A Proposal of Code Completion Problem for Java Programming Learning Assistant System

Htoo Htoo Sandi Kyaw, Shwe Thinzar Aung, Hnin Aye Thant, and Nobuo Funabiki

Abstract To enhance Java programming educations in schools, we have developed a Web-based *Java Programming Learning Assistant System (JPLAS)* that provides a variety of programming assignments to cover different learning stages. For the first stage, JPLAS offers the *element fill-in-blank problem* where students study the grammar and code reading through filling the blank elements, composed of reserved words, identifiers, and control symbols, in a high-quality code. Unfortunately, it has been observed that students can fill the blanks without reading the code carefully, because the choice is limited for each blank. In this paper, we propose a *code completion problem* as a generalization of the element fill-in-blank problem. To solve the drawback, it does not explicitly show blank locations in the code, which expects students to carefully read the code to understand the grammar and code structure. The correctness of the answer is verified through string matching of each statement with the filled elements and the corresponding correct one. Besides, to encourage students to study *readable code* writing, the correct statement satisfies the *coding rules* including the spaces. For evaluations, we generated six code completion and element fill-in-blank problems respectively, and asked ten students in two universities to solve them. Their solution results show that the code completion problem is much harder than the element fill-in-blank problem, and requires far deeper code reading and understanding of coding rules.

Department of Information Science, University of Technology, Yatanarpon Cyber City, Myanmar, e-mail: htoohtoosandikyaw.hhsk@gmail.com
Department of Electrical and Communication Engineering, Okayama University, Okayama, Japan, e-mail: funabiki@okayama-u.ac.jp

1 Introduction

Recently, *Java* has been widely used in various practical application systems in societies and industries due to the high reliability, portability, and scalability. Java was selected as the most popular programming language in 2015 [1]. Therefore, strong demands have appeared from industries in expending Java programming educations. Correspondingly, a plenty of universities and professional schools are currently offering Java programming courses to meet this challenge. A typical Java programming course consists of grammar instructions in the class and programming exercises in computer operations.

To assist Java programming educations, we have developed a Web-based *Java Programming Learning Assistant System (JPLAS)* [2]-[6]. In the JPLAS server, we adopt *Linux* for the operating system, *Tomcat* for the Web application server, *JSP* [7] and *Java* for application programs, and *MySQL* for database. Currently, JPLAS provides the *element fill-in-blank problem*, the *value trace problem*, the *statement element fill-in-blank problem*, and the *code writing problem* to support self-studies of Java programming at various learning stages.

Among the four problems, the *element fill-in-blank problem* is designed for novice students. In this problem, a high-quality Java code with several blank elements is given to students. They are requested to fill the blanks that are shown explicitly in the code by typing the correct ones. An *element* represents the least unit in a code, and includes a *reserved word*, an *identifier*, and a *control symbol*. The correctness of each answer from a student is verified through *string matching* with the corresponding original element in the code. This original element must be the unique grammatically correct answer for the blank to avoid confusions by novice students. Thus, we proposed a graph-based *blank element selection algorithm* to select such elements as many as possible automatically. Besides, to let students answer the problems in JPLAS even if the Internet is not available, we have implemented the *offline answering function*. The hash function and the message authentication technique are used here to avoid illegally copying answers among students.

Unfortunately, in the element fill-in-blank problem, students can know where the blank elements exist, because they are shown in the code. Besides, each blank usually has a limited choice of elements for the correct answer. As a result, they can answer the problem without reading the code carefully to understand the grammar and code structure. Actually, a big drop exits from the value trace problem, a small variant of the element fill-in-blank problem, to the statement fill-in-blank problem that requires some programing from scrach. It indicates that the element fill-in-blank problem is not sufficient in improving the Java programming skill.

In this paper, we propose a *code completion problem* as a generalization of the element fill-in-blank problem. To solve the drawback, it does not explicitly show the locations of blank elements in the code, so that students need to find them in the code and complete the whole statements. Then, it is expected that students carefully read the code to understand the grammar and code structure. The correctness of the student answer is verified through *string matching* of each answer statement with the filled elements and the corresponding correct one in the original code. Besides,

to encourage students to study *readable code* writing [8], the correct statement follows the *coding rules* including the spaces. For this purpose, the *coding rule check function* [9] is applied for the original code for the problem before use.

For evaluations, we generated six code completion and element fill-in-blank problems and asked ten students in our universities to solve them. Their solution results show that this problem is much harder and requires far more code reading than the previous one. Also, when we statistically test the two types of problems by using t-test, the result shows that they are significantly different from each other which confirms that the proposed code completion problem fills the gap between the element fill-in-blank problem and statement fill-in-blank problem.

The rest of this paper is organized as follows: Section 2 reviews the previous work. Section 3 presents the code completion problem generation. Section 4 shows the evaluation results of the proposal. Finally, Section 5 concludes this paper with some future works.

2 Previous Works

In this section, we review our previous works related to this paper.

2.1 Element Fill-in-blank Problem

The *element fill-in-blank problem* intends for a student to learn the Java grammar and basic programming skills through *code reading* [6]. In this problem, a Java code with several blank elements is shown to a student, where he/she needs to fill in the blanks. This Java code should be of high-quality, most worth for code reading. An *element* is defined as the least unit of a code, such as a reserved word, an identifier, and a control symbol. A *reserved word* signifies a fixed sequence of characters that has been defined in the Java grammar to represent a specific function, which should be mastered first by the students. An *identifier* is a sequence of characters defined in the code by the author to represent a variable, a class, or a method. A *control symbol* in this paper indicates other grammar elements such as "." (dot), ":" (colon), ";" (semicolon) , "(,)"(bracket), "{, }" (curly bracket).

2.2 Blank Element Selection Algorithm

The *blank element selection algorithm* [6] uses the *constraint graph* that is generated to describe the constraints in the blank element selection. It generates the fill-in-blank problem through the following five steps:

1. *Vertex generation for constraint graph:* each vertex represents a candidate element for being blank.
2. *Edge generation for constraint graph:* an edge is generated between any pair of two vertices or elements that should not be blanked at the same time.
3. *Compatibility graph generation:* by taking the complement of the constraint graph, the *compatibility graph* is generated to represent the pairs of elements that can be blanked simultaneously.
4. *Clique extraction:* a maximal clique of the compatibility graph is generated by a simple greedy algorithm to find the maximal number of blank elements with unique answers from the given Java code. This greedy algorithm repeats to: 1) select the vertex that has the *largest degree* in the compatibility graph for the clique, 2) remove this vertex and its non-adjacent vertices from the graph, until the graph becomes null.
5. *Fill-in-blank problem generation:* the ratio between the number of blanks for control symbols and that for other elements is controlled.

2.3 Coding Rules

Coding rules [9] represents a set of rules or conventions for producing high quality source codes. By following coding rules, the uniformity of the code will be maintained, which enhances the readability, maintainability, and scalability. *Coding rules* consist of *naming rules*, *coding styles*, and *potential problems*.

1. *Naming Rules : Naming rules* describe the rules for finding the naming errors in the source code. Here, the *Camel case* [10] is adopted as the common Java naming rule. For an identifier representing a variable, a method, or a method argument, the top character should be a lower case, where the delimiter character between two words should be an upper case. For an identifier representing a class, both of them should be an upper case. For an identifier representing a constant, any character should be an upper case. A full-spelling English word should be used for an identifier name, whereas Japanese or Roman Japanese should not be used.
2. *Coding Styles : Coding styles* indicate the rules for detecting the layout errors in the source code. They include the position of an indent or a bracket, and the existence of a blank space. By following coding styles, the layout of a source code will become more consistent and readable.
3. *Potential Problems : Potential problems* illustrate the rules for discovering the portions in the source code that can pass the compilation but may induce functional errors or bugs with high possibility. They include a *dead code* and *overlapping codes*. A *dead code* represents the portion in the source code that is not executed at all, and *overlapping codes* represent the multiple portions in the source code that have similar structure and functions to each other. By solving potential problems, the code can not only improve the maintainability and scalability but speed up the execution.

2.4 Offline Answering Function for Element Fill-in-blank Problem

In this paper, we use the *offline answering function* for the *element fill-in-blank problem* [11], to ask students in Myanmar universities where the Internet connections are not stable, to solve the generated problems.

1. The teacher accesses to the JPLAS server, selects the element fill-in-blank problems for the assignments, and download the set of the necessary files for them on online.
2. The teacher distributes the assignment files to the students by using a file server or USB memories.
3. Students install the files in their PCs, and answer the problems in the assignments using Web browsers on offline, where the correctness of each answer is verified instantly at the PCs using the JavaScript program.
4. Students submit their final answering results to the teacher by using a file server or USB memories.
5. The teacher uploads the results from the students to the JPLAS server to manage them.

3 Proposal of Code Completion Problem

In this section, we present the code completion problem and its generation procedure.

3.1 Overview of Code Completion Problem

In a code completion problem, a Java source code with several missing elements is shown to the students without specifying their existences. Then, each student needs to find the locations of the missing elements in the code and fill the correct ones. The correctness of each answer from a student is verified through string matching with the corresponding original statement in the code. For studying readable code writing, the code must satisfy the coding rules composed of *naming rules*, *coding styles*, and *potential problems*.

The code completion problem is generated by a teacher through the following steps:

1. Select a Java source code from the website or textbook that is worth of reading to study the current topic.
2. Apply *naming rules test* in *coding rule check function* to the source code, and fix the errors if found.
3. Apply *coding styles test* to the code, and fix the errors if found.
4. Apply *potential problems test* to the code, and fix the errors if found.

5. Register each statement in the code as the correct answer unit in string matching.
6. Apply the *blank element selection algorithm* to select the blanks in the code.
7. Remove the selected blank elements from the source code for the *problem code*.

For the automatic execution of this procedure, we implemented the necessary programs by Java and the script by Bash.

3.2 Source Code Selection

To clarify the procedure, we explain the details by using the code for class *FibonacciCalculator*. This class generates *Fibonacci series*, 0,1,1,2,3,5,8,13,21,⋯, recursively, such that each subsequent number becomes the sum of the previous two numbers [12]. There are two base cases: Fibonacci(0) and Fibonacci(1).

3.3 Application of Coding Rule Check Function

The three tests in the coding rule check function are applied to the source code. **code 1** shows the corrected code that is used for the problem.

code 1

```
1  import java.math.BigInteger;
2  /**
3   * FibonacciCalculator
4   * @author student
5   */
6  public class FibonacciCalculator {
7     private static final BigInteger TWO = BigInteger.valueOf(2);
8     /**
9      * recursive fibonacci method
10     * @param number : to calculate fibonacci
11     * @return BigInteger : returns the fibonacci result
12     */
13    public static BigInteger calculateFibonacci(BigInteger number) {
14       if (number.equals(BigInteger.ZERO) || number.equals(BigInteger.ONE))
15          return number;
16       else
17          return calculateFibonacci(number.subtract(BigInteger.ONE))
18          .add(calculateFibonacci(number.subtract(TWO)));
19    }
20    /**
21     * displays the fibonacci values from 0−40
22     * @param args used
23     * @return Nothing
24     */
25    public static void main(final String[] args) {
26       for (int counter = 0; counter <= 40; counter++)
27          System.out.println("Fibonacci of " + counter + " is: "
28          + calculateFibonacci(BigInteger.valueOf(counter)));
29    }
30 }
```

3.4 Blank Element Selection and Removal

Then, the code which has passed the *coding rule check function* is applied to the *blank element selection algorithm* [6] to select blank elements from the code. Finally, the specified blank elements are removed to generate the problem code for the code completion problem. **code 2** shows the generated one.

code 2

```
1  import java.math.BigInteger;
2  /**
3   * FibonacciCalculator
4   * @author student
5   */
6  public FibonacciCalculator {
7     private BigInteger TWO = BigInteger.valueOf(2);
8     /**
9      * recursive fibonacci method
10     * @param number : to calculate fibonacci
11     * @return BigInteger : returns the fibonacci result
12     */
13    public BigInteger calculateFibonacci(BigInteger number) {
14       if (number.equals(BigInteger.ZERO) || .equals(.ONE))
15          return number;
16
17          calculateFibonacci(numbersubtract(BigInteger.ONE))
18          .add(calculateFibonacci(number.subtract()));
19    }
20    /**
21     * displays the fibonacci values from 0−40
22     * @param args used
23     * @return Nothing
24     */
25    public void main(final [] args) {
26       (int counter = 0; counter <= 40;++)
27          .out.("Fibonacci of " + + " is: "
28          + (BigInteger.valueOf(counter)));
29    }
30 }
```

3.5 Correct Answer

The correct answer is given for each whole statement that includes the spaces or tabs if there, so that the simple string matching can be used for correctness verification. For example, for line 26 in **code 2**, the correct answer is given in **code 3**, which includes two tabs before *for*, one space after *for*, and seven spaces inside of *()*. This strict style intends students to follow the coding rules.

code 3

```
1     for (int counter = 0; counter <= 40; counter++)
```

3.6 Problem Complexity Analysis

To mathematically analyze the solution difficulty difference between the element fill-in-blank problem and the code completion problem, we compare the total number of answer selections for all the blanks, if the solution for any blank is selected completely randomly for the problems that are generated from the same source code by selecting the same blank elements. This analysis assumes the following notations:

- s: number of statements in the code
- n: number of blank elements at one statement
- m: number of candidate elements to fill in each blank

In the *element fill-in-blank problem*, the answer for a blank can be selected independently. Thus, the total number of selections to fill in all the blanks, N_{ef}, is proportional to the number of blanks, which is given by $O(m \times n \times s)$. In the *code completion problem*, the answers for all the blanks at each statement must be selected at the same time. Thus, all the possible combinations of candidates for each statement must be considered, which is given by $O(m^n)$. As a result, the total number of selections to fill in all the blanks, N_{cc}, is $O(m^n \times s)$.

In the above example code, if $m = 50$ is assumed, N_{ef} is 800 (= 50×16), and N_{cc} is $135,250$ (= $50 \times 5 + 50^2 \times 4 + 50^3 \times 1$) where five statements have one blank, four statements have two blanks, and one statement has three blanks.

4 Evaluation

In this section, we evaluate the code completion problem in JPLAS through applications to ten students in two universities who have studied Java programming for more than one year.

4.1 Problem Assignments in Evaluation

In order to compare the solution performances by students between element fill-in-blank and code completion problems, we generated six pairs of both problems such that each pair has the similar difficulty with each other. The sources codes cover variable, array, collection, recursive, and polymorphism. Then, these problems are divided into two groups by selecting one problem for each of the six pairs, so that each group consists of three element fill-in-blank and code completion problems. Next, we asked ten students to solve them. These students are randomly divided into two groups, and the five students in each group are assigned one group of six problems.

Table 1: Correct solution rates (%).

	element fill-in-blank	code completion
ave.	96.6	81.6
SD	6.17	21.33

Table 2: T-test result.

item	value
observation	10
test statistics	2.96721213028104
test critical two-tail	2.262157163
Alpha-level	0.05
P(T≤t) two-tail	0.015773669

4.2 Solution Results by Students

Table 1 shows the solution results by students for the problems. Here, the average and the standard deviation on the correct solution rate (%) for each student are summarized. Clearly, the code completion problem exhibits the worse result than the element fill-in-blank, although their original source codes have similar difficulty and the same number of blanks is generated for the same source code.

Then, to confirm the abovementioned result statistically, we apply *T-test* that can determine if the two sets of data are significantly different from one another. *T-test* is one of statistical tests used for hypothesis testing. The *null hypothesis* is assumed to show no difference between them. Then, it is decided to accept or reject this null hypothesis. According to [13], there are two approaches to determine whether the null hypothesis is accepted or rejected. In the *critical value approach*, if *test statistics* is greater than *critical value*, the null hypothesis is rejected in favor of the alternative hypothesis. In the *P-value approach*, if *P-value* is less than or equal to *Alpha-level*, the null hypothesis is rejected. In this paper, to fully reject the null hypothesis, we adopt both approaches.

Table 2 shows the *T-test* result for solution results. The null-hypothesis is assumed that there is no difference between solution results of the element fill-in-blank problem and those of the code completion problem. In Table 2, *test statistics* is greater than *test critical*, which means the rejection of the null hypothesis. Also, *P-value* is smaller than *Alpha-level*, which means the rejection of it. Therefore, it is concluded that the solution result statistics of students are significantly different between the two problems, and the code completion problem is harder than the element fill-in-blank problem.

5 Conclusion

This paper proposed the *code completion problem* as a generalization of the element fill-in-blank problem for *Java Programming Learning Assistant System*. This problem does not explicitly show blank locations in the code, expecting students to carefully read the code to understand grammar and code structure. The correct-

ness of the answer is verified through string matching of the whole statement and the corresponding correct one. For evaluations, six code completion and element fill-in-blank problems using source codes for different topics were generated, and ten students in two universities solved them. The comparison of solution results between two problems shows the new problem is far more difficult. In future works, we will generate a variety of code complete problems using various codes and apply them to students in Java programming courses to verify the effectiveness.

References

1. Cass, S.: The 2015 Top Ten Programming Languages, http://spectrum.ieee.org/computing/software/the-2015-top-ten-programming-languages/?utm_so
2. Funabiki, N., Matsushima, Y., Nakanishi, T., Amano, N.: A Java programming learning assistant system using test-driven development method. Int. J. Comput. Sci. 40(1), 38-46 (2013)
3. Zaw, K. K., Funabiki, N., Kao, W.-C.: A proposal of value trace problem for algorithm code reading in Java programming learning assistant system. Inf. Eng. Express. 1(3), 9-18 (2015)
4. Ishihara, N., Funabiki, N., Kao, W.-C.: A proposal of statement fill-in-blank problem using program dependence graph in Java programming learning assistant system. Inf. Eng. Express. 1(3), 19-28 (2015)
5. Tana, Funabiki, N., Zaw, K. K., Ishihara, N., Matsumoto, S., Kao, W.-C.: A fill-in-blank problem workbook for Java programming learning assistant system. Int. J Web Inform. Sys. 13(2), 140-154 (2017)
6. Funabiki, N., Tana, Zaw, K. K., Ishihara, N. Kao, W.-C.: A graph-based blank element selection algorithm for fill-in-blank problems in Java programming learning assistant system. IAENG Int. J. Comput. Sci. 44(2), 247-260 (2017)
7. Murach J., Urban, M.: Murach ' s Java servlets and JSP (3rd ed.). Mike Murach & Associates, Inc. (2014)
8. Boswell D., Foucher, T.: The art of readable code. O ' Reilly (2011)
9. Funabiki, N., Ogawa, T., Ishihara, N., Kuribayashi, M., Kao, W.-C.: A proposal of coding rule learning function in Java programming learning assistant system. In: Proc. CISIS, 561-566 (2016)
10. CamelCase definition, http://searchsoa.techtarget.com/definition/CamelCase
11. Funabiki, N., Masaoka, H., Ishihara, N., Lai, I-W., Kao, W.-C.: Offline answering function for fill-in-blank problems in Java programming learning assistant system. In: Proc. ICCE-TW. 324-325 (2016)
12. Deitel P. J., Deitel, H. M.: Java: How to Program, 9th Edition. Prentice Hall (2011)
13. Hypothesis Testing, https://onlinecourses.science.psu.edu/statprogram/node/137