Proceedings of the SICE Annual Conference 2016
Tsukuba, Japan, September 20-23, 2016

A Method of Program Refactoring based on Code Clone Detection and Impact Analysis

Masakazu Takahashi[1†], Reiji Nanba[2], Yunarso Anang[1], Tatsuya Uchiyama[1], and Yoshimichi Watanabe[1]

[1]Department of Computer Science and Engineering, University of Yamanashi, Yamanashi, Japan
(Tel : +81-55-220-8585; E-mail: {mtakahashi, g14dma01, g12mk003, nabe}@yamanashi.ac.jp)
[1]Department of Civil Engineering, Dai-Ichi Institute of Technology, Kagoshima, Japan
(Tel : +81-55-220-8585; E-mail: r-nanba@daiichi-koudai.ac.jp)

Abstract: This paper proposes a method that aggregates similar portions in a program into one common portion and redesigns current program structure to appropriate program structure. When a new function is added to an existing program, the function tends to be developed by copying & pasting a portion in the program and modifying its portion. As a result, it becomes to exist many similar portions in the program. In the case that error modifications or changes occurs in the similar portion, appropriate modifications are required to the all similar portions. It would be considered that the quality and efficiency of those tasks are decreased. So that, this paper proposes a method that detect similar portions with minor modifications and a method that aggregates those similar portions to one appropriate common portion with well-defined program structure. As a result, current program becomes to be refined to the program that can accommodate future modification or changes properly.

Keywords: Code Clone, Impact Analysis, Refactoring

1. INTRODUCTION

When developing a new program or when adding and modifying program functions, in many cases, programmers copy certain portions of the program and paste them, while revising them, in order to complete development of the program. This method retains a number of similar program portions (code clones: CC) within the program. Programs including a large number of CCs have the following problems: Those portions related to CCs are difficult to change, errors in changes are highly likely to be produced, and the process of making changes becomes inefficient. These become issues when using programs for a long time.

This paper proposes a method to redesign a program, written in Java as a representative object oriented programming language (OOPL), into a proper program structure by detecting CCs within the program. Moreover, this paper proposes a method to clarify scopes of verification associated with addition and modification of the program (referred to as the program change). This proposed method clarifies scopes where verification is needed in the unit of method, filed, and line in order to streamline re- verification. This can recreate a program including CCs into a proper program structure without overlapped portions. By doing so, this method can verify the adequacy of the recreated program. Additionally, this proposed method makes it easier and more efficient to subsequently change the program, in order to achieve the program structure which is adequate to use for a longer time.

2. RELATED STUDIES

This chapter describes the previous studies related to the proposed method. The previous studies are broadly categorized into those regarding CC detection, those regarding refactoring, and those regarding impact analysis.

First, studies regarding CC detection are described in the following section. CCs are detected in the unit of characters, expressions, or lines. The character-based CC detection process can detect CCs in any unit; however, this process can detect only completely-matched CCs. Furthermore, this process takes considerable time to detect CCs because of conducting matching for each single character. The expression-based CC detection process deletes unnecessary blanks, line breaks, and comments preliminarily, while replacing variable names and numerical values with specific characters. Therefore, this process can detect program portions where variable names and numerical values have been changed as CCs. As is the case of expression-based CC detection, the line-based CC detection process deletes unnecessary blanks, line breaks, and comments and replaces variable names and numerical values with specific characters, and then calculates the hash value of each line. Once this hash value is calculated, CCs can be detected quickly [1].

Next, studies regarding refactoring are described. Design patterns are known as the standard of proper program design[2]. Introduction of design patterns into program design can enhance readability and maintainability of program written in OOPL. Gamma et al. have proposed 23 design patterns. Moreover, Fowler et al. have organized representative refactoring methods (referred to as refactoring formats hereinafter) in a catalog format [3]. Inoue et al. have proposed support methods and tools for implementing refactoring for a part of refactoring formats. Furthermore, they have proposed benchmarks which can serve as information for determining a proper refactoring format by using the CC distribution status, CC length, and CC location information. These benchmarks are referred to as metrics[4]. Representative metrics include the Number of Related Variables (NRV, which is the average number of externally defined variables referred within CCs), and the Number of Substituted Variables (NSV, which is the average number of variables assigned within CCs).

Finally, studies regarding impact analysis are described.

978-4-907764-50-0 PR0001/16 ¥400 ©2016 SICE

Impact analysis is a method to clarify the scope of program which might be affected when the program is modified. Kung have proposed a method to clarify program classes which might be under influence of program modification based on the class firewall concept [5]. However, this method had a problem in that methods and fields not modified (referred to as members hereinafter) would be included in the range under influence. Jang have proposed a method to clarify methods and fields under modification by using the Member Dependency Graph which indicates the access relationship between operations and properties [6]. However, this method also had a problem in that a large amount of program portions dependent from influence can be included within members. As mentioned above, the traditional methods were inefficient in verifying program portions under influence of program modification.

3. PROPOSED PROGRAM REFACTORING METHOD

This paper proposes a method that implements program refactoring by detecting CCs within a program in order to make it easier and more efficient to modify the program. By doing so, this method achieves a program structure that can be used for a long time.

Fig.1 shows the proposed refactoring procedure. Refactoring is conducted on every single pair of CCs. As preparation for refactoring, STEP 1 detects CCs from a program before modification. This step also creates a diagram referred to as a Member Access Graph (MAG) which indicates the access relationship between methods and fields before modification, and a diagram referred to as a Member Override Graph (MOG) which indicates the inheritance relationship of methods. STEP 2 selects a pair of CCs (referred to as a CC pair), which conducts refactoring, from detected CCs. This step then analyzes the content of the CC pair in order to create information necessary for determining the refactoring format. Where refactoring is not necessary or where refactoring is judged to be impossible to conduct, this step returns to the initial step and selects the next CC pair. STEP 3 modifies the program based on information of refactoring. This process is done manually. When refactoring has been completed, this step creates the MAG and MOG of the program modified. STEP 4 then implements impact analysis regarding the relevant modification based on the MAG and MOG before and after modification. The scope of the program which is under the influence of the portions modified differs between MAG and MOG before and after modification. Based on this result, the program is verified. This verification process is done manually. This completes refactoring for one CC pair. Afterward, the program is refined and elevated to a proper structure by repeating the above-mentioned procedure from STEP 1 through STEP 4. The following section describes what is actually done in each step in detail.

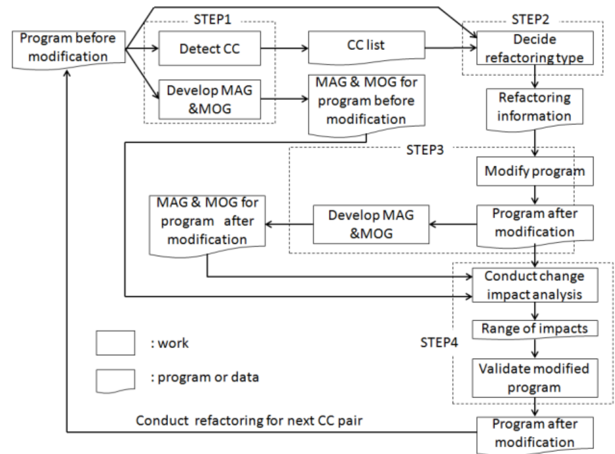

Fig.1 Proposed Refactoring Procedure

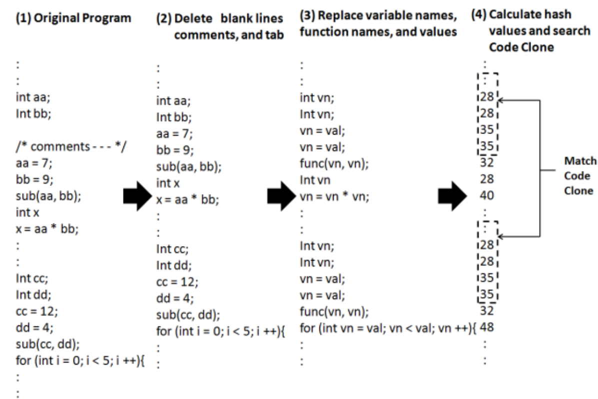

Fig.2 Flow of Cole Clone Detection

3.1 Preprocessing for Refactoring (STEP 1)

This section describes the operation of STEP 1. This operation consists of CC detection and the creation of MAG and MOG. Each of the tasks is as follows:

(1) CC detection

CCs are generated when a program is developed by copying, pasting, and modifying the existing portions of the program. Variable names and numerical values differ in many CCs, while they are not completely matched. Therefore, a better method which can detect CCs although the program is slightly modified. Fig.2 shows the flow of CC detection including slight program modification. First, information which has nothing to do with program execution, such as blanks, comments, or tabs within the program is deleted. Second, variable names, function names, and numerical values are replaced with specific characters, and the basic program structure is clarified. Third, the hash value is calculated per each line of the program with the basic program structure clarified. This hash value is calculated by obtaining and adding ACII codes of each line. Finally, portions, where the hash value of each line matches more than n (the value of n is changeable as necessary), are detected as a CC list.

(2) Creation of MAG and MOG

Programs written by OOPL execute their services based on combinations of method calls between objects and

references of fields. MAG graphically indicates the relationship between method calls and references of fielsds. MAG expresses the relationship of method calls in the directed line from the calling source to the calling destination, while expressing the field reference relationship in the directed line from the reference source to the reference destination. MOG graphically indicates overwritten methods accompanied by class inheritance, implementation of methods which are defined abstractly, and fields encapsulation. MOG expresses overwritten methods in the directed line from the method to be overwritten to the behavior to be overwritten. Fig. 3 shows the corresponding relationship between MAG and MOG. The program shown in Fig. 3 (a) has the following classes: BaseClass, DerivedClass, and Application. In addition, the BaseClass class has the operation methods, method1(int x) and method1(String x), the DerivedClass class has method1(int x), and the Application class has method2() and method3(). The method2() method (call source) of this program calls the call destination, method1(String x) of the DerivedClass class, while the method3() method (call source) calls the call destination, method1(int x) of the BaseClass class. Fig. 3 (b) shows MAG where these relationships are connected with directed lines. Next, the method1(int x) method of the DerivedClass class (operation to overwrite) overwrites the method 1(int x) method of the BaseClass class (operation to be overwritten). Fig.3 (c) shows MOG where these relationships are connected with directed lines.

3.2. Determination of Refactoring Formats (STEP 2)

This section describes the operation of STEP 2. This operation consists of the determination of CC pairs which conduct refactoring and the determination of CC pairs. CC pairs are determined simply by choosing CC pairs. Therefore, the determination of refactoring formats is described below.

Refactoring formats are determined based on refactoring information which is obtained by analyzing the program part of CC pairs under refactoring. Refactoring information consists of the start line of the CC, the end line, the number of external variables used, the number of methods called, and the parent class. The start and end lines of the CC are obtained from the CC list. The number of external variables, the number of methods called, and the parent class are obtained from MAG and MOG. Table1 indicates the refactoring formats treated in this research and the judging criteria for applying these formats.

3.3. Implementation of Refactoring (STEP 3)

STEP 3 modifies the program based on refactoring information, after which this process creates the MAG and MOG of the program which has been modified. Program modification is done manually. The creation of the MAG and MOG of the program which has been modified is the same operation as that described in section 3.1 (2).

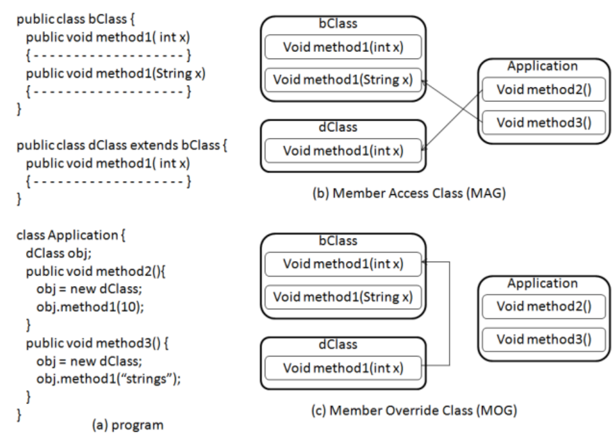

Fig.3 Relationships between MAG and MOG

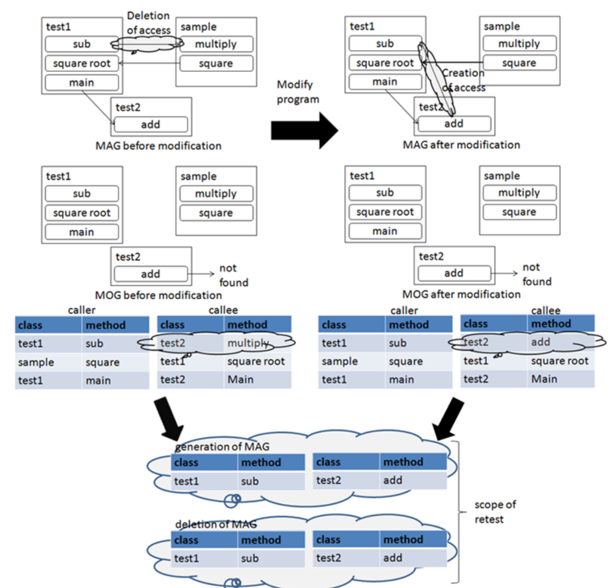

Fig.4 Sample of Change Impact Analysis

3.4. Implementation of Impact Analysis (STEP 4)

This section described the operation STEP 4. Impact analysis is a method to identify a scope of a program under the impact of modification. Scopes under impact are clarified in the unit of classes, members, or lines. This research clarifies a program scope which needs to be verified in two stages due to program modification. The first stage clarifies a program scope under the impact of program modification in the unit of members, while the second stage clarifies a program scope under the impact of program modification in the unit of lines in methods.

The member-based impact analysis which is the first stage clarifies the scope under impact in the unit of members by clarifying differences in MAG and MOG between before and after program modification. First, the relationship of access to members which appeared and vanished due to program modification is extracted based on comparison of MAG before and after program modification. In a similar manner, the override relationship of methods which appeared and vanished due to program modification is then extracted based on

comparison of MOG before and after program modification. These differences become members which are under impact of program modification. The following shows an example of change in a MAG. Fig.4(a) shows the MAG before modification and the output result of this MAG by using the MAG/MOG creation tool (described later in 3.5), and Fig.4(b) shows the MAG after modification and the output result of this MAG by using the MAG/MOG creation tool. This program consists of the test1 class, the test2 class, and the sample class. The test1 class has the operation methods, subclass, square root, and main, the test2 class has the add method, and the sample class have the multiply and square methods. In addition, the sub method of the test1 class before modification accesses to the multiply method of the sample class, the main method of the test1 class accesses the add method of the test2 class, and the square method of the sample class access to the square root method of the test1 class. On the other hand, after modification, the sub method of the test1 class accesses to the add method of the test2 class, the main method of the test1 class accesses to the add method of the test2 class, and the square method of the sample class accesses to the square root method of the test1 class. Comparison of Fig.4(a) and (b), shows that access of the sub method of the test1 class to the multiply method of the sample class vanishes, while the access of the sub method of the test1 class to the add method of the test2 class increases. This status is shown by the shaded portion in Fig.4. This portion becomes members which are affected by modification.

The second stage, sentence-based impact analysis, clarifies those lines under impact by using static program slicing for the affected portions extracted in the first stage. Static program slicing is a method which focuses on any variable in the program in order to extract only program portions (lines) necessary for calculating the variable focused on. These program portions are referred to as static slices. Where any input data is given to static slices, the same calculation result as the original program is obtained for the variables focused on. Static slices are created by tracing the dependency relationship of data and control between the program lines in the member inversely within the variable calculation process. Fig.5 shows an example of static slice extraction procedure with the focus on the program argument, variable x. The figures listed on the left side of Fig.5 indicate the number of program lines. First, the variable x in the 7th line of the original program is focused on. This variable x in the 7th line is calculated by using the variable y in the 5th line. The variable x is initialized in the 1st line. On the other hand, the variable y in the 5th line is calculated in the 4th line. This variable y in the 4th line is initialized in the 2nd line. According to the results above, as shown in the right side of Fig.5, the static slices for the variable x in the 7th line are the lines 1, 2, 4, 5, and 7.

3.5. Creation of a Refactoring Support Tool

This chapter describes a tool that supports the operation of each STEP in Fig.1. This prototype is composed of the following subtools: CC detection tool, MAG/MOG creation tool, refactoring format proposal tool, and impact analysis tool. Each subtool is explained as follows:

The CC detection tool detects CCs containing slight changes made within a program. This tool is used by STEP 1 in Fig.1. The input of this tool is a program, while the output is a CC list. Fig.6 shows an example of a CC list output by the CC detection tool. The CC list contains the paths of all files where CCs exist, the start/end lines of CCs of all CCs.

Table1 Refactoring formats and their judging criteria

Refactoring format	Refactoring method	Judging criteria
Extract Method	Integrates those operations (CCs) overlapping within the same class by creating a new method.	CCs exist within the same class. One CC exists (there are a few CCs).
Pull Up Method	Integrates those operations (CCs) overlapping within the same subclass having the same super class by creating a new operation in the super class.	Has the same super class. CCs exist within the subclass.
Extract Class	Integrates multiple operations (CCs) overlapping within the same class by creating a new class.	CCs exist within the same class. Multiple CCs exist (there is a number of CCs).
Extract Super Class	Integrates multiple operations (CCs) overlapping within the class without having the same super class by creating a new parent class. The original class becomes the subclass of the super class.	Does not have the same super class. Multiple overlapping CCs exist within the class.
Parameterized Method	Integrates those overlapping operations within the same class, where only the values to be used within operations differ, by making the values to be used one argument.	CCs exist within the same class. Multiple CCs exist (there is a number of CCs). Values to be used within CCs differ.
Pull Up File	Integrates multiple subclasses with the same super class having the same property by giving the super class properties.	Has the same super class. Overlapping properties exist within the subclass.

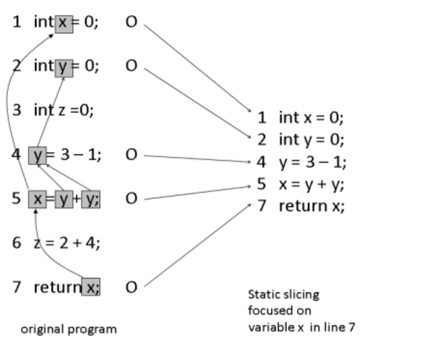

Fig.5 Sample of Static Program Slicing

Code Clone No	File Path	Start Line	End Line
1	c:¥testCode¥c2.java	13	34
1	c:¥testCode¥c2.java	45	69
2	c:¥testCode¥c2.java	13	34
2	c:¥testCode¥c3.java	33	54
3	c:¥testCode¥c2.java	13	34
3	c:¥testCode¥c3.java	89	113
:	:	:	:

Fig.6 Example of Code Clone List

c:¥testCode¥c2.java	
Start Line	13
End Line	34
Number of Used Externally Defined Variables	0
Number of Method Call	0
Parent Class	c1
Total of Codes	12
Start Line	45
End Line	69
Number of External Variables Used	0
Number of Method Called	0
Parent Class	c1
Total of Codes	12
Applicable Refactoring Type	Extarct Method
	Parameterized Method

Fig.7 Example of Refactoring Information

The MAG/MOG creation tools creates the MAG and MOG of a program. This tool is used by STEPS 1 and 3 in Fig. 1. The input of this tool is a program, while the output is MAG and MOG. The lower of Fig. 4 shows an example of the MAG as the output of the MAG/MOG creation tool. MAG contains classes and members to which the members of the call source belong, and classes and methods to which the members of the call destination belong. MOG contains overwriting methods and methods to be overwritten. The refactoring format proposal tool creates refactoring information of CC pairs selected. This tool is used by STEP 2 in Fig. 1. The input of this tool is a program, a CC list (one CC pair selected from the CC list), MAG and MOG, while the output is the refactoring format. Refactoring information contains the total number of CC lines, the number of externally defined variables used, the parent class, and the number of method calls. Fig.7 indicates refactoring information as the output of the refactoring format proposal tool. The impact analysis tool clarifies program scopes which are under impact of program modification based on the unit of members and lines from the program including MAG and MOG before and after program modification. This tool is used by STEP 4 in Fig. 1. The input of this tool is a modified program and MAG/MOG before and after modification, while the output is the impact scope. The lower right of Fig. 4 shows the impact scope as the output of the impact analysis tool.

4. EVALUATION OF THE PROPOSED METHOD AND TOOL

This section describes the results of evaluation for the proposed method and the tool. The evaluation was conducted by two engineers who have same level of skills and experiences (they had 4-6 years programming experiences using OOPL and attended a two-days lecture for refactoring).

An evaluation procedure is described in this paragraph. First, one engineer (engineer A) conducted refactoring for prepared programs using the proposed method and the tool. Next, other engineer (engineer B) conducted refactoring for prepared programs without the proposed method and the tool. Last, three engineers (engineer A, engineer B, and engineer C who had same skills and experiences) compared their refactoring results, and evaluated adequacy of refactoring.

Test cases are described in this paragraph. Left side in Table 2 shows a list of test cases. Test case 1 - 5 is used for the evaluation of refactoring that single refactoring format is applied. Test case 6 - 8 is used for the evaluation of refactoring that plural refactoring formats are applied simultaneously. This is because that there were reports that the most CC could be applied "pull up + parameterized method" and "extract method + parameterized method", and those CC was added some slight modifications, such as changing variable names and constant values [7]. Table 2 shows the test results. Regarding the test case 1 to 5, engineer A and B could conduct refactoring to all test cases appropriately. Regarding the test case 6 to 8, engineer A could conduct refactoring to all test cases appropriately. While, engineer B could not conduct refactoring of the test case 6. Engineer B applied only "parameterized method" for the test case 6, because Engineer B missed to apply "pull up method." Consequently, appropriate refactoring had not done.

Table 2 List of test cases

No	Test Case	Eng. A	Eng.B
1	Pull up field	Success	Success
2	Pull up method	Success	Success
3	Extract method	Success	Success
4	Extract super class	Success	Success
5	Parameterized method	Success	Success
6	Parameterized method + Pull up method	Success	Fail
7	Parameterized method + Extract method	Success	Success
8	Parameterized method + Extract super class	Success	Success

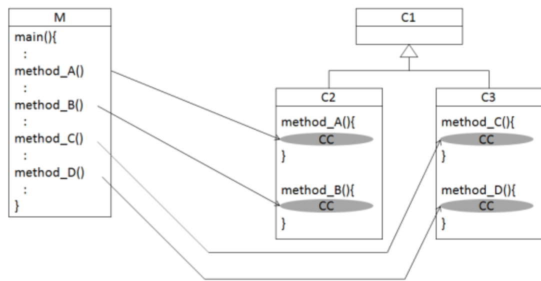

Fig.8 Program Structure of Test Case 6

```
String method_A(int a) {
  int tmp = 0;
  tmp = a%2;
  String s = null;
  if (tmp == 1) {
    s = "odd";
  }
  else {
    s = "even"
  }
  return s;
}

String method_B(int x) {
  int tmp = 0;
  tmp = x%2;
  String s = null;

// reminder equals 1.
  if (tmp == 1) {
    s = "odd";
  }
// reminder equals 0.
  else {
    s = "even"
  }
  return s;
}

String method_C(int a) {
  int tmp = 0;
  tmp = a%2;
  String n = null;
  if (tmp == 1) {
    n = "odd";
  }
  else {
    n = "even"
  }
  return n;
}

String method_D(int x) {
  int tmp = 0;
  tmp = x%2;
  String s = null;

// reminder equals 1.
  if (tmp == 1) {
    s = "odd";
  }
// reminder equals 0.
  else {
    s = "even"
  }
  return s;
}
```

Fig.9 Program List of Methods

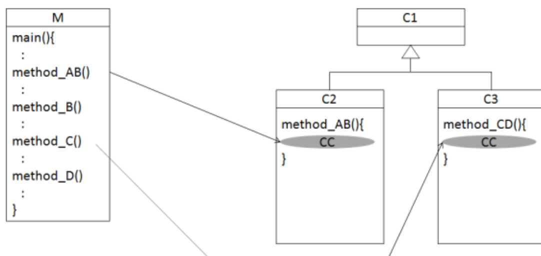

Fig.10 Refactoring Results (Engineer B)

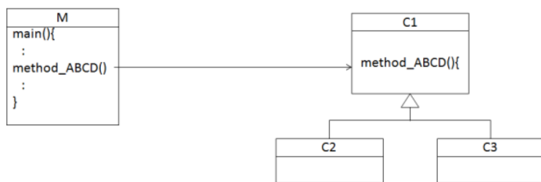

Fig. 11 Refactoring Results (Engineer A)

Here, refactoring results for test case 6 is described. Fig.8 shows program structure (class diagram) given in test case 6. This program has four classes, such as M, C1, C2, and C3. Class M has main method. And super class C1 has subclass C2 and C3. Method_A, method_B, method_C and method_D are CCs, and the name of variables in those methods and the values of constants were slightly changed. Fig.9 shows the programs of method_A, method_B, method_C and method_D. Those four methods have same operation, such as the methods divide a value of argument by 2, and return "odd" when the remainder of the division is one or return "even" when the remainder of the division is zero.

Accordingly, method_A and method_B can be aggregated as method_AB in subclass C2 using parameterized method. Likewise, method_C and method_d can be aggregated as method_CD in subclass C3. The engineer B conducted only those operations. Fig.10 shows a refactoring result conducted by the engineer B. While, merhood_AB and method_CD are aggregated to method_ABCD in super class C1 by using "pull up method." Fig.11 shows a refactoring result conducted by the engineer A.

From the above mentioned results, we could confirm the appropriate CC detection and aggregation were conducted using the proposed method and tool.

5. CONCLUSION

In this research, we proposed a program refactoring method based on CC detection and impact analysis, while producing a support tool. Applying the proposed method and the tool to the program, we confirmed that our method and tool were able to detect CCs within a program and aggregate the program by applying the proposed refactoring formats. As a result, we successfully achieved a program with a structure where the program can easily be modified based on the basic pattern of programming. Furthermore, we can expect that applying our proposed method immediately after the program has been developed and integrating CCs can enhance efficiency in subsequent program modification.

The future issues include how to judge the refactoring format to be applied where multiple refactoring formats can be applied. Moreover, we are going to make refactoring available for wide variety of programs.

REFERENCES

[1] B. Baker, "A Program for Identifying Duplicated Code", Proc. Of Computing Science and Statistics: 24th Symposium on the Interface, 24, pp.49-57, Mar.1992.

[2] E. Gamma et al., Design Patterns - Elements of Reusable Object-Oriented Software -, Addison-Wesley, 1995.

[3] M. Fowler et al., Refactoring: Improving The Design of Existing Code, Addison-Wesley, 1999.

[4] K. Inoue et al., "Identifying Refactoring Opportunities for Removing Code Clones with A Metrics-based Approach", Journal of Software Maintenance and Evolution: Research and Practice, vol.20, pp.435–461, 2008.

[5] D. Kung et al., "Class Firewall, Test Order, and Regression Testing of Object-Oriented Programs" Journal of Object-Oriented Programming, pp.51-65, 1995.

[6] Y. Jang et al., "Change Impact Analysis for A Class Hierarchy", Software Engineering Conference, pp.304-311, 1998.

[7] Y. Higo et al., "Refactoring Support Environment Based on Code Clone Analysis", Transaction of Institute of Electronics, Information, and Communication Engineers, Vol.J-88-D-I, pp.186-195, 2005.