A Concept of Value Trace Problem for Java Code Reading Education

Khin Khin Zaw* Nobuo Funabiki*
* Department of Electrical and Communication Engineering, Okayama University, Japan
p8lj1oji@s.okayama-u.ac.jp, funabiki@okayama-u.ac.jp

***Abstract*—To assist Java programming educations, we have developed a Web-based *Java Programming Learning Assistant System (JPLAS)*. JPLAS provides the *fill-in-blank problem* to help novice students studying Java programming at home, where some elements in a high-quality code including reserved words, identifiers, and control symbols are blanked to be filled by students. Unfortunately, this problem may be solved without reading out the algorithm in the code if students are familiar with Java grammar to some extent. In this paper, to further cultivate the code reading capability of students, we propose a concept of the *value trace problem* to ask the actual values of important variables in a code implementing some fundamental data structure or algorithm. A value trace problem can be generated by: 1) selecting a high-quality class code for the algorithm to be studied, 2) making the main class to instantiate the class in 1) if it does not contain the main method, 3) adding the functions to output variable values in questions into a text file, 4) preparing the input file to the code in 3) if necessary, 5) running this code to obtain the set of variable values in the text file, 6) blanking some values from the text file to be filled by students, and 7) uploading the final code, the blanked text file, and the correct answer file into the JPLAS server, and adding the brief description on the algorithm for a new assignment. To verify the feasibility of this concept, we manually generated five problems and asked four students with high Java programming skills in our group to solve them. Then, we analyzed the difficulty of the value trace problem for *Quick Sort*.**

***Keywords*-Java programming education, JPLAS, code reading, fill-in-blank problem, value trace problem, algorithm.**

I. Introduction

The programming language *Java* has high reliability and portability with excellent learning environments. Java has been extensively used for various practical systems in industries even at mission critical systems in large enterprises and small-sized embedded systems. Thus, the cultivation of Java programming engineers has been strongly demanded from industries. In fact, a lot of universities and professional schools are offering Java programming courses to deal with the demands. A Java programming course usually combines grammar instructions by classroom lectures and programming exercises by computer operations.

To help Java programming educations, we have developed the Web-based *Java Programming Learning Assistant System (JPLAS)* [1]- [4]. JPLAS provides the *element fill-in-blank problem* to support self-studies of students. In a fill-in-blank problem, a high-quality Java code with blanked elements is exhibited to students to fill the blanks with correct elements. An *element* represents the least unit in a Java code such as a *reserved word*, an *identifier*, and a *control symbol* [3]. A *reserved word* is a fixed sequence of characters that has been defined in Java grammar to represent the specified function, and should be mastered first by any student. An *identifier* is a sequence of characters defined in the code by the author to represent a variable, a class, and a method. A *control symbol* intends other elements such as "." (dot), ":" (colon), ";" (semicolon) , "(,)" (bracket), "{, }" (curly bracket). Any answer from a student is marked through comparison with the correct answer (string matching). The *fill-in-blank problem* intends for novice students to learn the Java grammar and basic programming.

To assist generating a feasible element fill-in-blank problem from a Java code such that the unique answer exists for any blank element, we have also proposed the *graph-based blank element selection algorithm*. This algorithm first generates a compatibility graph by selecting every candidate element in the code as a vertex, and connecting any pair of vertices by an edge if they can be blanked together. For this algorithm, we defined the conditions that a pair of elements can be blanked simultaneously. Then, the blank elements are selected by extracting a maximal clique of a compatibility of a graph [4].

Unfortunately, the element fill-in-blank problem can be solved mechanically without reading out the processing or algorithm in the Java code, if students are familiar with this problem and the Java grammar to some extent. Due to the unique answer constraint, only limited choices of elements may exist for many blanks. Actually, we have observed that as the number of solving element fill-in-blank problems increases, students could reach correct answers much faster than the beginning. Thus, a new problem that keeps the nature of filling blanks and marking answers through comparisons, but requires much deeper code reading is necessary for such students.

In this paper, we propose a concept of the *value trace problem* as a new type of a fill-in-blank problem. In this problem, students are questioned about actual values of important variables in a Java code implementing some algorithm such as the sorting. A value trace problem can be generated by: 1) selecting a high-quality class code for an algorithm, 2) making the main class to instantiate the class in 1) if it does not contain the main method, 3) adding the functions to write variable values in questions into a text file, 4) preparing the input file to the final code, 5) running the final code to obtain the set of variable values in the text file, 6) blanking some values from

978-1-4799-9958-3/15 $31.00 © 2015 IEEE
DOI 10.1109/IIAI-AAI.2015.188

the text file to be filled by students, and 7) uploading the final Java code, the blanked text file, and the correct answer file into the JPLAS server, adding the brief description on the algorithm for a new assignment. To verify the feasibility of this concept, we manually generated five problems, and asked four students who have high Java programming skills in our group to solve them. Then, we analyzed the difficulty of the value trace problem for *Quick Sort*.

The rest of this paper is organized as follows: Section II reviews the outline of JPLAS and the fill-in-blank problem. Sections III presents our concept of the value trace problem in JPLAS. Section IV shows simple evaluations of this concept. Section V provides the conclusion with future works of this paper.

II. JPLAS and Fill-in-blank Problem

In this section, we review the outline of JPLAS and the fill-in-blank problem.

A. Software Platform for JPLAS

Figure 1 illustrates the software platform for JPLAS. In the JPLAS server, we adopt the *Linux* for the operating system, *Tomcat* for the Web application server, *JSP/Servlet* for application programs, and *MySQL* for the database. The user can access to JPLAS through a Web browser.

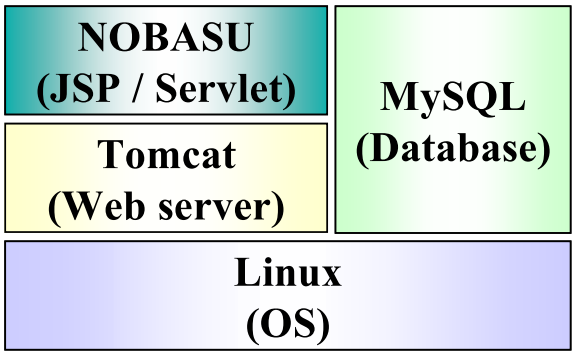

Fig. 1. Software platform for JPLAS.

B. Code Writing Problem in JPLAS

JPLAS provides the *code writing problem* in addition to the *element fill-in-blank problem* to support self-studies of students at various learning levels. The *code writing problem* in JPLAS intends for a student to learn writing a source code from scratch. The JPLAS function for this problem is implemented based on the *test-driven development (TDD) method* [5] using an open source framework *JUnit* [6]. It automatically tests the answer code on the server to verify the correctness when submitted from a student. Thus, a student can easily repeat the cycle of writing, testing, modifying, and resubmitting a code by him/herself. One problem of this function is that any student needs to write a code that can be tested by the *test code* that must be prepared by the teacher to test the correctness on *JUnit*. Thus, this function may not be suitable for a novice student who has just started learning Java programming.

C. Fill-in-blank Problem in JPLAS

1) Definitions of Terms: Here, we review definitions of terms for the *element fill-in-blank problem*. A *problem code* represents a Java source code that is used for a fill-in-blank problem. A *question* represents a blank to be filled inside the problem code. A *problem* consists of one problem code with several questions or blanks, their correct answers, and a comment on this problem. An *assignment* consists of a title, one or multiple problems, and a comment on the assignment. Usually, several assignments are given to students in each course, where JPLAS can support multiple courses at the same time. Any registered teacher in JPLAS can generate new problems and assignments using the shared database.

2) User Services: The fill-in-blank problem function consists of the *teacher service* and *student service*. The other process including the user registration uses the corresponding ones in the Java programming education assistant system. The teacher services includes the *Java code registration*, the *selection of reserved words and sample codes, the problem generation, assignment generation*, and *score reference*. The student services include the *assignment solution* and the *score reference*.

3) System Utilization Procedure: The utilization procedure for the element fill-in-blank problem in JPLAS is given as follows:

1) A teacher registers Java codes as *problem codes* in the database.
2) A teacher selects one code from the database.
3) A teacher selects the blank elements or *questions* from the code for a new *problem*.
4) A teacher generates and registers an *assignment* to his/her course by selecting problems and describing its title and comment.
5) A student selects an assignment.
6) A student submits answers to the problems in the assignment.
7) The server scores the answers and returns the results.
8) A student modifies incorrect answers and resubmits them to the server, if necessary.
9) A teacher and a student may refer to the score of the assignment.

III. Concept of Value Trace Problem

In this section, we present our concept of the value trace problems in JPLAS.

A. Generation Procedure of Value Trace Problem

The goal of the *value trace problem* in JPLAS for Java programming educations is to give students training opportunities of profoundly reading and analyzing a Java code that implements a fundamental data structure or an algorithm by asking to trace real values of important variables in the code. The *code reading* plays an essential role in writing high-quality codes for any programmer. It is also indispensable in modifying existing codes for some systems, which is common

in real worlds. A value trace problem is generated by a teacher with the following steps:

1) to select a high-quality class code for a fundamental data structure or an algorithm,
2) to make the main class to instantiate the class in 1) if it does not contain the main method,
3) to add the functions to write values of important variable in questions into a text file,
4) to prepare the input data file to be accessed by algorithm Java code and the teachers can modify the data in the input data file if necessary,
5) to run the algorithm Java code to obtain the set of variable values in the output text file,
6) to blank some values from the output text file to be filled by students,
7) to upload the final Java code, the blanked text file, and the correct answer file into the JPLAS server, and add the brief description on the algorithm and the problem for a new assignment.

In the following subsections, we describe the details of Steps 1), 2), 3), 6), and 7) using a Java code for *Shell Sort* [7] as a typical example. *Shell Sort* has been modified from *Insertion Sort* to speed up sorting unarranged input data. *Shell Sort* is a sequence of interleaved *Insertion Sort* procedures, where the interleave size is reduced after each pass until the size becomes 1, which identical to *Insertion Sort*.

B. Step 1): Java Code for Shell Sort

In this paper, we use the following Java code for *Shell Sort* in [7].

```
1:class Shellsort{
2:   public static void sort(int[] a){
3:     int n= a.length;
4:      int h;
5:      //find initial value of h
6:      for(h=n/2;h>0;h/=2){
7:          // sort according to h value
8:          for (int i=h; i<n; i++){
9:              int j=i;
10:              while(j>=h&&a[j-h]>a[j]{
11:                  int temp=a[j];
12:                  a[j]=a[j-h];
13:                  a[j-h]=temp;
14:                   j-=h;
15:              }
16:          }
17:      }
18:   }
19:}
```

C. Step 2): Generation of Main Class

Some Java codes may not include *main method* and contain only *class* like the code for *Shell Sort* in Section III-C. For convenience, we call it *algorithm class*. Then, we need to generate *main class* to contain *main method* to instantiate the class to run the code. In this paper, we manually generated *main class* by following the description in [8]. In this *main class*, the input data is also described, so that the input file in Step 4) can be skipped.

```
1: class Shellsort_Main{
2:   public static void main(String args[]){
3:     int[] a= new int[] {12,1,3,4,11};
4:     Shellsort.sort(a);
5:   }
6:}
```

D. Step 3): Adding Output Functions

An algorithm is regarded as a well-defined computational procedure that takes some input values, and produces some output values after a sequence of computational steps that transform the input values into the output values [9]. Thus, we believe that students can study and understand the main procedure of the fundamental data structure or the algorithm in the Java code by tracing the values of some important variables during transformations the input values to the output values. Then, it is necessary to add the necessary functions of writing such variable values into a text file in *main class* and *algorithm class*.

In this subsection, we explain the procedure here using *Shell Sort*. *Shell Sort* repeatedly divides the entire collection of data to be sorted into a set of sub-collections by taking every h-th element and applies *Insertion Sort* for each sub-collection. Here, h represents the interleave size. According to this algorithm, we regard that the following variables are essential for understanding it and should be traced at each iteration by students in the *value trace problem* for *Shell Sort*:

(1) the interleave size h,
(2) the application results of *Insertion Sort*.

Then, we modify the *algorithm class* by adding the functions to output the necessary values of them into a text file as follows:

```
1:import java.util.ArrayList;
2:class Shellsort{
3:public static void sort(int[] a){
4:public static ArrayList lis=new ArrayList();
5:  int n= a.length;
6:  int h;
7:  //find initial value of h
8:  for(h=n/2;h>0;h/=2){
9:      // sort according to h value
10:     for (int i=h; i<n; i++){
11:         int j=i;
12:          while(j>=h&&a[j-h]>a[j]{
13:              int temp=a[j];
14:              a[j]=a[j-h];
15:              a[j-h]=temp;
16:              j-=h;
17:           }
18:             int[]b=a.clone();
19:             lis.add(b);
20:      }
21: System.out.println("The value of h = "+h);
22:  for(int k = 0 ; k < lis.size() ; k++ ){
23:     for(int j=0; j < Shellsort.lis.
           get(k).length; j++){
24:        System.out.print(""+Shellsort.
         lis.get(k)[j]+",");
25:     }
26:     System.out.println();
27:   }
```

```
28:   list.clear();
29:  }
30: }
31:}
```

E. Step 6): Blanking Values for Problem Generation

In this subsection, we discuss how to make the blanking some values from the text file to be filled by students. The complete output text file for the value trace problem is as follows:

```
1://output result
2:  value of h = 2
3:value of a[]
4:3,1,12,4,11,
5:3,1,12,4,11,
6:3,1,11,4,12,
7:value of h =1
8:value of a[]
9:1,3,11,4,12,
10:1,3,11,4,12,
11:1,3,4,11,12,
12:1,3,4,11,12,
```

In *Shell Sort*, the interleave size h is the most important parameter. For each h, *Insertion Sort* is applied for the data set with this interleave size. Thus, to understand the code for *Shell Sort*, students should trace the values of h from the initial one to the final and the data sorting results for each h. To let students trace the data sorting results, we actually blank the whole line in the text file where some data is changed from the previous line, in addition to the first line. Then, we blank the corresponding lines in the text file as follows:

```
1://output result
2:  value of h = _1_
3:value of a[]
4:_2_, _3_ ,_4_, _5_ , _6_ ,
5:  3,  1,  12,  4,  11,
6:_7_,_8_,_9_,_10_,_11_,
7:value of h =_12_
8:value of a[]
9:_13,_14_, _15_ , _16_, _17_,
10: 1,  3 ,   11,   4,   12,
11:_18_,_19_,_20_,_21_,_22_ ,
12:  1,   3,   4,  11,  12,
```

Figure 2 illustrates the user interface of selecting the text file and blanking the output data for a value trace problem.

F. Step 7): Generating Assignment

After we prepared the final Java codes of *main class* and *algorithm class*, the blanked text file, and the correct answer file, we upload them to the JPLAS server using the existing function. Then, we add brief description on the algorithm and the problem to help students to understand it.

Figure 3 illustrates the user interface for generated blanked function, where the assignment title, the comment, the problem code, the answer forms, and the answering buttons are shown. The problem code contains all the necessary information including the final Java codes and the blanked text file.

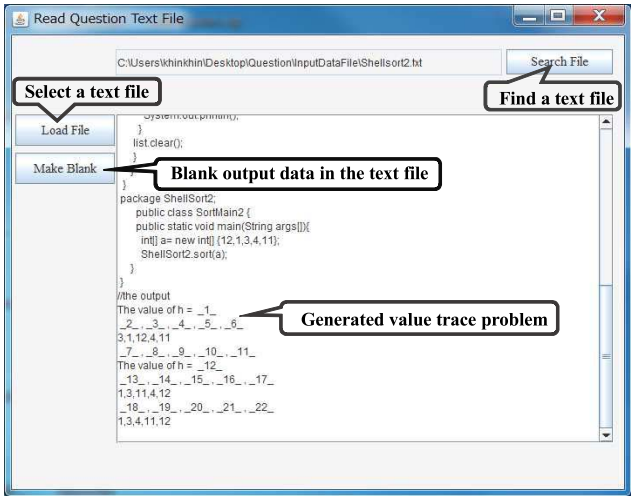

Fig. 2. Interface for blanking output data.

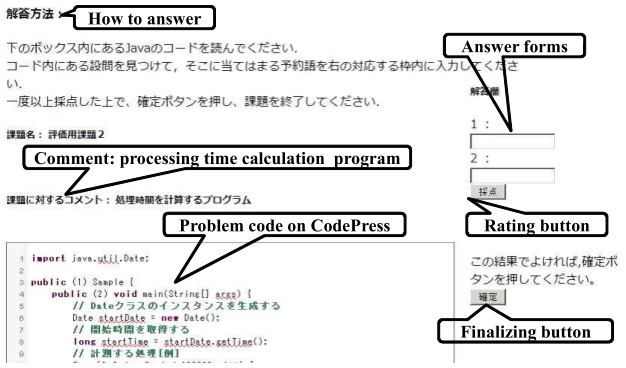

Fig. 3. Interface for assignment answering

IV. Evaluation

In this section, we evaluate the proposed concept of the value trace problem in JPLAS.

A. Analysis of Generated Problem

For this evaluation, we generated the value trace problem for *Shell Sort*. The generated code file for this evaluation is shown here. The codes from line 1 to line 31 describe *algorithm class* with added output functions to a text file. The codes from line 32 to line 36 describe *main class*. The codes from line 38 to line 49 describe the problem to be solved by students. There are 17 blanks in this problem where students have to fill the correct numbers by reading and understanding the codes. Here, actually two different values appear for h.

```
1:import java.util.ArrayList;
2:class Shellsort{
3:public static void sort(int[] a){
4:public static ArrayList lis=new ArrayList();
5:  int n= a.length;
6:  int h;
7:  //find initial value of h
```

```
8:  for(h=n/2;h>0;h/=2){
9:      // sort according to h value
10:     for (int i=h; i<n; i++){
11:         int j=i;
12:          while(j>=h&&a[j-h]>a[j]{
13:              int temp=a[j];
14:              a[j]=a[j-h];
15:              a[j-h]=temp;
16:              j-=h;
17:           }
18:             int[]b=a.clone();
19:             lis.add(b);
20:      }
21: System.out.println("The value of h="+h);
22:  for(int k = 0 ; k < lis.size() ; k++ ){
23:     for(int j=0; j < Shellsort.lis.
           get(k).length; j++){
24:        System.out.print(""+Shellsort.
          lis.get(k)[j]+",");
25:     }
26:     System.out.println();
27:   }
28:   lis.clear();
29:  }
30: }
31:}
32:class Shellsort_Main{
33:  public static void main(String args[]){
34:     int[] a= new int[] {12,1,3,4,11};
35:     Shellsort.sort(a);
36   }
37:}
38://output result
39:  value of h = _1_
40:value of a[]
41:_2_, _3_ ,_4_, _5_ , _6_ ,
42: 3 ,1, 12,4, 11 ,
43:_7_,_8_,_9_,_10_,_11_,
44:value of h =_12_
45:value of a[]
46:_13,_14_, _15_ , _16_, _17_,
47: 1,  3  ,  11 ,   4 ,  12  ,
48:_18_,_19_,_20_,_21_,_22_ ,
49:  1 ,  3 ,  4  ,  11 , 12 ,
```

B. Solutions of Five Problems

Then, we generated five value trace problems using Java codes for *Stack*, *Queue*, *Bubble Sort*, *Shell Sort* and *Quick Sort* [11], and asked four students in our group who have sufficient skills and knowledge for Java programming. After solving all the problems correctly, we asked them to comment the difficulty of each problem with five levels where level (1) is the easiest and level (5) is the most difficult, and the approximate time spent to solve each problem with four levels where (1) is less than 10 min, (2) is about 15 min, (3) is about 20 min. and (4) is longer than 25 min. Table I show the results.

Then, we checked the number of submissions of their answers for each problem by each student. Table II shows that they generally submitted answers more times for P4 and P5, which supports the result in Table I.

C. Difficulty of Quick Sort

The results in the previous subsection show that the value trace problem for *Quick Sort* is the most difficult. Here, we analyze the reason.

TABLE I
QUESTIONNAIRE RESULTS.

problem ID	data structure or algorithm	# of blanks	difficulty level	time level
P1	Queue	20	(1)	(1)
P2	Stack	16	(1)	(1)
P3	Bubble sort	15	(1)	(1)
P4	Shell Sort	22	(1)	(3)
P5	Quick Sort	34	(5)	(4)

TABLE II
RECORDS OF STUDENTS

student ID	P1	P2	P3	P4	P5
S1	1	1	5	3	14
S2	12	5	2	24	20
S3	1	3	1	10	19
S4	3	3	6	9	17

Quick sort employs the divide-and-conquer strategy. It starts by picking an element from the data list as the *pivot*. Then, it reorders the data list so that all the elements with values less than the *pivot* come before the *pivot* and the other elements come after it, which is often called *partitioning*. Then, it recursively applies the same procedure to the sub-lists at the left side and the right side of the pivot, until the whole list is sorted.

In the following value trace problem for *Quick Sort*. The codes from line 1 to line 38 describe *algorithm class* with added output functions to a text file, the codes from line 39 to line 45 describe *main class*, and the codes from line 46 to line 60 describe the problems to be solved by students. 30 blanks are prepared in this problem for students to fill the correct values by reading and understanding the codes, where *p* represents the variable for *pivot*.

```
1:class Quicksort{
2:public static int partition(int array[],
   int left, int right){
3:     int p,tmp,i,j;
4:     p=array[left];
5:     i=left;
6:     j=right+1;
7:   System.out.println("The pivot :" + p);
8:   for(;;){
9:    while(array[++i]<p) if(i >= right) break;
10:   while(array[--j]>p) if(j <=left) break;
11:        if (i >= j) break;
12:        tmp=array[i] ;
13:        array[i]=array[j];
14:        array[j]=tmp;
15:   }
16:     if(j!=left){
17:     tmp=array[left];
18:     array[left]=array[j];
19:     array[j]=tmp;
20:     }
21:    System.out.print("The output is ");
22:    for(int k:array){
```

```
23:      System.out.print(k);
24:      System.out.print(" ");
25:    }
26:   System.out.println();
27:   return j;
28: }
29: public static void quicksort(int a[],
        int left, int right){
30:    int i;
31:    if(right>left){
32:      i=partition(a,left,right);
33:      System.out.println();
34:      quicksort(a,left,i-1);
35:      quicksort(a,i+1,right);
36:    }
37: }
38:}
39:public class Quickmain {
40: public static void main (String[] args) {
41:  int [] arr={65,70,75,80,85,60,55,50,45};
42:  QuickSort.quicksort(arr,0,arr.length-1);
43:
44:  }
45:}
46://the output is
47:The pivot  : _1_
48:The output is _2_ ,_3_ ,_4_ ,_5_,_6_,
                  _7_ ,_8_ ,_9_ ,_10_
49:The pivot  : _11_
50:The output is _12_,_13_,_14_, _15_,_16_,
                  _17_ ,_18_ ,_19_ ,_20_
51:The pivot:_21_
52:The output is 50 , 45 , 55 , 60 , 65 ,
                  85 , 80 , 75 , 70
53::The pivot:_22_
54:The output is 45 , 50 , 55 , 60 , 65 ,
                  85 , 80 , 75 ,70
55:The pivot :_23_
56:The output is _24_,_25_,_26_,_27_,_28_,
                 _29_ ,_30_ , _31_ ,_32_
57:The pivot:_33_
58:The output is 45 , 50 , 55 , 60 , 65 ,
                  70,  80 ,75  , 85 ,
59:The pivot  :_34_
60:The output is 45,  50 , 55 , 60 , 65 ,
                  70,  75 ,  80 , 85 ,
```

In *Quick Sort*, the pivot p is the most important parameter. For each p, the data arrangement is applied for each data set. Thus, to understand the code for *Quick Sort*, students should trace the values of p from the initial one to the final and the data arrangement results for each p. Thus, we blanked the corresponding lines that in the output file that is shown as follows:

```
1://the output is
2:The pivot :65
3:The output is 60, 45, 50, 55, 65,
                  85, 80, 75, 70
4:The pivot :60
5:The output is 55, 45, 50, 60, 65,
                  85, 80, 75, 70
6:The pivot :55
7:The output is 50, 45, 55, 60, 65 ,
                  85, 80, 75, 70
8:The pivot :50
10:The output is 45, 50, 55, 60, 65,
                   85, 80, 75,  70,

11:The pivot :85
12:The output is 45, 50, 55, 60, 65,
                   70, 80, 75, 85,
13:The pivot :70
14:The output is 45, 50, 55, 60, 65,
                    70, 80, 75, 85,
15:The pivot :80
16:The output is 45, 50, 55, 60, 65,
                    70, 75, 80, 85,
```

V. Conclusion

In this paper, we proposed a concept of the *value trace problem* for algorithm Java code reading in the *Java Programming Learning Assistant System (JPLAS)*. For simple evaluations, we generated five value trace problems using Java codes for two data structures and three sorting algorithms, and asked four students to solve them after uploading them into JPLAS. We analyzed the difficulty of the value trace problem for *Quick Sort*. In future studies, we will implement programs to assist generating value trace problems as best as possible, generate value trace problems for a variety of data structures and algorithms, and apply them to Java programming courses to evaluate the effectiveness of the value trace problem in Java programming educations.

References

[1] N. Funabiki, Y. Matsushima, T. Nakanishi, and N. Amano, "A Java programming learning assistant system using test-driven development method," IAENG Int. J. Computer Science, vol. 40, no. 1, pp. 38-46, Feb. 2013.

[2] N. Funabiki, Y. Fukuyama, Y. Matsushima, and T. Nakanishi, "An extension of fill-in-the-blank problem function in Java programming learning assistant system," Proc. Humanitarian Tech. Conf. (HTC 2013), pp. 95-100, Aug. 2013.

[3] Tana, N. Funabiki, T. Nakanishi, and N. Amano, "An improvement of graph-based fill-in-blank problem generation algorithm in Java programming learning assistant system," 2013 Int. Work. ICT Beppu, Dec. 2013.

[4] Tana, N. Funabiki and N. Ishihara, "A proposal of graph-based blank element selection algorithm for Java programming learning with fill-in-blank problem," to appear in Int. MultiConf. Eng. Comput. Sci., March 2015.

[5] K. Beck, *Test-driven development: by example*, Addison-Wesley, 2002.

[6] JUnit, http://www.junit.org/.

[7] Shell sort, http://www.thelearningpoint.net/computer-science/arrays-and-sorting-shell-sort-with-c-program-source-code.

[8] JavaMain, http://csis.pace.edu/~bergin/KarelJava2ed/ch2/javamain.html.

[9] Algorithm, http://people.cis.ksu.edu/~tamtoft/CIS775/F08/Slides/01.pdf.

[10] Sorting, http://faculty.simpson.edu/lydia.sinapova/www/cmsc250/LN250_Weiss/L12-ShellSort.htm.

[11] Quicksort, http://www.algolist.net/Algorithms/Sorting/Quicksort.