A Code Completion Problem in Java Programming Learning Assistant System

Htoo Htoo Sandi Kyaw, Su Sandy Wint, Nobuo Funabiki, and Wen-Chung Kao

***Abstract*—As an efficient object-oriented programming language, *Java* has been extensively used in a variety of applications around the world. To assist Java programming educations, we have developed a *Java Programming Learning Assistant System (JPLAS)*, which provides a great number of programming assignments to cover different levels of learning. For the first learning stage, JPLAS offers the *element fill-in-blank problem (EFP)* to study Java grammar through *code reading*. EFP asks students to fill in the blank elements in a given source code. However, EFP can be solved relatively easily, because the choice of the correct answer is limited for each explicit blank. In this paper, we propose a *code completion problem (CCP)* to overcome this drawback in EFP. To be specific, CCP does not explicitly show the locations of missing elements in the code. Instead, CCP will ask students to complete every statement in the code by filling in the correct elements at the correct locations. When the whole statement becomes equal to the original one, it is regarded as the correct answer. For evaluations, we generated CCP instances in both online/offline JPLAS, and asked university students from Myanmar, Japan, China, Indonesia, and Kenya to solve them. The results confirmed that CCP is harder than EFP, the *two-level marking* and the *hint function* are effective in improving solution performances of students, and the *difficulty level* for EFP is applicable in CCP.**

***Index Terms*—Java programming, JPLAS, code completion problem, blank element selection algorithm, hint function, difficulty level**

I. Introduction

Nowadays, *Java* has been widely applied in societies and industries due to its reliability, portability, and scalability. Moreover, Java was selected as the most popular programming language in 2019 [1]. Therefore, strong demands have appeared from IT industries in expanding Java programming educations. Correspondingly, a plenty of universities and professional schools are offering Java programming courses to meet this challenge. Generally, a Java programming course consists of grammar instructions and programming exercises.

To advance Java programming educations by assisting programming exercises, we have developed a *Java Programming Learning Assistant System (JPLAS)*. That is, JPLAS provides various programming exercise problems, such as the *element fill-in-blank problem (EFP)* [2][3], the *value trace problem (VTP)* [4], the *statement fill-in-blank problem (SFP)* [5], and the *code writing problem (CWP)* [6], to support self-studies of Java programming at different learning stages. For any exercise problem in JPLAS, the answer from a student is marked automatically using the program in the system.

Manuscript received August 21, 2019; revised April 17, 2020.

Htoo Htoo Sandi Kyaw, Su Sandy Wint, and Nobuo Funabiki are with the Department of Electrical and Communication Engineering, Okayama University, Okayama, Japan, e-mail:pxs93q36@s.okayama-u.ac.jp and funabiki@okayama-u.ac.jp.

Wen-Chung Kao is with the Department of Electrical Engineering, National Taiwan Normal University, Taipei, Taiwan, e-mail:jungkao@ntnu.edu.tw.

Originally, JPLAS has been implemented as an online Web application system, so that a teacher can easily manage the JPLAS studies while numerous students are using a single database. In this *online JPLAS*, the JPLAS server adopts *Linux* for the operating system, *Tomcat* for the Web application server, *JSP/Java* for application programs, and *MySQL* for the database [7]. A student may freely access to any problem in JPLAS and answer the questions using a Web browser. Figure 1 demonstrates the software platform for JPLAS.

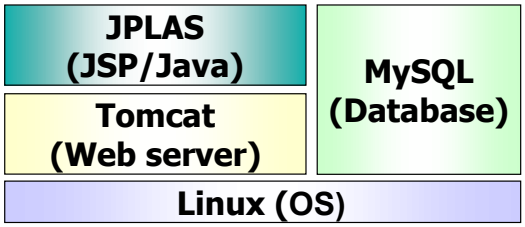

Fig. 1: JPLAS server platform.

In addition, the *offline answering function* has been implemented to allow students to answer the problems in JPLAS even if the Internet is unavailable [12]. For solving in this *offline JPLAS*, the problem assignment delivery and answer submission can be accomplished with a USB. A student can answer the questions in EFP and VTP using a Web browser, and those in SFP and CWP using *Eclipse*.

For the first learning stage, JPLAS offers the *element fill-in-blank problem (EFP)* to study Java grammar through *code reading*. In EFP, a Java source code with blank elements is given to students, where the blanks are shown explicitly in the code. Then, the students are requested to fill in the blanks by typing the correct elements. An *element* represents a least unit in a code, which 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.

In EFP, the original element in the source code must be the unique correct answer for each blank to avoid causing novice students confusions. Thus, we proposed a graph-based *blank element selection algorithm* to select such elements most automatically.

Nevertheless, in EFP, students will know where the blank elements exist in the code, because they are shown explicitly. Besides, each blank usually has a limited choice of elements for the correct answer. As a result, they may solve EFP without reading the code carefully or entirely understanding the grammar and code structure.

In this paper, we propose a *code completion problem (CCP)* [9] to overcome the drawback of EFP. In contrast to EFP, CCP does not show where the missing elements exist.

Then, it will ask students to complete each statement by filling in the correct elements at the ideal positions. When the whole statement becomes equal to the original in *string matching*, it is regarded as the correct answer.

For evaluations, we implemented the CCP instance generating program by modifying the program and the answering/-marking interface for EFP in both online/offline JPLAS [10]. Then, we generated six CCP instances using highly readable source codes [11] that were obtained by applying the *coding rule check function* [12] to the original codes, and asked 20 university students from Myanmar, Japan, China, Indonesia, and Kenya to solve them with the answer interface. The results confirmed that CCP is harder than EFP, and the *two-level marking* and the *hint function* are effective in improving solution performances of students for CCP.

The rest of this paper is organized as follows: Section II discusses related works in literature. Section III reviews our preliminary works to CCP. Section IV proposes CCP. Section V and VI present the implementation of CCP for offline and online respectively. Section VII evaluates CCP. Finally, Section VIII concludes this paper with future work.

II. Related Works In Literature

In this section, we discuss works in literature related to the automatic marking of student answers and the programming study through code reading.

In [13], Brusilovsky et al. developed the *QuizPACK* system that can generate parameterized exercises for the C language and automatically evaluate the correctness of student answers by comparing them to the correct ones provided by the teacher.

In [14], Ihantola et al. presented a systematic literature review for automated assessments of programming assignments, which includes major features and approaches from the pedagogical and technical points of view.

In [15], Jin et al. described a new technique to represent, classify, and use source codes written by novices as the base for the automatic hint generation for programming tutors, using the linkage graph representation. The linkage graph representation is used to record and reuse a student work as a domain model and use an overlay comparison between the in-progress work and the complete solution in a twist on the classic approach to the hint generation. The algorithm mentioned in this paper could generate hints over 80% of the problems that students will encounter.

In [16], Tung et al. developed a programming exercise management system, namely *Programming Learning Web (PLWeb)*, which consists of two main components, the server and the integrated development environment (IDE). The IDE in PLWeb, which is a modified version of *jEdit* [17], is used not only as the authoring tool for instructors to compose exercises but also as the novice-friendly editor for students to study programming and to submit their solutions. For the automatic marking of student programs, the IDE calls the pre-installed compiler in their computers to compile and execute them in the editing area.

In [18] Ichinco et al. performed an exploratory study of novices using examples to complete programming tasks. To analyze programming behaviors, they defined the "realization point" as the time when a participant discovered the crucial concept in an example. It is observed that a participant may take a long time to reach the realization point because the time he/she spent on executing the example code was longer than on reading the example code.

In [19], Staubitz et al. described how practical programming exercises could be provided, and examined the landscape of potentially helpful tools for automated assessments in massive open online courses. They discussed various automated assessment methods including the coding style assessment, which is similar to the coding rule learning function in our study.

In [20], Griffin discussed several lines of research, in order to support the premise that people learning programming can do more effectively and efficiently if they spend as much time on *deconstructing* codes as on writing codes. The term *deconstruction* is referred to as reading, tracing, and debugging a code.

In [21], Keuning et al. performed a systematic literature review of a variety of automated feedback generation tools to identify what kind of feedback is provided, what techniques are used to generate the feedback, how adaptable the feedback is, and how these tools are evaluated. This review is useful in implementing new automatic marking and hint functions.

In [22], Kakeshita et al. developed a programming education support tool called *Pgtracer*. *Pgtracer* utilizes fill-in-the-blank questions composed of a source code and a trace table. The blanks in the code and the trace table must be filled by the students to improve the code reading while solving the questions.

In [33], Yokata et al. developed a web application of an adaptive tutoring system with formative assessment. The formative assessment is generated using Educator's knowledge structure map, Learner's knowledge structure map and the relative distance which measures the difference between the machine generated solution and user's solution evaluated at some point. The system generates the mathematical expression as a problem and automatically check the learner's answer and give hint from the formative assessment to let the learner notice his/her mistakes.

In [34], Satoh et al. described a technique to make code/output correspondence in runtime of a given program for program understanding. Program understanding is important for novice programmers, because they need to read and understand sample program code as a previous step of writing codes. This technique is applied in program visualization tool which has two panes, a code reviewer and an output text area. The tool let the student to read the program code in a code reviewer pane and when a student clicks a statement in the program code, the tool highlights the output of that statement in the output text area.

In [35], Fei et al. analyzed various VB programming automatic scoring method. The method consists of dynamic evaluations of the function of event codes of students' programs and static evaluations of interfaces' designs, respectively. In a dynamic scoring, API Hook is applied to record the messages which are triggered by any human operators while the standard answer programs are being run; meanwhile, the system uses VB 6.0 to compile and run students' programs, and it will send the recorded messages to the running students' programs to drive them to run automatically, then sends out the results of students' programs,

which is at running, to the specified file with the output code, and match the results with the standard answers. In a static scoring, string matching algorithms are used to carry on form information matching, control information in a form matching, event keyword matching. The test results show that automatic scoring method results are stable, and they are consistent with the results of manual reviewers.

In [36], Yokota et al. developed a mathematic learning software named JCALC as an adaptive tutoring system. The educators' knowledge structure is utilized to diagnose a student's knowledge structure. For students to encourage continue studying by using JCALC, not only hint giving function but also learning management function is provided. Hint is first displayed by explaining what to do to solve the problem. Still if students cannot solve although referring to the first hint, second hint is provided by displaying the hint for which the experienced educator give.

In [37], Jain et al. developed an educational tool for understanding algorithm building and learning programming language. The tool provides an innovative and a unified graphical user interface for development of multimedia objects, educational games and applications. It also provides an innovative method for code generation to enable students to learn the basics of programming languages using drag-n-drop methods for image objects.

In [38], Lun et al. studied component interaction testing to examine quality of software architecture. Components are used to composite the software architecture and communicate with each other by their interfaces. The authors presented a set of component path coverage criteria for the test, and two algorithms to realize the automatic generation of the corresponding component paths according to the criteria and an experimental method to analyze the component interaction. This component interaction testing should be considered to obtain high quality source codes for CCP in future works.

In [39], Zaw et al. proposed the informative test code approach to the code writing problem in JPLAS for studying the three important object-oriented programming concepts, namely, *encapsulation*, *inheritance*, and *polymorphism*, which should be mastered by every student. The test code describes the necessary information including names, access modifiers, data types of member variables and methods, to implement the source code using the concepts. By writing a source code to pass the test code, a student can learn how to use them.

III. Preliminary Works

In this section, we review the *element definition*, the *blank element selection algorithm*, and the *coding rule check function* as our preliminary works to the code completion problem in this paper.

A. Element Definition

An *element* is defined as the least unit of a source code, which covers a reserved word, an identifier, a conditional operator, 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. It is expected that students should master the proper use of a *reserved word* in learning programming. An *identifier* is a sequence of characters defined in the code by the author to represent a *variable*, a *class*, or a *method*. A *conditional operator* is used in a conditional statement to determine the state. A *control symbol* in this paper indicates other grammar elements such as "." (dot), ":" (colon), ";" (semicolon) , " (,) " (bracket), "{, }" (curly bracket).

B. Blank Element Selection Algorithm

The *blank element selection algorithm* selects a maximal number of feasible blank elements from a given source code using a graph theory [2]. The first step of the algorithm generates the *compatibility graph* by selecting a candidate element for blank from the source code as a *vertex* and then, by connecting a pair of vertices with an *edge* if they can be blanked together. To fulfill this purpose, the conditions that a pair of elements cannot be blanked simultaneously have been defined.

The second step extracts a *maximal clique* [23] of the compatibility graph, in order to locate a maximal set of feasible blank elements. Empirically, it has been observed that EFP becomes more difficult as a larger number of elements are blanked [2]. By blanking a subset of the selected elements, a variety of fill-in-blank problems can be generated with different levels.

The details of the algorithm procedure are described as follows:

1) *Vertex generation for constraint graph:* each possible element for being blank is selected from the source code and is regarded as a *vertex* of a *constraint graph*.
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 to satisfy the uniqueness.
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 detect 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 of 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.

C. Coding Rule Check Function

The *coding rules* [12] represent a set of the rules or conventions for producing source codes of high quality. By following the coding rules, the uniformity of the code will be maintained, which enhances the readability, maintainability, and scalability. The *coding rules* consist of *naming rules*, *coding styles*, and *potential problems*. We have implemented the *coding rule check function* to automatically confirm whether the source code follows the *coding rules*, and suggest the code parts that do not follow them if available.

1) *Naming Rules : naming rules* describe the rules for identifying the naming errors in the source code. Here, the *Camel case* [24] 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. An English word should be used as 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 the 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 of 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 of the source code that is not executed at all, and *overlapping codes* signify 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.

IV. Proposal of Code Completion Problem

In this section, we propose the *code completion problem (CCP)* in JPLAS.

A. Overview of CCP

In CCP, a source code with several missing elements is shown to the students without specifying their existences. Then, a student needs to locate the missing elements in the code and fill in the correct ones there. The correctness of the answer from a student is verified by applying *string matching* to each statement in the answer to the corresponding original statement in the code. Only if the whole statement is matched, the answer for the statement will become correct. Furthermore, merely one incorrect element will result in the incorrect answer.

B. CCP Instance Generation Procedure

An instance of CCP can be generated through the following five steps:

1) Select a source code from a Website or a textbook that is worth of reading to study the current topic.
2) Apply the *coding rule check function* to the source code and fix the errors in the code if found.
3) Register each statement in the source code as the correct answer unit for string matching.
4) Apply the *blank element selection algorithm* to select the blank elements from the source code.
5) Remove the selected blank elements from the source code to generate the *problem code* in a new CCP instance.

```
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 }
```

Fig. 2: Code 1

By using the Java programs and the Bash script, this procedure can be executed automatically.

C. Example of CCP Instance Generation

Next, we explain the details of each step for the CCP instance generation using a sample code.

1) Source Code Selection: As a sample code, the class *FibonacciCalculator* is selected here. 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 [25]. There are two base cases: Fibonacci(0) and Fibonacci(1).

2) Application of Coding Rule Check Function: The coding rule check function is applied to the source code for class *FibonacciCalculator*. Figure 2 shows the corrected source code after the application.

3) Application of Blank Element Selection and Removal: Then, the *blank element selection algorithm* is applied to the corrected source code to select the blank elements. Finally, the selected blank elements are removed from the code as shown in Figure 3.

D. Problem Solution by Student

A student can solve CCP instances using the answering interface on a Web browser either offline or online, according to the availability of the Internet connection. In the interface, each input form corresponds to one statement in the source code. Then, the student needs to complete each statement by filling in all the missing elements at each input form.

```
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 }
```

Fig. 3: Code 2

E. Two-Level Answer Marking

The answer is marked through *string matching* of the whole statement on the JPLAS server using the Java program for online, or at the student browser using the JavaScript program for offline. The statement in the student answer and the corresponding statement in the original code are compared.

This marking is executed in two levels in our implementation. The *first level marking* is to compare the statements after removing the *spaces* and *tabs* from them [26]. In Java grammar, any number of spaces or tabs can be inserted in the statement. Besides, it is difficult to distinguish between multiple spaces and one tab. To avoid confusions of novice students, the first level marking does not consider the spaces and tabs in string matching.

However, to encourage a student to be aware of *readable code* which follows the coding rules, the locations of tabs and spaces are important. Thus, the *second level marking* is to compare the statements including the spaces and tabs. If the answer code contains a missing tab or space, or extra one, the warning feedback will be returned to the student in the answering interface.

V. Implementation for Offline JPLAS

In this section, we present the implementation of the code completion problem for offline JPLAS.

A. Operation Flow

Figure 4 illustrates the operation flow for CCP in offline JPLAS.

1) *CCP instance download*: a teacher accesses to the JPLAS server, selects the CCP instances for the assignment, and downloads the required files into the own PC on online.

TABLE I: Files for distribution.

File name	Outline
css	CSS files for Web browser
index.html	HTML file for Web browser
page.html	HTML file for correct answers
jplas2015.js	js file for reading the problem list
distinction.js	js file for checking the correctness of answer
jquery.js	js file for use of jQuery
sha256	js file for use of SHA256
storage.js	js file for Web storage

2) *Assignment distribution*: the teacher distributes the assignment files to the students by using a file server or USB memories.
3) *Assignment answering*: the students receive and install the files on their PCs, and answer the CCP instances in the assignment using Web browsers on offline, where the correctness of each answer is verified instantly at the browsers using the JavaScript program.
4) *Answering result submission*: the students submit their final answering results to the teacher by using a file server or USB memories.
5) *Answering result upload*: the teacher uploads the answering results from the students to the JPLAS server to manage them.

B. File Generation

Table I shows the necessary files with their specifications for CCP in offline JPLAS. These files are designed for view, marking, and answer storage.

C. Cheating Prevention

In offline JPLAS, the correct answers need to be distributed to the students so that their answers can be verified instantly on the browser. To prevent disclosing the correct answers, they will be distributed after taking hash values using *SHA256* [27]. In addition, to avoid generating the same hash values for the same correct answers, the assignment ID and the problem ID are concatenated with each correct answer before hashing. Then, the same correct answers for different blanks are converted to different hash values, which ensures the independence among blanks.

D. Problem Answer Interface

Figure 6 illustrates the answering interface for CCP on a Web browser in offline JPLAS. “Problem Code” shows the *problem code* of the CCP instance. “Answer Code” shows the answer forms where each line corresponds to one statement in the problem code that may involve missing elements. Then, a student is requested to complete every statement by filling in all the missing elements with each form.

With the default browser function, “tab” cannot be input to insert a tab in the statement, since it is used to move to another input form. However, the second-level marking for CCP checks the correctness of the tabs. Thus, in our implementation, the input of “tab” is made possible in the answering interface by the adopting *JavaScript* function shown in Figure 5 [26]:

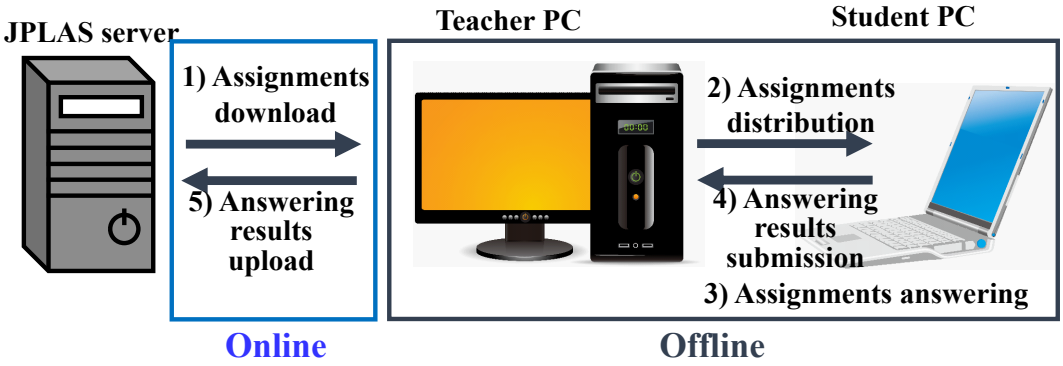

Fig. 4: Operation flow for CCP in offline JPLAS.

```
function enableTab(){
  var textareas = document.getElementsByName('intext');
  var count = textareas.length;
  for (var i = 0; i < count; i++) {
    textareas[i].onkeydown = function(e) {
    if(e.keyCode == 9 || e.which == 9) {
      e.preventDefault();
      var s = this.selectionStart;
      this.value = this.value.substring(0,this.selectionStart) + "
      " + this.value.substring
      (this.selectionEnd);
      this.selectionEnd = s+1;
      e.preventDefault();
      return false;
      }
    }
  }
}
```

Fig. 5: Code 3

E. Answer Marking

When a student clicks the answer button in the interface, the two-level marking is applied.

1) First-level Marking: First, the first-level marking applies to the answer. Here, after any space or tab is removed from the answer code, each statement is compared with the correct one without a space or tab. If they are different, the corresponding input form of the statement is highlighted by *pink* to suggest that at least one character in the answer statement is different from the correct one, and the marking is aborted. Otherwise, the following second-level marking is applied.

2) Second-level Marking: In the second-level marking, the whole statement, including spaces and tabs, will be compared between the answer code and the original code. If they are different, it is highlighted in *yellow*. Otherwise, the form is not highlighted at all.

F. Hint Function

To help a student who submits the answer for a plenty of times and still cannot solve a CCP instance, the *hint function* will be implemented. This function can be initiated after a certain number of incorrect submissions and be used by requesting the incorrect statement number [28].

The bottom two rows in Figure 6 show the input and output of the hint function. Here, the student inputs "3" as the incorrect statement number to request the hint. Then, the hints on the corresponding statement appear, such that the location of the missing element, *int*, is highlighted with *blue*, and the mistyped element, *tmp*, is highlighted with *purple*. It is noted that *tmp* should be *temp*. By referring to the highlighted elements, it is expected that the student can solve them.

G. Answer Result Submission

All the answering results of a student are kept in the browser using *localStorage* for *Web Storage* [29]. It can store up to 5MB data and keep it even after the shutdown of the PC. With this tool, students are allowed to keep an eye on their progress of studies for the assignments.

When a student submits the results, all the answering results in the Web storage are written into a text file. Then, the student submits this text file by using a USB memory or an E-mail to the teacher. To prevent the student from falsifying or plagiarizing the results of other students at submissions, the message authentication technique is adopted. Using SHA256 [27], the hash value of the answering result with the student ID is calculated before submission. Then, the coincidence between this hash value and that of the submitted data is evaluated.

VI. Implementation for Online JPLAS

In this section, we present the implementation of the code completion problem for online JPLAS.

A. Problem Data in Database

In online JPLAS, each instance in any problem type is managed by a single text field in the corresponding table in the *MySQL* database. This problem data field contains the markers shown in Table II to manage various data that are necessary for each problem type.

TABLE II: Markers in problem data field.

Marker	description
//@JPLAS answer	following paragraph contains correct answers
//@JPLAS statement	following paragraph contains problem statement
//@JPLAS output	following paragraph contains html format data for interface
null	problem code

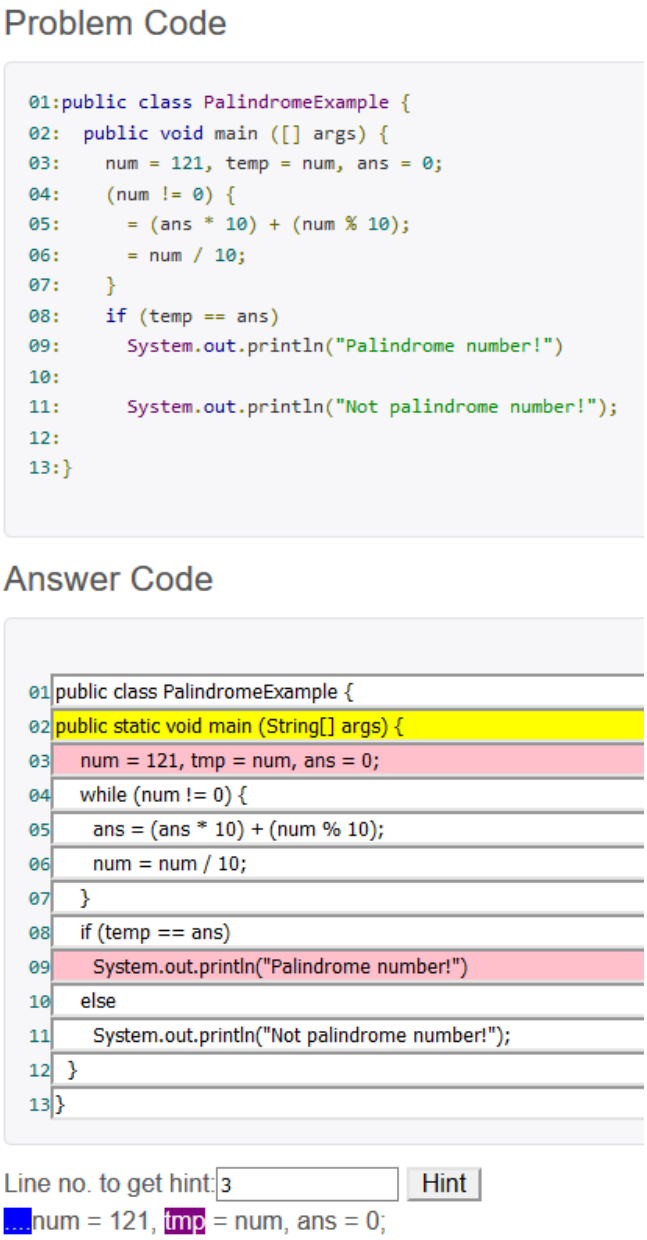

Fig. 6: Interface of problem answering in offline JPLAS.

B. Problem Answer Interface

The answer interface for offline JPLAS including the *hint function* is used for online JPLAS. Thus, a student can use both the offline and online JPLAS easily. The submitted answer is stored in the database directly.

C. Answer Marking

The two-level marking for offline JPLAS is also applied for online JPLAS. The implementation in the server adopts the *responsibility chain* design pattern. Here, first, the *string matching test* is performed at the two-level marking for CCP. Then, if this test is passed, the *compiling test* will be automatically initiated to examine the correctness of the answer code, because the first-level test excludes the spaces and tabs from it.

VII. Evaluations

In this section, we evaluate the *code completion problem (CCP)* for Java programing study in four steps. The first step evaluates the difficulty of CCP if compared with EFP. The second step evaluates the effectiveness of the two-level marking and the hint function implemented in CCP. The third step investigates the effect of CB for CCP instances. And finally, the fourth step observes how the readability of the source code affects the solution performance by students.

A. First-Step Evaluation

In the first-step evaluation, we excluded the *first-level marking* and the *hint function* from the answer interface for CCP in both offline/online JPLAS, because they were not implemented for EFP.

1) Problem Instances: To compare the solution performances by students between CCP and EFP, we generated six pairs of CCP and EFP instances such that the two instances in each pair appear to have the similar level of difficulty. Table III shows their overviews. The generated instances are divided into two groups by selecting one instance from each of the six pairs, so that each group consists of three CCP and EFP instances, respectively.

TABLE III: Instances for first-step evaluation.

pair ID	grammar topic	LOC		#of missing elements	
		CCP	EFP	CCP	EFP
1	variable	23	14	15	16
2	array	17	30	18	19
3	collection	30	17	19	18
4	recursive	14	23	16	15
5	method overloading	20	25	24	6
6	polymorphism	25	20	6	24

2) Solution Results by Students: Then, we asked 20 university students from Myanmar, Japan, China, Indonesia, and Kenya who have studied Java programming for more than one year to solve the generated instances in either group. That is, each student is randomly assigned one group, such that the equal number of students will solve one group.

Table IV demonstrates the average and the standard deviation (SD) of the correct solution rates (%) per student for CCP and EFP. Clearly, CCP exhibits the worse result than EFP, although the original source codes have similar difficulties and the same number of blanks was generated from the same source code. This result has confirmed that CCP is more difficult than EFP, which requires more careful code reading.

TABLE IV: Summary of correct solution rates for first-step evaluation.

correct rate (%)	CCP without improvements	EFP
ave.	82.46	93.92
SD	16.95	9.36

3) T-test *Verification:* To confirm the above-mentioned result statistically, *T-test* was applied here, which can determine if the two sets of data are significantly different from one another. *T-test* is one of the statistical tests used for hypothesis testing.

The *null hypothesis* is assumed to show no difference between the solution results for CCP and those for EFP. Then, it is decided to accept or reject this null hypothesis. According to [30], 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* will be 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, we adopt both approaches.

Table V indicates the *T-test* result. In Table V, *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 more challenging than the element fill-in-blank problem.

TABLE V: *T-test* result for first-step evaluation.

observation	20
test statistics	2.645645529
test critical two-tail	2.042272456
Alpha-level	0.05
P(T≤t) two-tail	0.012854482

4) Problem Complexity Analysis: To mathematically analyze the solution difficulty difference between CCP and EFP, the total number of possible answer selections for the same blanks generated from the same source code is compared, assuming that the solution for any blank is selected completely randomly. The following notations are used:

- N_{CCP}: total number of answer selections for CCP
- N_{EFP}: total number of answer selections for EFP
- 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 CCP, the answers for all the blanks at one statement must be selected at the same time. Thus, all the possible combinations of the candidates for each statement must be considered, which is given by $O(m^n)$. Hence, N_{CCP} is given by $O(m^n \times s)$.

In EFP, the answer for each blank can be selected independently. Thus, the total number of selections to fill in all the blanks is proportional to the total number of blanks. As a result, N_{EFP} is given by $O(m \times n \times s)$.

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

B. Second-Step Evaluation

In the second-step evaluation, we assessed the effects of the *first-level marking* and the *hint function* in the answer interface for CCP, which have been implemented to improve the correct solution rate.

1) Improved Answer Interface: The *first-level marking* compares the answer code with the original source code without considering the tabs and spaces. The marking in the first-step evaluation includes the tabs and spaces in the code. Thus, students must fill in the tabs and spaces at the same locations of the original code, which can make CCP too difficult for them.

The *hint function* suggests the locations of errors in the answer code when a student requests it. This function is available after he/she fails to answer it correctly at certain times. By referring to the hints, it is expected that a student can solve the CCP instance even if he/she cannot come up with a solution after submitting the answer several times.

2) Solution Results by Students: For the second-step evaluation, the same six CCP instances in the first-step evaluation were applied to 20 students who are different from the first-step evaluation. Table VI shows the average and the standard deviation (SD) of the correct solutions rates (%). Clearly the correct solution rate was improved here by adopting the *first-level marking* and the *hint function*.

TABLE VI: Summary of correct solution rates for second-step evaluation.

correct rate (%)	CCP with improvements
ave.	99.55
SD	1.49

3) Difficult CCP Instance: The correct solution rate for each student becomes close to 100% in the second-step evaluation. However, two students did not reach 100% at ID 4. To analyze the reason, this instance will use the source code in Figure 2 that generates *Fibonacci series*, 0, 1, 1, 2, 3, 5, 8, 13, 21, ..., recursively. In *Fibonacci series*, each subsequent number becomes the sum of the previous two numbers [25]. Besides, this source code uses *BigInteger* class that can be used for mathematical operations that involve big integers that can be outside of the range of a primitive data type [31]. Due to the use of the *recursive* call and *BigInteger* class, this problem becomes difficult for them. It will be critical to prepare more materials to study these hard topics using JPLAS including CCP, which will be in future works.

C. Third-Step Evaluation

In the third-step evaluation, we investigate how the *continuous blank number CB* [3] affects the solution performance of students. CB represents the maximum number of continuously blanked elements in a problem code. It is introduced to control the difficulty, because a problem code becomes harder in general when more blanked elements continue. The answer interface in the second step evaluation is used here.

1) Problem Assignment to Students: In this evaluation, we selected six source codes that are different from the previous evaluations but have similar difficulty, and generated problem codes with $CB = 3$. Then, we asked 11 students in Myanmar, Japan, Indonesia, and Kenya who have studied Java programming for more than one year, to solve each problem within five minutes. Table VII shows the problem ID (PID), the grammar topic, LOC, and the number of missing elements in each problem. When compared with Table III for $CB = 1$, the number of missing elements is much increased while LOC is similar.

2) Solution Performance by Students: Table VIII shows the average and the standard deviation (SD) of the correct solution rates (%) in the third-step evaluation. When compared with Table VI, the average is decreased and the SD is

TABLE VII: Instances for third-step evaluation.

problem ID	grammar topic	LOC	#of missing elements
1	wrapper class	23	41
2	class: method	23	33
3	exception	26	24
4	recursive	17	30
5	class: method	29	65
6	class: method	17	29

TABLE VIII: Summary of correct solution rates for third-step evaluation.

correct rate (%)	CCP with CB = 3
ave.	88%
SD	6.02

increased, which implies that the larger CB can increase the difficulty of CCP.

3) Correlation between Difficulty Level and Correct Rate: Then, we analyze the correlation between the *difficulty level* in [32] and the correct solution rate by the students in the second and third-step evaluations. *Difficulty level* has been introduced as the index to represent how difficult an *element fill-in-blank problem (EFP)* is to students. Figure 7 indicates that the *strong correlation* ($r = -0.76$) exists between the difficulty level and the correct rate. Thus, it is concluded that *difficulty level* is a proper index for CCP and a larger CB can increase the difficulty for CCP.

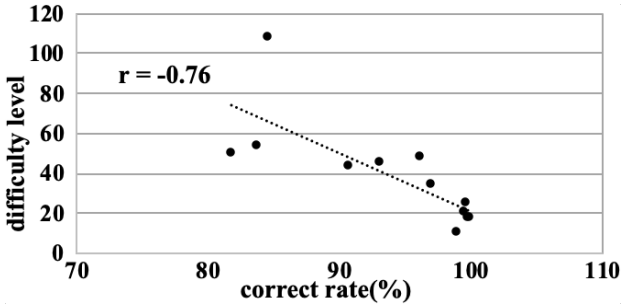

Fig. 7: Correlation between difficulty level and correct rate.

D. Fourth-Step Evaluation

In the fourth-step evaluation, we observe how the readability of the source code affects the solution performance by students in CCP. Again, the improved answer interface is adopted.

1) Problem Assignment to Students: As less-readable source codes for students, we newly choose six codes that may have an unfamiliar library class of "BigInteger", a large LOC, or very short identifier names such as "h", "m", or "s", and generate CCP instances in Table IX. Then, we asked 14 students in Myanmar, Japan, China, Indonesia, and Kenya to solve them.

TABLE IX: Instances for fourth-step evaluation.

problem ID	grammar topic	LOC	#of missing elements
1	method overloading	20	10
2	recursive	13	13
3	looping	9	7
4	exception	51	27
5	exception	47	25
6	recursive	24	12

TABLE X: Summary of correct solution rates for fourth-step evaluation.

correct rate (%)	CCP
ave.	75.57%
SD	9.74

2) Solution Performance by Students: Table X shows the result summary in the fourth-step evaluation. When compared with Tables VI and VIII, the solution performance by students becomes worst. Thus, the readability of source codes should be carefully examined at generating CCP instances for novice students.

3) Correlation between Difficulty Level and Correct Rate: Then, we analyze the correlation between the difficulty level and the correct solution rate in the fourth-step evaluation. Again, Figure 8 suggests the strong correlation ($r = -0.96$) between them.

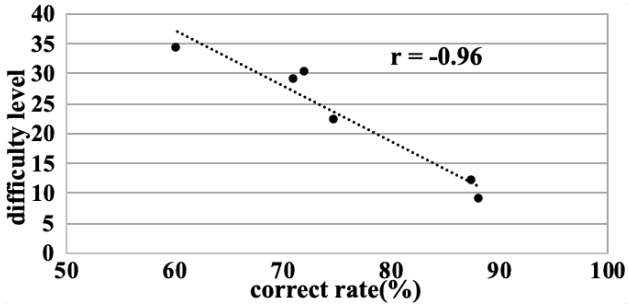

Fig. 8: Correlation between difficulty level and correct rate.

VIII. Conclusion

In this paper, we proposed the *code completion problem (CCP)* for *Java Programming Learning Assistant System (JPLAS)*. CCP asks students to complete the given source code by filling in the correct elements at the correct positions without explicitly showing the locations of the blanks, unlike the *element fill-in-blank problem (EFP)*. For evaluations, we generated CCP instances, and asked university students in Myanmar, Japan, China, Indonesia, and Kenya to solve them. The results confirmed the effectiveness of CCP with the *two-level marking* and the *hint function* in Java programming study, and the *difficulty level* for EFP in CCP. In future works, we will prepare materials to study hard grammar and programming topics using JPLAS, generate a variety of CCP instances, and apply them in Java programming courses.

References

[1] Interactive: The Top Programming Languages, IEEE Spectrum, https://spectrum.ieee.org/static/interactivethe-top-programming-languages-2019.

[2] Nobuo Funabiki, Tana, Khin Khin Zaw, Nobuya Ishihara, and Wen-Chung Kao, "A Graph-based Blank Element Selection Algorithm for Fill-in-blank Problems in Java Programming Learning Assistant System," IAENG International Journal of Computer Science, vol. 44, no. 2, pp247-260, 2017.

[3] Tana, Nobuo Funabiki, Khin Khin Zaw, Nobuya Ishihara, Shinpei Matsumoto, and Wen-Chung Kao, "A Fill-in-blank Problem Workbook for Java Programming Learning Assistant System," Internal Journal of Web Information Systems, vol. 13, no. 2, pp140-154, 2017.

[4] Khin Khin Zaw, Nobuo Funabiki, and Wen-Chung Kao, "A Proposal of Value Trace Problem for Algorithm Code Reading in Java Programming Learning Assistant System," Information Engineering Express, vol. 1, no. 3, pp9-18, 2015.

[5] Nobuya Ishihara, Nobuo Funabiki, and Wen-Chung Kao, "A Proposal of Statement Fill-in-blank Problem using Program Dependence Graph in Java Programming Learning Assistant System," Information Engineering Express, vol. 1, no. 3, pp19-28, 2015.

[6] Nobuo Funabiki, Yukiko Matsushima, Toru Nakanishi, Kan Watanabe and Noriko Amano, "A Java Programming Learning Assistant System using Test-driven Development Method, " IAENG International Journal of Computer Science, vol. 40, no.1, pp38-46, 2013.

[7] Nobuya Ishihara, Nobuo Funabiki, Minoru Kuribayashi, and Wen-Chung Kao, "A Software Architecture for Java Programming Learning Assistant System," International Journal of Computer & Software Engineering, vol. 2, no. 1, pp116-122, 2017.

[8] Nobuo Funabiki, Hiroki Masaoka, Nobuya Ishihara, I-Wei Lai, and Wen-Chung Kao, "Offline Answering Function for Fill-in-blank Problems in Java Programming Learning Assistant System," Proceedings of IEEE International Conference on Consumer Electronics-Taiwan 2016, 27-29 May, Nantou, Taiwan, pp324-325.

[9] Htoo Htoo Sandi Kyaw, Shwe Thinzar Aung, Hnin Aye Thant, and Nobuo Funabiki, "A Proposal of Code Completion Problem for Java Programming Learning Assistant System," Proceedings of The 10-th International Workshop on Virtual Environment and Network-Oriented Applications 2018, 4-6 July, 2018, Matsue, Japan, pp855-864.

[10] Htoo Htoo Sandi Kyaw, Nobuo Funabiki, Nobuya Ishihara, Minoru Kuribayashi, and Khin Khin Zaw, "Implementation of Code Completion Problem in Online Java Programming Learning Assistant System, " Proceedings of The Institute of Electronics, Information and Communication Engineering General Conference 2019, 19-22 March, Tokyo, Japan, pp93-94.

[11] Dustin Boswell and Trevor Foucher, The art of readable code, O'Reilly, 2011.

[12] Nobuo Funabiki, Takuya Ogawa, Nobuya Ishihara, Minoru Kuribayashi, and Wen-Chung Kao, "A Proposal of Coding Rule Learning Function in Java Programming Learning Assistant System," Proceedings of The 8th International Workshop on Virtual Environment and Network-Oriented Applications 2016, 6-8 July, Fukuoka, Japan, pp561-566.

[13] Peter Brusilovsky and Sergey Sosnovsky, "Individualized Exercises for Self-assessment of Programming Knowledge: An Evaluation of QuizPACK," Journal on Educational Resources in Computing, vol. 5, no. 6, 2005.

[14] Petri Ihantola, Tuukaa Ahoniemi, Ville Karavirta, and Otto Seppala, "Review of Recent Systems for Automatic Assessment of Programming Assignments," Proceedings of The 10th Koli Calling International Conference on Computing Education Research 2010, October, Koli, Finland, pp86-93.

[15] Wei Jin, Tiffany Barnes, John Stamper, Michael Eagle, Matthew W. Johnson, and Lorrie Lehmann, "Program Representation for Automatic Hint Generation for A Data-driven Novice Programming Tutor," Lecture Notes in Computer Science: Proceedings of International Conference on Intelligent Tutoring Systems 2012, 14-18 June, Chania, Greece, vol 7315, pp304-309.

[16] Sho-Huan Tung, Tsung-Te Lin, and Yen-Hung Lin, "An Exercise Management System for Teaching Programming," Journal of Software, vol. 8, no. 7, pp1718-1725, 2013.

[17] Slava Pestov, John Gellene, and Alan Ezust, "jEdit 4.5 user's guide," Sep. 2012, http://www.jedit.org/users-guide/index.html.

[18] Michelle Ichinco and Caitlin Kelleher, "Exploring Novice Programmer Example Use," Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing 2015, 18-22 October, Atlanta, USA, pp63-71.

[19] Thomas Staubitz, Hauke Klement, Jan Renz, Ralf Teusner, and Christoph Meinel, "Towards Practical Programming Exercises and Automated Assessment in Massive Open Online Courses," Proceedings of IEEE International Conference on Teaching, Assessment, Learning for Engineering 2015, 10-12 December, Zhuhai, China, pp23-30.

[20] Jean M. Griffin, "Learning by Taking Apart: Deconstructing Code by Reading, Tracing, and Debugging," Proceedings of The 17th Annual Conference on Information Technology Education 2016, September, Boston Massachusetts, USA, pp148-153.

[21] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren, "Towards a Systematic Review of Automated Feedback Generation for Programming Exercises," Proceedings of The 2016 ACM Conference on Innovation and Technology in Computer Science Education 2016, 9-13 July, Arequipa, Peru, pp41-46.

[22] Tetsuro Kakeshita and Miyuki Murata, "Application of Programming Education Support Tool pgtracer for Homework Assignment," International Journal of Learning Technologies and Learning Environments, vol. 1, no. 1, pp41-60, 2018.

[23] Michael R. Garey and David S. Johnson, Computers and intractability: A guide to the theory of NP-completeness, Freeman, New York, 1979.

[24] CamelCase definition, http://searchsoa.techtarget.com/definition/CamelCase.

[25] Paul J. Deitel and Harvey M. Deitel, Java: how to program, 9th Edition, Prentice Hall, 2011.

[26] Htoo Htoo Sandi Kyaw, Nobuo Funabiki, Minoru Kuribayashi, and Khin Khin Zaw, "Three Improvements for Code Completion Problem in Java Programming Learning Assistant System," Proceedings of Information Processing Society of Japan-Special Interest Group on Programming 2019, 13-14 April, Okayama, Japan.

[27] SHA-256 Cryptographic Hash Algorithm, http://www.movable-type.co.uk/scripts/sha256.html.

[28] Htoo Htoo Sandi Kyaw, Nobuo Funabiki, and Minoru Kuribayashi, "An Implementation of Hint Function for Code Completion Problem in Java Programming Learning Assistant System", Proceedings of Forum on Information Technology 2019, 3-5 September, Okayama, Japan, pp307-308.

[29] HTML5 Web Storage, http://www.w3schools.com/html/html5_webstorage.asp.

[30] Hypothesis Testing, https://onlinecourses.science.psu.edu/statprogram/node/137.

[31] BigInteger Class in Java, https://www.geeksforgeeks.org/biginteger-class-in-java/.

[32] Nobuo Funabiki, Shinpei Matsumoto, Su Sandy Wint, Minoru Kuribayashi, and Wen-Chung Kao, "A Proposal of Recommendation Function for Solving Element Fill-in-blank Problem in Java Programming Learning Assistant System," Proceedings of International Conference on Network-Based Information System 2019, 5-7 September, Oita, Japan, pp247-257.

[33] Hisashi Yokata, "On Developing an Adaptive Tutoring System with Formative Assessment for Mobile Learning," Proceedings of The World Congress on Engineering and Computer Science 2013, 23-25 October, San Francisco, USA, pp174-177.

[34] Miyu Satoh and Seikoh Nishita, "Correlating Program Code to Output for Supporting Program Understanding," Proceedings of The International MultiConference of Engineering and Computer Scientists 2019, 13-15 March, Hong Kong, pp180-183.

[35] Tan Peng Fei, Li Yan Heng and Zhang Chang Yun, "Research of VB Programming Automatic Scoring Method Based on the Windows API," Proceedings of The International MultiConference of Engineering and Computer Scientists 2012, 14-16 March, Hong Kong, pp235-239.

[36] Hisashi Yokota, "On Development of an Adaptive Tutoring System Utilizing Educator's knowledge Structure," Proceedings of The World Congress on Engineering and Computer Science 2011, 19-21 October, San Francisco, USA, pp260-264.

[37] Anshul K. Jain, Manik Singhal, and Manu Sheel Gupta, "Educational Tool for Understanding Algorithm Building and Learning Programming Languages," Proceedings of The World Congress on Engineering and Computer Science 2010, 20-22 October, San Francisco, USA, pp292-295.

[38] Lijun Lun, Xin Chi, and Hui Xu, "Testing Approach of Component Interaction for Software Architecture," IAENG International Journal of Computer Science, vol. 45, no. 2, pp353-363, 2018.

[39] Khin Khin Zaw, Win Zaw, Nobuo Funabiki, and Wen-Chung Kao, "An Informative Test Code Approach in Code Writing Problem for Three Object-oriented Programming Concepts in Java Programming Learning Assistant System," IAENG International Journal of Computer Science, vol.46, no. 3, pp445-453, 2019.

[40] Nobuo Funabiki, Tana, Khin Khin Zaw, Nobuya Ishihara, and Wen-Chung Kao, "Analysis of fill-in-blank problem solutions and extensions of blank element selection algorithm for Java programming learning assistant system," Proceedings of The World Congress on Engineering and Computer Science 2016, 19-21 October, San Francisco, USA, pp237-242.