An Improvement Method for Program Structure Using Code Clone Detection, Impact Analysis, and Refactoring Formats

Masakazu Takahashi *, Reiji Nanba **, Yunarso Anang *, and Yoshimichi Watanabe *

Abstract: This paper proposes a method that aggregates similar portions (code clones) in the developed program and redesigns the current program structure to an adequate program structure. The procedure of the proposed method is the following: Code clones are detected, and their characteristics are extracted, the design methods that code clones are aggregated are proposed, the program is modified, and the affected portions in the program are clarified and retested. Furthermore, a tool that supports the proposed method is created. As a result of applying the proposed method and the tool, it is found that 45% CCs are aggregated and 13% instructions of the program are deleted. Additionally, it is found that an adequate program structure is realized by using the proposed method and the tool.

Key Words: code clone, impact analysis, refactoring, object oriented method.

1. Introduction

This paper proposes a method that detects similar portions in the program (Code Clone: CC), aggregates CCs, and redesigns CCs into an adequate program structure. This makes the maintenance for the program easy. This redesign method is called refactoring.

At first, the reasons why the programs contain many CCs are explained. When developing a new program or when adding and modifying program functions, in many cases, programmers copy, paste, and modify certain portions of the program in order to complete development of the program. This method retains a number of CCs within the program. Programs containing a large number of CCs have the following problems: Those portions related to CCs are difficult to change, errors in changes are likely to be produced, and the process of making changes becomes inefficient. These become issues when maintaining programs.

Next, the outline of the paper is explained. Takahashi et al. proposes the refactoring method, but sufficient application and evaluation have not conducted [1]. So, this paper mainly describes the results of applications for existing programs, written in Java as an Object Oriented Programming Language (OOPL). The procedure of the proposed method is the following: The proposed method detects CCs in the program and analyzes the characteristics of CCs, such as the start line, the end line, the number of used externally defined variables, the number of method calls, and the parent classes. Based on the extracted characteristics, refactoring methods are proposed. The programmer modifies the current program to the program with an adequate structure. Moreover, scopes of tests related to the modification of the program (referred to as the modification) are clarified using the impact analysis method. The scopes are shown in the units of methods, fields, and lines. And the scopes are tested for adequacy of the modified program. This procedure can recreate a proper program structure without overlapped portions. Additionally, this proposed method makes it easier and more efficient to change the program, and it contributes to maintain the program for a longer time.

2. Related Works

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 this paragraph. CCs are detected in the units 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 detects unnecessary blanks, line breaks, and comments preliminarily, while replacing variable names and numerical values with specific symbols. Therefore, this process can detect program portions where variable names and numerical values have been changed as CCs. The line-based CC detection process detects unnecessary blanks, line breaks, and comments and replaces variable names and numerical values with specific symbols, and then calculates the hash value of each line. Once this hash value is calculated, CCs can be detected quickly [2],[3]. While Monden et al. report that the rate of the CC portion in the legacy systems is about 40% when combined tests were finished [4].

Next, studies regarding refactoring are described. Design patterns are known as the standard of proper program design. Introduction of design patterns into program design can enhance readability and maintainability of programs written in OOPL. Gamma et al. have proposed 23 design patterns [5].
Moreover, Fowler et al. have organized representative refactoring methods (referred to as refactoring formats) in a catalog format [6]. 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 [7]. 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. Impact analysis is a method to clarify the scope of programs which might be affected when the program is modified. Kung has proposed a method to clarify program classes which might be under the influence of program modification based on the class firewall concept [8]. However, this method had a problem that methods and fields not modified (referred to as members hereinafter) would be included in the scope under the influence. Jang has 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 [9]. Yokomori has identified a range of regression test by analyzing member access relationships [10]. Traon has identified a range of regression test by analyzing design specification written in Unified Modelling Language [11]. However, these methods also have a problem in that a large amount of program portions dependent from influence can be included within the members. As mentioned above, the traditional methods were inefficient in verifying program portions under the 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.

Figure 1 shows the proposed refactoring procedure. As preparation for refactoring, STEP 1 detects CCs from a program before modification and develops their list (referred to as CC list). 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. Still the opportunity losses for CC refactoring are avoided to keep the CC list, MAGs, and MOGs that are developed first until the refactoring process will be completed (this process detail is mentioned in Section 3.2). STEP 2 selects a pair of CCs (referred to as a CC pair) from the CC list, which conducts refactoring. 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 modified program. 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. If more refactoring is required when the first refactoring is finished, CC list is developed again, CCs that can be conducted refactoring are confirmed, and those CCs are conducted refactoring again. This procedure is conducted repeatedly until there is no CC that should be conducted refactoring.

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.

3.1.1 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 is required. Figure 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 as shown in Fig. 2 (2). Second, variable names, function names, and numerical values are replaced with specific symbols as shown in Fig. 2 (3), 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 as shown in Fig. 2 (4). This hash value is calculated by obtaining and adding ASCII 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.

3.1.2 Creation of MAG and MOG

Programs written in 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 fields. 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 overridden methods accompanied by class inheritance, implementation of methods which are defined abstractly, and fields encapsulation. MOG expresses overridden methods in the directed line from the method to override to the method to be overridden. Figure 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 has the field y, the method1(int x), and the method1(String x), the DerivedClass has the method1(int x), and the Application has the field z, the field obj, the method2(), and the method3(). The method1(int x) and method1(String x) in the BaseClass refers to y in the BaseClass, the method1(int x) in the DerivedClass refers to y in the BaseClass, the method2() refers to z and obj and calls method1(int x) in the DerivedClass class, while the method3() refers to z and obj and calls method1(String x) in the BaseClass class. Figure 3 (b) shows MAG where these relationships are connected with directed lines. Next, the method1(int x) in the DerivedClass class overrides the method 1(int x) in the BaseClass class. Figure 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 refactoring formats. 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 portion 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. Table 1 indicates the refactoring formats treated in this research and the judging criteria for applying these formats. Because there is a case that the refactoring formats cannot be applied simply, the detailed refactoring method is considered individually based on the proposed refactoring formats. For example, in the case when the plural return values are handed over from a refactoring portion to an original program portion, a method that those values are handed over using instance that includes those values as fields is considered.

3.3 Implementation of Refactoring (STEP 3)

STEP 3 modifies the program based on refactoring information, after that 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.

3.4 Implementation of Impact Analysis (STEP 4)

This section describes the operation of STEP 4. Impact analysis is a method to identify a scope of a program under the impact of modification. The proposed method modifies several dozen program lines in the unit. Consequently the affected portions become wide range, and it requires conducting complex works and longer time by using only program slicing technique. The object oriented program consists of members; the program written in OOP has a characteristic that the service is realized by delegating the authority between members. As modified portions for refactoring are classes and members, the differences of access and override relationships of the program between before and after modification are a method that those values are handed over from a refactoring portion to an original program portion, a method that those values are handed over using instance that includes those values as fields is considered.
Table 1 Refactoring formats and their judging criteria.

<table>
<thead>
<tr>
<th>Refactoring Format</th>
<th>Refactoring Method</th>
<th>Judging Criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extract Method</td>
<td>Integrate overlapping operations (CCs) within the same class into one newly developed method in the same class.</td>
<td>CCs exist within the same class. One kind of CC exists (or a few kinds of CC exist).</td>
</tr>
<tr>
<td>Pull up Method</td>
<td>Integrate overlapping operations (CCs) in the subclasses that have one super class into one newly developed method in the super class.</td>
<td>Subclasses have a same super class. CCs exist in the subclasses.</td>
</tr>
<tr>
<td>Extract Class</td>
<td>Integrate multiple overlapping operations (CCs) in the same class into a newly developed class.</td>
<td>CCs exist in the same class. Many kinds of CCs exist.</td>
</tr>
<tr>
<td>Extract Super Class</td>
<td>Integrate multiple overlapping operations (CCs) in the classes that do not have a same super class into a newly developed parent class (super class). The original classes become the subclasses of the parent class.</td>
<td>Classes do not have a same super class. Multiple overlapping operations exist in the classes.</td>
</tr>
<tr>
<td>Parameterized Method</td>
<td>Integrate overlapping operations (CCs) within the same class, where only the used values differ, into a newly developed method that the values are handed over as the arguments.</td>
<td>CCs exist within the same class. Values that are used within the CCs differ.</td>
</tr>
<tr>
<td>Pull up Field</td>
<td>Integrate same fields in the subclasses that have a same super class into newly added fields in the super class.</td>
<td>Subclasses have a same super class. Overlapped fields in the subclasses exist.</td>
</tr>
</tbody>
</table>

Fig. 4 Sample of change impact analysis.

The ride relationship of methods which appeared and disappeared due to program modification is then extracted based on comparison of MOG before and after program modification. These differences become members which are under the impact of program modification. The following shows an example of change in a MAG. Figure 4 (a) shows the MAG before modification and the output result of this MAG by using the MAG/MOG creation tool (described 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 sub method, square root method, and main method, the test2 class has the add method, and the sample class has the multiply and square methods. In addition, the sub method in the test1 class before modification accesses to the multiply method in the sample class, the main method in the test1 class accesses to the add method in the test2 class, and the square method in the sample class access to the square root method in the test1 class. On the other hand, after modification, the sub method in the test1 class accesses to the add method in the test2 class, the main method in the test1 class accesses to the add method in the test2 class, and the square method in the sample class accesses to the square root method in the test1 class. Comparison of Fig. 4 (a) and (b) shows that access of the sub method in the test1 class to the multiply method in the sample class disappears, while the access of the sub method in the test1 class to the add method in the test2 class increases. This status is shown by the cloud shaped portion in Fig. 4. This portion becomes members which are affected by the modification. Though the affected portions related to the methods are described only, affected portions related to the fields can be identified in the same manner.

The second stage, line-based impact analysis, clarifies those lines under the 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 are 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. Figure 5 shows an example of the 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 section describes a support tool in Fig. 1. This tool is composed of the following sub-tools: CC detection, MAG/MOG creation, refactoring format proposal, and impact analysis tool. Each sub-tool 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. Figure 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. The MAG/MOG creation tool creates the MAG and MOG of a program. This tool is used by STEP 1 and STEP 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 overriding 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 format contains the total number of CC lines, the number of externally defined variables used, the parent class, and the number of method calls. Figure 7 indicates refactoring format as the output of the refactoring format proposal tool. Additionally, Fig. 8 shows the sample screen of refactoring order sheet that shows the method for refactoring a CC pair based on the refactoring format. The left side of the refactoring order sheet shows the current program structure, and the right side of the refactoring order sheet shows the program structure that the refactoring will be done. 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 part 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 is conducted in two stages. In the first stage, adequacy and sufficiency of the proposed method and the tool are evaluated. In the second stage, effectiveness and serviceability of the proposed method and the tool are evaluated by applying those to the existing program.

4.1 Evaluation of Adequacy and Sufficiency for the Proposed Method and Tool

This section describes adequacy and sufficiency of the proposed method and the tool. The evaluation was conducted by two engineers who have the same level of skills and experience. (They have 4 years to 6 years of programming experience using OOPL and attend a two-day 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, the other engineer (engineer B) conducted refactoring for prepared pro-
Table 2 List of test case.

<table>
<thead>
<tr>
<th>No</th>
<th>Type</th>
<th>Engineer A</th>
<th>Engineer B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Pull up Field</td>
<td>Success</td>
<td>Success</td>
</tr>
<tr>
<td>2</td>
<td>Pull up Method</td>
<td>Success</td>
<td>Success</td>
</tr>
<tr>
<td>3</td>
<td>Extract Method</td>
<td>Success</td>
<td>Success</td>
</tr>
<tr>
<td>4</td>
<td>Extract Class</td>
<td>Success</td>
<td>Success</td>
</tr>
<tr>
<td>5</td>
<td>Parameterized Method</td>
<td>Success</td>
<td>Success</td>
</tr>
<tr>
<td>6</td>
<td>Parameterized Method + Pull up Method</td>
<td>Success</td>
<td>Fail</td>
</tr>
<tr>
<td>7</td>
<td>Parameterized Method + Extract Method</td>
<td>Success</td>
<td>Success</td>
</tr>
<tr>
<td>8</td>
<td>Parameterized Method + Extract Class</td>
<td>Success</td>
<td>Success</td>
</tr>
</tbody>
</table>

grams without the proposed method and the tool. Last, three engineers (engineer A, engineer B, and engineer C who had the 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 types of test cases. Test cases 1–5 are used for the evaluation of refactoring that single refactoring format is applied. Test cases 6–8 are 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 “Parameterized Method + Pull up Method” and “Parameterized Method + Extract Method”, and those CC was added some slight modifications, such as changing variable names and constant values [12]. Right side in Table 2 shows the test results. Regarding the test case 1 to 5, engineer A and engineer B could conduct refactoring to all test cases appropriately. Regarding the test cases 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 been done.

Here, refactoring results for test case 6 is described. Figure 9 shows the program structure (class diagram) given in test case 6. This program has four classes, such as M, C1, C2, and C3. Class M has the main method. And super class C1 has subclasses C2 and C3. Method_A, method_B, method_C, and method_D are CCs, and the names of variables in those methods and the values of the constants were slightly changed. Figure 10 shows the programs of method_A, method_B, method_C, and method_D. Those four methods have the 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 applying Parameterize Method. Method_C and method_D can be aggregated as method_CD in subclass C3 in the same manner. The engineer B conducted only those operations. Figure 11 shows a refactoring result conducted by the engineer B. While method_AB and method_CD are aggregated to method_ABCD in super class C1 by using “Parameterized Method + Pull Up Method.” Figure 12 shows a refactoring result conducted by the engineer A.

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

4.2 Evaluation of the Effectiveness and Serviceability for the Proposed Method and Tool

This section describes effectiveness and serviceability that are evaluated by applying the proposed method and the tool to the existing program. To evaluate those, six kinds of program, such as programs A to F, are inputted into the tool and are conducted refactoring. The developers of programs A to E have the same level skills and experiences (they have four to six years of experiences developing Java program). The developer who creates program F has a lot of skills and experiences in comparison with the other developers, because program F is open source software. Program A is written in Java and is a tool that is used for analyzing overridden and reference relationships between fields and methods. Program B is a tool that is used for analyzing the causes of specific fault in the control program (conducting Fault Tree Analysis). Program C is a tool that analyzes all
Through long-term-use of the program, it is considered that the program was redesigned into an appropriate program structure by adequate maintenance and refactoring. But this process in the operation phase requires many costs. Consequently, we recommend that a newly developed program undergoes refactoring as soon as possible. This makes the program structure adequate, and this reduces the total costs related to the maintenance of the program. Additionally, the program that has undergone refactoring ensures a long time use.

As a result, CCs in the program can be integrated and the program can be redesigned into an adequate program structure by applying the proposed method and the tool. It is considered that the program becomes to be used for a long time and the total development cost of the program will be reduced because the program is not newly developed. This contributes to enterprise management. We recommend that the refactoring for the program is conducted immediately after the program is developed.

### 5. Conclusion

This paper proposes a refactoring method that detects CCs in the developed program, redesigns the current program structure into an adequate program structure, and aggregates CCs. Additionally, a refactoring support tool is developed. To evaluate the proposed method and the tool, they are applied to the existing programs. As a result, 45% CCs in the developed program could be aggregated, and 13% instructions of the program were reduced. Furthermore, we found that the developed programs are redesigned into the programs that have an adequate program structure. Those explain that the proposed method and the tool have sufficient ability to redesign the program. From those results, it is found that the developed program has a lot of CCs, and many CCs can be redesigned into an adequate program structure by applying both “Extract Method” and “Parameterize Method” simultaneously. We strongly recommend that the refactoring should be conducted as soon as possible when the program is developed. It is considered that refactoring contributes to reducing the maintenance costs throughout the lifecycle of the developed program.

The future issues include how to judge the refactoring formats to be applied where multiple refactoring formats can be applied. Moreover, we are going to increase applicable refactoring formats.

### References


---

**Masakazu Takahashi** (Member)

He 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, Japan from 2005 to 2008. He is presently a professor in University of Yamanashi, Japan.

His research interests include software engineering and safety.

**Reiji Nanba**

He received B.E. degree in 1999 from Daiichi Institute of Technology, Japan. M.E. 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 presently an associate professor in Daiichi Institute of Technology. His research field is in Civil and Environmental Engineering and Engineering Education.

**Yunarso Anang**

He 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. Starting from 2014, he is also a Ph.D. student at University of Yamanashi, Japan. His research interests include software engineering.

**Yoshimichi Watanabe**

He 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.