IJCAT - International Journal of Computing and Technology, Volume 5, Issue 1, January 2018
ISSN (Online) : 2348-6090
www.IJCAT.org

An Efficient Merging Method for Code Clones and Gapped Code Clones Using Software Metrics

[1] Masakazu Takahashi; [2] Yunarso Anang; [3] Reiji Nanba; [4] Yoshimichi Watanabe

[1,4] Graduate School Dept. Div. of Engineering, , University of Yamanashi
4-3-11 Takeda, Kofu, Yamanashi, 400-8511, Japan

[2] Dept. of Computational Statistics, ,Institute of Statistics
JI. Otto Iskandardinata No. 64C, Jakarta Timur, 13330, Indonesia

[3] Dept. of Environmental Engineering, Daiichi Institute of Technology
1-10-2 Kokubu-Chuo, Kirishima, Kagoshima, 899-4395, Japan

Abstract - A program fragment that is created by copying & pasting an existing program is called Code Clone (CC). A program fragment that some instructions are added, deleted, and modified is called Gapped Code Clones (GCC). In general, many CCs and GCCs that exist in the program decrease readability and maintainability of the program. This study proposes an effective detection method of CCs and GCCs that are suitable for merging by calculating software metrics related to the complexity of the control flow, independent from other program portions, and non-dependency of programming language specification. Additionally, this study recommends the merging procedure of CCs and GCCs by calculating software metrics related to the program structure of CCs and GCCs. As a result of the application of the proposed method to the existing programs, it is confirmed that the adequate merging of CCs and GCCs is conducted.

Keywords - *Code Clone, Gapped Code Clone, Merging, Refactoring, Maintainability, Readability*

1. Introduction

When a function is added to a certain program, we occasionally copy and paste part of the program by conducting minor changes to the variables and constants so as to achieve the function. This results in existing many similar fragments within the program. Such program fragments are referred to as code clones (hereinafter referred to as CCs). [1] Additionally, for some CCs, instructions are added, deleted, or modified. These CCs that have differences are referred to as gapped code clones (hereinafter referred to as GCCs). [2] In the following sections, CCs and GCCs are described as CCs/GCCs.

CCs/GCCs that exist in the program may reduce software maintainability. Therefore, software engineers have considered that they want to merge CCs/GCCs into one location. However, it has been clarified that there exist different types of CCs/GCCs, such as those that are never modified and those with less readability and less effectiveness in spite of a required cost when merging. Hence, a method for selecting CCs/GCCs that are expected to improve readability and maintainability by the merging and for proposing CCs/GCCs merging type and procedure are desired. Through this paper, focusing on programs developed by using Java, we propose a method for detecting and for merging functionally coherent CCs/GCCs that are independent of other program portions. This method efficiently detects and merges CCs/GCCs that are effective in merging. There are several units for detecting CCs/GCCs, such as characters, tokens (a group of characters separated with spaces or semicolons), lines, blocks (a group of code lines such as if blocks or methods), and other similar items.

In this paper, we attempt to detect CCs/GCCs in units of code lines or blocks so that they are designed as a method or a class that is commonly used within the program. To detect CCs/GCCs, we apply the Smith-Waterman algorithm (hereinafter referred to as SWA) which can detect similar portions within a character string. Then the proposed method calculates the software metrics (hereinafter referred to simply as metrics) for the functionality of the CCs/GCCs and determines the CCs/GCCs that should be merged. Finally, the proposed method proposes the proper CCs/GCCs merging type and procedure by calculating the metrics that are associated with the program structure of CCs/GCCs. We apply the proposed method to the existing programs to evaluate

whether CCs/GCCs are adequately and efficiently merged or not by the proposed method.

2. Related Studies for Detecting and Merging CCs/GCCs

Related studies can be broadly classified into those related to CCs/GCCs detection, those related to the characteristics of CCs, those related to software metrics, and those related to refactoring.

This section describes studies related to CCs/GCCs detection. The studies related to CC detection can roughly be categorized into the studies on detection methods in the program unit, and on detection methods focusing on the program structure. The LCS algorithm is a representative algorithm for detecting CCs in the program unit. [3] The LCS-based CC detection method detects CCs by obtaining the longest common subsequence for the units of the functions, methods, and blocks which are included in the source code. [4] The CC detection method that focuses on the program structure analyzes the program and expresses the program structure by using a syntactic tree or dependence graphs in order to detect similar structures. By doing so, this method detects CCs. [5, 6] Murakami proposed a method for detecting GCCs within the program in a token unit by using the SWA [7, 8] which detects similar fragments from two character strings. [9]

This section describes studies related to the characteristics of CCs. At first, it was considered that CC should be merged. However, Higo et. al indicated the issue that some CCs remained unmodified when certain CCs were modified. [10] They made it possible to detect such unmodified CCs by checking the conflicts in the variables existing in the CCs. Gode et. al analyzed the frequencies and risks of changing CCs. They indicated that approximately 50% of CCs were unchanged, and approximately 10% of CCs were changed more than twice. They also indicated when the CCs were changed twice or more, they have a negative effect due to careless mismatches. [11]

This section describes studies related to the program metrics. Fenton reported a wide variety of metrics that could be used for software development. [12] Furthermore, Hatano et al. proposed a method for clarifying the program structure by using C & K metrics for programs written by the object-oriented programming language, so as to rewrite the program structure into a more proper structure based on the program structural values. [13]
This section describes studies related to refactoring. Refactoring is a technique to rewrite the existing program into a new program with an adequate program structure without making any changes in the existing program's functions. Fowler proposed a representative refactoring method. [14] Higo et al. developed a tool called Aries that selects CCs suitable for refactoring within the program. [15]

3. Selecting CCs/GCCs and Proposing a Merging Method

The proposed method extracts CCs/GCCs from the target program and calculates their metrics. Based on these values, the proposed method selects CCs/GCCs that can be merged and proposes an efficient merging method. First, section 3.1 explains the outline of the proposed method. Section 3.2 explains individual techniques that compose the proposed method. Section 3.3 describes support tools.

3.1 Outline of the Proposed Method

Fig.1 shows the outline of the proposed method. The proposed method consists of five steps. STEP 1 detects CCs/GCCs that exist within the target program. STEP 2 calculates the metrics of each CC/GCC. STEP 3 selects the candidates of CCs/GCCs that can be merged effectively by using calculated metrics. STEP 4 calculates the metrics related to the program structures of these candidates and proposes a merging method based on the calculated metrics. STEP 5 merges CCs/GCCs in accordance with the proposed merging type and procedure.

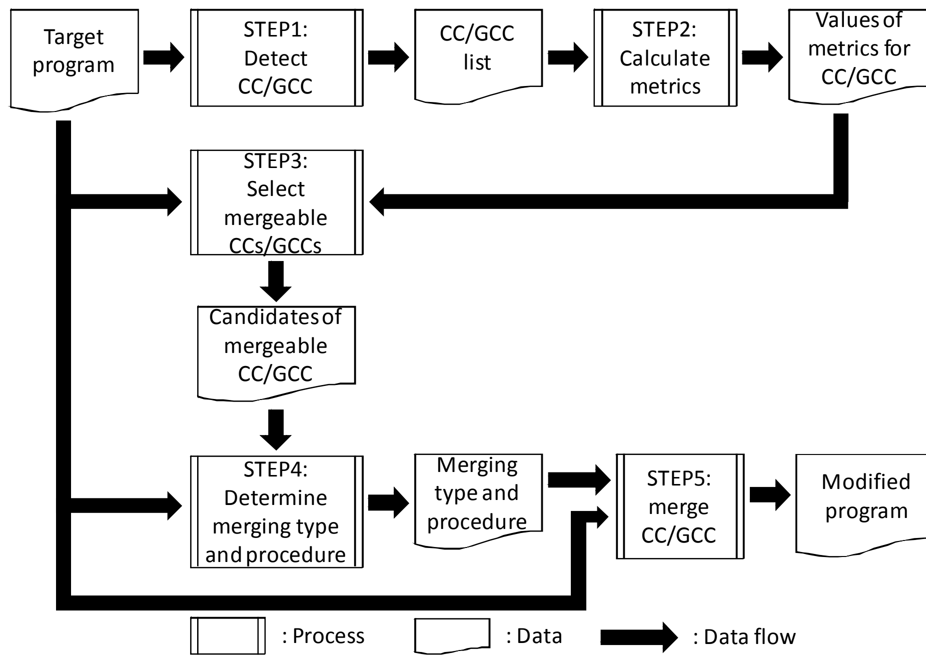

Fig. 1 Outline of the Proposed Method

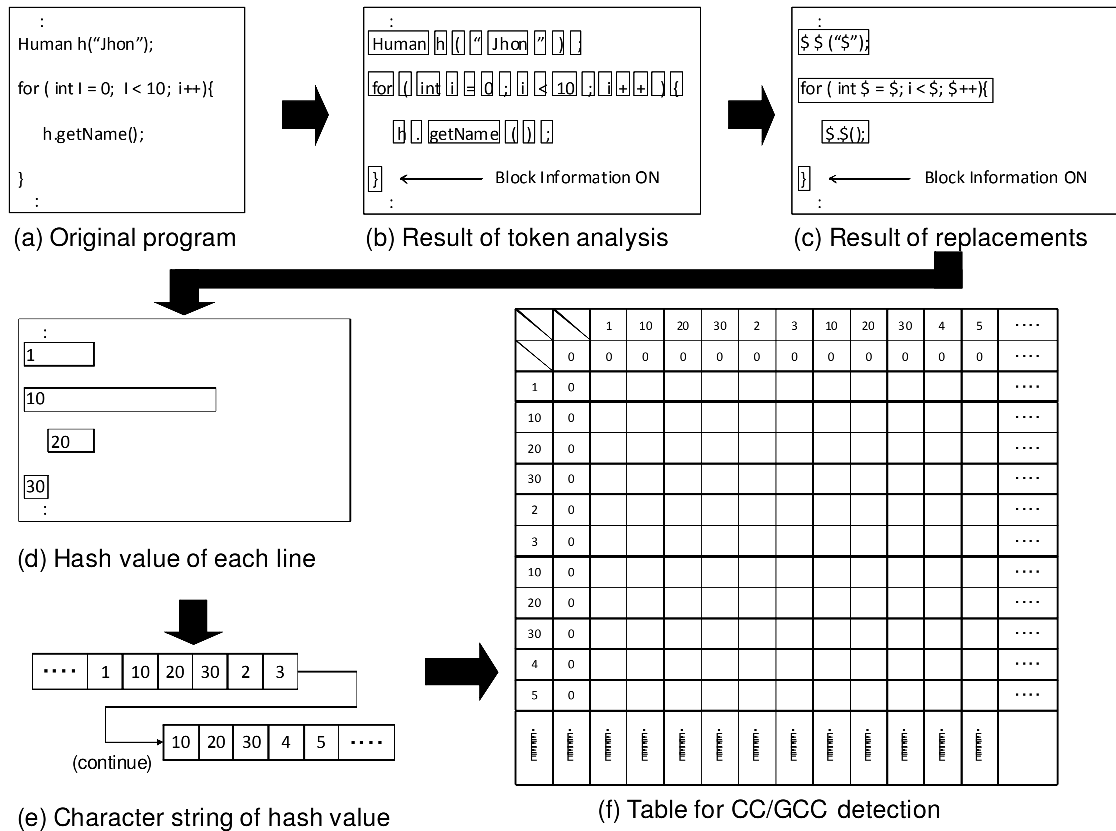

Fig.2 Creation of SWA Table for Detecting CCs/GCCs

3.2.1 Detect CCs/GCCs (STEP1)

This section describes STEP 1 shown in Fig.1. By using the SWA, STEP 1 detects CCs/GCCs that exist within the target program. The SWA is an algorithm that detects similar partial character strings within character strings. The proposed method calculates the hash value in each code line of the target program and creates a string that consists of a sequence of hash values. Then by detecting matched portions in the sequence of hash values, the proposed method detects CCs and GCCs. The SWA uses parameters of {match, mismatch, gap}. Varying these parameters can change the size of the tolerable gap. Here, "match" indicates the weight when compared characters within the string match, "mismatch" indicates the weight when compared characters do not match, and "gap" indicates the weight when a character is inserted into the matched partial character strings.

The following describes how to detect CCs/GCCs by using the SWA. Here, the value for each of match, mismatch, and gap is set as 1, -1, and -1, respectively.

(1) Create a table for CCs/GCCs detection

(a) By analyzing the program of Fig.2 (a), identify tokens as shown in Fig.2 (b).

(b) As shown in Fig.2 (c), replace the variable names, values, function names, sub-routine names, class names, and method names with special characters, for example "$". This replacement absorbs differences accompanied with lesser modifications.

(c) As shown in Fig.2 (d), identify portions (code lines) separated with "{ }" and ";" in order to hash each code line.

(d) As shown in Fig.2 (e), create strings while setting the hash of each code line to one character. Here, the number of characters (the number of program code lines) is expressed as m.

(e) Create a two-dimensional table with m+2 rows and m+2 columns. Thereafter, each cell of row i and column j is expressed as c(i, j), while the value of c(i, j) is expressed as $v_{i,j}$.

(f) As shown in Fig.2 (f), store string elements from $v_{1,3}$ to $v_{1,m+2}$ in sequence. Similarly, store string elements (hash values) from $v_{3,1}$ to $v_{m+2,1}$. Afterward, enter 0 (zero) from $v_{2,2}$, $v_{2,3}$ to $v_{2,m+2}$, and from $v_{3,2}$ to $v_{m+2,2}$.

(2) Calculate the cells of the table

(a) As shown in Fig.3 (a), calculate and set $v_{i,j}$ by using the equations below. Here, the diagonal cells absolutely match, while the upper-right and the lower left elements become same values (symmetric matrix). Therefore, there is no need to calculate the upper-right elements.

$$v_{i,j}\ (2 \le i, 2 \le j) = \max \begin{cases} v_{i-1,\ j-1} + s(a_i, b_j), \\ v_{i-1,\ j} + gap, \\ v_{i,\ j-1} + gap, \\ 0. \end{cases} \tag{1}$$

$$s(a_i, b_j) = \begin{cases} match\ (a_i = b_j), \\ mismatch\ (a_i \ne b_j). \end{cases} \tag{2}$$

(b) As shown in Fig.3 (b), when the $v_{i,j}$ calculated above is not 0, set a tracking pointer that refers to the calculated c(i, j) from the cell used for cell value calculation (c(i-1, j-1), c(i-1, j), or c(i, j-1)).

(c) Calculate all the cells of the table.

(3) Detect CCs/GCCs

(a) As shown in Fig.3 (c), beginning at the cell where $v_{i,j}$ is maximized within the table, trace back the cells until the pointer's value becomes 0, while recording the tracing path.

(b) The character strings that are created by connecting the values of the cell in the 1st row along the recorded path and connecting the values of the cell in the 1st column along the recorded path become a pair of GCC. In the case of Fig.3 (c), similar strings are "10-20-30" and "10-20-4-30." (The underlined character becomes a GAP. The for-loop of Fig.2 (a) is a GCC.) In the case that the value of $v_{i,j}$ does not decrement even once when the pointer traces back to the cells, the detected string is considered to be a CC.

Generally, a program contains plural CCs/GCCs. In (3) (b), tracing the cells beginning from the cell that satisfies the following conditions detects all CCs/GCCs.

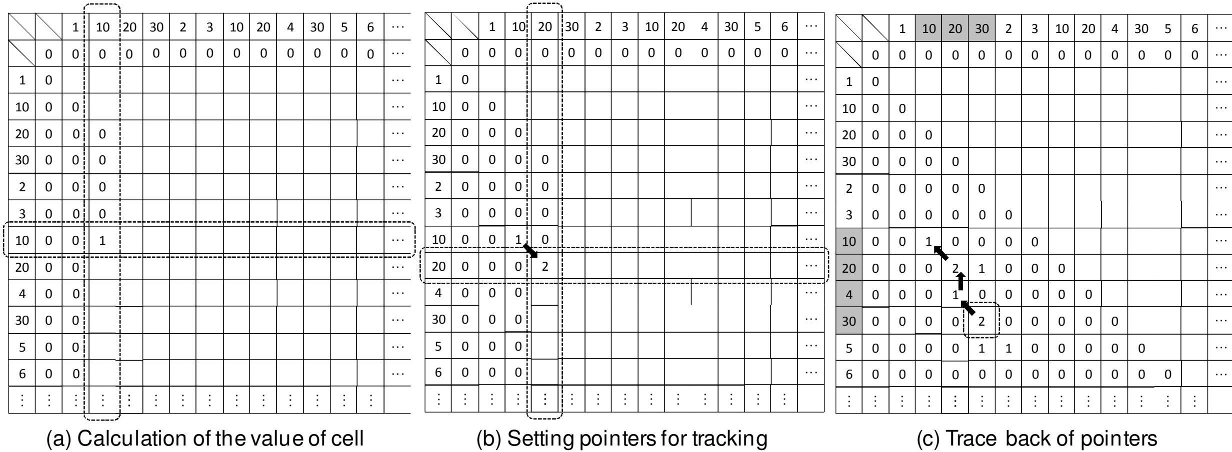

Fig. 3 Creation of SWA Table for Detecting CCs/GCCs.

$$v_{i,j} > 0 \tag{3}$$

$$v_{0,j} = v_{i,0} \tag{4}$$

Equation (3) indicates that CCs/GCCs continue until the relevant cell. Equation (4) indicates that the final lines of CCs/GCCs match. When CCs/GCCs are detected in the line unit, these equations add information that those lines ending with "}" are likely to be a block. The engineer checks when merging.

3.2.2 Calculate Metrics (STEP2)

This section describes STEP 2 shown in Fig.1. In this study, metrics that evaluate internal-logic complexity, independence, and the programming language dependency are used to improve program readability and maintainability by merging CCs/GCCs.

(1) LOC (Lines of Code)

LOC is a metric that indicates the number of program lines. LOC does not include comments or empty code lines. LOC is used for calculating other metrics.

(2) CYCR (Cyclomatic Complexity Rate)

CYC is a metric that indicates the program's control flow complexity. CYC has the number that the total number of execution paths in the program and 1 are added (the number of branch instructions, such as if, while, for, switch, etc. and 1 added). [16] CYCR is the value of CYC divided by LOC. CCs/GCCs with greater CYCR values have complicated control flows.

$$CYCR = \frac{CYC}{LOC} \tag{5}$$

(3)COB (Cohesion of Blocks)

COB is a metric that indicates the cohesion of the program fragment. [17] Cohesion means the degree of coordination of variables used for a block (a program fragment surrounded by parentheses, "{" and "}") in the program. Where the number of blocks within the program is b, the number of variables used in the program is v, the j-th variable used in the program is V_j, and the number of blocks in the program where variable V_j is used is $\mu(V_j)$, COB is defined as flows. CCs/GCCs with greater CYCR values become highly independent.

$$COB = \frac{1}{b}\frac{1}{v}\sum_{j}^{v} \mu(V_j) \tag{6}$$

(4) RNR (Ratio of Non-Repeated Elements)

RNR is a metric that indicates the proportion of non-repeated instructions included in the program. A program contains repeated programming-language-dependent instructions (routine phrases). [18] These portions are defined by the programming language specifications, and difficult to merge. CCs/GCCs with greater RNR values are likely to describe procedures. Where the LOC of the

whole CCs/GCCs is LOC_{whole} and the LOC of repeated statements is $LOC_{repeated}$, RNR is defined as follows.

$$RNR = 1 - \frac{LOC_{repeated}}{LOC_{whole}} \quad (7)$$

3.2.3 Selecting Mergeable CCs/GCCs (STEP3)

This section describes step 3 shown in Fig.1. By using the metrics obtained, STEP 3 selects CCs/GCCs that can be merged.

Here, the following section describes how to select CCs/GCCs, which can be merged, by using the metrics. First, CCs/GCCs that are independent on the programming language specifications are extracted by using RNR. Next, CCs/GCCs are narrowed down by using CYCR and COB in order to determine CCs/GCCs that can have benefits of merging. In order to determine the merging possibilities, we analyzed the existing programs to determine the thresholds for the metrics. We used three programs, A, B, and C. Each program has LOC between 1000 and 2500 while having 10 to 15 classes. Program A calculates a proper temperature in an agricultural plant, program B calculates the weight applied to the building's roof under various natural conditions such as rainfall, wind, and snow, and program C calculates travel expenses for a business trip. While calculating the metrics of CCs/GCCs within each program, two engineers determined the thresholds for metrics by judging whether CCs/GCCs could be merged or not. Since each metric's domain varied, we normalized each domain to [0, 1] by using equation (8). Here, x_i indicates the relevant metric's value, x_{max} indicates the maximum value of the relevant metric, and x_{min} indicates the minimum value of the relevant metric.

$$normalized_value = \frac{(x_i - x_{\min})}{(x_{\max} - x_{\min})} \quad (8)$$

Table 1: Thresholds of CYCR and COB for Merging

Program	A	B	C
Number of CCs/GCCs	6	8	6
Number of mergeable CCs/GCCs	3	5	3
Rate of Merging [%]	50	63	50
Threshold of CYCR	0.48	0.58	0.55
Threshold of COB	0.64	0.55	0.51

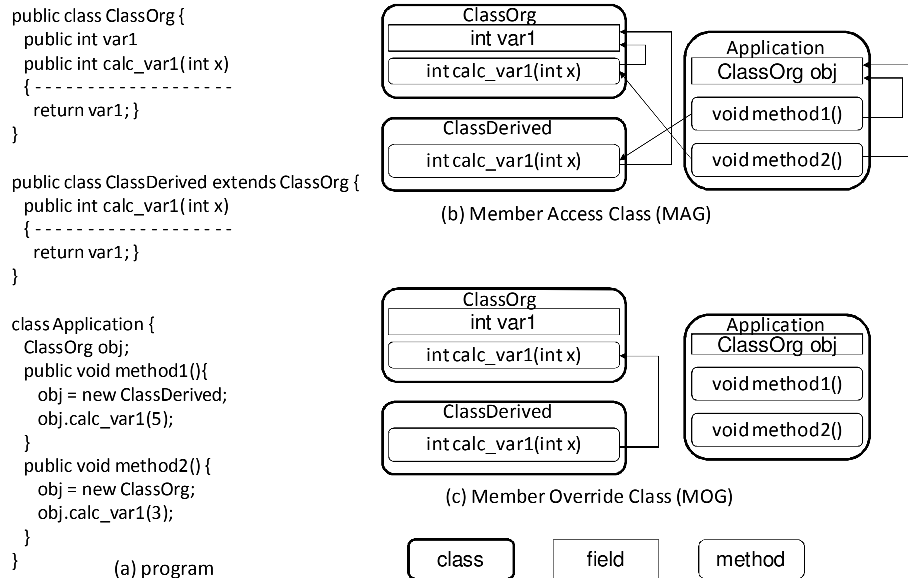

Fig.4 Relationship between Program, MAG, and MOG

Table 1 shows the number of CCs/GCCs within each program and the threshold of each metric that is determined to be merged. Based on these results, the threshold for RNR of CCs/GCCs that can be merged is set to 0.50, 0.58 for CYCR, and 0.64 for COB. CCs/GCCs that satisfy those conditions are the program fragments that are the portions describing specific procedures that are independent of programming language specification, the portions having similar program control structures, and the portions being highly independent of other

portions. Merging these CCs/GCCs could improve program readability and maintainability.

3.2.4 Determine Merging Type and Procedure for CCs/GCCs (STEP3)

This section describes STEP 4 shown in Fig.1. STEP 4 calculates the metrics related to the program structures of CCs/GCCs and determines the merging type and procedure. Table 2 lists merging type, judgment criteria (metrics that are used for determination), and specific merging procedures. We selected those methods suitable to merge CCs/GCCs that are proposed as the refactoring formats by Fowler. The proposed method determines the merging method based on information about the starting lines of CC and GCC, the ending lines of CC and GCC, the starting line of the gap, the ending line of the gap, classes to which CCs/GCCs belong, super classes, subclasses, attributes that are referred to (in classes and other classes), and methods used (in classes and other classes). The proposed method determines whether the merging is conducted or not, based on the number of attributes that are referred to and the number of methods used. In this study, merging is implemented when these numbers are 1 or smaller. CCs/GCCs can be merged even when these values are 2 or greater. However, this situation makes merging more complicated, while the program readability is not improved to any great extent. Therefore, CCs/GCCs are not merged in this case. Information about the starting lines, the ending lines, the starting line of the gap, and the ending line of the gap are obtained by using the CC/GCC detection method described in section 3.2.1. Other information is obtained by developing a Member Access Graph (MAG) and a Member Override Graph (MOG). [19] The MAG is a graph that expresses the relationship between the method call and attribute reference. The MAG expresses the method call relation in directed segments from the caller to the call target while expressing the attribute reference relation in direct segments from the referrer to the reference target. The MOG is a graph that expresses the overriding relationship with the inheritance of the method and the implementation of the abstract method. The MOG expresses method overriding and implementing in the directed segment from the method that overrides and the method to be overridden. Fig.4 shows the relationship between the program, MAG, and MOG. The program of

Table 2: List of Merging Type, Judgment Criteria, and Merging Procedure

Merging Type	Judgment Criteria (all the condition are satisfied)	Merging Procedure
Extract Method	● CCs/GCCs belong to the same class.	● Creates a new method in the class for merging.
Pull Up Method	● CCs/GCCs belong to the same class. ● CCs/GCCs belong to the same super class. ● CCs/GCCs exist in different subclasses.	● Creates a new method in the super class for merging.
Extract Class	● CCs/GCCs belong to the same class. ● Plural variety of CCs/GCCs exists.	● Creates a new class. ● Creates a new method in the class. ● Merges CCs/GCCs in the method.
Extract Super Class	● CCs/GCCs belong to different super classes. ● Plural variety of CCs/GCCs exists in the class.	● Creates a new super class. ● Changes the original class into a subclass. ● Creates a new method in the super class. ● Merges CCs/GCCs in the method.
Parameterized Method	● CCs/GCCs belong to the same class. ● Different constants are used for CCs/GCCs.	● Creates a new method. ● Set the constants as the arguments for the method.
Pull Up Filed	● CCs/GCCs belong to the same super class. ● CCs/GCCs exist in different subclasses. ● CCs/GCCs have the same attribute	● Transfers the attribute to the super class.
Create Template Method	● Extract Method or Pull Up Method is established ● There is a GCC (gaps are included) ● Input data and output data necessary for calculating the gap are different. ● The gap calculation result is used for the subsequent portion of the GCC's gap.	● Moves the GCC's common parts to the super class. ● Creates abstract methods that have same signatures of gap parts of GCC in the super class. ● Implements the abstract method in the subclass.

Fig.4 (a) has the following classes, ClassOrg, ClassDerived, and Application. ClassOrg has the calc_var1(int x) method, ClassDerived has the calc_var1(int x) method, and Application has two methods, such as method1() and method2(). In this program, method1() in the Application class calls calc_var1(int x) in the ClassDerived, and method2() calls calc_var1(int x) in the ClassOrg class. Fig.4 (b) shows the MAG that links the relationship between these method calls with directed segments. Additionally, calc_var1(int x) in the ClassDerived class overrides calc_var1(int x) in the ClassOrg class. Fig.4 (c) shows the MOG that links the overriding relationships with directed segments.

3.2.5 Merge CCs/GCCs (STEP5)

This section describes STEP 5 shown in Fig.1. Applying the specific procedure for the merging type determined by STEP 4 (see Table 2), STEP 5 merges CCs/GCCs. This step is conducted manually.

3.3 Support Tools for the Proposed Method

To evaluate the proposed method, we developed the CC/GCC detection tool, the metrics calculation tool, and the merging procedure proposal tool. The CC/GCC detection tool outputs information according to each CC/GCC, such as the name of the file, the starting and ending code line numbers, the starting and ending code line numbers of the gap. The metrics calculation tool outputs the following information according to each CC/GCC, LOC, CYC, COB, RNR, MAG, and MOG. [20] The merging proposal tool show merging procedure visually.

4. Evaluation of the Proposed Method

This chapter evaluates the utility of the proposed method and the prototype tools while applying them to programs that are actually in use. Five kinds of programs, from A through E, are inputted into the prototype tools, and CCs/GCCs in them are merged. Here, CCs or GCCs having 20 lines or more were detected, while the following parameters were given to the SWA: match = 2, mismatch = -3, and gap = -2. The programs A through E used for this evaluation were developed by individual programmers having four to six years of programming experience and equivalent skills. Program A checks the overridden relation between methods and the reference relation between attributes. Program B analyzes and traces the cause of particular malfunctions of a control program (Fault Tree Analysis). Program C supports exhaustive consideration of possible failures of a control program (Failure Mode and Effects Analysis). Program D extracts all the code lines that are necessary for calculating the values of particular

Table 3: Application Results of the Proposed Method

	A	B	C	D	E	F	Average
LOC before Change	3135	4271	2408	1582	5927	18026	
LOC after Change	2724	3387	1972	1374	5307	17545	
Detected CC	30	51	19	14	42	32	
Detected GCC	4	3	2	3	4	3	
Candidates for Integrated CC	14	23	11	6	17	20	
Candidates for Integrated GCC	3	3	2	2	4	3	
Integrated CC	13	23	11	6	15	19	
Integrated GCC	3	2	2	1	3	2	
Reduced LOC (Total)	411	884	436	208	620	481	
Reduced LOC (CC)	364	806	403	180	533	407	
Reduced LOC (GCC)	47	78	33	28	87	74	
Average Size (Total)	26	35	34	30	34	23	30
Average Size (CC)	28	35	37	30	36	21	31
Average Size (GCC)	16	39	17	28	29	37	27
Integrated / Candidates(total)[	94	96	100	88	86	91	93
Integrated Rate (CC) [%]	93	100	100	100	88	95	96
Integrated Rate (GCC) [%]	100	67	100	50	75	67	76
Reduced Rate (Total) [%]	13	21	18	13	10	3	9
Reduced Rate (CC) [%]	12	19	17	11	9	2	8
Reduced Rate (GCC)[%}	1	2	1	2	1	0	1

IJCAT - International Journal of Computing and Technology, Volume 5, Issue 1, January 2018
ISSN (Online) : 2348-6090
www.IJCAT.org

Table 4: Application Results of Applied Merging Types

	A (Total)		B (Total)		C (Total)		D (Total)		E (Total)	
	CC	GCC	CC	GCC	CC	GCC	CC	GCC	CC	GCC
Pull Up Method	1		1		2		0		1	
	1	0	1	0	1	1	0	0	1	0
Pull Up Field	0		1		0		0		0	
	0	0	1	0	0	0	0	0	0	0
Extract Method	14		20		9		8		14	
	11	3	19	1	9	0	6	2	14	0
Extract Class	1		2		1		0		1	
	1	0	2	0	1	0	0	0	0	0
Extract Super Class	0		0		0		0		2	
	0	0	0	0	0	0	0	0	0	2
Parameterize Method	13		20		10		8		15	
	10	3	19	1	9	1	6	2	15	0
Create Template Method	0		1		0		0		0	
	–	0	–	1	–	1	–	0	–	1

variables within a program (program slicing). Program E is the present prototype tools described in the previous section, 3.3.

Table 3 shows the CC/GCC merging results of each program. The meaning of each line of Table 3 is as follows: “LOC before/after Change” indicates the program’s LOC before/after merging, “Detected {CC, GCC}” indicates the number of detected {CCs, GCCs} within the program, “Candidates for Integrated {CC, GCC}” indicates the number of {CCs, GCCs} that were determined to be merged by the proposed method, “Integrated {CC, GCC}” indicates the number of {CCs, GCCs} that were actually merged, “Reduced LOC {Total, CC, GCC}” indicates the LOC of {total CCs and GCCs, only CCs, only GCCs} that were reduced by merging, “Average Size {Total, CC, GCC}” indicates the average LOC of {total CCs and GCCs, only CCs, only GCCs}, “Integrated/Candidates {Total, CC, GCC}” indicates the percentage of the number of {total CCs and GCCs, only CCs, only GCCs} merged actually and the number of {total CCs and GCCs, only CCs, only GCCs} selected by the proposed method for merging, and “Reduced Rate {Total, CC, GCC} indicates the percentage of LOC of {total CCs and GCCs, only CCs, only GCCs} that were reduced by merging. Since each program had a different scale, it did not make sense to directly compare LOC and the number of CCs/GCCs. On the other hand, the Average Size was 32 LOC, which was almost the same value of all the programs. This was because the threshold for CCs/GCCs detection was 20 LOC or greater, and the scale level for a programmer to easily understand and develop the program function by copying and pasting code is about 30 LOC. Integrated/Candidate (CC) reached 96%. This confirmed that the CCs selected by the proposed method could be almost merged. However, Integrated/Candidate (GCC) was 79%, which was lower than CC. This was because some GCCs including several gaps were not merged. Therefore, Integrated/Candidate (Total) was 93%. Reduced Rate was 10%. Because the number of CCs/GCCs varied depending on the completion rate of each program, this rate varied significantly.

Table 4 shows the merging types applied to CCs/GCCs. The proposed method might apply plural merging types simultaneously. Therefore, the number of CCs/GCCs did not match the number of merging types.

The breakdown for the merging type for CCs of program A is as follows: The Pull Up Method was applied to 1 CC, the Extract Method was applied to 1 CC, the Extract Method and the Parameterized Method were simultaneously applied to 10 CCs, and the Extract Class was applied to 1 CC. As for GCC merging types, the Extract Method and the Parameterized Method were simultaneously applied to three GCCs. Those merged GCCs had simply one gap, and the gap was independent of other portions of GCCs. Therefore, the gap was moved to the back of the GCC. Two programmers confirmed that the applied merging types and procedures were adequate.

The breakdown for the merging type for CCs of program B is as follows: The Pull Up Method was applied to 1 CC, the Pull Up Field was applied to 1 CC, the Extract Method and the Parameterized Method were

simultaneously applied to 19 CCs, and the Extract Class was applied to 2 CCs. As for the breakdown of GCC merging types, the Extract Method and the Parameterized Method were simultaneously applied to 1 GCC, and the Create Template Method was applied to 1 GCC. The former merging was done because of the same reason as program A. The latter merging was done because there was only one gap. However, other GCCs that were not merged had plural gaps. Two programmers confirmed that applied merging types and procedures were adequate.

The breakdown for the merging type for CCs of program C is as follows: The Pull Up Method was applied to 1 CC, the Extract Method and the Parameterized Method were simultaneously applied to 9 CCs, and the Extract Class was applied to 1 CC. As for the breakdown of GCC merging types, the Pull Up Method and the Parameterized Method were simultaneously applied to 1 GCC, and the create Template Method was applied to 1 GCC. These GCCs were merged because of the same reasons as program B. Two programmers confirmed that applied merging types and procedures were adequate.

As for CC merging types for program D, the Extract Method and the Parameterized Method were simultaneously applied to 6 CCs. As for GCC merging types, the Extract Method and the Parameterized Method were simultaneously applied to 2 GCCs. These GCCs were merged because of the same reasons as program B. Two programmers confirmed that applied merging types and procedures were adequate.

The breakdown for the merging types for CCs of program E is as follows: The Pull Up Method and the Parameterized Method were simultaneously applied to 1 CC, and the Extract Method and the Parameterized Method were simultaneously applied to 14 CCs. As for GCC merging types, the Extract Super Class were applied to 2 GCCs, and the Create Template Method was applied to 1 GCC. As for the former GCCs, the gap portion was independent of the common portion. Therefore, the common portion was set as the Super Class, while the gap portion remained in the Child Class. The latter merging was done because there was only one gap. However, other GCCs that were not merged had plural gaps. Two programmers confirmed that applied merging types and procedures were adequate.

These application experiments confirmed that the Extract Method and the Parameterized Method were frequently applied to merge CCs/GCCs. This was because CCs/GCCs are frequently developed by copying and pasting, as well as modifying, adding, and/or deleting small-scale program fragments that are coherent as a function.

5. Summary and Future Works

Through this paper, we proposed a method for selecting CCs/GCCs for merging by calculating CYCR, COB, and RNR of CCs/GCCs detected by using the SWA. Furthermore, the proposed method also determines a proper merging type from the metrics related to the program structures of those selected CCs/GCCs. We applied the proposed method to the existing programs and confirmed that the proposed method can adequately merge 96% of CCs that were determined to be merged. We also confirmed that the proposed method can properly merge 79% of GCCs that were determined to be merged. The proposed method made it possible to merge CCs/GCCs more efficiently than merging based on exhaustive analysis of all CCs/GCCs. The low percentage of merging GCCs was due to plural gaps existing in GCCs. These GCCs were not merged in this study. This was because we determined that merging such GCCs would be less effective in improving readability and maintainability in comparison with the increasing cost of merging work. The above-mentioned results confirmed that use of the proposed method enabled efficient merging of CCs/GCCs.

The future issues include the introduction of new metrics so as to improve the rate of judging CCs/GCCs that can be merged. Moreover, while preparing more merging types, we will improve program readability and maintainability.

References

[1] Y. Higo, S. Kusmoto and K. Inouse, "A Survey of Code Clone Detection and its Related Techniques", IEICE Transaction on Information and Systems D, Vol. 91-D, No. 6, 2008, pp. 1465-1481.

[2] K. M. Bergman, L. L. Lau and D. Notkin, "An Ethnographic Study of Copy and Paste Programming Practices in OOPL, Proceedings 2004 International Symposium on Empirical Software Engineering, 2004, pp. 83-92.

[3] L. Bergroth, H. Hakonen and T. Raita, "A Survey of Longest Common Subsequence Algorithms", Proceedings 7th International Symposium on String Processing and Information Retrieval, 2000, pp.39-48.

[4] C. K. Roy and J. R. Cordy, "NiCad: Accurate Detection of Near-miss Intentional Clones Using Flexible Pretty-

printing and Code Normalization", Proceedings 16th International Conference on Program Comprehension, 2008, pp. 172-181.

[5] L. Jiang, G. Misherghi, Z. Su and S. Glondu, "DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones", Proceedings the 29th International Conference on Software Engineering, 2007, pp. 96-105.

[6] J. Krinke, "Identifying Similar Code with Program Dependence Graphs", Proceedings the 8th Working Conference on Reverse Engineering, 2001, pp. 301-309.

[7] S. Temple and W. Michael, "Identification of Common Molecular Subsequences", Journal of Molecular Biology, Vol. 147, 1981, pp. 195–197.

[8] H. Murakami, K. Hotta, Y. Higo, H. Igaki and S. Kusumoto, "Gapped Code Clone Detection with Lightweight Source Code Analysis", Proceedings ICPC 2013, 2013, pp. 93-102.

[9] H. Murakami, K. Hotta, Y. Higo, H. Igaki and S. Kusumoto, "Gapped Code Clone Detection Using the Smith-Waterman Algorithm" , IPSJ Journal, Vol. 55, No. 2, 2014, pp. 981-993 .

[10] Y. Higo and S. Kusumoto, "How Often Do Unintended Inconsistencies Happen? - Deriving Modification Patterns and Detecting Overlooked Code Fragments-", Proceedings 28th IEEE International Conference of Software Maintenance, 2012, pp.222-231.

[11] N. Gode and R. Koschke, "Frequency and Risks of Changes to Clones", Proceedings ICSE'11, no page number, 2011, 10pages.

[12] N. Fenton and J. Bieman, "Software Metrics -A Rigorous and Practical Approach - Third Edition", CRC press, 2014.

[13] K. Hatano, Y. Nomura, H. Taniguti and K. Ushijima, "A Mechanism to Support Automated Refactoring Process Using Software Metrics", IPSJ Journal, Vol.44, No.6, 2003, pp.1548-1557

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

[15] Y. Higo, T. Kamiya, S. Kusumoto and K. Inoue, "Refactoring Support Environment Based on Code Clone Analysis", IEICE Transactions D, Vol. J88-DI, No.2, 2005, pp.186-195.

[16] T. McCabe, "Complexity Measure", IEEE Transactions on Software Engineering, Vol.2, No 4, 1976, pp 308-320.

[17] M. Ioka, N. Yosida, T. Masai and K. Inoue, "Ranking Candidates for Applying Template Method Pattern with a Cohesion Metric COB", Technical Report of IEICE KBSE, vol.111,No.169, 2011, pp.57-62.

[18] Y. Higo, T. Kamiya, S. Kusumoto and K. Inoue, "Method and implementation for investigating code clones in a software system", Information and Software Technology, Vol.49, Issues 9-10, 2007, pp.985-998.

[19] R. Yokomori, K. Kondou, F. Ohata and K. Inoue, "Impact Analysis System for Changes on Object-Oriented Programs", IEICE Transactions D, Vol. J86–D–I,No.3, 2003, pp.150–158.

[20] M. Takahashi, R. Nanba, Y. Anang and Y. Watanabe, "An Improvement Method for Program Structure Using Code Clone Detection, Impact Analysis, and Refactoring Formats", SICE Journal of Control, Measurement, and System Integration, Vol.10, No.3, 2017, pp. 184-191.

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 and with University of Yamanashi since 2008. He is a professor in University of Yamanashi since 2014. His research interests include software engineering and safety.

Yunarso Anang received B.E. and M.E. degree in software engineering from University of Yamanashi in 1995 and 1997 respectively, and received Ph.D in engineering also from University of Yamanashi in 2017. He was with SYNC Information System, Inc., Japan from 2000 to 2007 as a senior engineer. He is a lecturer in Institute of Statistics, Indonesia, since 2008. His research interests include software engineering and quality.

Reiji Nanba received B.E. degree in 1999 from Daiichi Institute of Technology, Japan, and ME degree in 2003 and Ph.D. degree in 2008 from 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.

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.