IJCAT - International Journal of Computing and Technology, Volume 3, Issue 7, July 2016
ISSN : 2348 - 6090
www.IJCAT.org

A Proposal of Refactoring Method for Existing Program Using Code Clone Detection and Impact Analysis Method

[1] Masakazu Takahashi, [2] Yunarso Anang, [3] Reiji Nanba, [4] Naoya Uchiyama, [5] Yoshimichi Watanabe

[1,5] Dept. of Research Interdisciplinary Graduate School, Div. of Engineering, Univ. of Yamanashi
Kofu, Yamanashi 400-8511, Japan

[2,4] Dept. of Education, Integrated Graduate School of Medicine, Engineering, and Agricultural Science, Univ. of Yamanashi
Kofu, Yamanashi 400-8511, Japan

[3] Dept. of Environmental Engineering, Daiichi Institute of Technology

Abstract - This paper proposes a method that redesigns a program with an inappropriate structure into a program with appropriate structure. When a program has been using for a long period, many functional additions and modifications to the program have occurred. Those are realized by copying & pasting a part of the program and modifying it. As a result, the program becomes to have inappropriate structure and to contain many similar portions. Those similar parts make future maintenances of the program difficult. So, this paper proposes a method that clarifies similar portions in the program and integrates all similar parts into a common portion by using Code Clone detecting method. Furthermore, this paper proposes a method that the current program structure translates into the appropriate program structure by using a refactoring method when integrating code clones. Consequently, the program becomes to be able to adjust functional additions and modifications in the future. Additionally, the program can maintain for a long period.

Keywords **- Refactoring, Code Clone, Impact Analysis, Legacy System, Maintenance.**

1. Introduction

This paper proposes a method to redesign program that can be maintained easily for a long period by summarizing similar portions in the program and by rewriting the original program structure into the appropriate program structure. Many modifications to an existing program consist of addition and/or modification of functions. Hereinafter, generic term both for addition and modification is called a modification. Existing programs have many similar program portions (hereinafter, similar program portion is called as Code Clone: CC) [1] and inappropriate program structures after modifications during long period of operation, because modifications of functions are done by copying & pasting existing program parts and rewriting of existing programs.

This paper proposes a method to redesign an existing program written in Java, such as representative Object Oriented Programming Language (OOPL), without CC and with appropriate program structure by summarizing CC, redesigning and rewriting existing program. Here in after, this work is called as refactoring. Additionally, to avoid regression errors when conducting refactoring, a method to specify test scopes accompanied by refactoring is proposed. In the proposed method, for test efficiency, test scope is clarified in two steps: in the unit of members in the classes and the unit of lines within methods. By conducting those works, existing programs are rewritten to appropriate program structures without CC, rewritten programs can be modified easily and efficiently, and rewritten programs become easy to be maintained for a long period.

The organization of the rest of this paper is as follows. Chapter 2 describes the related studies. Chapter 3 describes the detection of CCs, refactoring, identification of re-verification scopes, and a prototype tool we developed.

Chapter 4 evaluates the proposed method. Finally, chapter 5 describes the future perspectives of the proposed method and tool.

2. Related Studies

This chapter describes the previous studies related to the proposed method. The previous studies are broadly categorized into those regarding CC detections, those regarding refactoring, those regarding impact analysis, and those regarding CCs included in legacy systems.

First, studies regarding CC detection are described as follows. CCs are detected in the unit of characters, expressions, or lines of program codes [2]. 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 a lot of 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 the 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 the specific characters, and then calculates the hash value of each line. Once this hash value is calculated, CCs can be detected quickly. Many tools implemented to those CC detection methods are developed as prototype tools.

For example, the CC detection tool, CCfinder [3], extracts some information related to CCs, such as lengths of CCs, position, distributions (classes and members that CCs belong.). Furthermore, the CC analysis tool, Aries [4] analyses calling relationships between methods in CC, and substitution and reference relationships of fields. This information that shows characteristics of CCs are called metrics. Representative metrics are shown as follows: the Number of Referred Variables (NRV): average number of external defined variables referred in program portions, the Number of Substituted Variables (NSV): average number of variables substituted in program portions, the Dispersion of Class Hierarchy (DCH): maximum distance between each program portions in class hierarchy, Deflation (DFL): reduced lines of code when summarizing common program portions, Length (LEN): maximum length of CC, and Population (POP): number of program portions in CC.

Next, studies regarding refactoring are described. Refactoring is a method to redesign existing program into the program with appropriate structure. Refactoring is conducted by modifying the existing program, checking it, and redesigning it in step by step. Those processes avoid falling into a state that the modified program behaves unexpectedly. Design patterns are known as the standard of the proper program design. 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 [5]. Moreover, Fowler et al. have organized representative refactoring methods (hereinafter, referred to as refactoring formats) in a catalog formats [6]. Higo et al. have proposed support methods and tools for proposing refactoring formats [7, 8, 9].

At third, studies regarding impact analysis are described. Impact analysis is a method to clarify the scope of program which might be affected when the program is modified. Kung et al. have proposed a method to clarify program classes which might be under influence of program modification based on the class firewall concept[10]. However, this method had a problem in that methods and attributes (hereinafter, generic term both for method and attribute is called as members) not modified would be included in the range under influence. Jang et al. have proposed a method to clarify methods and attributes under modification by using the Member Dependency Graph which indicates the access relationship between methods and attributes[11]. However, this method also has a problem in that a large amount of program portions dependent from influence can be included within members. As mentioned above, the existing methods were inefficient in testing program scopes under influence of program modification.

At last, studies regarding CCs in legacy systems are described. Legacy systems are systems that are difficult to maintain functions because of many changes for a long period of operation. Monden et al. investigated numbers of CCs in enterprise (backbone) systems that are developed by five software development vendors[12]. As a result, when combined tests were finished, the rate of CC portion was over 40% in the whole program. Additionally, it was shown that rate of CC portion in large scaled systems tends to be higher.

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 an appropriate program structure that can be used for a longer period of time robustly.

Figure 1 shows the proposed refactoring procedure written in Data Flow Diagrams (DFD). Here, we assume that processes in the DFD are executed sequentially along the process ID in DFD. Rectangles in Figure 1 show processes, and directed lines show data flows. Furthermore, we define that refactoring is conducted on every single pair of single type in CCs. The reason is that unexpected behaviors may be caused by a program with inappropriate modifications when all CCs are summarized, redesigned, and rewritten simultaneously. Therefore, refactoring is conducted for single pair of CCs (hereinafter, the pair is called as CC pair) in step by step, checking and investigating by engineers. As shown in DFD0 in Figure 1, refactoring process consists of four processes. As shown in DFD1 in Figure 1, the process1.1 detects all CCs from a program before modification and develops a list of CCs (hereinafter, this list is called as CC list). The process1.2 creates a diagram referred to as a Member Access Graph (MAG) which indicates the access relationship between methods and attributes before modification, and a diagram referred to as a Member Override Graph (MOG) which indicates the inheritance relationship of methods before modification. As shown in DFD2 in Figure 1, the process2.1 selects a CC pair from CC list which will be target of refactoring. The process2.2 then analyzes the content of the CC pair in order to create information necessary for determining the refactoring format (hereinafter, this information is called as refactoring information. This information is described in section 3.2). If refactoring is not required, refactoring is judged to be impossible to conduct, this step returns to the initial step, the process2.1, and selects the next CC pair. As shown in DFD3 in Figure 1, the process3.1 modifies the program based on refactoring information.

This process is done manually. When refactoring has been completed, the process3.2 creates the MAG and MOG of the modified program. As shown in DFD4 in Figure 1, the process4.1 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 modified portions differs between MAG and MOG before and after modification. Based on this result, the process4.2 tests the modified program. This test process is done manually. This completes refactoring for one CC pair. Afterward, the program is refined and elevated to an appropriate structure by repeating the above-mentioned procedure from process1 through process4. The following section describes what is actually done in each step (process) in detail.

3.1 Analysis of Program before Refactoring (process 1)

This section describes the operation of process1. This operation consists of CC detection and the creation of MAGs and MOGs. Each of the tasks is as follows:

(1) CC detection (process 1.1)

CCs are generated when a program is developed by copying & pasting, and rewriting the existing portions of the existing program. Variable names and numerical values differ in many CCs, while they are not completely matched. Therefore, a method which can detect CCs with the same program structure but slightly rewritten is required. Figure 2 shows the detection flow of CCs which have the same program structure. First, information which has nothing to do with program execution, such as blanks,

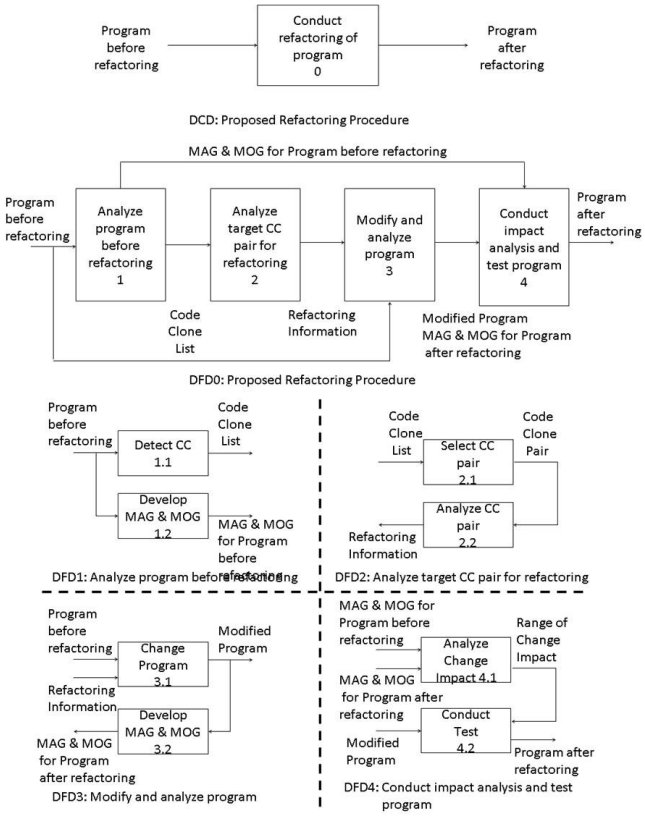

Fig.1 Proposed refactoring procedure

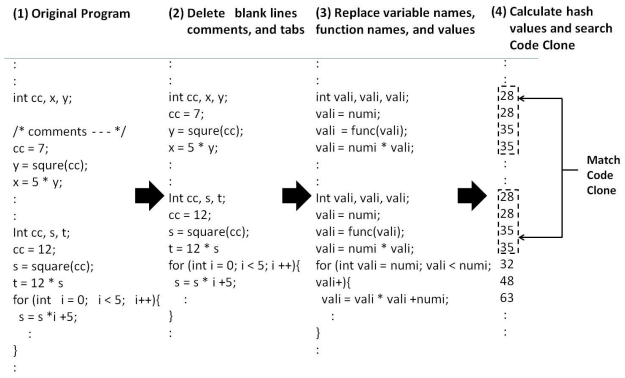

Fig.2 Flow of code clone detection

comments, line breaks, or tabs within the program is deleted. At this time, instruction that is written in plural lines for readability is rewritten into instruction that is written in single line. 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 the ACII codes of each character, adding them, and then dividing it with the value of *m*. The value of *m* is the index size of hash table (*m* shows variation that hash function can classify). Finally, portions, where the hash value of each line matches more than *n*, are collected and listed into a CC list. The value of *n* shows minimum lines that we consider the similar program portion to be a CC. The value of *m* and *n* is changeable as necessary. In this research, we settled that the value of *m* is *50* and the value of *n* is *30* based on our experiences.

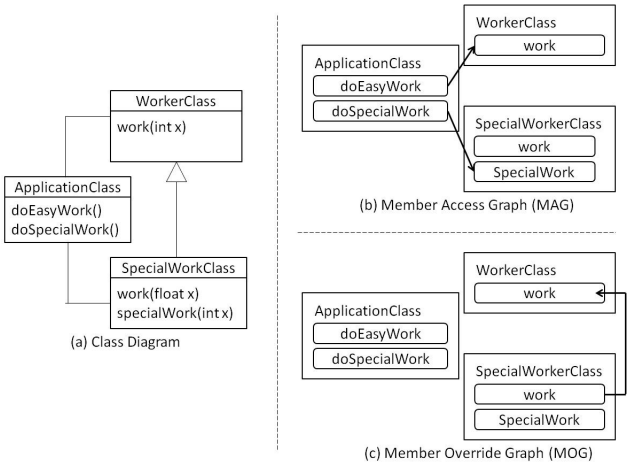

Fig.3 Sample of MAG and MOG

(2) Creation of MAG and MOG (process1.2)

Programs written in Java execute their services based on combinations of method calls between objects and references of attributes. This research is using MAG and MOG to represent those services. MAG graphically indicates the relationship between method calls and references of attributes. MAG expresses the relationship of method calls in the directed line from the calling source (caller) to the calling destination (callee), while expressing the attribute reference relationship in the directed line from the reference source to the reference destination. MOG graphically indicates overridden methods accompanied by class inheritance, implementation of methods which are defined abstractly, and attribute encapsulation. MOG expresses overridden methods in the directed line from the method to override to method to be overridden.

Figure 3 shows the corresponding relationship between MAG and MOG. The program shown in Figure 3 (a) has the following classes: *WorkerClass*, *SpecialWorkerClass*, and *ApplicationClass*. In addition, the *WorkerClass* has *work(int x)* method, the *SpecialWorkerClass* has *specialWork(int x)* method and *work(float x)* method, and the *ApplicationClass* has *doEasyWork()* method and *doSpecialWork()* method. The *doEasyWork()* in the *ApplicationClass* calls the *work(int x)* in the *WorkerClass*, and the *doSpecialWork()* in the *ApplicationClass* calls the *specialWork(int x)* in the *SpecialWorkerClass*. Therefore, MAG is developed by drawing directed line from the *doEasyWork()* in the *ApplicationClass* to the *work(int x)* in the *WorkerClass*, and directed line from the *doSpecialWork()* in the *ApplicationClass* to the *spcialWork(int x)* in the *SpecialWorkerClass*. Figure 3 (b) shows MAG. Rectangles with bold lines show classes, rectangles with narrow lines show methods. Next, the work(int x) in the *SpecialWorkerClass* overrides the *worker(int x)* of the *WorkerClass*. Figure 3 (c) shows MOG. Therefore, MOG is developed by drawing directed line from the *work(float x)* in the *SpcialWorkerClass* to the *work(int x)* in the *WorkerClass*. The meanings of rectangles in the MOG are same as meanings in MAG.

3.2 Analysis of Target CC Pair for Refactoring (Process 2)

This section describes the operation of process2. This operation consists of the determination of a CC pair which conducts refactoring (process2.1), and the determination of refactoring formats(process2.2). A CC pair is determined simply by choosing a CC pair from CC list. Therefore, the

determination of refactoring formats is described below (process2.2).

Refactoring formats are determined based on program before modification and refactoring information which is obtained by analyzing the program portions of CC pairs under refactoring. Refactoring information consists of the start line of the CC, the end line of the CC, the total number of CC lines, the number of external variables used, the number of called methods, the parent class, and refactoring formats. The start and end lines of the CC are obtained from the CC list. The types and number of external variables, the types and number of called methods, and the parent classes are obtained from MAGs and MOGs. The reason that those values are included in refactoring information is because it provides a better way to obtain effects of refactoring in the case that a program portions have single structural cohesion. The structural cohesion means that program portion contains whole instruction blocks, such as for(){}, while(){}, and switch(){}. Furthermore, if the combination of the program portion and the program portions surrounding it is weak, it will be easier to obtain effects of refactoring. The weak combination means that there are less assignments and references to filed variables, and less calling of methods. When CC has those characteristics, the harmful effects of summarizing CCs are less, and refactoring will be easy to conduct.

Table 1 indicates the refactoring formats treated in this research and the judging criteria for applying these formats.

Additionally, the proposed method indicates refactoring method based on the decided refactoring format. Because appropriate refactoring operation has already been decided according to the CC format, the proposed method indicates refactoring method by showing both current and modified program structure written in Class diagrams.

Table1 list of refactoring format and judging criteria

Refactoring format	**Refactoring procedure**	**Judging criteria**
Extract Method	Integrates those operations (CCs) overlapping within the same class by creating a new method.	CCs exist within the same class. A few external variables exist. A few calling methods exist.
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.	CCs do not exist within the same class. CCs have the same super class. CCs within the subclass exist.
Extract Class	Integrates multiple operations (CCs) overlapping within the same class by creating a new class.	CCs exist within the same class. Plural CCs exist.
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.	CCs do not exist within the same class. CCs do not have the same super class. Plural types of CCs exist within the different 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. Different types of external variables are used in the CCs. Different types of method are used in the CCs Different values are used within CCs.
Pull Up Field	Integrates multiple subclasses with the same super class having the same attributes by giving the super class attributes.	CCs have the same super class. CCs have the overlapping attributes within each subclass.

3.3 Modification and Analysis Program (Process 3)

Process3 modifies the program based on refactoring information and creates MAG and MOG of program after modification. 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).

3.4 Implementation Impact Analysis and Tests (Process 4)

This section described the operation process4. 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 members in the classes and in the unit of lines within methods. 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 in the classes, while the second stage clarifies a program scope under the impact of program modification in the unit of lines within 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 appear and disappear 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 disappear due to program modification is then extracted based on comparison of MOGs before and after program modification. These differences become members which are under impact of program modification.

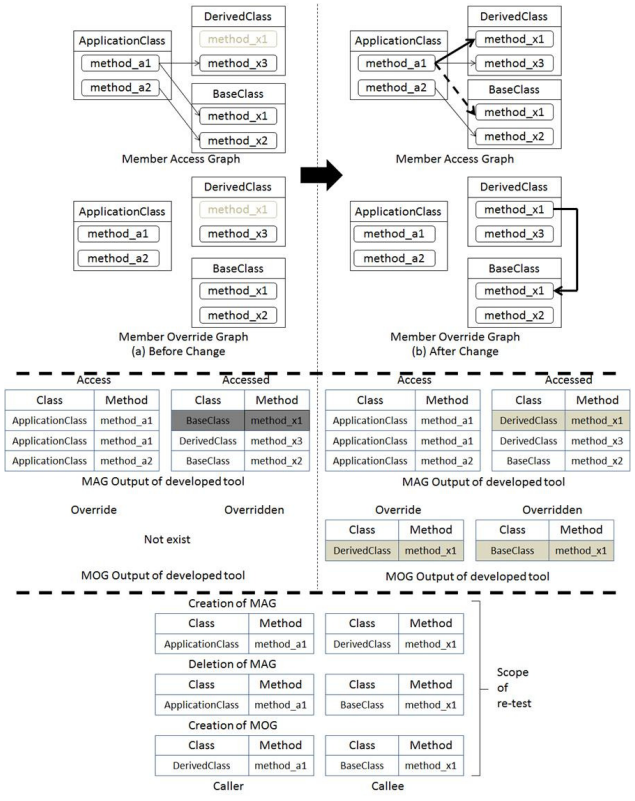

Fig.4 Sample result of impact analysis using MAG and MOG

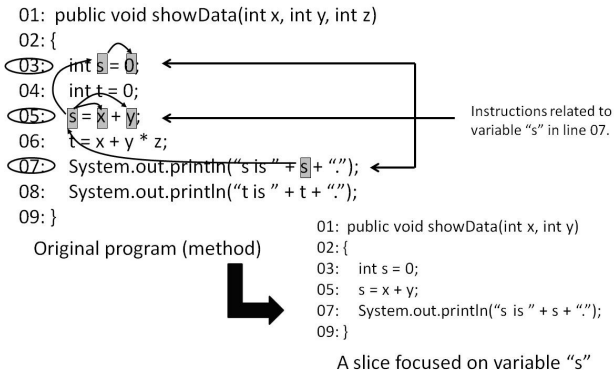

Fig.5 Procedure of static slicing

Figure 4 shows a proposed process specifying impact scopes from MAGs and MOGs before and after modification. Left side in Figure 4(a) shows MAG and MOG before modification. Targeted program has three classes, such as *ApplicationClass*, *BaseClass*, and *DerivedClass*. Before modification, the ApplicationClass has method_a1 and method_a2, the BaseClass has method_x1 and *method_x2*, and the *DerivedClass* has *method_x3*. The *method_a1* in the *ApplicationClass* accesses the *method_x1* in the *BaseClass* and the *method_x3* in the *DerivedClasss*, and the *method_a2* in the *ApplicationClass* accesses the *method_x2* in the *BaseClass*. There is no override of methods (MAG does not have any directed line). After modification, the *method_x1* is added to *the DerivedClass*, and the *method_x1* in the *DerivedClass* overrides the *method_x1* in the *BaseClass*. As a result, the method_a1 in the *ApplicationClass* accesses both the *method_x1* and the *method_x3* in the *DerivedClass*. Upper right side in Figure 4(a) shows MAGs after modification, and lower right side in Figure 4(a) shows MOGs after modification. Here, bold directed line in the figure shows newly created access, and bold dotted direct line in the figure shows disappeared access. When using MAGs/ MOGs developing tool described in section3.5, left side of Figure 4(b) shows MAGs and MOGs before modification, and right side of Figure 4(b) shows MAG and MOG after modification. By comparing MAGs and MOGs before and after modification, scopes of change impact are clarified. In this case, deep gray cell shows a disappeared access after modification, and light gray cells show newly appeared access relationship and newly appeared override relationships after modification. Figure 4(c) shows change impact scopes based on the differences of access and override relationships before and after modification. These cells are dark and light gray color cells that are shown in MAGs and MOGs before and after modification. In this case, there is no deletion of MAG. Normally this information contains creation/deletion of MAGs and MOGs.

The second stage, sentence-based impact analysis, clarifies those lines of the program code under impact by using static program slicing for the affected portions extracted in the first stage. Ripples of influence between methods are propagated through arguments. Influence of change impact is considered two types: the case that the arguments influence the called method, and the case of arguments influence callee method. In the former case, change impact scopes can be obtained by extracting following lines: lines that use arguments, lines that assign result calculated using arguments to new variables, lines that use assigned variables, and lines that assign result calculated using assigned variables to new variables. In the latter case, change impact scopes can be obtained by extracting lines according to reverse sequence of program execution. This executing process is called static program slicing. 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. When any input data is given to static slices, the same calculation result as the original program is obtained for the variables focused on. Figure 5 shows an example of static slice extraction procedure. The numbers listed on the left side of Figure 5 indicate the number of program lines. First, the variable *s* in the 07th line of the original program is focused on, and this variable *s* in the 07th line is calculated by using the variable *x* and *y* in the 05th line. The variable *s* is initialized in the 03rd line. While, the variable *x* and *y* in the 05th line is passed as arguments of *showData* method in the 01st line. Additionally, making those sliced lines executable, *{* in 02nd line and *}* in 09th line are added to those sliced lines. As shown in right side of Figure 5, the static slices focused on *s* in 07th line are the lines *{01, 02, 03, 05, 07, 09}*. This static slice can calculate the value of *s*, when *x* and *y* are given.

3.5 Creation of a Refactoring Support Tool Prototype

This section describes a prototype that supports the operation of each process in Figure 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 process1.1 in Figure 1. The input of this tool is a program, while the output is a CC list. Figure 6 shows an example of a CC list output by the CC detection tool. Though the output format of CC list is Comma-Separated Value (hereinafter, CSV) type, Figure 6 shows CC list as table format in considering readability of contents. The CC list contains the paths of all files where CCs exist, the start/end lines of CCs of all CCs. The MAG/MOG creation tool creates the MAG and MOG of a program. This tool is used by process1.1 and process3.2 in Figure 1. The input of this tool is a program, while the output is MAG and MOG. MAG contains classes and members to which the members of the call source belong (caller), and classes and methods to which the members of the call destination belong (callee). MOG contains overriding methods and methods to be overridden. Figure 7 shows an example of MAG and MOG output by the MAG/MOG creation tool. Though the output format of MAG/MOG is CSV type, Figure 7 shows MAG and MOG as table format in considering readability of contents. The refactoring format proposal tool creates refactoring information of selected CC pair. This tool is used by process2.2 in Figure 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 information. Refactoring information contains the start line of CC, the end line of CC, the total number of CC lines, the number of externally defined variables used, the number of method calls, the parent class, and refactoring formats.

Figure 8 indicates refactoring information as the output of the refactoring format proposal tool. Though the refactoring information is CSV type, Figure 8 shows the refactoring information as table type format in consideration for readability of contents. Additionally, Figure 9 shows a sample screen of the proposal for refactoring format of the cc pair. The left side of this screen shows a program structure (class diagrams) before refactoring, and the right side of this screen shows a program structure (class diagrams) after refactoring. The input of this tool is a refactoring format, while the output is the refactoring order sheet. The refactoring order sheet shows how to modify the program, and this is used when engineers modify the program. 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 MAGs and MOGs before and after program modification. This tool is used by process4.1 in Figure 1. The input of this tool is a modified program and MAGs/ MOGs before and after modification, while the output is the impact scopes. The impact scopes contain appeared/disappeared accesses and override relationships. Figure 10 shows a sample output of impact scope output by the impact analysis tool. Though the output format of impact scope is CSV type, Figure 10 shows impact scope as table format in considering readability of contents.

4. Evaluation of the Proposed Method and Tool Conclusions

This chapter describes the results of evaluation for the proposed method and the developed tool. In order to evaluate them, we conducted refactoring by inputting two programs to

the tool. One was the tool described in section 3.5 (which was just completed with refactoring not conducted), and the other was an open-source build tool, Ant (Ver. 1.9.4, where it had been used for a long period). Before applying the method, the developed tool has 5927 lines (hereinafter, LOC) and 24 classes, while Ant has 18026 LOC and 51 classes.

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 Sample of CC list

MAG			
CALL SOURCE		CALL DESTINATION	
CLASS	METHOD	CLASS	METHOD
ApplicationClass	method_a1	DerivedClass	method_x1
ApplicationClass	method_a1	DerivedClass	method_x3
ApplicationClass	method_d2	BaseClass	method_x2
MOG			
OVERRIDE SOURCE		OVERRIDE DESTINATION	
CLASS	METHOD	CLASS	METHOD
DerivedClass	method_x1	BaseClass	method_x1

Fig.7 Sample of MAG and MOG output

c:¥testCode¥c2.java	
Start Line	13
End Line	34
Total Net CC Lines	22
Number of Externally Defined Variables	0
Number of Method Calls	0
Parent Class	c1
Start Line	45
End Line	69
Total Net CC Lines	22
Number of Externally Defined Variables	0
Number of Method Calleds	0
Parent Class	c1
Applicable Refactoring Formats	Extarct Method
	Parameterized Method

Fig.8 Sample of refactoring information

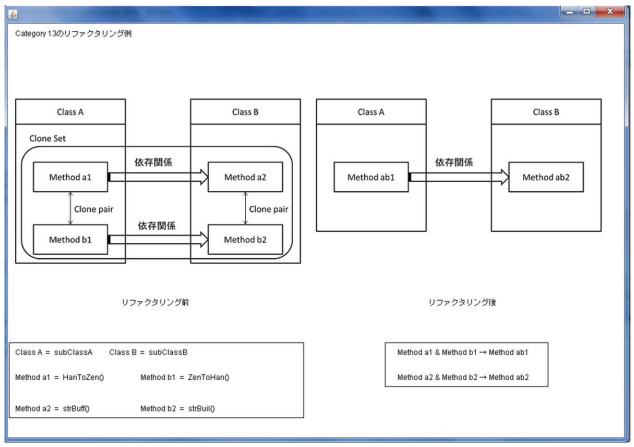

Fig.9 Sample screen of refactoring order sheet

CREATION OF MAG			
CALL SORCE		CALL DESTINATION	
CLASS	METHOD	CLASS	METHOD
ApplicationClass	method_a1	DerivedClass	method_x1
DELETION OF MAG			
CALL SORCE		CALL DESTINATION	
CLASS	METHOD	CLASS	METHOD
ApplicationClass	method_a1	BaseClass	method_x1
CREATION OF MOG			
OVERRIDE SOURCE		OVERRIDE DESTINATION	
CLASS	METHOD	CLASS	METHOD
DerivedClass	method_a1	BaseClass	method_x1
DELETION OF MOG			
OVERRIDE SOURCE		OVERRIDE DESTINATION	
CLASS	METHOD	CLASS	METHOD

Fig.10 Sample of impact scope output

Table 2: Application result of the tool prototype

Result	Prototype	Ant
LOC before change	5927	18026
Detected CCs	42	32
Integrated CCs	23	19
LOC after Change	4994	17619
Reduced LOC	933	407

Table 3: Breakdown of proposed refactoring formats

Refactoring format	Prototype	Ant
Pull up method	1	3
Extract method	14	16
Extract class	1	0
Parameterized method	13	16
Cannot Integrate	8	13

Table 2 shows the result of applying the proposed method, and Table 3 shows the breakdown for detected CCs and

applied refactoring formats. LOC in Table 2 indicates the number of program lines. The total sum of refactoring formats per program in Table 3 does not correspond to the number of detected CCs that is indicated in Table 2. This is because multiple types of refactoring formats which are applicable to one CC were proposed by the tool.

The results of Table2 are explained in the following. At first, the developed tool had 42 CCs when the tool was developed immediately after, the tool proposed that 23 CCs of them are judged to be able to conduct refactoring. 19 CCs that could not be conducted refactoring had following reasons: CCs did not exist in the same class, and CCs did not have same super class. In those cases, if summarizing CCs, the coupling increased too much. As a result of engineer's investigation for refactoring formats recommended by the tool, 15 CCs could conduct refactoring. The representative reasons that CCs did not conduct refactoring in spite of the proposals for refactoring are as follows: CC was a part of "if then else" instruction block et al. (2 cases), and CC used many external variables (6 cases). As a result of conducting refactoring, LOC of the tool became 4994 LOC, and 944 LOC was reduced. This accounts for 16% of the entire program. The rate of LOC reduction due to CC integration was large. This is because no refactoring had not been conducted for refining the program structure yet. On the other hand, ANT had 32 CCs with adequate operation experience. The tool prototype proposed that 30 CCs of them are judged to be able to conduct refactoring. 2 CCs that could not be conducted refactoring had following reasons: CCs did not exist in the same class, CCs did not have same super class. As a result of engineer's investigation for refactoring formats recommended by the tool, 19 CCs could conduct refactoring. The representative reasons that CCs did not conduct refactoring in spite of the proposals for refactoring are as follows: CC was a part of "if then else" block et al. (9 cases), and CC used many external variables (4 cases). As a result of conducting refactoring, LOC of ANT became 17619 LOC, and 407 LOC was reduced. This accounts for 2% of the entire program. The rate of LOC reduction due to CC integration was small, because the program structure was properly elevated to a certain degree by use and changes that had been made in the past.

The results of Table3 are explained in the following. As for the developed tool, the breakdown for refactoring formats includes one simple application of the Pull UP Method, one simple application of the Extract Method, one simple application of the Extract Class, and 13 simultaneous applications both of the Extract Method and the Parameterized Method. When it comes to ANT, the breakdown for refactoring formats includes three simple applications of the Pull UP Method, and sixteen simultaneous applications both of the Extract Method and the Parameterized Method. In both of the tool and ANT, the Extract Method and the Parameterized Method were more used as the refactoring formats. This result was considered that there were many CCs that were created by copy & paste of original program, and modifying the constant values in CCs. As for "Cannot Integrate" in Table 3, one CC was actually part of the "if then else" command. Other CCs were that external variables were frequently used and methods were often called within the CC. Therefore, CC integration was hard to implement only by changing and increasing arguments. We made plural programmers examine the above refactoring results, and they confirmed that all refactoring processes were properly implemented.

Judging from the above described results, we were able to confirm that application of the proposed method and the tool that we developed could make it possible to *refactor* a program with a proper program structure by integrating CCs. Moreover, it is found that only few CCs can be integrated for a program which is being used for a long period. As for such programs which are planned to be used for a longer period, therefore, CCs need to be integrated in the appropriate timing, and it is necessary to redesign an appropriate program structure that is easy and efficient to maintain and add functions.

5. Conclusion

This paper proposed a method that summarizes CCs in the existing program written in Java and redesigns the existing program into the program with appropriate program structure using CC detection and impact analysis method. Additionally, our prototype that supports conducting the proposed method was developed. As a result of applying the proposed method and the tool, we found that CCs in the existing programs were summarized and redesigned to appropriate program structure. If the proposed refactoring method is applied to the existing program, they will be able to be achieved to increase adequacy and efficiency of modifying the existing program.

The future issues include how to judge the refactoring format to be applied where multiple refactoring formats can be applied, the support for modifications of a program where the proposed refactoring format is applied, and the testing the modified program. When the structure (design) of the program is too much inappropriate, we cannot obtain enough effects because inappropriate program structure requires a lot of costs for redesigning and rewriting of the program. The examples of inappropriate program structure are as follows: too less classes (inappropriate class structure), too large method (it is difficult to identify impact scope), and too much

method calls and references of external variables et al. We will also consider influence when conducting refactoring for modules referred externally. Furthermore, we also consider the expansion of proposed method for applying the embedded program domain. The proposed method is useful for applying to enterprise programs, while it is difficult to apply to embedded programs. For example, enterprise program means accounting, allowance, and material resource planning programs, while embedded program means machinery control program. The reasons are as follows: generally, embedded program uses many global variables (external defined variables), and real-time operations that are executed by interrupts have to be considered. Additionally, based on the viewpoint of operation streamlining, we consider automation or semi-automation of these processes in order to fulfill our support tool. Moreover, by increasing applicable refactoring formats and giving feedbacks of the application results to the tools, we are going to try to make refactoring available for a wide variety of programs.

References

[1] Baker, B.S, A Program for Identifying Duplicated Code, Proc. of Computing Science and Statistics: 24th Symposium on the Interface, No. 25, 1992, pp.49-57.

[2] Baker, B.S, On Finding Duplication and Near-Duplication in Large Software System, Proc. of the Second IEEE Working Conf. on Reverse Engineering, 1995, pp.86-95.

[3] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue, CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code, IEEE Transactions on Software Engineering, Vol. 28, No. 7, 2002, pp. 654- 670.

[4] Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue, Aries: Refactoring Support Tool for Code Clone, Proceedings of 3rd Workshop of Software Quality, 2005, pp.53-56.

[5] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Elements of Reusable Object-Oriented Software. Addison-Wesley Publishing Company, 1995.

[6] Martin Fowler, Kent Beck, John Brant, William Opdyke, don Roberts, Refactoring: Improving the Design of Existing Code, Addison Wesley Longman Inc1999.

[7] Yoshiki Higo, Yasushi Ueda, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue, On Software Maintenance Process Improvement Based on Code Clone Analysis, Journal of Information Processing Society of Japan, vol. 45, No.5, 2004, pp. 1357-1366.

[8] Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, Katsuhiro Inoue, Refactoring Support Environment Based on Code Clone Analysis, IEIC Transaction, vol.J88-D-I, No.2, 2005, pp.186-195.

[9] Norihito Yoshida, Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoti and Katsuro Inoue, On Refactoring Support Based on Code Clone Dependency Relation, Journal of Information Processing Society of Japan, vol.3, no.48, 2007, pp.1431-1442.

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

[11] Yoon K. Jang, Heung S. Chae, Yong R. Kwon, and Doo H. Bae, Change Impact Analysis for a Class Hierarchy, Proc. of Software Eng. Conf., 1998, pp.304-311.

[12] Monden A., Saito S. and Matsumoto K., capturing industrial experiences of software maintenance using product metrics, Proc.of 5th World Multi-Conf. on Systems, Cybernetics and Informatics, Vol.2, 2001, pp.394-399.

Masakazu Takahashi received B.S. degree in 1988 from Rikkyo University, Japan, and M.S. degree in 1998, Ph.D. degree in 2001, both in Systems Management from University of Tsukuba, Japan. He was with Ishikawajima-Harima Heavy Industries Co., Ltd. from 1988 to 2004. He was with Shimane University from 2005 to 2008. He is a professor in University of Yamanashi, Japan since 2014. His research interests include software engineering and safety.

Yunarso Anang received B.E. degree in 1995 and M.E. degree in 1997 both in software engineering from University of Yamanashi, Japan. He was with SYNC Information System, Inc., Japan from 2000 to 2007, as a senior system engineer. From 2008, he is with Institute of Statistics (Sekolah Tinggi Ilmu Statistik), Jakarta, Indonesia, as a lecturer in computer science. From 2014, he is a Ph.D. student at University of Yamanashi, Japan. His research interests include software engineering.

Reiji Nanba received B.E. degree in 1999 from Daiichi Institute of Technology, Japan. ME degree in 2003 and Ph.D. degree in 2008, Shimane University, Japan. He was an Assistant Professor in Daiichi Institute of Technology in 2008. He is an associate professor till now in Daiichi Institute of Technology in 2011. His research field is mainly in Civil and Environmental Engineering and Engineering Education.

Naoya Uchiyama received B.E. degree in 2012 and M.E. degree in 2014 both in University of Yamanashi, Japan. He is with TOKAI group. His research field is network security.

Yoshimichi Watanabe received the B.S. and M.S. degrees in computer science from University of Yamanashi, Japan in 1986 and 1988 respectively and received D.S. degree in computer science from Tokyo Institute of Technology, Japan in 1995. He is presently an associate professor of the Department of Computer Science and Engineering at University of Yamanashi. His research interests include software development environment and software quality.