On the Level of Code Suggestion for Reuse

Akio Ohtani, Yoshiki Higo, Tomoya Ishihara, and Shinji Kusumoto
Graduate School of Information Science and Technology, Osaka University,
1-5, Yamadaoka, Suita, Osaka, 565-0871, Japan
Email: {a-ohtani,higo,t-ishihr,kusumoto}@ist.osaka-u.ac.jp

***Abstract*—Code search techniques are well-known as one of the techniques that helps code reuse. If developers input queries that represent functionality that they want, the techniques suggest code fragments that are related to the query. Generally, code search techniques suggest code at the component level of programming language such as class or file. Due to this, developers occasionally need to search necessary code in the suggested area. As a countermeasure, there is a code search technique where code is suggested based on the past reuse. The technique ignores structural code blocks, so that developers need to add some code to the pasted code or remove some code from it. That is, the advantages and disadvantages of the former technique are disadvantages and advantages of the latter one, respectively. In this research, we have conducted a comparative study to reveal which level of code suggestion is more useful for code reuse. In the study, we also compared a hybrid technique of the two techniques with them. As a result, we revealed that component-level suggestions were able to provide reusable code more precisely. On the other hand, reuse-level suggestions were more helpful to reuse larger code.**

***Keywords*-Code search, Code reuse, Code clone**

I. Introduction

Software reuse is known as a promising way to promote efficient software development. Keyword-based code search systems are a kind of frameworks that support code reuse [1], [2], [3]. Such systems return source code (code fragments) related to a given query (keywords that a developer inputs). However, existing keyword-based code search systems have a drawback. They suggest code at a component level of the programming language. Thus, they sometimes suggest extra (unnecessary) code together with necessary one. Developers need to seek code that they actually want to reuse from the suggested code. Most of the existing systems suggest code at file-level or class-level, so that developers need to search a small piece of reusable code from a large amount of suggested code if they want to implement small functions. Besides, developers do not always need code at the same level [4]. Sometimes they need a whole class but at other times they need only several lines of code. Search systems should suggest code at the level of developers' demands.

As a way to solve the above issue, Ishihara et al. proposed a code search technique that suggests code at the level of the past reuse [5]. This technique suggests code that has been reused. In the technique, code clones (hereafter, clones) across software systems are regarded as reused code. That is, detecting clones among systems is regarded as identifying code reuse among them. Before now, some techniques have been proposed to detect clones across systems [6], [7], [8].

However, suggesting code at reuse-level requires manual code addition or deletion after developers copy and paste suggested code as it is. That is because reused code does not necessarily match with the structural code blocks such as class, method, or simple block in methods. Developers do not need to seek code that they actually want to reuse from a large amount of suggested code, but they need to add some code to the pasted code or remove some code from it. The advantage and disadvantage of the component-level suggestion are the disadvantage and advantage of the reuse-revel suggestion, respectively.

It is unclear which of the two techniques supports developers more efficiently and more effectively than the other one. Consequently, in this research, we have conducted a comparative study on the following techniques.

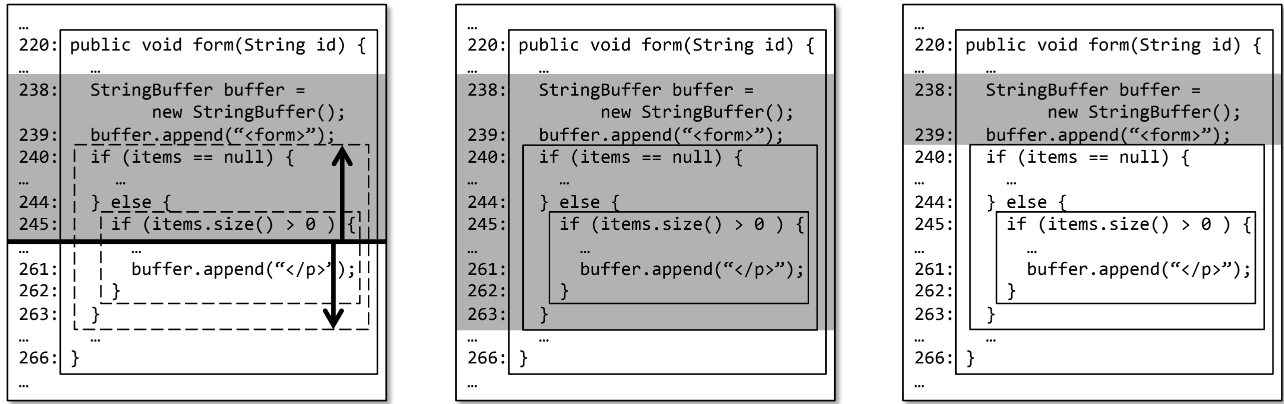
- **Technique-A:** the first technique is a component-level code suggestion [1], which suggests code at the method-level[1]. In this paper, we do not introduce this technique any more in this paper due to space limitation.
- **Technique-B:** the second technique is a code suggestion based on past reuse [5].
- **Technique-C:** the third technique is a revised version of the second technique. That is, suggested code is identified based on the past reuse, but they are reshaped to match with the structural units of programming language like the first technique. The third technique is our proposed technique in this paper.

The remainder of this paper is organized as follows: In Section II, we introduce our previous technique (Technique-B), and in Section III we explain our proposed technique (Technique-C); in Section IV reports the experimental result, then we discuss it in Section V; threats to validity in the experiment is described in Section VI; lastly, we conclude this paper in Section VII.

II. Technique-B: Reuse-level Code Search

Ishihara et al. proposed a code search technique that suggests code reused in the past [5]. Their technique firstly detects the past code reuse in advance. When a developer inputs a query, it suggests reused code related to the query. The highlighted area in Figure 1(a) is an example of code

[1]The technique originally suggests code at class-level. But, in this experiment, we developed a tool that suggests code at method-level with the component-level code search technique because the class-level suggestion is too coarse in this comparative study.



(a) Before using proposed technique (b) Enlarged suggested area (c) Shrunk suggested area

Fig. 1. Example of enlarging and shrinking suggested code by using the proposed technique

suggested by Ishihara's technique. Contrary to Technique-A, their technique ignores borders of code blocks.

The technique leverages the index-based clone detection [9] to detect past code reuse quickly. Code reuse is the most common cause that clones occur in source code [10], [11]. Thus, detecting clones can identify past code reuse.

Ishihara's technique consists of two procedures, *Code Analysis* and *Code Suggestion*. The *Code Analysis* procedure detects clones from a given set of source files and extracts keywords from each of the detected clones. Clones and keywords are stored into a database. The *Code Suggestion* procedure suggests code related to query input by developers. The *Code Suggestion* procedure utilizes the database created by the *Code Analysis* procedure for suggesting code.

III. Technique-C: Reuse-level Code Search considering Block Borders

The proposed technique is an enhancement of Ishihara's technique. We explain the proposed techniques by using Java code examples because currently our implementation handles Java language. However, it is not difficult to expand it to other programming languages.

Ishihara's technique ignores block borders in source code, so that suggested code occasionally includes unnecessary program statements or lacks necessary ones for completing tasks. In Figure 1(a), the hatched area is the code suggested by Ishihara's technique, and the dashed rectangles mean their block borders that are split by the suggested code. Thus, if we copy and paste the code as it is, the pasted code requires further modifications such as deleting unnecessary program statements or adding necessary ones.

Hereafter, we use term **decoupled block**, which means a code block where the border of a given clone exists. Two if-blocks that start from the 240 line and the 245 line are decoupled blocks of the clone in Figure 1(a).

The proposed technique suggests cloned code with consideration for code block borders. The proposed technique suggests code with the code range adjustment by considering decoupled blocks. More concretely, the proposed technique has two ways to adjust code ranges.

- Enlarging code range so as to include a decoupled block
- Shrinking code range so as to exclude a decoupled block

Decoupled blocks can occur at the top and the bottom of clones. For each decoupled block, the proposed technique adjusts code range with the above two strategies.

Figure 1 shows an example of code range adjustment. In Figure 1(a), the highlighted code is a clone, and the two if-blocks that start from the 240 line and the 245 line are its decoupled blocks. If we adopt the strategy *enlarging*, the highlighted code in Figure 1(b) is suggested to developers. Conversely, if we adopt the strategy *shrinking*, the highlighted code in Figure 1(c) is suggested. There are not partially-included sub-blocks in both of the suggested code ranges.

A. Procedures

Figure 2 shows an overview of the proposed technique. The procedure of the proposed technique is similar Ishihara's technique [5], but it has some additional operations. The hatched items show the additional operations.

The proposed technique consists of two procedures, *Code Analysis* and *Code Suggestion*. The *Code Analysis* procedure includes the following 5 steps.

STEP-A1: detecting clones from given source files. The number of clones for each clone group is counted.
STEP-A2: extracting keywords from the detected clones.
STEP-A3: computing CRS (*component rank* scores) for methods that include the detected clones.
STEP-A4: identifying code blocks in given source files.
STEP-A5: adjusting code ranges for each of the clones by comparing it with the code blocks.

The *Code Suggestion* procedure includes the 3 steps.

STEP-S1: ranking code fragments that include keywords related to a given query.
STEP-S2: selecting an adjusted code range for a code fragment selected by the developer.
STEP-S3: suggesting the adjusted range to the developer.

The remainder of this section explains the added steps.

B. STEP-A4: Block Detection

The start and end lines of code blocks in given source files are identified. In our implementation, we use Java Development Tools (JDT) for obtaining the block positions in Java source code. The tool traverses the abstract syntax tree of each method to identify the following types of code blocks:

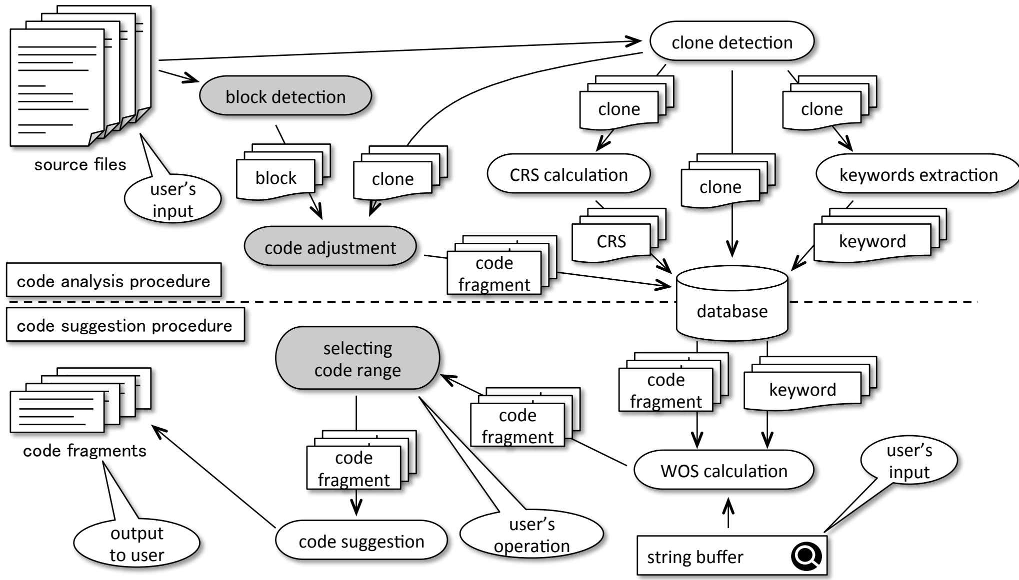


Fig. 2. Overview of proposed technique

do-while, for, foreach, if, switch, synchronized, try, and while. If a node is recognized as one of those types, the start and end lines are computed by using APIs in JDT.

C. STEP-A5: Code Adjustment

Each clone is compared with code blocks. If the top border or bottom border of the clone is dividing sub-blocks, code ranges that include the sub-blocks and exclude them are obtained. In the case of Figure 1(a), the two adjusted code ranges shown in Figures 1(b) and 1(c) are obtained.

D. STEP-S2: Selecting Code Range

In this step, the proposed technique ranks adjusted code ranges of a clone selected by the developer. The ranking is performed by computing a metric **misalignment**.

Here, we assume that $L(c)$ is a set of lines included in a given code fragment c, and c_o, c_1, c_2, $\cdots$, c_n are a clone and its adjusted code ranges.

Misalignment (m) between a clone (c_o) and its adjusted code range (c_i) is computed with the following formula.

$$m(c_o,c_i) \quad = \quad |L(c_i) \cap \overline{L(c_o)}| + |\overline{L(c_i)} \cap L(c_o)| \qquad (1)$$

Just after a developer selects a clone, the adjusted code range whose misalignment is the smallest is selected as the default code range. She/he can see other adjusted code ranges by operating the front-end GUI.

In the case of Figure 1, the two adjusted code ranges have the following misalignment values.

$$m(c_o,c_1) = 18, \qquad m(c_o,c_2) = 6$$

c_o, c_1, and c_2 mean code fragments in Figures 1(a), 1(b), and 1(c), respectively. In this case, adjusted code range c_2 has higher priority than c_1.

IV. Experiment

In this section, we describe the experiment that we compared the three code search techniques A, B, and C.

A. Experimental Design

Nine research participants attended the experiment. Each of them was given nine tasks, and she/he implemented Java code for each task with an assigned technique. The technique assignment was performed as follows: firstly, the participants were divided into three groups; then, we assigned one of the three technique to each group.

Before participants' implementation, the authors prepared several test cases for each task and gave a lecture to let them know how to use the tools of the techniques. In a given task, the participants read the specifications firstly and then they implemented code with an assigned tool. The participants decided keywords by themselves. We imposed no restriction on the number of code search. They were able to input any keywords as long as they wanted. Their screens were captured as motion pictures. If any one of the following conditions is satisfied, the task was regarded as finished.

1) For the non-GUI tasks, the implemented code passed all the test cases. Herein, a non-GUI task means a task that does not make any GUI component.
2) For the GUI tasks, the participants checked the GUI of their implementations by using the printed GUI examples for the tasks. If they regarded their GUI matched with the example, then the authors checked the behavior of the GUI by operating them.
3) The participants searched code fragments by using a specified tool 5 minutes, but they were not able to find reusable code. This condition was introduced because participants had only 20 minutes for each task. If they spend much time for searching code, it is difficult for them to complete tasks within the time limit even if they find reusable code.
4) Twenty minutes passed before any of the above 3 conditions was satisfied.

In the cases of (1) and (2), we regarded that the task was successfully finished. But in the cases of (3) and (4), we regarded that the task failed.

We imposed the two restrictions for the participants.

- They were able to use only one of the tools for a given task. Using two or three tools in the task was prohibited.
- They had to reuse code at least 1 line of code suggested from the tool.

B. Research Participants

The research participants in this experiment were two undergraduate students, five master's course students, two Ph.D. candidates in the department of computer science at Osaka University. All the participants had at least half year experience of Java programming and they had developed at least 5,000 lines of Java code in the past. Their experiences are on their exercise lessons and their research activities. The nine participants were divided into three groups with the consideration for their Java experiences so that the average programming skills of the three participants in the three groups are approximately the same.

C. Source Code Used for Making Database

Techniques-BC need a database to suggest code when they take keywords as their inputs (see Figure 2). In this experiment, we made a database from *UCI* [7]. The size of *UCI* dataset is huge: it includes 13,192 projects, 2,127,877 Java source files, and 20,449,896 methods.

We detected clones with Ishihara's technique [5], and then we detected block borders for each of the clones. The detected clones and its block borders were registered to the database. It took about 8 hours to finish the operations from *UCI* dataset. The database creation had been performed before we decided tasks because we needed to use the database to decide tasks.

D. Tasks

In this experiment, the authors prepared 9 tasks, each of which was implementing a Java method that met the given specifications. When we were deciding the tasks, we made sure that all the 3 techniques were able to suggest (a part of) reusable code if they took appropriate keywords as their inputs from research participants. Concretely speaking, we browsed source code of many clones in the database one by one, and if possible we created a task that the cloned code or its surrounding code was reusable on. On the other hand, Technique-A can suggest code in any method if they take keywords used in the method. By deciding tasks based on clones in the database, we were able to ensure that all the three techniques suggested reusable code for the tasks if they took appropriate keywords.

The participants were given signatures of the method and Javadoc comments including the specifications. They implemented the bodies of the methods. For non-GUI tasks, the authors prepared some test cases. For GUI tasks, the authors implemented the tasks and executed the programs. Then, the authors captured the screens and printed them for distributing to the participants. The whole bodies of the methods excepting signatures and Javadoc comments were regarded as participants' implementations.

The following list is the collections of the tasks that participants did in the experiment.

- **T1**: sorting the numbers included in a given string.
- **T2**: removing vowels and changing capitals into small letters in a given string.
- **T3**: implementing a simple window with three buttons (red, yellow, and blue) by using Swing, which is a Java GUI library. If a button is clicked, the background of the window is changed to the color.
- **T4**: storing a given string (the 1st parameter) into a file with a given name (the 2nd parameter).
- **T5**: sorting string in the alphabetical order.
- **T6**: performing a multiplication of two matrices
- **T7**: counting the number of words in a given text file.
- **T8**: implementing a simple window with three labels by using Swing. The strings on the labels are given by parameters.
- **T9**: performing exclusive-OR operation on given two byte arrays.

Each group did the tasks with the tools shown in Table I. Unfortunately, there were tasks where the participants failed implementations. Table II shows the number of failed tasks for each of the tools. There were 81 tasks in this experiment, and 34 out of them were failed.

E. Measures

In this experiment, we leveraged the following three measures to investigate to what extent each of the techniques was able to support code reuse. Those measures were computed by watching the motion pictures carefully.

- **time:** this is a difference between starting time and finishing time. Starting time is a clock time when a participant inputs a first character to the method body or a query to the tool for searching code. Finishing time is a clock time when all the test cases were passed. All the test cases were able to run by batched processing because the authors had built an Ant task for the test cases.
- **usage rate:** this measure represents how accurately the tool suggests reusable code to participants. This is a fraction where the denominator is the number of the suggested program statements and numerator is the number of reused program statements in the suggested ones. *Usage rate* can be represent with

TABLE I
ASSIGNMENT OF TOOLS FOR GROUPS

	T1, T2, T3	T4, T5, T6	T7, T8, T9
G1	Technique-A	Technique-B	Technique-C
G2	Technique-C	Technique-A	Technique-B
G3	Technique-B	Technique-C	Technique-A

TABLE II
NUMBER OF FAILED TASKS FOR EACH TOOL

Tool	T1	T2	T3	T4	T5	T6	T7	T8	T9
Technique-A	1	1	2	3	1	2	1	0	1
Technique-B	1	1	1	1	1	2	1	1	1
Technique-C	1	2	2	1	1	3	2	0	0

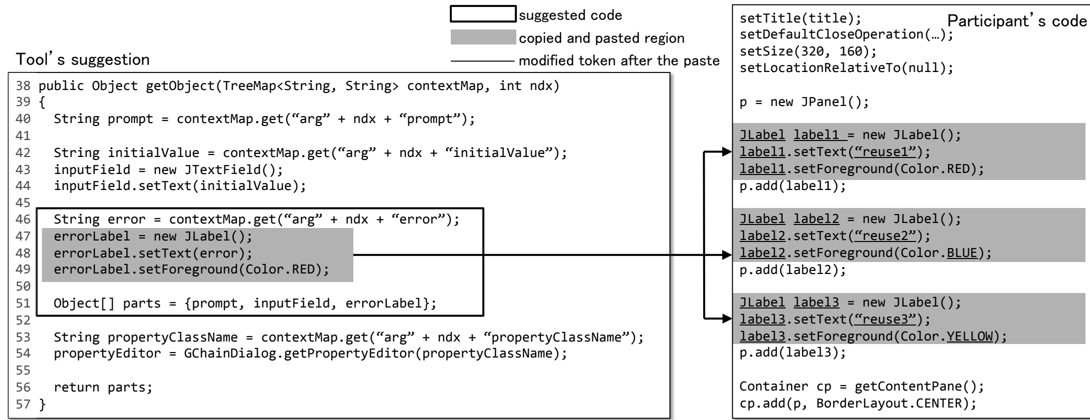


Fig. 3. Example of measures calculation

the following formula.

$$Usage\ Rate \quad = \quad \frac{\#\ of\ reused\ statements}{\#\ of\ suggested\ statements}$$

contribution rate: this measure represents how much code reuse contributes to the implementation. This is a fraction where the denominator is the number of all the program statements in participant's code and the numerator is the number of reused program statements in the code. *Contribution rate* can be represent with the following formula.

$$Contribution\ Rate \quad = \quad \frac{\#\ of\ reused\ statements}{\#\ of\ all\ statements}$$

Figure 3 is an example of computing *usage rate* and *contribution rate*. The left window is a code suggestion by the proposed technique, and the right window is a source file that the developer is implementing. The rectangle region in the left window is suggested code and the hatched region is reused code. The code is pasted in the developer's code three times. The developer implements 19 program statements and 9 of them come are reused code. In this case, *usage rate* becomes 0.6 (3/5) and *contribution rate* becomes 0.47 (9/19).

F. Results

Figure 4 shows the values of *usage rate* for all the tasks. The tasks where the height of bar graph is 0 are failed ones. The graph also shows the average of *usage rate* for each technique on each task. The right-most graph shows the average on all the tasks. The average of all the tasks shows that Technique-C marked a higher average than Techniques-AB.

Figure 5 shows the values of *contribution rate* for the tasks in the same fashion as Figure 4. The value of Technique-C is greater than the other two techniques in 4 out of the 9 tasks and the average of Technique-C on all the tasks is higher than Techniques-AB.

Figure 6 shows the values of *time* for the tasks. Regarding the average on all the tasks, Technique-C took less *time* than Technique-A but it took more *time* than Technique-B.

To wrap up, Technique-C suggested reusable code more accurately than the other two techniques and code reuse with Technique-C was the most helpful to implement tasks. However, the time for implementation with Technique-C was longer than Technique-B.

V. Discussion

In this section, we discuss some factors that affected the results shown in Figures 4, 5, and 6.

A. Long Implementation Time with Technique-C

While both the *usage rate* and *contribution rate* of Technique-C has higher values than the other two techniques, the *time* of Technique-C was not shortest in the three techniques. The reason was that the participants took longer time to operate the GUI of Technique-C than the GUIs of the other techniques. The GUI of Technique-C had buttons for changing suggested code ranges of a selected clone. There were some tasks where the participants changed the suggested code ranges by pushing the buttons plenty of times and looked for reusable code. On the other hand, the GUIs of Techniques-AB are very simple. They just suggest code of the selected code fragment. From this finding, we can say that default code range is very important to shorten the implementation time.

B. Some Cases Where Technique-C's Usage and Contribution Rates are Low

There are some tasks where both the *usage rate* and *contribution rate* of Technique-C was lower than the other two techniques. The reason was that the participants overlooked reusable code outside the suggested range.

Figure 7 shows such a case. The solid-rectangle area is suggested to the participant and the dotted-rectangle area is reusable code. The solid-rectangle area is the clone but the proposed technique excluded the available code by adjusting code range. The available code is just below the suggested area but the participant overlooked it. Even if available code existed in the viewer of Technique-C, it was overlooked because it was not highlighted correctly.

One promising way to avoid overlooking is using data dependencies between program statements [12]. In the case of Figure 7, the cloned code includes some variable declarations ("*converted*", "*spaces*", "*ats*", and "*linefeeds*"). Those variables are referenced in the reusable code. If the proposed technique adjusted code ranges with consideration for such data dependencies, the code area including the reusable code will get higher priority than the one excluding it. In such a situation, the participant will not overlook the available code.

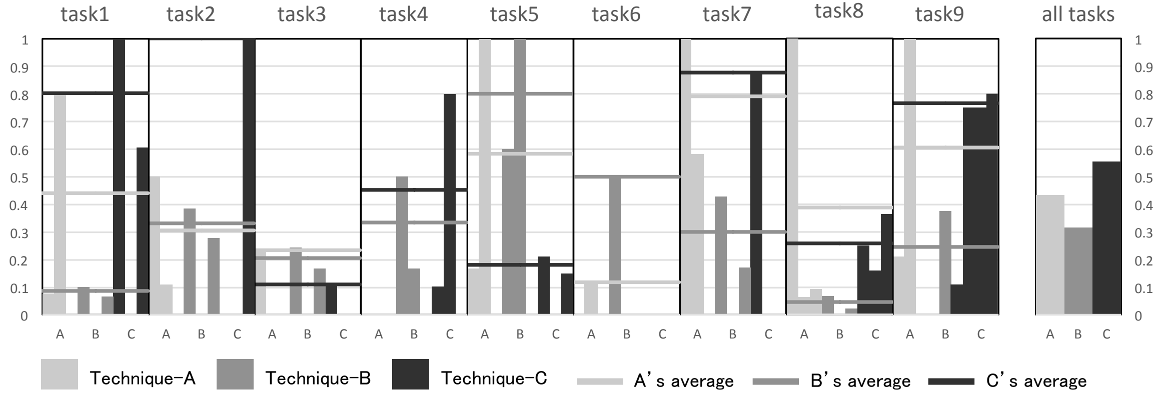


Fig. 4. Usage rate

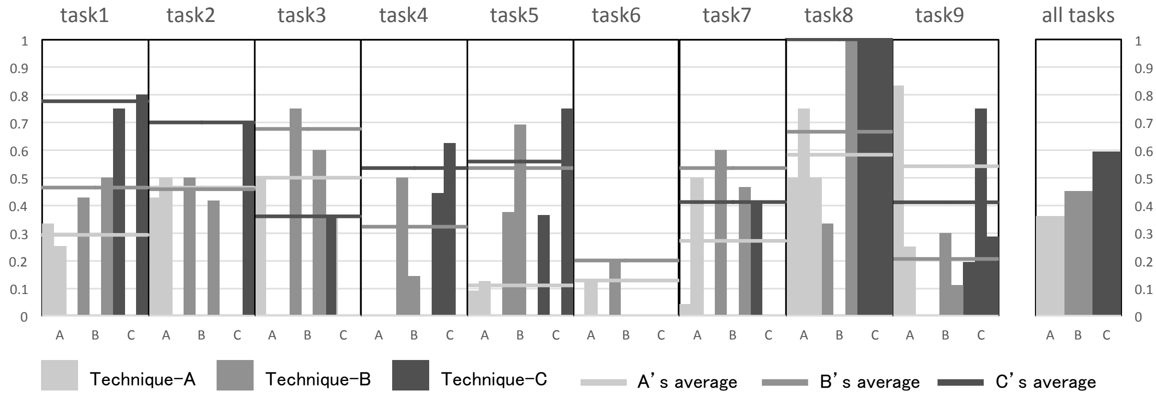


Fig. 5. Contribution rate

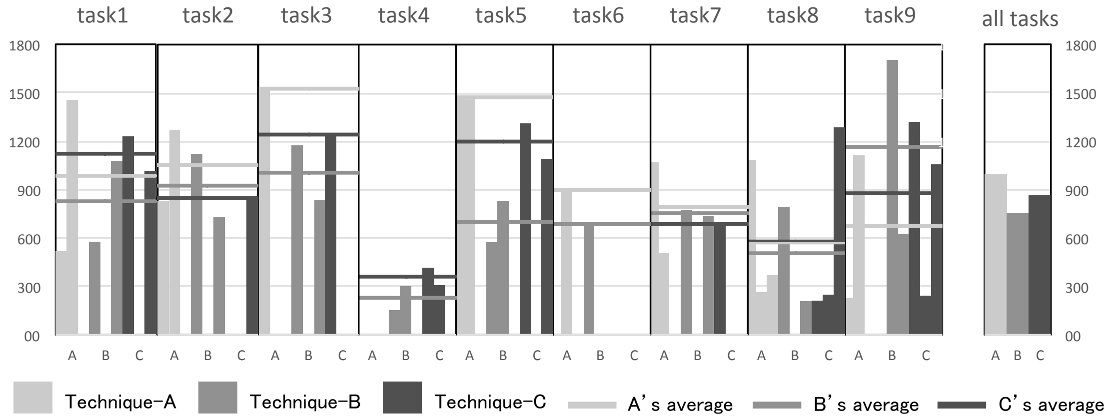


Fig. 6. Timing

C. Usage and Contribution Rates

Technique-A is the second place in *usage rate*, but it is worst in *contribution rate*. These facts mean that the block-level suggestion is more precisely providing reusable code than the reuse-level suggestion. But, the latter is more helpful to reuse code as much as possible than the former. Consequently, we can say that the hybrid technique has advantages of both the two techniques.

D. Developers' Faith In Code Suggestion

If a developer believes that suggested code is really reusable, she/he might insist on reusing it. In such case, implementation with code reuse may take longer time than implementing code from scratch. In this experiment, we imposed the participants to reuse at least 1 line of code for each task. This restriction was intended to collect enough data of code reuse, but this restriction can be interpreted as a big faith to code suggestion.

If a developer does not believe code suggestion so much, she/he will give up searching reusable code after she/he input different keywords a few times. There should have been some cases in the experiments where the participants would have been able to finish the tasks without code reuse.

VI. Threats to Validity

Here, we describe threats to validity in the experiment.

Research participants: all the research participants had at least half year experiences of Java programming and each of them had developed at least 5000 lines

```
…
public static String convertTabs(String text) {
  boolean preformatted = false;
  String converted = "";
  int spaces = 0;
  int ats = 0;
  int linefeeds = 0;
  for (int i = 0; i < text.length(); i++) {
    if (text.charAt(i) == ' ') {
      spaces++;
    …
    } else {
      spaces = 0;
      ats = 0;
    }
    …
  }
  return converted;
}
…
```

suggested code
reusable code cloned region

Fig. 7. A code suggestion where a developer overlooked reusable code

of code. If participants with different experiences had joined the experiment, we might have gotten a different result.

Experimental restriction: we imposed the participants to reuse code at least 1 line of suggested code in each task. This restriction was intended to collect enough data of code reuse. However, this restriction may have affected the values of three measures.

Clones: we detected clones for Techniques-BC. Those clones were detected by a detection tool, and so the detected clones must include false positives. We had not removed the false positives before the participants performed tasks. False positives are not reused code but they are anyway duplicated code. In the experiment, we do not think the presence of false positives had a negative impact on Techniques-BC.

Task: the three groups implemented 9 tasks in the different orders. The order of the tasks that they implemented might affect the three measures.

Tool: the three groups used the three techniques in the different orders. The order of using the techniques might impact on the experimental result.

Database: all the tools used in the experiments suggest code that had been registered to the database in advance. Consequently, the suggested code vary if the database has a different set of source files. Besides, code reuse generate clones between different software systems. If we reuse unreliable code, the pasted code may requires many modifications in the future. Thus, it is very important to build a database with reliable code.

VII. Conclustion

In this research, we conducted an experiment where three kinds of keyword-based code search techniques were compared: the first technique suggested code at method-level. The second one suggested code that had been reused in the past. The third technique is a hybrid of the first and second techniques. That is, the third technique suggests code based on the past reuse, but it adjusted suggested code range with consideration for the code blocks.

In the experiment, we compared the three technique from the three points. The first point is how accurately the techniques were able to suggest reusable code. The second point is to what extent code reuse contributes to the implementations. The third point is time required for implementing code with the techniques.

The hybrid technique marked the best scores on the first and second points, but it ranked the second in the implementation time. The experimental result indicates that the method-level code suggestions are able to provide reusable code more precisely than the reuse-level ones. On the other hand, the reuse-level code suggestions are more helpful to reuse larger code than the method-level ones.

In the future, we are going to introduce data dependencies for deciding appropriate code ranges for suggestion. This is because we found some cases where the code range adjustments were not appropriate and the participants overlooked reusable code. Introducing data dependencies will be able to recognize cohesive code chunk in the source code. Suggesting cohesive code chunk will be more helpful to assist code reuse.

References

[1] K. Inoue, R. Yokomori, T. Yamamoto, M. Matsushita, and S. Kusumoto, "Ranking Significance of Software Components Based on Use Relations," *IEEE Transactions on Software Engineering*, vol. 31, no. 3, pp. 213–225, 2005.

[2] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby, "A Search Engine for Finding Highly Relevant Applications," in *Proceedings of the 32nd International Conference on Software Engineering*, 2010, pp. 475–484.

[3] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: Finding Relevant Functions and Their Usage," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 111–120.

[4] M. Umarji, S. E. Sim, and C. V. Lopes, "Archetypal internet-scale source code searching," in *IFIP Advances in Information and Communication Technology*, B. Russo, E. Damiani, S. Hissam, B. Lundell, and G. Succi, Eds. Springer, 2008.

[5] T. Ishihara, K. Hotta, Y. Higo, and S. Kusumoto, "Reusing Reused Code," in *Proceedings of the 20th Working Conference on Reverse Engineering*, 2013, pp. 368–372.

[6] T. Ishihara, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto, "Inter-Project Functional Clone Detection Toward Building Libraries - An Empirical Study on 13,000 Projects," in *Proceedings of the 19th Working Conference on Reverse Engineering*, 2012, pp. 387–391.

[7] J. Ossher, H. Sajnani, and C. Lopes, "File Cloning in Open Source Java Projects: The Good, the Bad, and the Ugly," in *Proceedings of the 27th International Conference on Software Maintenance*, 2011, pp. 283–292.

[8] Y. Sasaki, T. Yamamoto, Y. Hayase, and K. Inoue, "Finding File Clones in FreeBSD Ports Collection," in *Proceedings of the 7th Working Conference on Mining Software Repositories*, 2010, pp. 102–105.

[9] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt, "Index-based Code Clone Detection: Incremental, Distributed, Scalable," in *Proceedings of the 26th International Conference on Software Maintenance*, 2010, pp. 1–9.

[10] D. Rattan, R. Bhatia, and M. Singh, "Software Clone Detection: A Systematic Review," *Information and Software Technology*, vol. 55, no. 7, pp. 1165–1199, 2013.

[11] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach," *Science of Computer Programming*, vol. 74, no. 7, pp. 470–495, 2009.

[12] S. P. Reiss, "Semantics-based Code Search," in *Proceedings of the 31st International Conference on Software Engineering*, 2009, pp. 243–253.