修士学位論文

題目

Investigating Clone Metrics of Merged Code Clones in Java Programs

指導教員

井上克郎 教授

報告者

Choi Eunjong

平成 24 年 2 月 7 日

大阪大学 大学院情報科学研究科

コンピュータサイエンス専攻 ソフトウェア工学講座

平成 23 年度 修士学位論文

Investigating Clone Metrics of Merged Code Clones in Java Programs

Choi Eunjong

Abstract

A code clone represents similar code fragments in source code. It has been said that the existence of code clones makes software maintenance more difficult because when a defect is included in a code fragment corresponding to the code clone, the other code fragments comprise of the same code clone should be inspected for the same kind of defect.

Even though developers would like to merge code clones into a single method to improve the maintainability of source code, they do not know what characteristics of code clones are appropriate for perform refactoring and which refactoring patterns could be applied. Therefore, information on the characteristics of code clones that were performed refactoring and its applied refactoring patterns is necessary.

In this study, I investigate the characteristic of code clones that were performing refactoring and their applied refactoring patterns in Java programs. In the approach, at first, I get history data of open source projects from software repository. Then, I detect Java methods that were performed refactoring between previous and current versions. Next, I identify pair of cloned methods that were performed refactoring. Finally, I investigate the characteristics of code clones that were performed refactoring and their applied refactoring patterns.

Keywords

Code Clone
Refactoring
Clone Metrics

目 次

1 Introduction **3**

2 Background **5**

2.1 Code Clone 5

2.1.1 Terminologies 5

2.1.2 Causes of Code Clones 5

2.2 Refactoring 7

2.2.1 Purpose of Refactoring 7

2.2.2 Refactoring Patterns to Code Clones 8

3 Proposed Approach **14**

3.1 Definition 14

3.2 Identify Clone Refactoring 14

3.3 Measurement of Clone Characteristics 18

4 Case Study **20**

4.1 Target System 20

4.2 Result 20

4.2.1 Difference on Similarity 21

4.2.2 Difference on a Length 22

4.2.3 Class distance 23

5 Related work and discussion **24**

6 Conclusion **26**

Acknowledgments **27**

参考文献 **28**

1 Introduction

The cost of software development is one of crucial issues. Especially, software maintenance occupies over 65 percentages of total software costs [14] due to the many factors that make software maintenance difficult. The existence code clones in source code is one of the main factors that occurs time and cost consuming tasks in the maintenance. when a defect is included in a code fragment corresponding to the code clone, the other code fragments comprise of the same code clone should be inspected for the same kind of defect. If target software consists of 1MLOC or more lines, it takes much times and efforts to detect defects from all code clones, and then, this task become critical software maintenance problem.

Refactoring [2] is one of the considerable activities that alleviates problems due to code clones because code clones can be merged into a single method by performing refactoring. However, all code clones are not always appropriate for refactoring. For example, code clones derived from programming idiom are necessary to be existed in source code for specific implementation such as checking the close after open the stream in Java programming, or personal dialects of individual developers [7].

Although code clones are not always appropriate for performing refactoring, programmers would like to find and modularize common functionalities in an important system such as social infrastructure, financial system. Theses systems will be maintained in next 10 years as a part of the infrastructure of a society. Due to the limited development time and cost, it is not acceptable for developers to check manually all the detected code clones in source code. Therefore, information on what sort of code clones were performed refactoring and how do code clones were performed refactoring is necessary to help developers when they would like to performing refactoring to code clones.

In this study, I investigate the characteristic of code clones that were performed refactoring and their applied refactoring patterns using history data of Java open source projects. I select 7 refactoring patterns(*extract class*, *extract method*, *extract superclass*, *form template method*, *parameterize method*, *pull up method*, and *replace method with method object*) can be used to merge code cones into a single method. In case study, I get history data of open source projects from software repository, and then, I select methods that were applied predefined refactoring patterns from the outputs of the state of art of REF-FINDER [10][8]. REF-FINDER is a refactoring detection tool between a pair of Java program versions. Next, I identify a pair of cloned methods using similarity metrics, usim(undirected similarity)[9]. Finally, I investigate the characteristics of code clones that were performed refactoring and their applied refactoring patterns.

The remaining part of the paper is structured as follows: Section 2 explains the backgrounds of this study. Section 3 describes proposed approach, while Section 4 explains the case study. Section 5 describes Related Work and Section 6 concludes.

2 Background

In this section, I discuss terminologies on code clone and refactoring to give clear idea of this study.

2.1 Code Clone

A code clone represents lexically, syntactically, or semantically similar code fragments in source code. Many literatures on code clone and its techniques have been published in the past two decades. This section shortly explains basic terminologies in code clone research, *Clone Pair*, *Clone Set* and then discusses the causes of code clones.

2.1.1 Terminologies

Clone relation represents an equivalence relation (i.e. reflexive, transitive, and symmetric relations) on code fragments. It holds between two code fragments if and only if they are the same sequences of code fragments[6], and the following terminologies are defined in terms of clone relation.

Clone Pair : A pair of code fragments if the clone relation holds between them

Clone Set : A group of code fragments where the clone relation holds between any clone pairs.

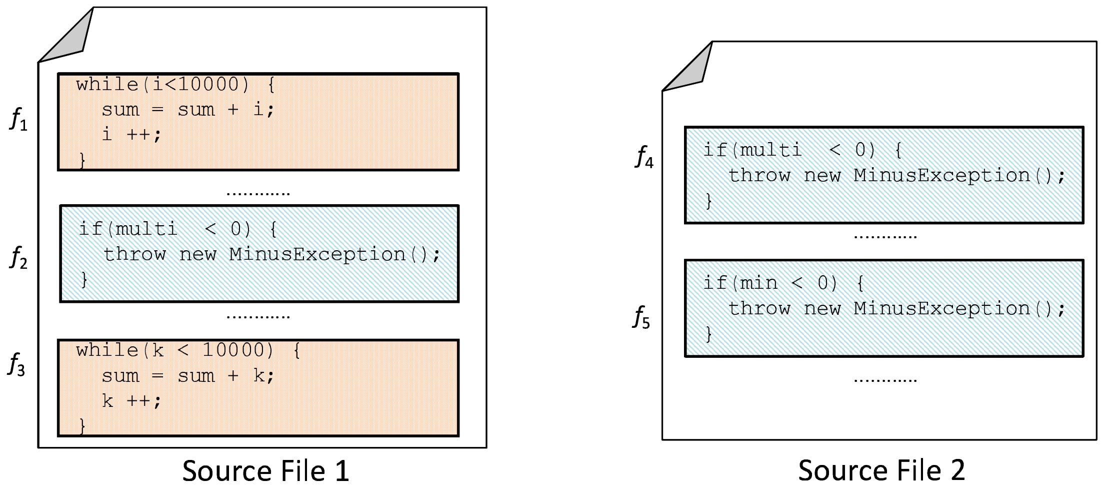
Figure 1 illustrates an example of clone pair and clone set. In Figure 1, f_1 and f_3 consist of a clone pair. In addition, f_2 and f_4, f_2 and f_5, and f_4 and f_5 are clone pairs. Moreover, in case of clone set, f_1 and f_3 and f_2, f_4, and f_5 comprise each clone set. In brief, Clone Pair and clone sets in In Figure 1 are

Clone Pair : $(f_1,\ f_3)$, $(f_2,\ f_4)$, $(f_2,\ f_5)$, $(f_4,\ f_5)$

Clone Set : $\{f_1,\ f_3\}$, $\{f_2,\ f_4,\ f_5\}$

2.1.2 Causes of Code Clones

Well-known cause of code clones is copying existing code fragment and then pasted it with or without modification. Some programmers can not help copying code fragment from existing source code and pasting it due to the limited time, or lack of knowledge



```
while(i<10000) {
  sum = sum + i;
  i ++;
}
```

```
if(multi  < 0) {
  throw new MinusException();
}
```

```
while(k < 10000) {
  sum = sum + k;
  k ++;
}
```

```
if(multi  < 0) {
  throw new MinusException();
}
```

```
if(min < 0) {
  throw new MinusException();
}
```

図 1: Code Pair and Clone Set

of project. On the contrary, some code clones are created unintentionally. For instance, some solution patterns are recursively implemented base on programmer's memory to solve specific problem without intention . C. K. Roy et. al. discussed various reasons for why code clones are introduced in the source code[11].

Developments Strategy

Some code clones are introduced in software systems because of reusing code, logic, design and/or an entire system and the way of system developing. Sometime, code fragments created by development tools become code clones because these tools are often use the same template to generate the identical or similar logic.

Maintenance Benefits

Code fragments are reused in software systems to obtain several maintenance benefits in some systems.. For instance, reusing the existing code fragments by copying and adapting to the new product is recommended tasks in financial system to preserve high reliability and avoid high risk of create new code logic.

Overcoming underlying Limitations

Some code clones are introduced in software systems due to the limitations of the programming languages, and constraints associated with the ability of programmers. If software system is large, programmers have a difficulty to understand overall of assigned software system within a given period of time. Therefore, they reuse existing code fragments because it is an easy way of implementation at hand.

Cloning by Accidents

Some code clones are introduced in software systems by accidents. For instance, when a developer implementates a specific function using specific APIs, he/she should implementate series of function calls and/or other ordered sequences of commands that are defined by API. These sequence orderings sometimes become code clone.

2.2 Refactoring

Fowler defines refactoring as a change to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior[2]. This section shortly discusses of purpose of refactoring and explains refactoring patterns that can be used to merged code clones into a single method.

2.2.1 Purpose of Refactoring

Modifying existing code is a risky task. It might takes a time and efforts. In addition, unexpectable defects might be introduced to software systems due to the modification. However, despite of many risks of modification of existing code, it is worth modifying existing code by performing refactoring because we can achieve many benefits as the results of refactoring[2].

Improve the Design of Software

The design of the program will decay without refactoring. As programmers change code (changes to realize short-term goals or changes made without a full comprehension of the design of the code) the code loses its structure. It becomes harder to see the design by reading the code. Regular refactoring helps code retain its shape(e,g, to eliminate duplicated code). By eliminating the duplicates, programmer ensures that the code says everything alone and only one, which is the essence of good design.

Improve Understandability of Code

A programmer sometimes has to modify code written by other programmers or code that he/she wrote in the past, but he/she does not remember things about it. In this cases, he/she have to understand the code before modifying. However, It takes many efforts to understand the code that are not refactored. A bits of code might be wrong place in code or structure of code is too complicate to understand. Refactoring makes code cleaner and better communicate its purpose. Refactoring provides higher levels of understanding code.

Help to Find Bugs

Help in understanding the code also helps programmer spots bugs. If a programmer refactor code, he/she works deeply on understanding what the code dose, and he/she put that new understanding right back into the code. In addition by clarifying the structures of the program using refactoring, a programmer clarifies certain assumptions he/she has made, to the point at which even he/she can not avoid spotting the bugs.

Help Programmer Develop Fast

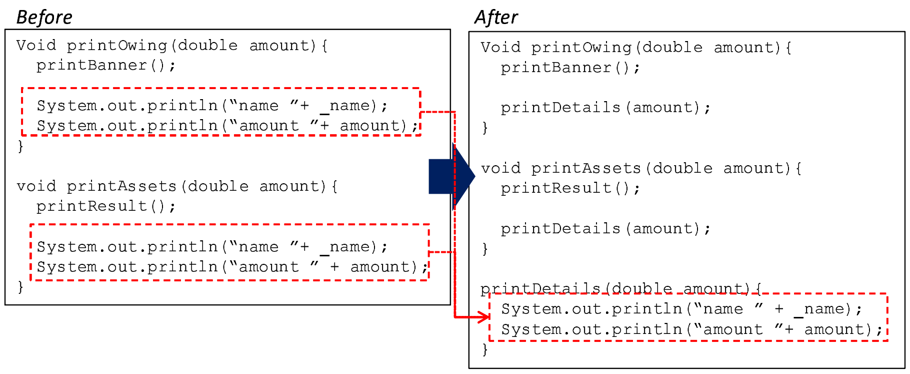
All the earlier points come down to this: Refactoring helps developers develop code more quickly. A good design is essential for rapid software development. Indeed, the whole point of having a good design is to allow rapid development. Without a good design, a developer can progress quickly for a while, but soon the poor design starts to slow developers down. A developer spend time finding and fixing bugs instead of adding new function. Changes take longer as developers try to understand the system and find the duplicated code. New features need more coding as developers patch over a patch that patches a patch on the original code base. A good design is essential to maintaining speed in software development. Refactoring helps developers develop software more rapidly, because it stops the design of the system from decaying. It can even improve a design.

2.2.2 Refactoring Patterns to Code Clones

Fowelr says the best time to perform refactoring to code is when code is stinky, so called bad smell. He chooses duplicated code as the best condition in the stink parade. The following refactoring patterns are available for removing duplicated code from 72 refactoring patterns are written in Folwer's book[2].

Extract Method

Extract method can be applied to code that is too complicate or long to understand its purpose. It also can be applied to remove duplicated code that have the same expression in two methods of the same class. Figure 2 illustrates an example of ap-



```
Void printOwing(double amount){
  printBanner();

  System.out.println(“name ”+ _name);
  System.out.println(“amount ”+ amount);
}

void printAssets(double amount){
  printResult();

  System.out.println(“name ”+ _name);
  System.out.println(“amount ” + amount);
}
```

```
Void printOwing(double amount){
  printBanner();

  printDetails(amount);
}

void printAssets(double amount){
  printResult();

  printDetails(amount);
}

printDetails(double amount){
  System.out.println(“name ” + _name);
  System.out.println(“amount ”+ amount);
}
```

図 2: Example of *extract method*

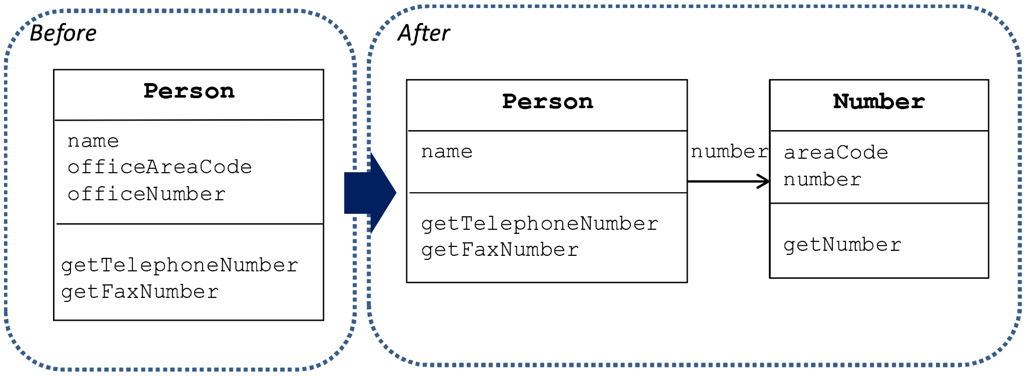


図 3: Example of *extract class*

plying *extract method* to remove duplicated code. Before refactoring, two duplicated statements are exist in two methods(*printOwing()* and *printAssets()*). But after refactoring, duplicated statements are extracted as a new method(*printDetails()*) and the old statements are replaced by a caller statement of new method.

Extract Class

If class is too big to understand easily or complicated, *extract class* can be applied. It also applies to similar methods or fields exists in the same class or unrelated classes. Figure 3 illustrates an example of applying *extract class* to remove duplicated code. After refactoring, a new class(*Number*) is created then a link from the old(*Person*) is added to the new class. Moreover, duplicated parts in similar methods(*getTelephoneNumber()* and *getFaxNumber()*) are moved from the old class into the new class.

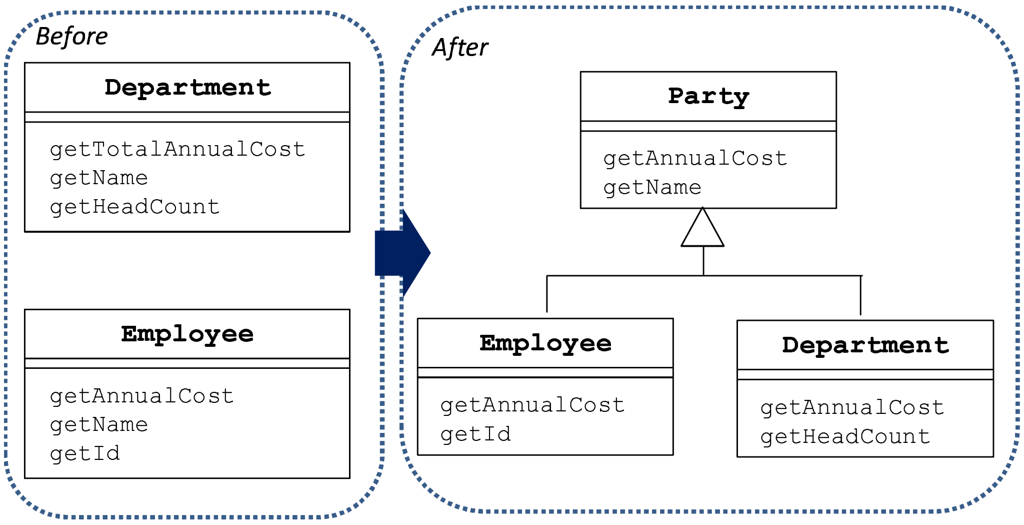


図 4: Example of *extract superclass*

Extract Superclass

Extract Superclass can be applied when two or more classes have similar features but not having a common parent class. Figure 4 illustrates an example of applying *extract superclass* to remove duplicated code. Method *getTotalAnnualCost()* in class *Department* and Method *getAnnualCost()* in class *Employee* is similar. Method *getName()* in class *Department Employee* is identical before refactoring. But after refactoring, a super class(*Party*) is created and common similar features move to the superclass; Method *getName()* are moved to the superclass by using Pull Up Method. Method *getAnnualCost()* in super class has the same signature from Method *getTotalAnnualCost()* in class *Department* and Method *getAnnualCost()* in class *Employee* in old the old version. The difference is achieved by overiiding method *getAnnualCost()* in each subclass.

Form Template Method

If a developer would like to merge two similar methods that perform similar steps in the same order, yet the steps are different from subclasses into a superclass, *form template method* can be used. In this case, A developer can move the similar methods to the superclass and allow polymorphism to play its role in ensuring the different steps to do their things differently. This kind of method is called a templated method. Figure 5 illustrates an example of *Form Template Method* to remove duplicated code. Before refactoring, subclass *ResidentialSite* and *LifelineSite* have similar but not the

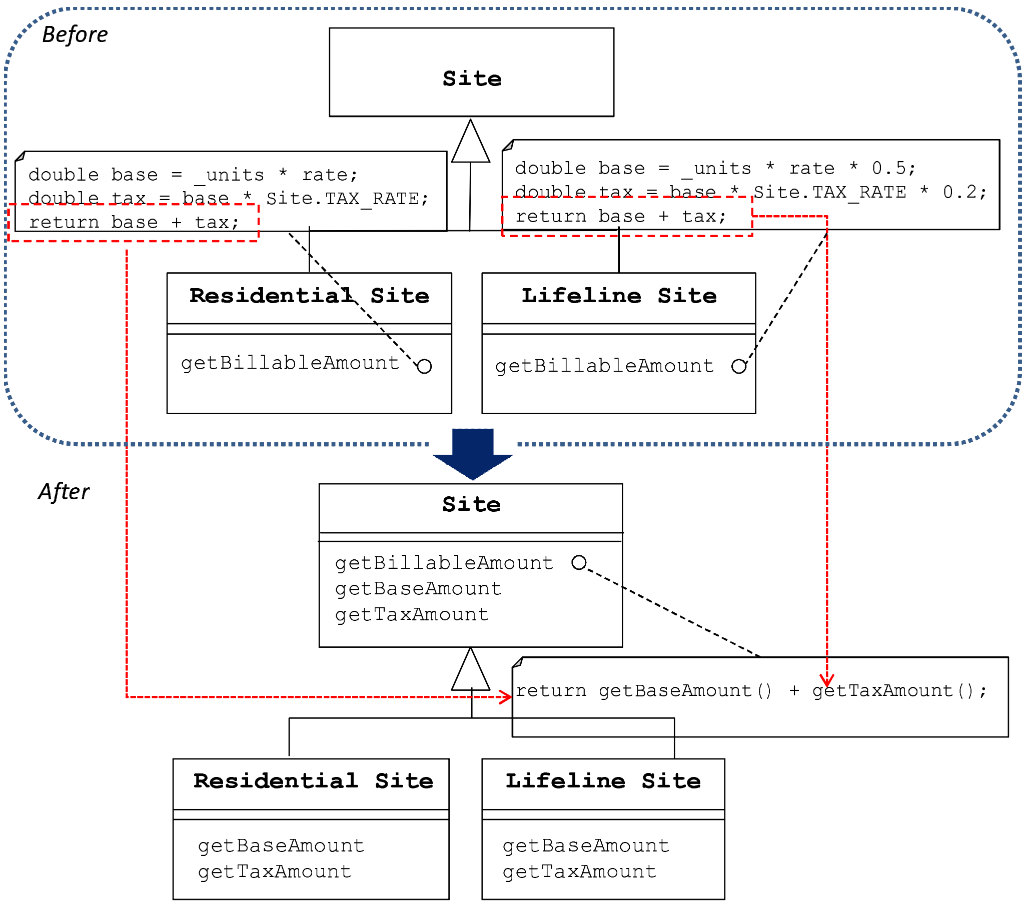


図 5: Example of *form template method*

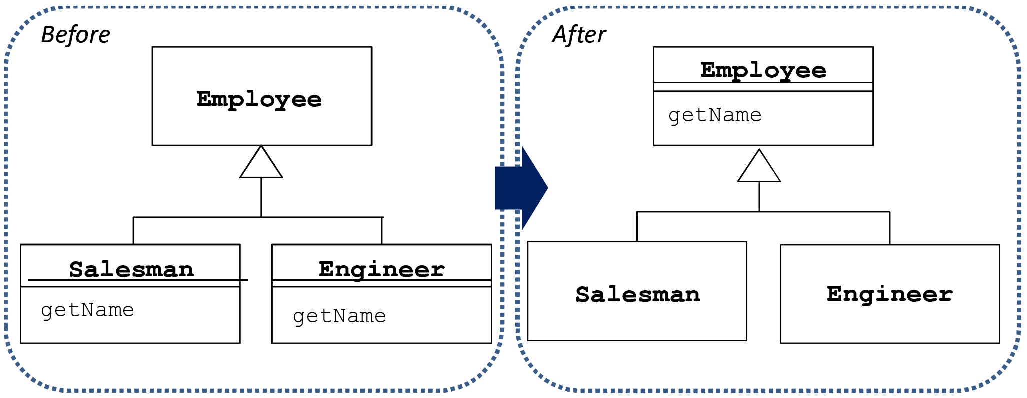


図 6: Example of *pull up method*

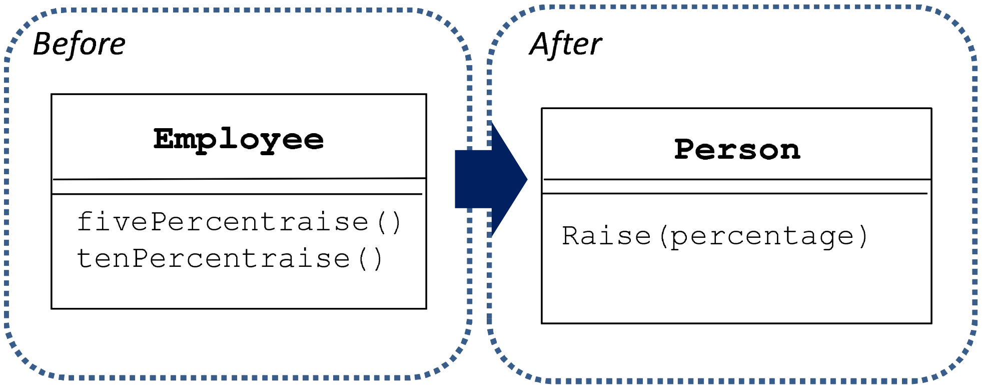
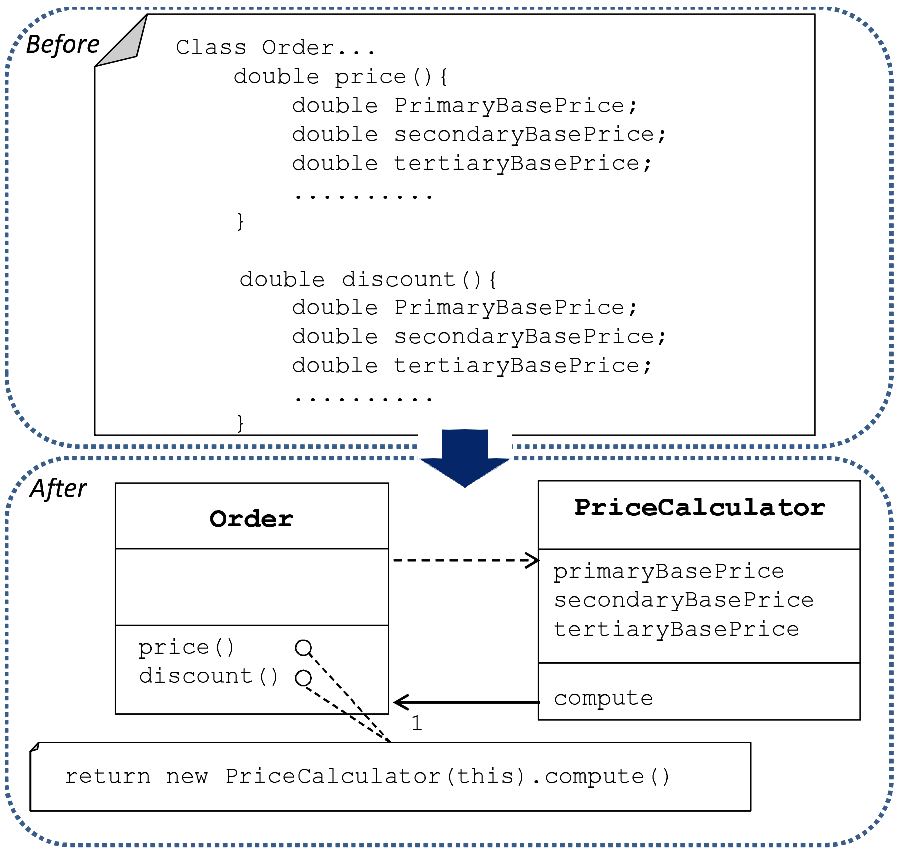


図 7: Example of *parameterize method*



```
Class Order...
    double price(){
        double PrimaryBasePrice;
        double secondaryBasePrice;
        double tertiaryBasePrice;
        ..........
    }

    double discount(){
        double PrimaryBasePrice;
        double secondaryBasePrice;
        double tertiaryBasePrice;
        ..........
    }
```

```
return new PriceCalculator(this).compute()
```

図 8: Example of *replace method with method object*

same method(*getBillableAmount()*). But after refactoring, identical signature are moved into the superclass(*Site*) and the differecne is achieved by overiiding method *getBaseAmountt()* and *getTaxAmmount()* in each subclass.

Pull Up Method

Pull up method can be applied to methods have the same body in subclasses. Figure 6 illustrates an example of *pull up method* to duplicated code. Before refactoring, class *Saleman* and *Enginer* have identical method *getName()*. But after refactoring, method *getName()* are moved to the common superclass(*Employee*).

Parameterize Method

If several methods do similar things but with different values contained in the method body, replacing the separate methods with a single method that handles the variations by parameters, so called *parameterize method* is effective refactoring patterns. Such a change removes duplicated code and increases flexibility, because programmer can deal with other variations by adding parameters. Figure 7 illustrates an example of *earameterize method* to remove duplicated code. Before refactoring, class *Employee* has two methods *fivePercentRaise()* and *tenPercentRaise()* that do the similar things but with different values contained in the method body. But after refactoring, created method(*raise(percentage)* that uses a parameter.

Replace Method with Method Object

Replace method with method object can be applied to the a long method that uses local variables in such a way that developer can not apply *extract method*. Figure 8 illustrates an example of *replace method with method object* to removed duplicated code. Before refactoring, two methods(*price()* and *discount()*) use local variables that can not *extract method*. But, after refactoring, all theses local variables into fields on the new class(*PriceCalculator*).

3 Proposed Approach

This section discuss the approaches to investigate the characteristics of code clones that were performed refactoring and its applied refactoring patterns. The approach consists of the following steps: (1) get history of software projects from software repository. (2) identify code clones that were performed refactoring from extracted files. (3) investigate characteristics of code clones that were performed refactoring and its applied refactoring patterns.

3.1 Definition

In this section, I present my own definition of terminologies in terms of code clones that were performed refactoring. These definitions help to clarify my discussion of proposed approach :

Clone Refactoring Refactoring activity that merge clone pair into a single method

Clone Refactoring Patterns Refactoring patterns that can be used to perform clone refactoring. As described in Section 2.2.2, they are *extract class*, *extract method*, *extract superclass*, *form template method*, *parameterize method*, *pull up method*, and *replace method with method object* can be used to merge code clones into a single method

3.2 Identify Clone Refactoring

To Identify clone refactoring, first, I detect methods that were performed refactoring between two versions, previous and current versions, then identify clone pair that were performed refactoring.

To detect methods that were performed refactoring, I use REF-FINDER, a refactoring detection tool with high accuracy [10][8]. It takes decomposes given two program versions as a database of logic facts about code elements (packages, classes and interfaces, methods, and fields), structural dependencies (containment, overriding relationships, subtyping relationships, method calls, and field accesses), and the content of code elements (e.g., if-then-else control structures in a method-body). Using theses facts, it infers concrete refactoring instances by converting a template logic rule into a logic query, and then invoking the query on the database using a Tyruba logic programming system[13]. The following are template logic rules on clone refactoring patterns[10].

[**Template Logic Rule 1**] added_type(newtFullName, newtShortName, pkg2)
$\wedge$ before_type(tFullName, tShortName, pkg)
$\wedge$ after_type(tFullName, tShortName, pkg)
$\wedge$ added_field(fFullname, X, tFullname)
$\wedge$ added_fieldoftype(fFullName, newtFullName)
$\wedge$ (move_field(fShortName, tFullName, newtFullName)
$\vee$ move_method(mShortName, tFullName, newtFullName))
$\rightarrow$ extract_class(newtFullName, tFullName)

[**Template Logic Rule 2**] added_method(newmFullName, newmShortName, tFullName)
$\wedge$ similarbody(newmFullName, newmBody, mFullname, mBody)
$\wedge$ after_method(mFullName, X, tFullName)
$\wedge$ added_calls(mFullName, newmFullName)
$\rightarrow$ extract_method(mFullName, newmFullName, newmBody, tFullName)

[**Template Logic Rule 3**] added_subtype(tFullName, subtFullName)
$\wedge$ NOT(before_type(tFullName, X, X))
$\wedge$ (move_field(fShortName, subtFullName, tFullName)
$\vee$ move_method(mShortName, subTFullName, tFullName))
$\rightarrow$ extract_superclass(subtFullName, tFullName)

[**Template Logic Rule 4**] same_body(calleeMFullName1, new_mbody1, mFullName1, mbody1)
$\wedge$ same_body(calleeMFullName2, new_mbody2, mFullName2, mbody2)
$\wedge$ NOT(equals(mFullName1, mFullName2))
$\wedge$ NOT(equals(sub_tFullName1, sub_tFullName2))
$\wedge$ added_method(mFullname, mShortName, super_tFullName)
$\wedge$ deleted_method(mFullName1, mShortName, sub_tFullName1)
$\wedge$ deleted_method(mFullName2, mShortName, sub_tFullName2)
$\wedge$ added_calls(mFullName, calleeMFullName)
$\wedge$ added_inheritedmethod(calleemShortName, super_tFullName, sub_tFullName1)
$\wedge$ added_inheritedmethod(calleemShortName, super_tFullName, sub_tFullName2)
$\wedge$ after_method(calleeMFullName1, calleemShortName, sub_tFullName1)
$\wedge$ after_method(calleeMFullName2, calleemShortName, sub_tFullName2)
$\wedge$ after_method(calleeMFullName, calleemShortName, super_tFullName)
$\wedge$ after_subtype(super_tFullName, sub_tFullName1)

$\wedge$ after_subtype(super_tFullName, sub_tFullName2)

$\rightarrow$ form_template_method(super_tFullName, sub_tFullName1, sub_tFullName2, mFullName)

[Template Logic Rule 5] deleted_method(m1FullName, m1ShortName, tFullName)

$\wedge$ before_parameter(m1FullName, params1, X)

$\wedge$ deleted_method(m2FullName, m2ShortName, tFullName)

$\wedge$ before_parameter(m2FullName, params2, X)

$\wedge$ NOT(equals(m1ShortName, m2ShortName))

$\wedge$ added_method(newmFullName, newmShortName, tFullName)

$\wedge$ after_parameter(newmFullName, newparams, X)

$\rightarrow$ parameterize_method(newmFullName)

[Template Logic Rule 6] move_method(fShortName, tChildFullName, tParentFullName)

$\wedge$ before_subtype(tParentFullName, tChildFullName)

$\rightarrow$ pull_up_method(fShortName, tChildFullName, tParentFullName)

[Template Logic Rule 7] added_type(tFullName, tShortName, pkg)

$\wedge$ added_field(fFullName, fShortName, tFullName)

$\wedge$ added_fieldoftype(fFullName, callingtFullName)

$\wedge$ added_method(newmFullName, newmShortName, tFullName)

$\wedge$ after_method(mFullName, mShortName, callingtFullName)

$\wedge$ deleted_methodbody(mFullName, mBody)

$\wedge$ similarbody(newmFullName, newmBody, mFullName, mBody)

$\wedge$ added_calls(mFullName, newmFullName)

$\rightarrow$ replace_method_with_method_object(mFullName, tFullName)

Each template logic rule represents:

- Template Logic Rule 1 represents that an *extract class*] refactoring requires that field *fShortName* or method *mShortName* is moved from class *tFullName* in the old version to new class *newtFullName* in the new version and now field *fFullname* in class *tFullName* is declared to be class *newtFullName* type.

- Template Logic Rule 2 represents that an *extract method* refactoring requires that a new method *newFullName*'s body content *newmBody* is extracted from method *mFullName* in the old version and that *mFullName* now calls *newFullName*.

- Template Logic Rule 3 represents that an *extract superclass* refactoring requires that new superclass *tFullName* of class *subtFullName* is added in the new version and field *fShortName* or method *mShortName* is moved from class *subtFullName* in the old version to superclass *tFullName* in the new version.

- Template Logic Rule 4 represents that an *form template method* refactoring requires that class *super_tFullName* is a superclass of *sub_tFullName1* and *sub_tFull2* in the new version. Method *mFullName*, *calleeMFullName* is added in the class *super_tFullName* and *mFullName* calls *calleeMFullName*. A new method *calleMFullName1*'s body *new_mbody1* is extracted from method *mFullName1* in the old version and a new method *calleMFullName2*'s body *new_mbody2* is extracted from method *mFullName2* in the old version. Methods *caleemShortName* are inherited by class *sub_tTullName1*, *sub_tTullName2* from class *super_tFullName*.

- Template Logic Rule 5 represents that an *parameterize method* refactoring requires that similar method *m1FullName* and *m2FullName* have no parameter in the old version. Method *m1FullName* and *m2FullName* are deleted and a new method *newmFullName* is added in the new version.

- Template Logic Rule 6 represents that an *pull up method* refactoring requires that class *tParaentFullName* is a supercalss of class *tChildFullName* and method *fShotName* are moved from class *tChildFullName* in the old version to class *tParentFullName* in the new version.

- Template Logic Rule 7 represents that an *replace method with method object* refactoring requires that a filed *fFullName* and method *mFullName* are added in new class *newtFullName* in a new version. A method *newFullName*'s body content *newmBody* is extracted from method *mFullName* in the old version and *mFullName* now calls *newFullName*.

Next, I identify code clones that were performed refactoring using the information of REF-FINDER's output. Developers sometimes merge low similarity code clones into a single method after much thought. Figure 9 describes an example of merging low similarity code clones into a single method. In figure 9, a code clone is created by copying existing source code with modification and insertion of additional statement. However, this modification and insertion make code clones difficult to be detected by token base code clone detection tool(e. g. CCFinder[6]).

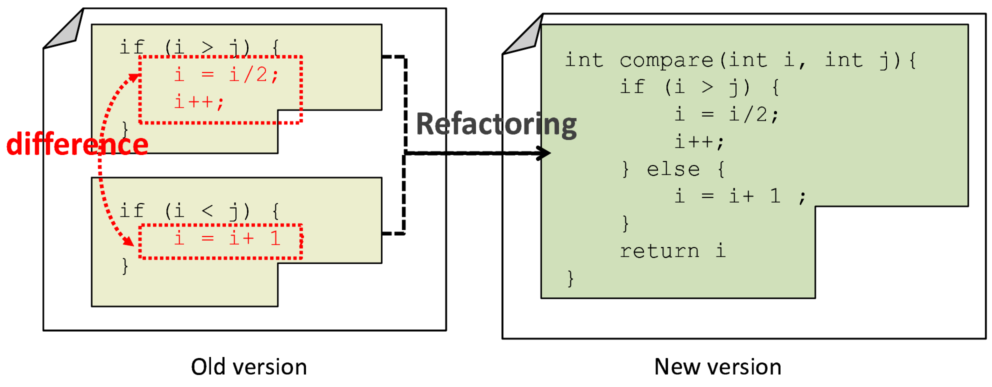


図 9: Example of merging low similarity code clones into a single method

To identify low similarity code clones, undirected similarity(usim)(i.e. a measurement about the similarity of two sequences suggested by Mende et. al[9]) is used. usim is defined by equation (1). It uses Levenshtein distance[12] that measure the minimal amount of changes necessary to transform one sequence of items into a second sequence of items. Each method that are performed refactoring is represented as normalized sequence $sf_x = norm(f_x)$. The normalization removes comments, line breaks and insignificant white space. The resulting edit distance $\Delta f_x, y = LD(sf_x, sf_y)$ then describes the number of items that have to be changed to turn method $f_x intointo f_y$. Levenshtein distance can be normalized to a relative value using the length of the corresponding sequence $l_x = len(sf_x)$.

$$usim\left(f_x, f_y\right) = \frac{\max\left(l_x, l_y\right) - \Delta f_x, y}{\max\left(l_x, l_y\right)} \times 100\,(\%) \qquad (1)$$

If usim value is over 40% between two methods, I define them as clone pair.

3.3 Measurement of Clone Characteristics

After identifying the clone pairs that were performed refactoring, I appliy following clone metrics to investigate the characteristics of code clones are appropriate for performing refactoring.

Similarity difference between clone pairs

Similarity measurement between two sequences, usim is used to identify code clones. Moreover, it is used to measure similarity between clone pairs that were performed clone refactoring. If usim value of clone pair is lower, it means clone refactoring is performed to less similar clone pair, and if usim value of clone pair is higher, clone refactoring is performed to more similar clone pair. This information helps developer to perform clone refactoring . For instance, let us pretend that *extract method* is mainly applied code clone with usim 50, if developers see clone pair with usim 50, they will consider to perform *extract method* preferentially.

The length difference between clone pairs

This metrics informs length difference between clone pair that were performed clone refactoring. If length difference between clone pair is lower, it means clone refactoring is performed to similar size of clone pair, and if length difference between clone pair is higher, it means that clone refactoring is performed to different size of clone pair. This information also helps developer to perform clone refactoring. For instance, let us pretend that *extract method* is mainly applied clone pair with 3 length difference, if developers see clone pair with 3 length difference, they will consider perform *extract method* preferentially y.

Class distance

Class distance represents the relationship of classes who contain clone pair. Class distance gives information of which clone refactoring can be applied to the code clones. For instances, code clones are existed in the same class, *extract method* refactoring pattern is can be applied or code clones are distributed in subclasses who have a common superclass, *pull up method* or *form template method* can be applied. Therefore, information on class distance gives developers clue of selecting clone refactoring patterns.

4 Case Study

I investigate the characteristic of the clone pairs that were performed refactoring as a case study. This section describes the details on the case study and its result.

4.1 Target System

To investigate the characteristics of clone pairs that were performed refactoring, I should choose subject software programs with reliable history information on refactoring activities. After all, I select jEdit, Columba, and Carol, the same projects that used in Prete's paper because their recalls and precisions are high enough. (Their overall precision and recall 79% and 95% respectively). Moreover, I also use same as revision pairs as they used in their paper because its refactoring information is reliable. I select 10 release pairs from overall projects. The following are details on each project and selected revision pairs correspond to each projects:

- jEdit[1] is a mature programmer's text editor written in Java. 2 release pairs(3.0-3.0.1, 3.0.2-3.1) are selected from this project.
- CAROL[2] is a library allowing to use different Remote Method Invocation(RMI) implementations written in Java. 2 release pairs(302-352, 352-449) are selected from this project.
- Columba[3] is an email client written in Java, featuring a user-friendly graphical interface with wizards and internationalization support. 6 release pairs(62-63, 389-421, 421-422, 429-430, 430-480, 480-481) are selected from this project.

4.2 Result

This section describes results of the investigation. I detect 43 refactoring patterns from overall subject projects. Table 1 shows the number of identified clone pairs and detected refactoring on each refactoring pattern. As described in table 1, only 5 refactoring patterns are detected from overall clone refactoring pattern. In terms of clone refactoring, only 4 refactoring patterns(*extract method*, *extract superclass*, *form template method*, *replace method with method object*) are detected. Figure 10 explains the number of identified

[1]http://jedit.org/
[2]http://carol.ow2.org/index.html
[3]http://sourceforge.net/projects/columba/

表 1: Number of identified clone pairs and detect refactoring on each refactoring pattern

Refactoring Patterns	#Identified Clone Pairs	#Detected Refactoring
Extract Class	-	-
Extract Method	11	25
Extract Superclass	1	12
Form Template Method	2	5
Parameterize Method	-	-
Pull Up Method	-	3
Replace Method with Method Object	17	43

refactoring methods in terms of clone refactoring. *replace method with method object* is most frequently applied refactoring pattern between the all refactoring patterns. The second frequently occurred clone refactoring patterns is *extract method*, next is *form template method*, and the final is *extract superclass*.

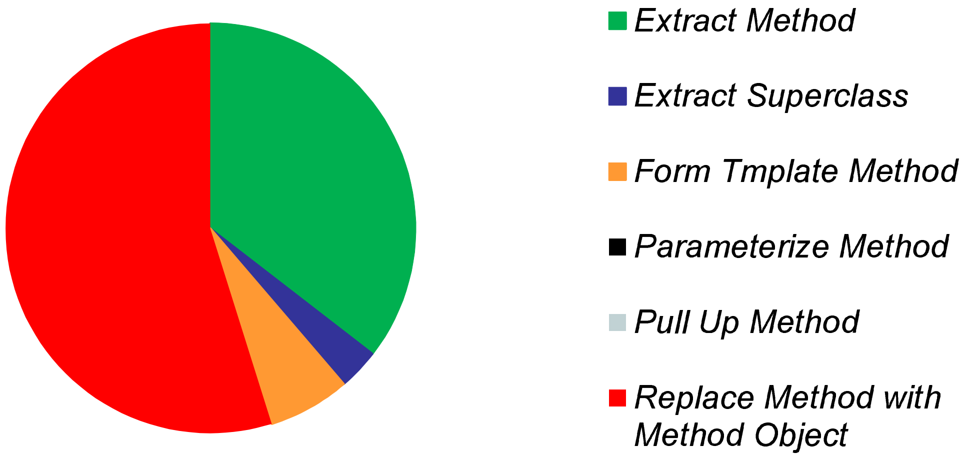


図 10: Number of identified refactoring methods

4.2.1 Difference on Similarity

Similarity differences on clone pairs that are performed clone refactoring are shown in Figure 11. As I mentioned in Section 3.3, I use usim to measure similarity between two sequences. If a pair of code fragments whose usim value is over 40, I define them as code clone, the minimum usim value of code clones is 40. Lower usim value represents that the similarity between a clone pair is lower. In contrast, higher usim value represents that the similarity between a clone pair is higher. The usim value of *extract method* pattern, *replace method with method object* pattern are low. and *extract superclass* pattern, *form template method* pattern are high. It means that *extract method* pattern, *replace method with method object* pattern are applied to the lowly similar clone pairs and *extract superclass* pattern, *form template method* pattern are applied to the highly similar clone pairs.

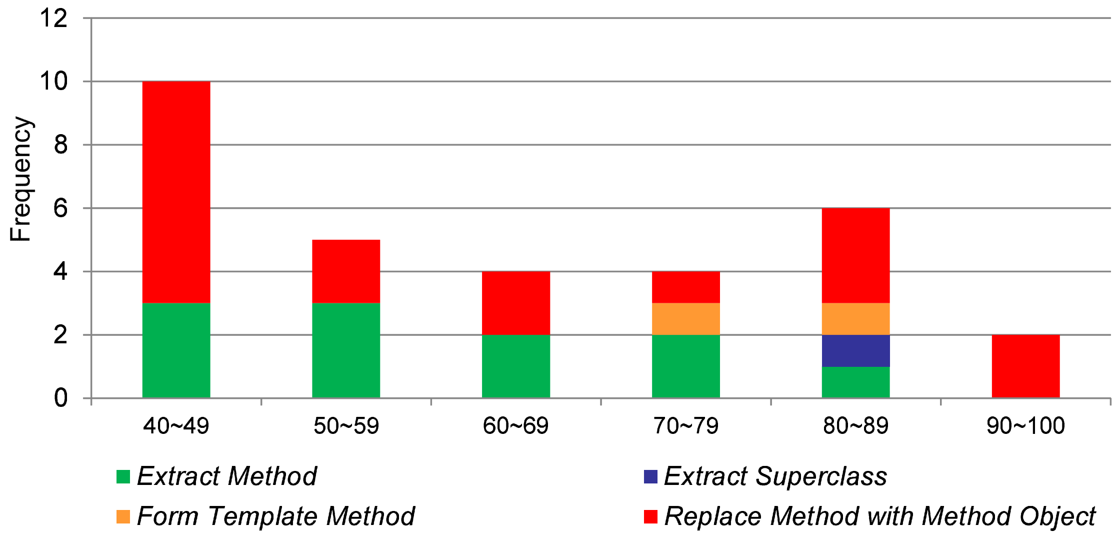


図 11: The sequence similarity between each clone refactoring pattern

4.2.2 Difference on a Length

Length difference on clone pairs that are performed clone refactoring are shown in Figure 12. As I mentioned in Section 3.3, lower length difference represents that the length difference between the clone pairs are little. In contrast, higher length difference represents that the length difference between the clone pairs are large. Length difference of *extract method* pattern, *extract superclass* pattern, and *form template method* pattern are low, and *replace method with method object* is various. It means that *extract method* pattern, *extract superclass* pattern, and *form template method* pattern are applied to clone pairs with little length difference, and *replace method with method object* is appled to various different length of the clone pairs.

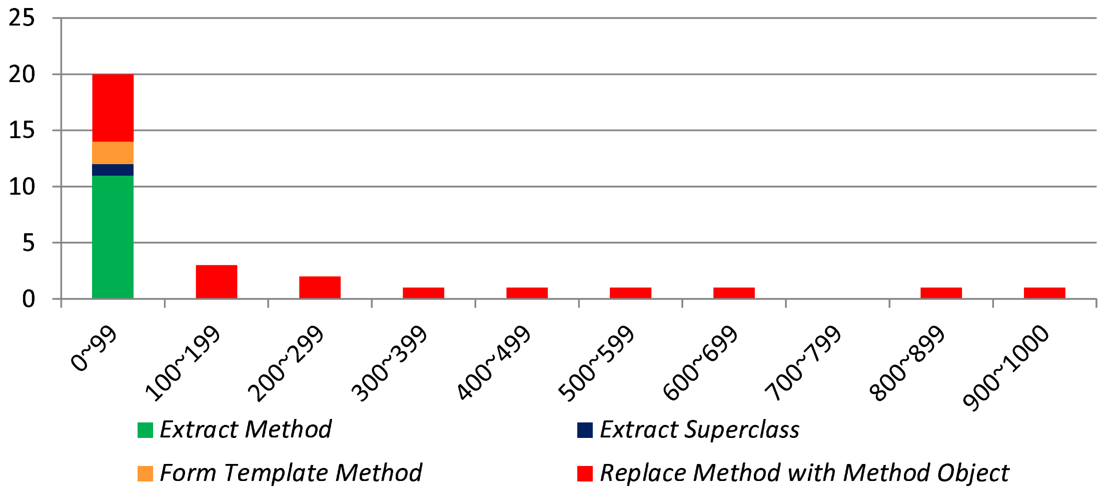


図 12: The length difference between each clone refactoring pattern

4.2.3 Class distance

Between detected refactoring patterns, only *replace method with method object* pattern can be applied to code clones between any classes because it has no condition of location of code clones before performing refactoring. Therefore, I investigate class distances on *replace method with method object* pattern. The class distances of *replace method with method object* pattern are shown in Figure 13. In *replace method with method object* pattern, clone pairs that are contained in the same package are most applied.

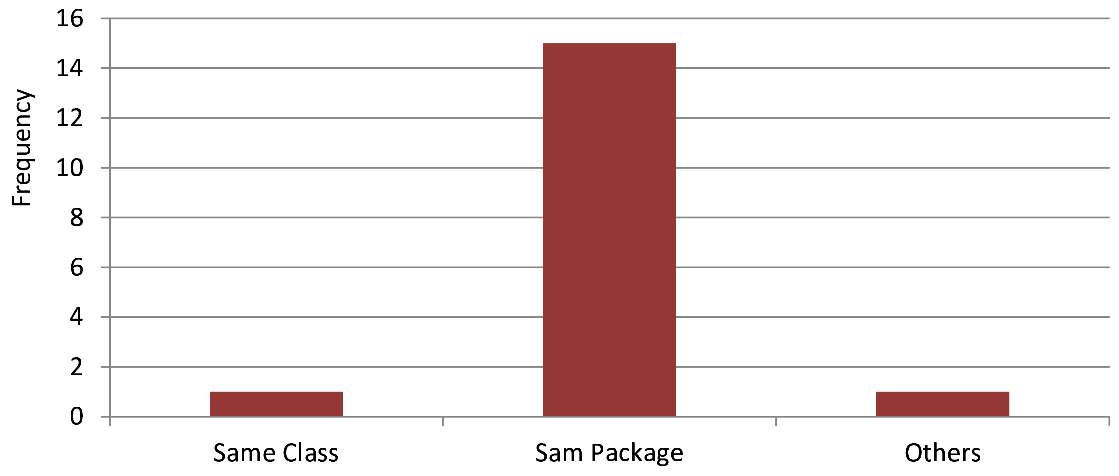


図 13: The class distance on replace method with method object

5 Related work and discussion

Jiang *et al.*[4] and Kapser *et al.*[7] pointed out that code clone detection tools using parameterized matching detected a lot of false positives. Jiang *et al.*[4] used textual filtering techniques to remove false positives from CCFinder's output. They removed clone papers code clones whose textural similarly falls below a certain threshold. Kapser *et al.*[7] proposed the following techniques to remove false positives from the output of their token-based clone detection tool.

- identifier names are not parameterized outside of methods.
- simple method calls are matched only if Levenshtein Disitance of those method names is small.
- logical structures (e.g., switch statements, if-else block) are matched if 50% of tokens in these structures are identical.

There is the possibility to make our method more effective by applying the filtering techniques proposed by Jiang *et al.* and Kapser *et al.* as the preprocessor or the postprocessor of our method.

CCFinderX[5] developed by Kamiya provides the metric TKS(S) that means the number of token types in code fragments belonging clone set S. The metric TKS is effective to remove clone sets not in need of developer's investigation (e.g., consecutive variable declarations) because those clones tend to have small number of token types. This means that there is the possibility of improving the effective of our method by use of the metric TKS in addition to the use of the metric RNR. However, clone sets with low RNR value include a lot of consecutive parts. They tend to have small number of token types and low TKS value. This correlation means that the use of both TKS and RNR are not significantly effective.

Balazinska *et al.* and Higo *et al.* characterize clone sets according to the ease of refactoring. Balazinska [1] *et al.* characterize clone sets by analyzing the following information:

- differences among the code fragments belonging to a clone set
- dependencies between the code fragment belonging to a clone set and its surrounding code

Higo [3] *et al.* proposed two metrics to represent the average number of the externally defined variables respectively referenced and assigned in the code fragments belonging to a clone set. The combination of our method and the techniques focusing on the ease of refactoring has the possibility to improve the effectiveness of clone set filtering.

6 Conclusion

This study investigates the characteristics of clone pairs that were performed refactoring to investigate the characteristics of code clones are appropriate for performing refactoring and which refactoring pattern is preferentially necessary for a tool support clone refactoring. I investigate 10 revision pairs from 3 Java open source projects using REF-Finder to detect refactoring and usim to identify code clones. I found characteristics of clone pairs that were performed(e.g. in case of *replace method with method object* is the most frequently applied refactoring pattern and they are applied to various length different clone pairs with little similarity in the same package)

As future work, I will investigate more revision pairs and more Java software projects. Moreover, I will use more metrics(e.g. cohension, cyclomatic complexity) to investigate the characteristics of clone pairs that were performed refactoring more accurately.

Acknowledgments

First and foremost, I would like to thank Professor Katsuro Inoue for giving me the opportunity to work with him. He provided advices and important direction to my thesis work. In addition, I am very grateful to Associate Professor Makoto Matsushita for his helpful comments on this study. I would like to express my gratitude to Assistant Professor Takashi Ishio and Yuki Manabe for their valuable comments and helpful criticism on this thesis.

Especially, I would like to express my gratitude to Assistant Professor Norihiro Yoshida in Nara Institute of Science and Technology for his comments, criticisms and advices on this thesis. I also would like to express my gratitude to Assistant Professor Higo Yoshiki in Kusumoto Laboratory for his valuable comments. Many Thanks to lots of friends in the Department of Computer Science, especially students in Inoue Laboratory.

And finally, I would like to express my gratitude to members of Yodogawa Rotary Club for their supports.

参考文献

[1] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. In *Proc. of WCRE 2000*, pp. 98–107, 2000.

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

[3] Yoshiki Higo, Shinji Kusumoto, and Katsuro Inoue. A metric-based approach to identifying refactoring opportunities for merging code clones in a Java software system. *J. Softw. Maint. Evol.: Res. Pract.*, Vol. 20, pp. 435–461, 2008.

[4] Z. M. Jiang and A. E. Hassan. A framework for studying clones In large software systems. In *Proc. of SCAM 2007*, pp. 203–212, 2007.

[5] T Kaimiya. In *http://www.ccfinder.net/doc/10.2/en/tutorial-ccfx.html.*

[6] 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, 2002.

[7] C. J. Kapser and M.l W. Godfrey. “Cloning considered harmful” considered harmful: patterns of cloning in software. *Empir Software Eng*, Vol. 13, pp. 645–692, 2008.

[8] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit. Ref-Finder: a refactoring reconstruction tool based on logic query templates. In *Proc. of ESE/FSE 2010*, pp. 371–372, 2010.

[9] T. Mende, R. Koschke, and F. Beckwermert. An evaludation of code similarity identification for the grow-and-prune model. *Journal of Software Maintenance*, Vol. 21, No. 2, pp. 143–169, 2009.

[10] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim. Template-based reconstruction of complex refactorings. In *Proc. of ICSM 2010*, pp. 1–10, 2010.

[11] C. K. Roy and J. R. Cordy. A survey on software clone detection research. *Technical Report*, Vol. 541, , 2007.

[12] Levenshtein VI. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, Vol. 10, pp. 707–710, 1966.

[13] K. D. Volder. *Type-oriented logic meta Programming*. PhD thesis, The University of British Columbia, 1998.

[14] S. W. L. Yip and T. Lam. A software maintenance survey. In *Proc. of APSEC 1994*, pp. 70–79, 1994.