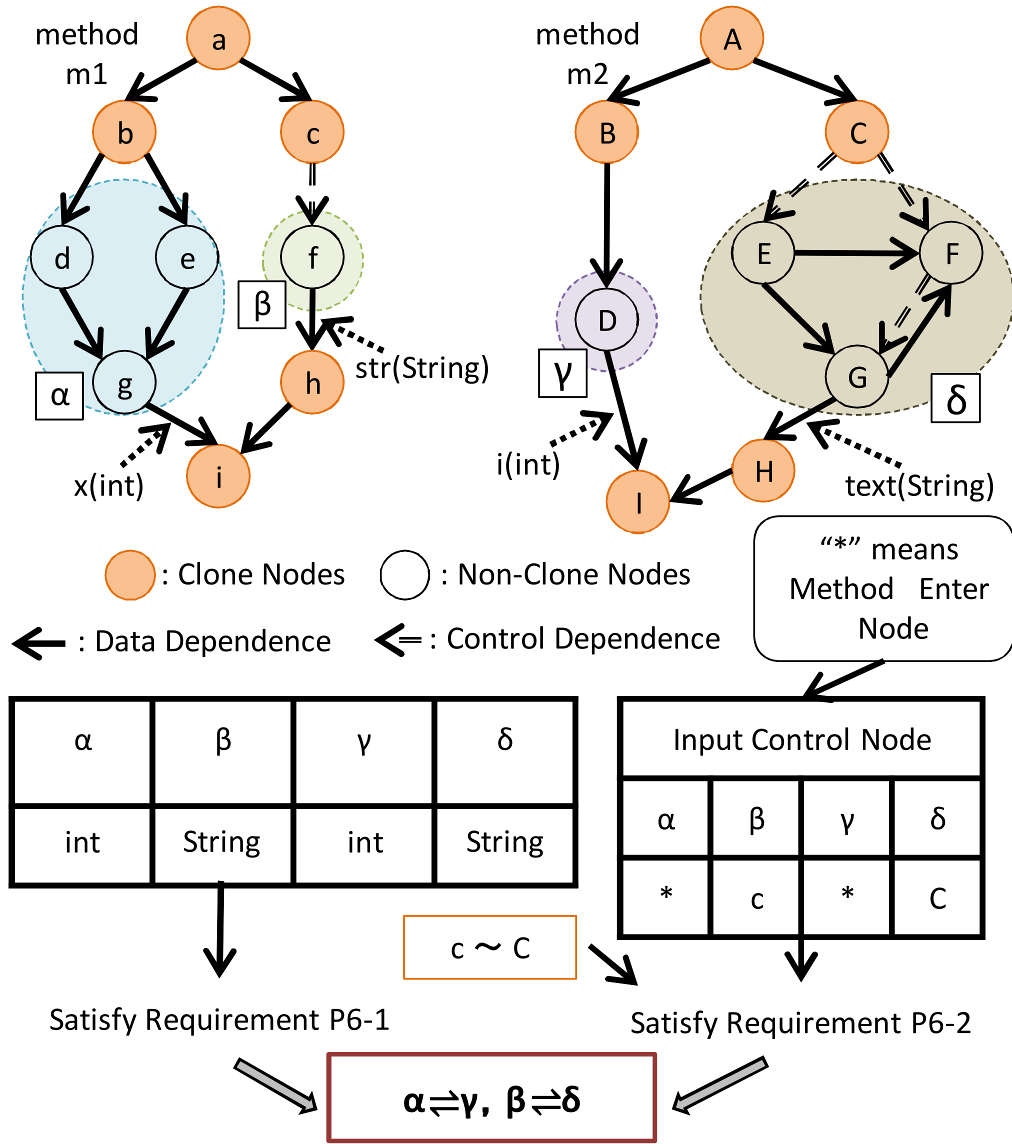


Figure 24: An Example of Pairwise Relationships

ENSs (S_{m_1}, S_{m_2}) as having same conditions for call if and only if they satisfy the formula (22).

$$(ICN_{S_{m_1}} \sim ICN_{S_{m_2}}) \vee (\exists S_1' \in DiffNodeSets(G_{m_1}) \exists S_2' \in DiffNodeSets(G_{m_2})$$
$$[S_1' \rightleftharpoons S_2' \wedge ICN_{S_{m_1}} \in S_1' \wedge ICN_{S_{m_2}} \in S_2']) \quad (22)$$

6.6.3 An Example of Pairwise Relationships Detection

Figure 24 shows an example of pairwise relationships detection. Due to space limitation, we do not write method enter nodes and control dependence from them in the figure.

In this example, there are two ENSs α and β in method $m1$, and there are also two ENSs γ and δ in method $m2$. Return values of EM_α and EM_γ are integer values, and return values of EM_β and EM_δ are string values. Consequently, two pairs of ENSs (α, γ) and (β, δ) satisfy Requirement P6-1.

Then, the proposed method checks the correspondence of call conditions. In this example, ICN_α and ICN_γ are the method enter nodes, which means that a pair of ENSs (α, γ) satisfies Requirement P6-2. In the case of (β, δ), ICN_β is c, and ICN_δ is C. Consequently, the pair of ENS (β, δ) satisfies Requirement P6-2 because $c \sim C$.

As a result, we get two pairs of ENSs (α, γ) and (β, δ) in this example.

7 Supporting for Method Groups

In this section, we describe the steps of proposed method for method groups. As described in 5.2, we use method pair information calculated in STEP-P1 to STEP-P6, therefore the steps from STEP-S1 to STEP-S6 are identical to from STEP-P1 to STEP-P6. Therefore, we describe the steps after STEP-S6 in the following subsections.

7.1 STEP-S7: Identify Method Groups

In this step, we detect groups of methods on which Form Template Method can be applied. In the reminder of this thesis, suppose that $m_1 \doteq m_2$ indicates that a pair of methods m_1 and m_2 is a refactoring candidate detected in the process described in 6.3.

Obviously, the binary relation $\doteq$ is a symmetric relation ($m_1 \doteq m_2 \Rightarrow m_2 \doteq m_1$). However, it is not a transitive relation. Assume that there are 3 methods m_1, m_2 and m_3, and $m_1 \doteq m_2$, $m_2 \doteq m_3$. In this case, there is at least 1 clone pair between m_1 and m_2, and between m_2 and m_3 because of the definitions of $\doteq$. However, there is no clone pair between m_1 and m_3 if all the clone pairs between m_1 and m_2 are not overlapped by any of clone pairs between m_2 and m_3. If there is no clone pair between m_1 and m_3, $m_1 \not\doteq m_3$ because of its definitions.

However, the proposed method temporarily regards a group of methods as a candidate method group if it satisfies the formula (23).

$$\forall m \in \mathit{MS}\ ,\ \exists m' \in \mathit{MS}\ (m \doteq m') \tag{23}$$

Under this definition, if $m_1 \doteq m_2$ and $m_2 \doteq m_3$ are satisfied, a group of methods m_1, m_2, and m_3 is regarded as a candidate method group regardless of whether $m_1 \doteq m_3$ is satisfied or not. If there is no clone pairs between m_1 and m_3, the proposed method omits the method group from candidate method groups in the next step.

7.2 STEP-S8: Detect Common and Unique Processes

In this step, the proposed method detects common processes and unique processes in every method group. Suppose that MS indicates a method group and G_{m_i} means the PDG of method m_i.

Statements must be duplicate between all the methods in the method group to be pulled up into a base class as a template method. We define $\mathit{CommonNodes}_{group}(G_{m_i})$ as a group of nodes in $V_{G_{m_i}}$ that are pulled up into a base class. The definition is shown in formula (24).

$$\mathit{CommonNodes}_{group}(G_{m_i}) := \{v_i \in V_{G_{m_i}} \mid \forall m_j \in \mathit{MS}\ ,\ \exists v_j \in V_{G_{m_j}}[v_i \sim v_j]\} \tag{24}$$

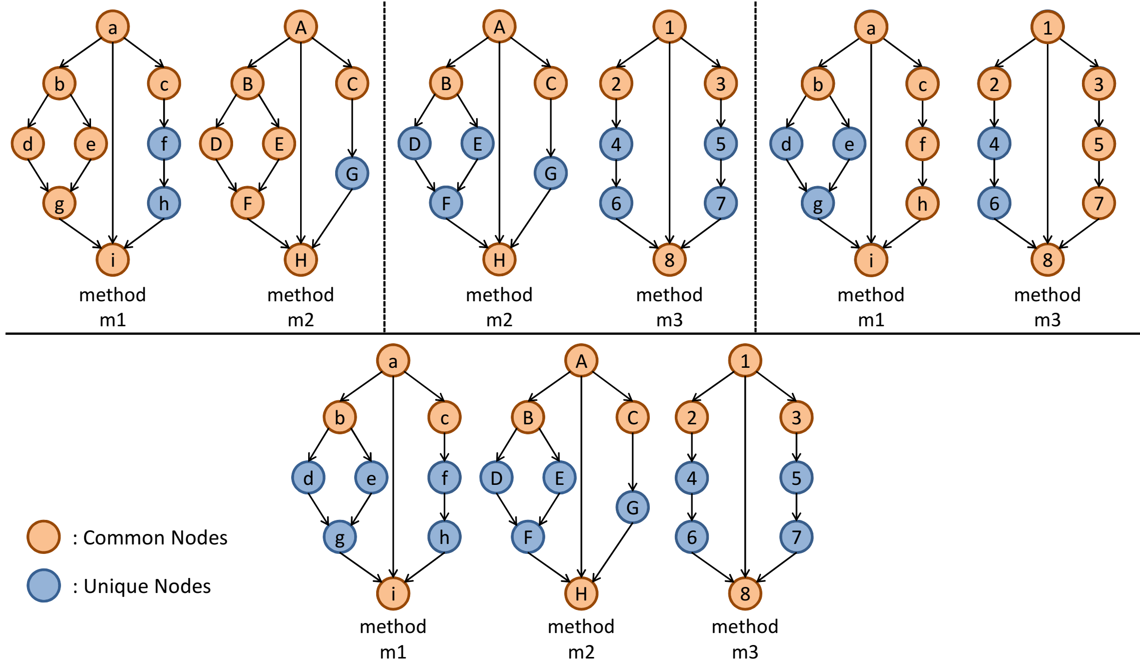


Figure 25: An Example of Method Group

We define $DiffNodes_{group}(G_{m_i})$ as a group of nodes that need to remain in the derived class that has method m_i. The definition is shown in the formula (25).

$$DiffNodes_{group}(G_{m_i}) := \{v_i \in V_{G_{m_i}} \mid v_i \notin CommonNodes_{group}(G_{m_i})\} \tag{25}$$

Figure 25 shows an example of method group. In this example, there are 3 methods ($m1$, $m2$, and $m3$) and all the pairs of them are detected as candidate method pairs, in other words $m1 \doteq m2$, $m2 \doteq m3$, and $m1 \doteq m3$. In this example, $CommonNodes_{group}(G_{m_1})$ and $DiffNodes_{group}(G_{m_1})$ become as follows.

$$\begin{aligned} CommonNodes_{group}(G_{m_1}) &= \{a, b, c, i\} \\ DiffNodes_{group}(G_{m_1}) &= \{d, e, f, g, h\} \end{aligned}$$

In some cases, some nodes included in $CommonNodes(G_{m_i})$ are omitted to make $CommonNodes_{group}(G_{m_1})$. Consequently, the amount of common processes on method groups might be quite small than that on method pairs. As a result, the number of elements in $CommonNodes(G_{m_i})$ might be less than the threshold of minimum code clone size that is specified by users. Therefore, the proposed method omits method groups if the number of their common nodes is less than the minimum clone size. Consequently, in the case that $m_1 \doteq m_2$, $m_2 \doteq m_3$, and $m_1 \not\doteq m_3$, the proposed method omit the method group that consists of m_1, m_2 and m_3.

In the next, the proposed method detects ENSs for every method in MS. There is no difference in the definitions of ENSs between method pairs and method groups because the detection of ENSs

is closed in each method.

7.3 STEP-S9: Detect Relationships on ENSs

In this step, the proposed method detects correspondences of ENSs between methods in MS.

Likewise on method pairs, the correspondence relationship means that ENSs in a correspondence relationship can be extracted as methods whose signatures are the same. As described in 6.6, the proposed method regards a pair of ENSs S_{m_1} and S_{m_2} as $S_{m_1} \rightleftharpoons S_{m_2}$ if they satisfy the requirements about their return values and their call conditions. We can detect this relationship by expanding that on method pairs.

Suppose that S_{m_1}, S_{m_2}, and S_{m_3} are ENSs in methods m_1, m_2, and m_3, respectively. In addition, assume that $S_{m_1} \rightleftharpoons S_{m_2}$ and $S_{m_2} \rightleftharpoons S_{m_3}$. Moreover, assume that EM_S means a method created by extracting an ENS S. Under this assumption, the types of return values $EM_{S_{m_1}}$ and $EM_{S_{m_2}}$ are the same. Moreover, those of $EM_{S_{m_2}}$ and $EM_{S_{m_3}}$ are also the same. Therefore, the return values of $EM_{S_{m_1}}$ and $EM_{S_{m_3}}$ are the same. Similarly, the call conditions for $EM_{S_{m_1}}$, $EM_{S_{m_2}}$, and $EM_{S_{m_3}}$ are same to each other. Consequently, the binary relationship $\rightleftharpoons$ is a transitive relation ($S_{m_1} \rightleftharpoons S_{m_2} \wedge S_{m_2} \rightleftharpoons S_{m_3} \Rightarrow S_{m_1} \rightleftharpoons S_{m_3}$).

Obviously, the binary relation $\rightleftharpoons$ is a symmetric relation ($S_{m_1} \rightleftharpoons S_{m_2} \Rightarrow S_{m_2} \rightleftharpoons S_{m_1}$). Moreover, it is also a reflexive relation ($S_{m_1} \rightleftharpoons S_{m_1}$). Consequently, the binary relation $\rightleftharpoons$ is an equivalence relation.

Therefore, we can detect correspondence relationships between 3 or more ENSs by detecting equivalent classes.

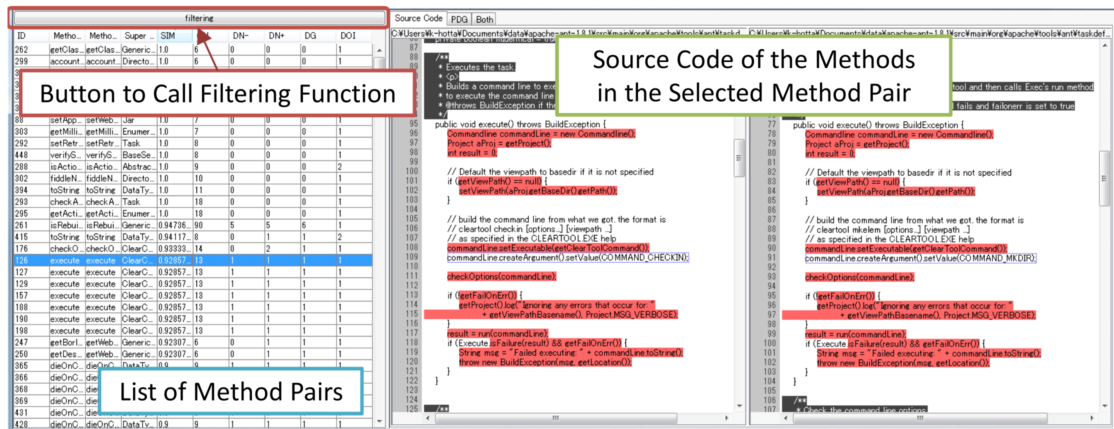


Figure 26: A Whole Snapshot of Creios (for Method Pairs)

8 Implementation

8.1 Overview

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.

The LOC of Creios is 17,290 with comments and white lines. It becomes 11,125 without comments and white lines. Moreover, Creios consists of 136 source files. In addition, it uses the external libraries as follows.

Scorpio: Scorpio is a PDG-based code clone detector. Creios uses it to detect code clones from the target source code [34].

MASU: MASU (*Metrics Assesment plugin platform for Software Unit*) is a source code analysis platform [53]. Creios uses it to analyze the source code. MASU is also used in Scorpio. MASU is an open source project in SourceForge.

JUNG: JUNG (*Java Universal Network/Graph Framework*) is a framework that provides software libraries for the modeling, analysis, and visualization of data that can be represented as a graph or network [54]. Creios uses it to visualize PDGs. JUNG is an open source project in SourceForge likewise MASU.

Creios has two modes. One is for method pairs, and the other is for method groups. We describe each of them in detail in the following subsections. Note that Creios does not modify

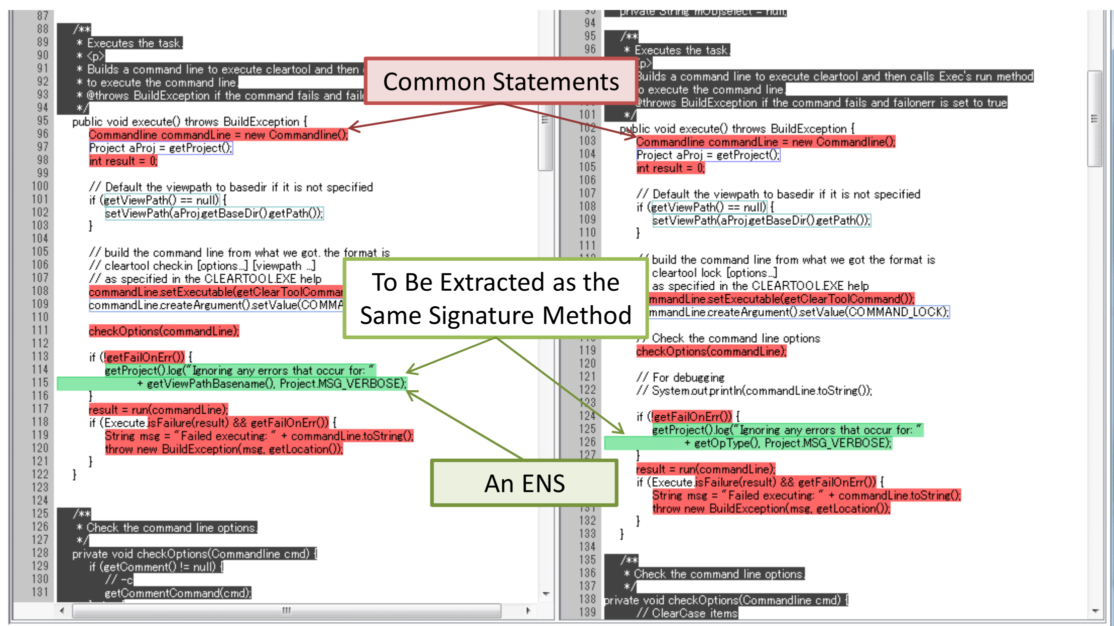


Figure 27: A Snapshot of Source Code View

the source code by itself. Therefore, users need to perform source code modification by their own effort.

8.2 Functionalities for Method Pairs

Figure 26 shows a snapshot of Creios for method pairs. The table shows all the candidate method pairs that Creios detected. When users select a method pair from the table, the source code of methods included in the pair is shown in the right panel.

Figure 27 shows a snapshot of source code view. In the source code view, common statements are highlighted with red. Statements highlighted by red mean that they should be pulled up into the base class as a template method. On the other hand, the other statements are unique processes in each method. Statements surrounded by the same color rectangles make an ENS. In addition, if users click statements that are not highlighted by red, an ENS includes the statement are highlighted. Moreover, if users click an ENS in one method, Creios also highlights the corresponding ENS in the other method. ENSs highlighted by the same color are under the correspondence relationship ($S_{m_1} \rightleftharpoons S_{m_2}$), which indicates that the methods created by extracting them have the same signature. Additionally, Creios shows the signature of method created from an ENS if users put cursor on the ENS.

Creios also has PDG view. Figure 28 shows a snapshot of PDG view. Each circle indicates a node of PDG, and each line indicates an edge of PDG. Nodes colored by red are nodes whose

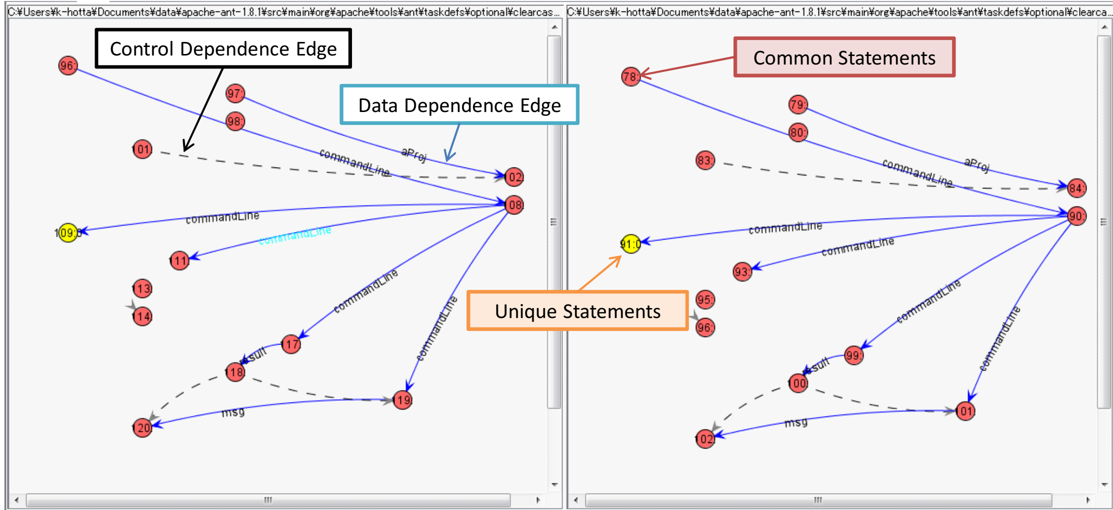


Figure 28: A Snapshot of PDG View

owner statements are included in common statements. The blue lines indicate data dependence edges, and the black broken lines indicate control dependence edges. The character string on each data dependence edge indicates the name of the variable that the edge represents. Note that Creios omits the method enter nodes and execute dependence edges in PDG view. To visualize PDGs, Creios uses APIs provided by JUNG.

Creios can show both the source code view and PDG view at a time. Figure 29 shows the apposing view of source code view and PDG view. The functionalities of the source code view and the PDG view in the apposing view are exactly same to the original ones.

In addition, Creios has a filtering function of method pairs with some metrics. All the metrics are calculated for each method pair. The metrics are as follows under the assumption that m_1 and m_2 are methods in a method pair, and G_m is the PDG of method m.

SIM: The similarity between two methods of each method pair (defined in the formula (26)).

$$SIM := \frac{|CommonNodes(G_{m_1})| + |CommonNodes(G_{m_2})|}{|V_{G_{m_1}}| + |V_{G_{m_2}}|} \tag{26}$$

CN: The number of nodes whose owner statements are included in common statements (defined in the formula (27)).

$$CN := |CommonNodes(G_{m_{1(2)}})| \tag{27}$$

DN+, DN-: The number of nodes whose owner statements are not included in common statements. Note that the values of this metric are different between each method. Therefore, we define DN+ as the larger one (formula (28)), and DN- as the smaller one (formula (29)),

Figure 29: A Snapshot of Apposing View of Source Code View and PDG View

respectively.

$$DN+ \quad := \quad max(|DiffNodes(G_{m_1})|, |DiffNodes(G_{m_2})|) \tag{28}$$

$$DN- \quad := \quad min(|DiffNodes(G_{m_1})|, |DiffNodes(G_{m_2})|) \tag{29}$$

LOC+, LOC-: The number of lines of each method. Obviously, the values of this metric are different between each method. Likewise DN+ and DN-, we define LOC+ as the larger one, and LOC- as the smaller one, respectively.

DG: The number of new methods that are created by extracting ENSs. DG is defined in the formula (30), where N is the number of ENSs that have their correspondents in the other method. Note that the 'correspondent' of an ENS S_1 is $\risingdotseq (S_1)$.

$$DG := |DiffNodeSets(G_{m_1})| + |DiffNodeSets(G_{m_2})| - N \tag{30}$$

DOI: The depth of inheritance from the common base class to the owner classes of the two methods. If the value is different for each method, we choose the larger one as the value of DOI.

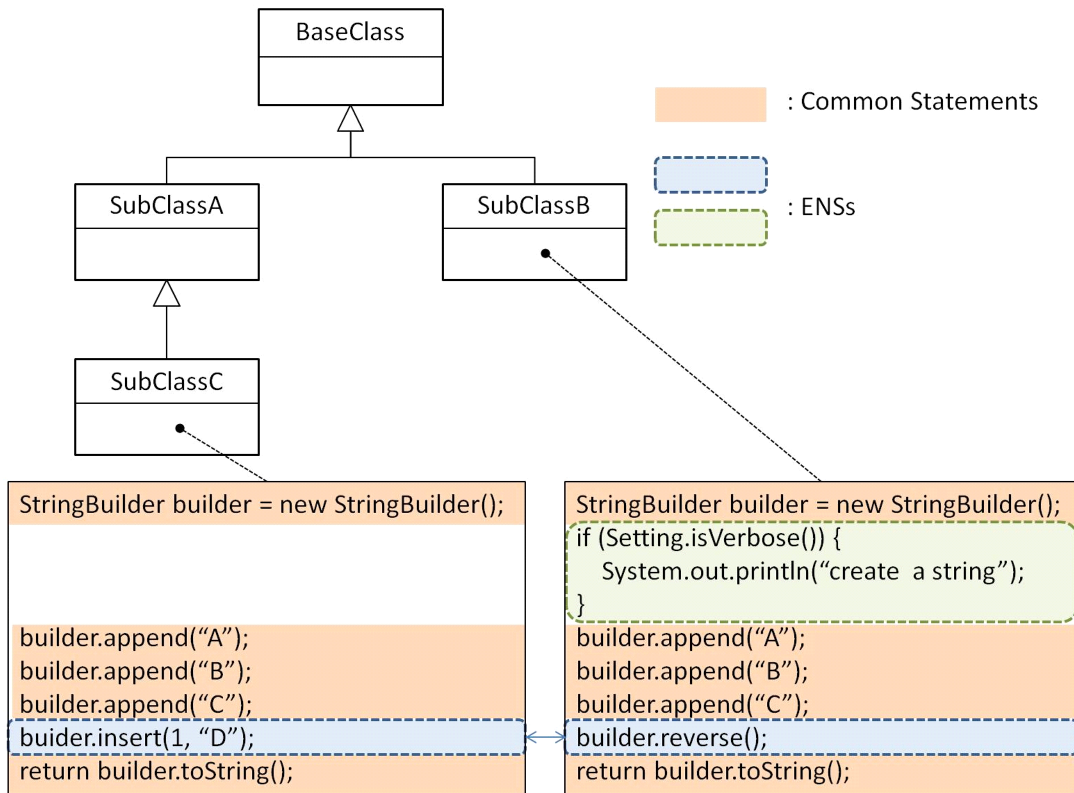


Figure 30: An Example of Candidate Method Pair

Table 1 shows the values of metrics of the method pair shown in Figure 30. Note that the values of inheritance depth from the common base class are different for each class that has the target method, therefore the larger value '2' is choosed as the value of DOI in this example.

Users can make a short list of candidate method pairs with the filtering function. The filtering function returns a list of method pairs whose metrics values are in the range that users specified. To call the filtering function, users push the button on the top of the table listing the method pairs.

A filtering view is launched when users push the button. Figure 31 shows a snapshot of the filtering view. The filtering view consists of three parts: a metrics graph, a list of metrics values, and a list of method pairs that pass the filtering. Figure 32 shows a metrics graph. Users specify the thresholds of each metric by dragging the graph. The area whose background color is gray

Table 1: The Values of Metrics in the Method Pair of Figure 30

SIM	CN	DN+	DN-	LOC+	LOC-	DG	DOI
0.769	5	3	1	9	6	2	2

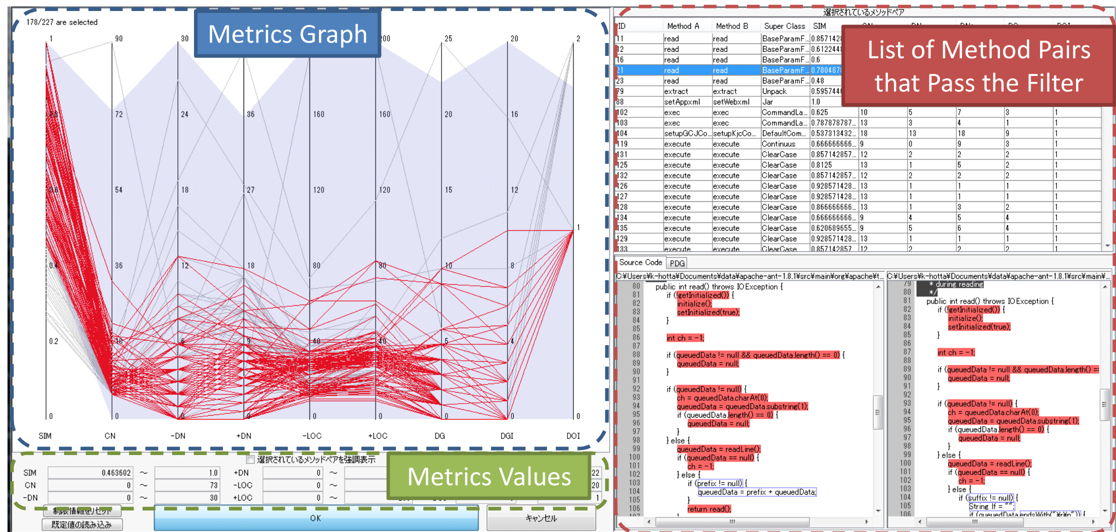


Figure 31: A Snapshot of Filtering View

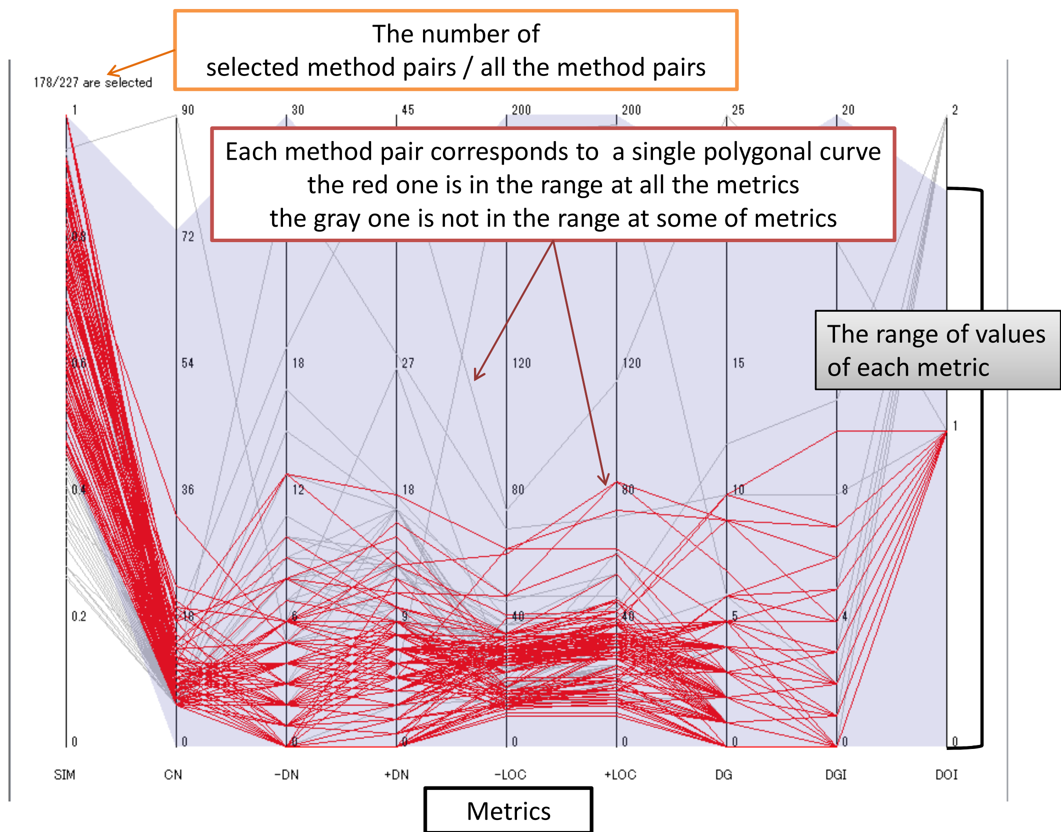


Figure 32: A Metrics Graph

Figure 33: View of the Metrics Values

Figure 34: A Whole Snapshot of Creios (for Method Groups)

indicates the range of thresholds for every metric, and the area whose background color is white indicates the outside of the range. In the metric graph, each polygonal curve corresponds a method pair. The polygonal curve becomes red if and only if all the metrics of the method pair represented by the polygonal curve are in the specified threshold. If any of the metrics is not in the threshold, the polygonal curve becomes gray. The specified lower limit and the specified upper limit of each metric are shown in the metrics values view (Figure 33). The list of selected method pairs is shown in the right of the filtering view. Users can view the source code and the PDGs of method pairs that are listed in the view. The functionalities of the source code view and the PDG view in the filtering view are exactly same to the original ones.

8.3 Functionalities for Method Groups

Figure 34 shows a snapshot of Creios for method groups. The left table shows all the candidate method groups that Creios detected. When users select a method group from table A, all the

methods in the selected method group are shown in the tables B-1 and B-2. Note that the tables B-1 and B-2 show the same contents. If users choose one of the methods in table B-1, the source code of the selected method is shown in the source code view C-1. Similarly, if users choose one of the methods in table B-2, its source code is shown in the source code view C-2. The source code view and the PDG view are the same to those of described in 8.2.

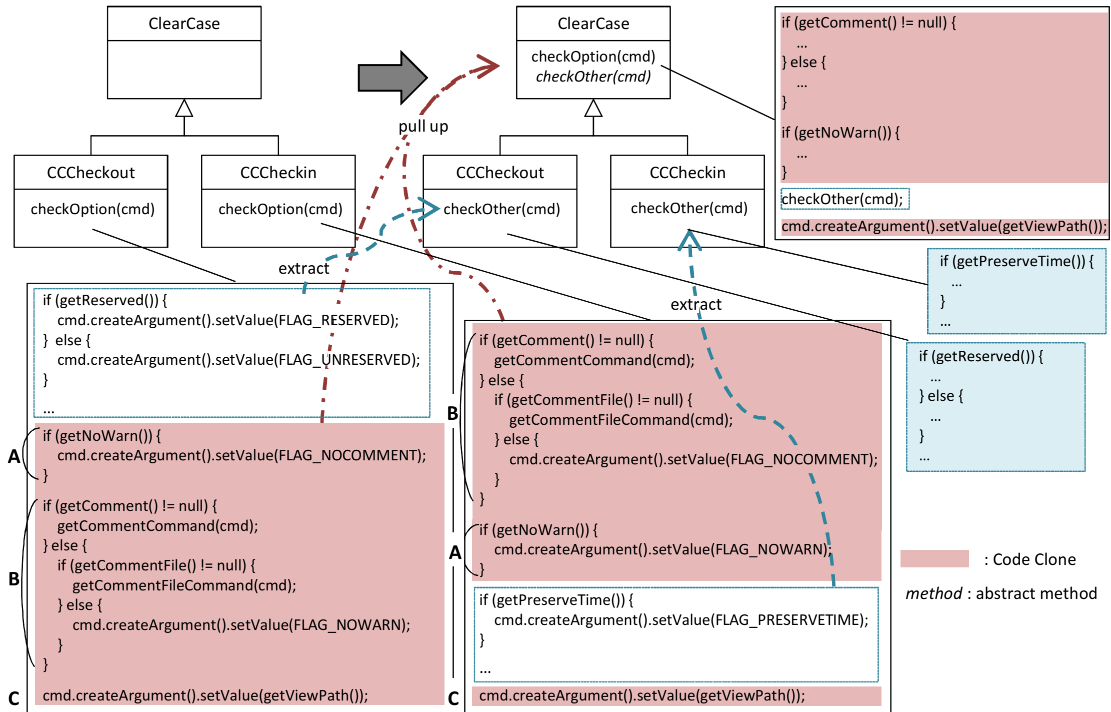


Figure 35: An Example of Application of Form Template Method with the Proposed Method

9 Evaluation

In order to evaluate the proposed method, we conducted experiments on two open source software systems. Table 2 shows the target software systems, their scale, and the environment of the experiments. The following subsections describe each of the experiments.

9.1 Evaluation of Supporting for Method Pairs

Table 3 shows the the number of detected candidates, and elapsed time to execute Creios on each target software system. The numbers of candidate method pairs are 226 and 45, so that it can be difficult for users to identify all the candidates manually. In addition, Creios can detect all the candidates in a few minutes although the target software systems have hundreds of source files.

Figure 35 shows a refactoring candidate in Ant detected by Creios and the result of the refac-

Table 2: Target Software Systems

Name	In Short	LOC	# of Files	Environment
Apache-Ant	Ant	212,401	829	CPU: Xeon 2.27GHz(8 core), RAM: 32GB
Apache-Synapse	Synapse	58,418	383	

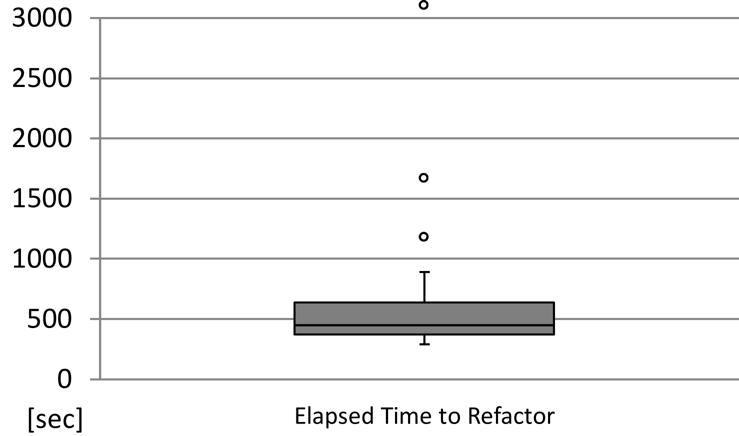


Figure 36: The Box-Plot of the Time to Apply Form Template Method on Synapse

toring. 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 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.

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 36 shows the box-plots of the time needed to apply refactorings. Because Creios suggests that all the candidates 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 3. As a result, we could apply refactorings in few

Table 3: The Number of Detected Candidates and Elapsed Time on Method Pairs

Name	# of Candidates	Elapsed Time [s]
Ant	226	178
Synapse	45	66

minutes in average nevertheless we are unfamiliar with the software.

9.2 Evaluation of Supporting for Method Groups

Table 4 shows the number of detected method groups, the number of them that have 3 or more methods, and elapsed time to detect them on each target software.

Likewise on method pairs, we applied Form Template Method refactoring to all the 6 method groups that the proposed method had suggested in Synapse. As a result, we successfully refactored all the candidates and confirmed that the behavior is preserved by using test suites.

9.3 Experiment with Subjects

9.3.1 Overview of the Experiment

We conducted an experiment with 7 subjects. All the subjects belong to Osaka University's Graduate School of Information and Technology (IST) or Osaka University's Department of Information and Computer Sciences in School of Engineering Science (ICS). The breakout of the subjects is that: 3 master's students in IST, 3 undergraduate students of fourth grader in ICS, and 1 research student in IST.

The objective of this experiment is to investigate the effectiveness of the proposed method as refactoring support method. In this experiment, subjects apply Form Template Method refactoring after a short introduction and practice, and we measure elapsed time that they need to finish the refactoring. All the subjects refactored 2 method groups described in 9.3.2. Subjects applied refactorings to one candidate method group with Creios, and they applied refactorings to the other candidate with *CCFinder*.

9.3.2 Target Method Groups

As described above, subjects applied Form Template Method to 2 method groups. The 2 method groups do not differ depending on the subjects. We call the 2 method groups **Candidate-A** and **Candidate-B**, respectively.

Table 4: The Number of Detected Candidates and Elapsed Time on Method Groups

Name	# of Candidates	# of Candidates (3 or more methods)	Elapsed Time [s]
Ant	48	18	195
Synapse	6	2	68

```
public PlanarImage executeDrawOperation() {
    BufferedImage bi = new BufferedImage(width, height,
                                         BufferedImage.TYPE_4BYTE_ABGR_PRE);
    Graphics2D graphics = (Graphics2D) bi.getGraphics();

    if (!stroke.equals("transparent") {
       BasicStroke bStroke = new BasicStroke(stroke_width);
       graphics.setColor(ColorMapper.getColorByName(stroke));
       graphics.setStroke(bStroke);
       graphics.draw(new Ellipse2D.Double(0, 0, width, height);
    {
    if (!fill.equals("transparent")) {
       graphics.setColor(ColorMapper.getColorByName(fill));
       grahpics.fill(new Ellipse2D.Double(0, 0, width, height));
    }
    for (int i = 0; i < instructions.size(); i++) {
       ImageOperation instr = ((ImageOperation) instructions.elementAt(i));
       if (instr instanceof DrawOperation) {
          PlanarImage img = ((DrawOperation) instr).executeDrawOperation();
          graphics.drawImage(img.getAsBufferedImage(), null, 0, 0);
       } else if (instr instanceof TransformOperation) {
          graphics = (Graphics2D) bi.getGraphics();
          PlanarImage image = ((TransformOperation) instr)
                .executeTransformOperation(PlanarImage.wrapRenderedImage(bi));
          bi = image.getAsBufferedImage();
       }
    }
    return PlanarImage.wrapRenderedImage(bi);
}
```

: Common Statements : ENSs

(a) Candidate-A

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

    if (getViewPath() == null) {
       setViewPath(aProj.getBaseDir().getPath());
    }

    commandLine.setExecutable(getClearToolCommand());
    commandLine.createArgument().setValues(COMMAND_MKBL);

    checkOption(commandLine);

    if (!getFailOnErr()) {
       getProject().log("Ignoring any errors that occur for: " +
                        getBaselineRootName(), Ptoject.MAG_VERBOSE);
    }

    result = run(commandline);
    if (Execute.isFailure(reault) && getFailOnErr()) {
       String msg = "Failed executing: " + commandLine.toString();
       throw new BuildException(msg.getLocation());
    }
}
```

: Common Statements : ENSs

(b) Candidate-B

Figure 37: Candidate Method Groups

Figure 37 shows the source code and the outputs of Creios on each candidate. The features of the 2 method groups are shown in Table 5.

9.3.3 Prodcedure of the Experiment

The procedure of the experiment consists of five steps as follows.

1. We give a brief introduction to subjects.
2. Subjects apply Form Template Method to a simple example.
3. We divide subjects into 4 groups. Table 6 shows the groups and assignments of each subject.
4. Subjects apply refactorings to the assigned method group.
5. Subjects apply refactorings to the other method group.

Introduction to subjects

At first, we gave an introduction to subjects about the background of this study and the experimental procedure. The introduction includes the following information.

- Code clones and their removal techniques.
- Form Template Method refactoring pattern.
- How to apply Form Template Method.
- How to use Creios.
- The procedure of the experiment.

Practice

Second, we have subjects practice applying Form Template Method with a simple method group. The target method group consists of 2 methods, and it contains 2 ENSs (note that we count a pair of ENSs (S_1, S_2) as 1 ENS if $S_1 \rightleftharpoons S_2$). The purposes of the practice are (1) to have subjects

Table 5: The Features of Target Method Groups

Label	# of Methods	# of Common Nodes	# of ENSs
Candidate-A	3	19	3
Candidate-B	12	9	5

understand refactoring steps of Form Template Method, and (2) to have subjects be familiar with the tool.

Grouping

In the next, we divide subjects into 4 groups. Table 6 shows the groups of subjects. As this table shows, the differences between each group are as follows:

- Which candidate do they refactor first?
- Which candidate do they use Creios?

Apply Refactoring

Subjects apply refactoring to the assigned target method group. For example, subjects in Group 1 refactor candidate A with Creios. We measure the elapsed time required to finish the refactoring for every subject.

9.3.4 Result

Table 7 shows the elapsed time to finish Form Template Method for every subject. The numeric characters in this table 'hh:mm:ss' indicates that the subject need hh hours and mm minutes and ss seconds to finish their refactoring tasks. For example, Subject 1 had finished applying refactoring on Candidate-A in 22 minutes and 45 seconds. 'N/A' means that the subject cannot finish refactoring in the case.

As the table shows, the time required to finish refactoring tasks varies greatly among subjects. There is also great variability among Candidate-A and Candidate-B; all the subjects required much time on Candidate-B than Candidate-A. This is because the degree of difficulty of Candidate-B

Table 6: Groups of Subjects

Group ID	Assigned Subjects	First Target and Using Tool	Second Target and Using Tool
1	Subject 1 Subject 2	Candidate-A Creios	Candidate-B *CCFinder*
2	Subject 5	Candidate-B Creios	Candidate-A *CCFinder*
3	Subject 6 Subject 7	Candidate-A *CCFinder*	Candidate-B Creios
4	Subject 3 Subject 4	Candidate-B *CCFinder*	Candidate-A Creios

is higher than that of Candidate-A. Table 8 shows the average time to finish the refactorings. As the table shows, the elapsed time with Creios is higher than that with *CCFinder* in Candidate-A, meanwhile the opposite result is shown in Candidate-B. As a result, Creios cannot reduce time required for the refactorings in the easier candidate, but it can reduse time required for the refactorings in the more difficult candidate. Therefore, Creios is useful in a case that the target method group is complex and it has a number of the methods.

Table 7: Elapsed Time to Finish Form Template Method Application

Subjects	Group ID	Candidate-A		Candidate-B	
Subject 1	1	with Creios	0:22:45	with *CCFinder*	0:50:30
Subject 2	1		0:52:00		1:17:00
Subject 3	4		0:19:22		N/A
Subject 4	4		0:09:58		0:27:45
Subject 5	2	with *CCFinder*	0:12:04	with Creios	0:25:30
Subject 6	3		0:22:55		0:50:28
Subject 7	3		0:35:20		1:04:15

Table 8: The Average Time

	Candidate-A	Candidate-B	Both
with Creios	0:28:14	0:46:44	0:34:54
with *CCFinder*	0:23:26	0:51:45	0:37:36
Both	0:25:50	0:49:15	0:36:09

10 Discussion

10.1 PDG Creation

There are some other dependences except data, control, and execute dependence that should be considered in PDGs. In the proposed method, dependence of break and continue statements and dependence of exception are considered. However, the proposed method does not consider dependence caused by the following factors.

- Library call.
- Alias.
- Presence of innner classes.
- Reflection.

Of these factors, we can consider dependence caused by library calls by giving the source code of libraries as additional input of the proposed method. However, it is quite difficult to give the source code of all the libraries that are used in the target software systems as the input.

In the experiments of this study, we cannot find instances that suffer any problems by dependence that are not considered in the proposed method. However, there is a risk that the proposed method suggest refactoring candidate incorrectly by these dependence. Thus, it is necessary to consider these factors to make the proposed method robust.

10.2 Detection of Common Statements

As described in 6.4, the proposed method omits clone pairs except the most largest one in the case that there are duplications of clone pairs. The purpose of this is to suggest more nodes as common statements. However, in some cases, this selection may be not appropriate. We can avoid this problem by delegating the selection to users. However, the proposed method does not have this function at present.

10.3 Candidates that Need to be Tailored

As we described in Section 9.1, we applied Form Template Method refactoring to 45 method pairs detected in Synapse on method pairs. In some cases, we had to make some modifications that Creios did not indicate, or we had to make some tailoring to the output of Creios to apply the pattern. Table 9 shows the modifications or adjustments needed to apply refactorings, and the number of candidates that needed them. The definitions of the terms in the table are as follows:

the term “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.

Issues of Visibility

The proposed method does not consider the visibility of methods and fields in the source code. Therefore, code fragments that should be pulled up into template method can 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 changes could cause vulnerability [55]. For fields, if fields have getters and setters, we can resolve this problem by using them.

Issues of ENSs and their Relationships

The proposed method automatically detects ENSs and correspondence relationships of ENSs. However, the automatically detected ENSs or relationships of ENSs may not fit with users' sensibilities. Although the automatically detected ENSs and their relationships do not always suitable, they can help users apply refactoring.

10.4 Detection of Method Groups

The proposed method forms method groups from all the methods that satisfy the requirements described in 6.3. However, it may be more suitable to form method groups from a subset of

Table 9: 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

the methods. We can improve this issue by delegating the selection of methods that should be included in method groups to users. However, the proposed method currently does not have this functionality.

10.5 Threats to Validity of the Experiment with Subjects

In the experiment with subjects, we confirmed that the proposed method reduces time to refactor in a case that the target is complex and there is a number of methods in the target method group. However, we found the opposite result in a case that the target is not complex. There might be bias of subjects' abilities, so that the result might occur. We may get another result with different grouping of subjects.

11 Conclusion

In this thesis, we proposed a new technique to assist developers to apply Form Template Method refactorings to code clones. It detects refactoring candidates automatically and suggests them to its users. It uses program dependence graphs as its data structure, which enables us to assist developers removing code clones having trivial differences that have no impact on the meanings of the program. Moreover, it can handle a group of three or more methods, which increases the practicality of code clones removal.

We implemented the proposed method as a tool, and conducted an experiment to evaluate the proposed method. We applied Form Template Method to all the candidates that the tool suggests in an open source software, and confirmed that we can refactor the candidates with preserving the behavior of the program.

As future works, 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. Also, we are going to expand the proposed method to delegate selections for the issues as follows.

- Methods that should be included in a method group.
- Common statements between methods in a method group.
- Statements that should be included in a set of nodes that should be extracted as a single method.
- Relationships of ENSs.

Moreover, we are going to improve the proposed method to be able to focus on code clones that affect maintainability significantly such as frequently modified code clones. Also, we are going to evaluate the effectiveness of the proposed method by additional experiments.

Acknowledgements

During this work, I have been fortunate to have received assistance from many people. This work could not have been possible without their valuable contributions.

First, I would like to express my sincere gratitude to my supervisor, Professor Shinji Kusumoto, at the Osaka University, for his considerate support, encouragement, and adequate guidance for this work.

Also, I would like to thank to Associate Professor, Kozo Okano, at the Osaka University for his guidance, valuable suggestions and discussions for this work.

I am also deeply grateful to Associate Professor, Hiroshi Igaki, at the Osaka University for his helpful comments and valuable suggestions.

I would like to express my heartfelt appreciation to Assistant Professor, Yoshiki Higo, at the Osaka University for his zealous coaching, continuous support, and encouragement throughout this work.

My sincere thanks go to all the subjects who take the time to the experiment for their effort, comments, and close cooperation for this work.

Finally, I would like to thank all of my friends in the Department of Computer Science at the Osaka University, especially the members in Kusumoto Laboratory, for their helpful advices, suggestions and assistance.

References

[1] Y. Higo, S. Kusumoto, and K. Inoue. A survey of code clone detection and its related techniques. *IEICE Transactions on Information and Systems*, Vol. 91-D, No. 6, pp. 1465–1481, June 2008. (in Japanese).

[2] T. Kamiya, Y. Higo, and N. Yoshida. Evolving and hot topics on code clone detection techniques. *Journal of Computer Software*, Vol. 28, No. 3, pp. 28–42, Aug. 2011. (in Japanese).

[3] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multi-linguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, Vol. 28, No. 7, pp. 654–670, July 2002.

[4] I. Baxter, A. Yahin, M. Anna L. Moura, and L. Bier. Clone detection using abstract syntax trees. In *Proc. of the 14th International Conference on Software Maintenance*, pp. 368–377, Mar. 1998.

[5] J.H. Johnson. Substring matching for clone detection tools. In *Proc. of the 10th International Conference on Software Maintenance*, pp. 120–126, Sep. 1994.

[6] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proc. of the 15th International Conference on Software Maintenance*, pp. 109–118, Aug. 1999.

[7] Z. Li, S. Myagmar, S. Lu, and Y.Zhou. Cp-miner : Finding copy-paste and related bugs in large-scale software code. *IEEE Transcations on Software Engineering*, Vol. 32, No. 3, pp. 176–192, Mar. 2006.

[8] R. Koschke, R. Falke, and P. Frenzel. Clone detection using abstract syntax suffix trees. In *Proc. of the 13th Working Conference on Reverse Engineering*, pp. 253–262, Oct. 2006.

[9] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard : Scalable and accurate tree-based detection of code clones. In *Proc. of the 29th International Conference on Software Engineering*, May 2007.

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

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

[12] Y. Higo and S. Kusumoto. Code clone detection on specialized pdgs with heuristics. In *Proc. of the 15th European Conference on Software Maintenance and Reengineering*, pp. 75–84, Mar. 2011.

[13] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proc. of the 12th International Conference on Software Maintenance*, pp. 244–253, Nov. 1996.

[14] J. Ossher, H. Sajnani, and C. Lopes. File cloning in open source java projects: The good, the bad, and the ugly. In *Proc. of the 27th International Conference on Software Maintenance*, pp. 283–292, Sep. 2011.

[15] Y. Sasaki, T. Yamaoto, Y. Hayase, and K. Inoue. File clone detection for a large scale software system. *IEICE Transactions on Information and Systems*, Vol. J94-D, No. 8, pp. 1423–1433, Aug. 2011. (in Japanese).

[16] N. Göde and R. Kosheke. Incremental clone detection. In *Proc. of the 13th European Conference on Software Maintenance and Reengineering*, pp. 219–228, Mar. 2009.

[17] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt. Index-based code clone detection: Incremental, distributed, scalable. In *Proc. of the 26th International Conference on Software Maintenance*, pp. 1–9, Sep. 2010.

[18] Y. Higo, Y. Ueda, M. Nishino, and S. Kusumoto. Incremental code clone detection: A pdg-based approach. In *Proc. of the 18th Working Conference on Reverse Engineering*, pp. 3–12, Oct. 2011.

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

[20] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus. Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, Vol. 27, No. 1, pp. 1–12, Jan. 2001.

[21] E. Murphy-Hill and A. P. Black. Breaking the barriers to successful refactoring. In *Proc. of the 30th International Conference on Software Engineering*, pp. 421–430, May 2008.

[22] T. Mens and T. Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, Vol. 30, No. 2, pp. 126–139, Feb. 2004.

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

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

[25] T. Masai, N. Yoshida, M. Matsushita, and K. Inoue. Supporting difference extraction for merging similar methods. In *IEICE Technical Report*, pp. 45–50, May 2010. (in Japanese).

[26] M. Ioka, N. Yoshida, T. Masai, Y. Higo, and K. Inoue. A tool support to merge similar methods with a cohesion metric cob. In *Proc. of the 3rd International Workshop on Empirical Software Engineering in Practice*, pp. 23–24, Nov. 2011.

[27] M. Weiser. Program slicing. In *Proc. of the 5th International Conference on Software Enginieering*, pp. 439–449, Mar. 1981.

[28] 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.

[29] 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.

[30] K. Inoue, T. Kamiya, and S. Kusumoto. Code-clone detection methods. *Computer Software*, Vol. 18, No. 5, pp. 529–536, 2001. (in Japanese).

[31] S. Bellon, R. Koschke, G. Antniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, Vol. 31, No. 10, pp. 804–818, Oct. 2007.

[32] CCFinderX. available at <http://www.ccfinder.net/ccfinderx-j.html>.

[33] CloneDR. available at <http://www.semdesigns.com/Products/Clone/>.

[34] Scorpio. available at <http://www-sdl.ist.osaka-u.ac.jp/~higo/cgi-bin/moin.cgi/scorpio>.

[35] W. F. Opdyke. *Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks*. PhD thesis, University of Illinois, 1992.

[36] Y. Higo, S. Kusumoto, and K. Inoue. Identifying refactoring opportunities for removing code clones with a metrics-based approach. In K. Cai, editor, *Java in Academia and Research*, chapter 3, pp. 57–82. Concept Press Ltd., 2011.

[37] M. Balazinska, E. Merlo, M. Dagenais, and B. Lague. Advanced clone-analysis to support object-oriented system refactoring. In *Proc. of the 7th Working Conference on Reverse Engineering*, pp. 98–107, Nov. 2000.

[38] R. Cottrell, J. J. Chang, R. J. Walker, and J. Denzinger. Determing detailed structural correspondence for generalization tasks. In *Proc. of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 165–174, 2007.

[39] R. Komondoor and S. Horwitz. Semantics-preserving procedure extraction. In *Proc. of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Language*, pp. 155–169, 2000.

[40] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy. An empirical study of code clone genealogies. In *Proc. of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 187–196, 2005.

[41] C. J. Kapser and M. W. Godfrey. "cloning considered harmful" considered harmful. *Empirical Software Enginieering*, Vol. 13, No. 6, pp. 645–692, Dec. 2008.

[42] 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.

[43] A. Monden, D. Nakae, T. Kamiya, S. Sato, and K. Matsumoto. Software quality analysis by code clones in industrial legacy software. In *Proc. of the 8th IEEE Internaitional Software Metrics Symposium*, pp. 87–94, June 2002.

[44] A. Lozano and M. Wermelinger. Evaluating the harmfulness of cloning: A change based experiment. In *Proc. of the 4th International Workshop on Mining Software Repositories*, May 2007.

[45] J. Krinke. Is cloned code more stable than non-cloned code? In *Proc. of the 8th International Working Conference on Source Code Analysis and Manipulation*, pp. 57–66, Sep. 2008.

[46] N. Göde and J. Harder. Clone stability. In *Proc. of the 15th European Conference on Software Maintenance and Reengineering*, pp. 65–74, Mar. 2011.

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

[48] F. Rahman, C. Bird, and P. Devanbu. Clones: What is that smell? In *Proc. of the 7th IEEE Working Conference on Mining Software Repositories*, pp. 72–81, May 2010.

[49] K. Hotta, Y. Sano, Y. Higo, and S. Kusumoto. Is duplicate code more frequently modified than non-duplicate code in software evolution?: An empirical study on open source software. In *Proc. of the ERCIM Workshop on Software Evolution and International Workshop on Principles of Software Evolution*, pp. 73–82, Sep. 2010.

[50] Y. Sasaki, K. Hotta, Y. Higo, and S. Kusumoto. Is duplicate code good or bad? an empirical study with multiple investigation methods and multiple detection tools. In *Proc. of the 22nd International Symposium on Software Reliability Engineering*, Nov. 2011.

[51] S. Lee, G. Bae, H. S. Chae, D. Bae, and Y. R. Kwon. Automated scheduling for clone-based refactoring using a competent ga. *Software: Practice and Experience*, Vol. 41, No. 5, pp. 521–550, Apr. 2010.

[52] 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*, pp. 105–114, Sep. 2011.

[53] MASU. avilable at <http://sourceforge.net/projects/masu/>.

[54] JUNG. avilable at <http://jung.sourceforge.net/>.

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

Algorithm 5 *ForwardSlice*(G_1, G_2, r_1, r_2, R_1, R_2)

Require: $G_1, G_2, r_1, r_2, R_1, R_2, r_1 = r_2$
Ensure: $R_1 \cong R_2$
1: $R_1 \overset{+}{\leftarrow} r_1$
2: $R_2 \overset{+}{\leftarrow} r_2$
3: **for all** $e_1 \in ForwardEdges(r_1)$ **do**
4:     **for all** $e_2 \in ForwardEdges(r_2)$ **do**
5:         $r'_1 \leftarrow head(e_1)$
6:         $r'_2 \leftarrow head(e_2)$
7:         **if** $r'_1 \neq r'_2$ **then**
8:             continue
9:         **end if**
10:         **if** $r'_1 \in R_1$ or $r'_2 \in R_2$ **then**
11:             continue
12:         **end if**
13:         **if** $r'_1 \in R_2$ or $r'_2 \in R_1$ **then**
14:             continue
15:         **end if**
16:         $ForwardSlice(G_1, G_2, r'_1, r'_2, R_1, R_2)$
17:     **end for**
18: **end for**

A Algorithms for Detecting Isomorphic Subgraphs

Suppose G_1 and G_2 are the target PDGs. The algorithm to detect isomorphic subgraphs between G_1 and G_2 with the forward slice is shown in Algorithm 5. Note that R_1 and R_2 must be initialized as empty sets to run this algorithm. In Scorpio, hash values are used to compare two nodes. Therefore, $r_1 = r_2$ indicates that the hash value of r_1 is equal to that of r_2. Also, the algorithm with the backward slice is shown in Algorithm 6. Both of the forward and backward slices are used to detect code clones in Scorpio.

Algorithm 6 *BackwardSlice*(G_1, G_2, r_1, r_2, R_1, R_2)

Require: $G_1, G_2, r_1, r_2, R_1, R_2, r_1 = r_2$

Ensure: $R_1 \cong R_2$

1: $R_1 \xleftarrow{+} r_1$
2: $R_2 \xleftarrow{+} r_2$
3: **for all** $e_1 \in BackwardEdges(r_1)$ **do**
4:     **for all** $e_2 \in BackwardEdges(r_2)$ **do**
5:         $r'_1 \leftarrow tail(e_1)$
6:         $r'_2 \leftarrow tail(e_2)$
7:         **if** $r'_1 \neq r'_2$ **then**
8:             continue
9:         **end if**
10:         **if** $r'_1 \in R_1$ or $r'_2 \in R_2$ **then**
11:             continue
12:         **end if**
13:         **if** $r'_1 \in R_2$ or $r'_2 \in R_1$ **then**
14:             continue
15:         **end if**
16:         $ForwardSlice(G_1, G_2, r'_1, r'_2, R_1, R_2)$
17:     **end for**
18: **end for**