A Study of Exercise Problems in Java Programming Learning Assistant System

March, 2018

Khin Khin Zaw

Graduate School of
Natural Science and Technology

(Doctor's Course)
Okayama University

Dissertation submitted to
Graduate School of Natural Science and Technology
of
Okayama University
for
partial fulfillment of the requirements
for the degree of
Doctor of Philosophy.

Written under the supervision of

Professor Nobuo Funabiki

and co-supervised by
Professor Satoshi Denno
and
Professor Yasuyuki Nogami

Okayama University, March 2018.

To Whom It May Concern

We hereby certify that this is a typical copy of the original doctor thesis of Ms. Khin Khin Zaw

Signature of

the Supervisor

Seal of

Prof. Nobuo Funabiki

Graduate School of

Natural Science and Technology

Abstract

As a reliable and portable object-oriented programming language, *Java* has been extensively used in a variety of practical systems. A large number of universities and professional schools are offering Java programming courses to meet these needs. To assist Java programming educations in schools, we have developed the Web-based *Java Programming Learning Assistant System (JPLAS)*. JPLAS mainly provides two types of exercise problems, called the *element fill-in-blank problem*, and the *code writing problem*, to support self-studies by students at various learning levels.

The *element fill-in-blank problem* intends for a novice student to learn the Java grammar and basic programming through code reading. To generate the feasible problem, we also proposed the *graph-based blank element selection algorithm* that selects the blank elements that satisfy the unique correct answers. The *code writing problem* intends for a student to learn the Java source code writing from scratch, after completing the grammar study. In this problem, we adopted the *test-driven development (TDD)* method using *JUnit* to verify its correctness of the answer code from the student. It tests the behaviors of the code via the test code.

However, through applications to students, we have observed that several drawbacks exist in the current exercise problems in JPLAS. First, in the *element fill-in-blank problem*, generally, students can solve the problems mechanically without reading carefully and understanding the code behaviors correctly, if they know the Java grammar, because of the limited choices for each blank. Thus, the difficulty of the problem should be controlled by changing the number and importance of blank elements. Then, in the *code writing problem*, on the other hand, a lot of students cannot solve harder problems that require multiple classes and methods. More detailed code specifications should be provided to help the students to design the proper classes and methods for the code.

In this thesis, we present the five contributions on advancements of exercise problems in JPLAS. In the first contribution, we present the extensions of the *blank element selection algorithm* to change the number of blank elements to control the difficulty level of the generated problem, and to additionally blank key elements such as conditional expressions. We verify the effectiveness of these extensions through applications to various Java codes and evaluations of how they affect the solution performances of the students. The results show that these extensions can control the number of blank elements and the problem difficulty, where the solution performance is greatly affected by them.

In the second contribution, we propose the *core element fill-in-blank problem* to enhance the code reading studies by novice students. In this problem, to control the importance of blank elements in terms of algorithm/logic implementations, we adopt the *program dependence graph (PDG)* in the *blank element selection algorithm*. We verify the effectiveness of this problem through applications of the problems using the codes for the graph theory or

fundamental algorithms to students with questionnaire. The results show the highly correct answer rates, nevertheless the code understanding is necessary to solve the problems. Many students commented that the problems are helpful to understand the behaviors of the algorithms.

In the third contribution, we propose the *value trace problem* as a new type of fill-in-blank problem. This problem asks students to trace the actual values of the important variables in a code implementing a data structure or an algorithm. To select the tracing values of the variables, we present the *blank line selection algorithm*. We verify the effectiveness of this problem through generated problems for sorting and graph theory algorithms to students with questionnaire. The results show that some problems are much more difficult than the *element fill-in-blank problem*. Many students commented that they are effective in understanding the algorithm in a code by code reading, and a long problem code is more difficult to read out because of the limited interface space in JPLAS.

In the fourth contribution, we study the *workbook design* for three *fill-in-blank problems* for use in a Java programming course to enhance the self-studies of Java programming by novice students. This design consists of 15 categories that are arranged in the conventional learning order of Java programming. In this thesis, we collect the Java codes from textbooks and Web sites, and discuss their use in a Java programming course. Then, element fill-in-blank problems are generated from source codes by using the extended *blank element selection algorithm*. For the preliminary evaluations, we assign several problems in the workbook to novice students. The results show that the problem codes including object array and double loops are difficult for the novice students.

In the last contribution, we propose the *informative test code* approach for the *code writing problem*. To help the students to solve harder problems that require multiple classes and methods, the *informative test code* describes the detailed specifications of the names, access modifiers, and data types of the classes, methods, and arguments. We verify the effectiveness of this approach through applications of generated test codes to students. The software metrics of student answer codes are also evaluated using *Eclipse Metric Plugin*. The results show that all the students could complete the qualitative codes using informative test codes, whereas most students could not complete them without the informative test code.

To complete the works on exercise problems in JPLAS, there are still a lot of issues to be solved. In future studies, we will further improve the *blank element selection algorithm*, improve the PDG generation method, generate the different element fill-in-blank problems using these algorithms for the workbook, improve the user interfaces for displaying the problem codes, improve the informative test code approach to avoid the drawbacks, prepare informative test codes for a variety of source codes, and assign these generated problems in JPLAS to students in Java programming courses.

Acknowledgements

It is my great pleasure to thank those who made this dissertation possible. I want to say so much, but I can hardly find the words. So, I'll just say that you are the greatest blessing in my life.

I owe my deepest gratitude to my supervisor, Professor Nobuo Funabiki, who has supported me throughout my thesis with his patience and knowledge. I am greatly indebted to him, whose encouragement, advice, and support from the beginning enabled me to develop the understanding of the subject, not only in scientific but also in life. He gave me wonderful advices, comments, and guidance when writing papers and presenting them. Thanks for making me what I am today.

I am heartily thankful to my co-supervisors, Professor Satoshi Denno and Professor Yasuyuki Nogami, for their continuous supports, guidance, and proofreading of this work.

I would like to acknowledge Japan International Cooperation Agency (JICA) for financially supporting my doctoral course study in Okayama University and Yangon Technological University (YTU) where I am working as a lecturer.

I would like to thank for the helpful discussions from many people including, Professor Toru Nakanishi, Associate Professor Minuro Kuribayashi, Mr. Nobuo Ishihara, Dr. Tana, and all the FUNABIKI Lab's members. Thank you for believing in me when I was too weak and exhausted to believe in myself. Thank you for pushing me, you have supported me in all the tough times that I have ahead and thank you to share your thoughts and experiences with me.

Last but not least, I am eternally grateful to my beloved family, who always encouraged and supported me throughout my life. Thank you for being with me all that difficult time. Your support and understanding gave me the strength to continue fighting.

List of Publications

Journal Paper

1. **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," Journal of Information Engineering Express, vol. 1, No. 3, pp. 9-18, September 2015.

2. **Khin Khin Zaw** and Nobuo Funabiki, "A design-aware test code approach for code writing problem in Java programming learning assistant system," International Journal of Space-Based and Situated Computing, vol.7, No.3, pp. 145-154, September 2017.

International Conference Paper

3. **Khin Khin Zaw** and Nobuo Funabiki, "A concept of value trace problem for Java code reading education," IIAI International Congress on Advanced Applied Informatics 2015, pp. 253-258, July 2015.

4. **Khin Khin Zaw**, Nobuo Funabiki, and Minoru Kuribayashi "A proposal of three extensions in blank element selection algorithm for Java programming learning assistant system," 2016 IEEE 5th Global Conference on Consumer Electronics, pp. 3-6, October 2016.

5. **Khin Khin Zaw** and Nobuo Funabiki "A core blank element selection algorithm for code reading studies by fill-in-blank problems in Java programming learning assistant system", The 7th International Conference on Science and Engineering 2016, pp. 204-208, December 2016.

6. Nobuo Funabiki, Shinpei Matsumoto, **Khin Khin Zaw**, and Wen-Chung Kao, "Applications of coding rule learning function to workbook codes for Java programming learning assistant system," The 7th International Conference on Science and Engineering 2016, pp. 1170-1175, December 2016.

7. **Khin Khin Zaw**, Nobuo Funabiki, and Wen-Chung Kao, "A proposal of informative test code approach for code writing problem in Java programming learning assistant system," The 8th International Conference on Science and Engineering 2017, pp. 260-265, December 2017.

Other Papers

8. **Khin Khin Zaw** and Nobuo Funabiki, "A blank line selection algorithm for value trace problem in Java programming learning assistant system," IEICE Society Conf., BS-6-2, pp. S19-S20, September 2015.

9. **Khin Khin Zaw** and Nobuo Funabiki, "A value trace problem for prim algorithm in graph theory in Java programming learning assistant system," 17th IEEE Hiroshima Section Student Symposium (HISS 2015), pp. 444-447, November 2015.

10. **Khin Khin Zaw** and Nobuo Funabiki, "A study of value trace problems for graph theory algorithms in Java programming learning assistant system," IEICE Tech. Report, SS-2016-01, pp. 159-164, January 2016.

11. **Khin Khin Zaw**, Nobuo Funabiki, and Minoru Kuribayash, "Extensions of blank element selection algorithm for Java programming learning assistant system," IEICE Tech. Report, ET-2016-06, pp. 41-46, June 2016.

12. **Khin Khin Zaw** and Nobuo Funabiki, "Preliminary evaluation of blank element selection algorithm for fill-in-blank problem in Java programming learning assistant system," IEICE Society Conf., BS-5-24, pp. S98-S99, September 2016.

13. **Khin Khin Zaw** and Nobuo Funabiki, "A blank element selection algorithm extension for algorithm code reading by fill-in-blank problems in Java programming learning assistant system," The 18th IEEE Hiroshima Section Student Symposium (HISS 2016), pp. 129-132, November 2016.

14. **Khin Khin Zaw**, Nobuo Funabiki, and Minoru Kuribayash, "Element fill-in-blank problems in Java programming learning assistant system," IEICE General Conf., BS-1-21, pp. S41-42, March 2017.

15. **Khin Khin Zaw** and Nobuo Funabiki, "A proposal of design-aware test code approach for code writing problem in Java programming learning assistant system," IEICE Society Conf., BS-7-8, pp. S56-S57, September 2017.

16. **Khin Khin Zaw** and Nobuo Funabiki, "An informative test code approach for code writing problem in Java programming learning assistant system," IEICE Tech. Report, SS-2017-10, pp. 31-36, October 2017.

List of Figures

List of Tables

List of Codes

Contents

Chapter 1

Introduction

1.1 Background

As a reliable and portable object-oriented programming language, *Java* has been extensively used variety of practical systems. They involve Web application systems, mission critical systems for large enterprises, and small-sized embedded systems. Thus, the cultivation of Java programming engineers has been in high demands amongst industries. As well, a great number of universities and professional schools are offering Java programming courses to meet these needs. A Java programming course usually combines grammar instructions by classroom lectures and programming exercises by computer operations. However, in programming exercises, a teacher can be overloaded in verifications of a lot of codes from students and in giving feedbacks with proper comments to them. If responses from the teacher becomes late, students may lose the learning motivations.

To help Java programming educations, we have developed the Web-based *Java programming learning assistant system (JPLAS)*. JPLAS adopts the *Ubuntu* for the operating system, *Tomcat* for the Web application server, *JSP* for the application programs with *HTML*, and *MySQL* for the database for managing the data. The user can access to JPLAS through a Web browser.

JPLAS has two user service functions: *teacher services* and *student services*. The teacher services includes the problem registration and the assignment generation. The student services include the assignment solution and the score reference. JPLAS mainly provides two types of exercise problems, namely the *element fill-in-blank problem* and, the *code writing problem*, to support self-studies of students at various learning levels. It has been expected that this system be a great help for reducing the teacher loads and improving the learning motivations of the students.

The *element fill-in-blank problem* in JPLAS intends for a student to learn the Java grammar and basic programming through code reading. This problem asks a student to fill the correct elements in the blank in a given Java code. The correctness of the answer is marked by comparing it with the original element in the code. Thus, the original element must be the unique correct answer for the blank.

In this problem, 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* is a fixed sequence of characters that has been defined in Java grammar to represent a specified function, and 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* intends other grammar

elements such as “ . ” (dot), “ : ” (colon), “ ; ” (semicolon), “ (,) ” (bracket), “ {, } ” (curly bracket).

In JPLAS, the teacher needs to register a new assignment with the problem code and answer in the database. Then, a student solves the problem through accessing to an assignment, checking the results, correcting and resubmitting the answer if necessary. To help a teacher to generate an *element fill-in-blank problem*, we also proposed a *graph-based blank element selection algorithm* that selects proper blank elements that satisfy the unique correct answers [1].

The *code writing problem* in JPLAS intends for a student to learn writing a Java source code from scratch. This problem asks a student to write a source code that satisfies the specifications given in the *test code*. In this problem, we adopted the *test-driven development (TDD) method* using *JUnit* [6][7]-[8]. *JUnit* automatically tests the answer code via the test code on the server, to verify its correctness when submitted from a student. In JPLAS, the teacher needs to register the assignment with the problem statement and the test code. Then, a student writes a source code through reading the statement and the test code, and testing, modifying and resubmitting the code if error occurs. As a target of user, we considered a student who may not be able to write a proper code, but who has studied simple Java codes in textbooks through exercises

However, through applications to students, we have observed that several drawbacks exist in the current problems. Firstly, in the *element fill-in-blank problem*, generally, the students can solve mechanically without reading carefully and understanding the code behaviors, if they are familiar with Java grammar. Due to the unique answer constraint, only limited choices of elements may exist for many blanks. Actually, we have found that as the number of solving element fill-in-blank problems increases, students could reach correct answers much faster than the beginning. Thus, the difficulty of the problem should be controlled by changing the number and importance of blank elements. In our previous studies [10][11], we found that as the number of blanks increases, the correct answer rates by students decreases. In the programming educations, the students should study the codes that implement some algorithms or logics such as standard inputs/outputs, data structure, fundamental algorithms including sorting, graph algorithms. Thus, in these codes, the blank elements should be considered not only by the grammar but also by their importance in the code in terms of algorithm/logic implementation.

On the other hand, in the *code writing problem*, a lot of students who are studying Java programming, cannot solve harder problems that require multiple classes and methods even after solving many simple problems. For example, the implementation of a graph theory algorithm is included in such problems, where the code needs the handling of the graph data in addition to the algorithm procedure. More detailed code specifications are necessary to help students to find the proper classes and methods for the code.

1.2 Contributions

In this thesis, motivated by the above mentioned problems, we propose the five advancements of the exercise problems for JPLAS as the contributions.

In the first contribution, we present the extensions of the *blank element selection algorithm* to change the number of blank elements to control the difficulty level of the generated problem, and to additionally blank key elements such as conditional expressions. Specifically,

first, we extend this algorithm by 1) adding operators in conditional expressions for blank candidates, 2) improving the edge generation method in the constraint graph to increase the number of blanks, and 3) introducing two parameters to change the frequency of selecting blanks [22] [23]. To verify the effectiveness of these extensions, we apply the extended *blank element selection algorithm* to 55 Java codes, and confirm that they can increase the number of blanks and control the problem difficulty. For the trial evaluation, we generated element fill-in- blank problems by applying this extended algorithm with different parameter values to investigate the solution performance of students. The results show that the number of blanks and the code length affect the solution performances by students.

In the second contribution, we propose the *core element fill-in-blank problem* to enhance the code reading studies by novice students. In this problem, a long Java source code implementing the graph theory or fundamental algorithm is used. To select the blank elements while controlling the importance in terms of algorithm/logic implementations, the *blank element selection algorithm* is extended by using the *program dependence graph (PDG)* together [28]. The students need to fill in blanks by reading and understanding the code structure. For evaluations, we apply the extended algorithm to four source codes for the graph theory or fundamental algorithms, and asked six students in our group to solve them. The results show that it has the highly correct answer rate, whereas code understanding is still necessary to solve the problems. The students commented that they are helpful in understanding the algorithm/logic implementation in a code.

In the third contribution, we propose the *value trace problem* as a new type of fill-in-blank problem [44]. It asks the students to trace the actual values of important variables in a Java code implementing some algorithms such as sorting and graph algorithms. Basically, the whole data in the line of the output data is blanked through executing the *blank line selection algorithm* [46], such that at least one data is changed from the previous one. To verify the proposal, we generated value trace problems for sorting and graph theory algorithms, and asked students in our lab. The results show that some problems are much more difficult than the *element fill-in-blank problem*. Some students commented that they are effective in understanding the algorithm in the Java code and the code reading, and that a long problem code is difficult to read out because of the one column page in JPLAS.

In the fourth contribution, we study the *workbook design* for the three *fill-in-blank problems* for use in Java programming courses to enhance self-studies of Java programming by students and to help preparing assignments for JPLAS by teachers [53][54]. This workbook design consists of 15 categories that are arranged in the conventional learning order of Java programming. Basically, the fill-in-blank problems are used for studying the grammar and basic programming skills through code reading. For evaluations, the suitable Java source codes are collected from textbooks and Web sites. Then, element fill-in-blank problems are generated from source codes by using the extended *blank element selection algorithm* for the workbook, and are assigned to students. The results show that the problem codes including the object array and double loops are more difficult for novice students.

In the last contribution, we propose the *informative test code* approach for the code writing problem in JPLAS [75][76]. To help the students to solve harder problems that require multiple classes and methods, the *informative test code* describes the detailed specifications of the names, access modifiers, and data types of the classes, methods, and arguments. By writing a source code to pass this test code, a student is expected to write a source code with the proper classes/methods in the model code. To verify effectiveness of this approach, first we prepared the informative test codes for five well known graph algorithms that consist

of multiple classes/methods and asked seven students to write the source codes for them. Then, the software metrics of student answer codes are evaluated using *Eclipse Metric Plugin*. The results showed that all the students completed the high-quality source codes for the harder problems using informative test codes, whereas most students could not complete them without informative test codes.

1.3 Contents of This Dissertation

The remaining part of this thesis is organized as follows.

In Chapter 2, we review the software architecture and two problems in JPLAS.

In Chapter 3, we propose the extensions of *blank element selection algorithm* for fill-in-blank problem in JPLAS.

In Chapter 4, we propose the *core element fill-in-blank problem*.

In Chapter 5, we propose the *value trace problem*.

In Chapter 6, we propose the *workbook design* for the three *fill-in-blank problems*.

In Chapter 7, we propose the *informative test code* approach for the *code writing problem*.

Finally, in Chapter 8, we conclude this thesis with some future works.

Chapter 2

Overview of JPLAS

In this chapter, we introduce the outline of *Java Programming Learning Assistant System (JPLAS)*.

2.1 Software Platform for JPLAS

Figure 2.1 illustrates the software platform for JPLAS. In JPLAS, *Ubuntu-Linux* is adopted for OS. The current system is running on *VMware* for portability. *Tomcat* is used as the Web server for *JSP*. *JSP* is a script language that can embed Java codes within HTML codes. *Tomcat* returns the dynamically generated Web page to the client. *MySQL* is adopted as the database for managing the data in JPLAS.

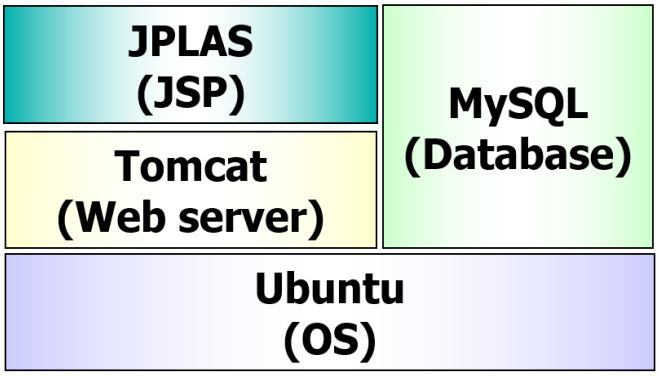

Figure 2.1: Software platform for JPLAS

2.2 User Services

The functions of JPLAS consist of *teacher service functions* and *student service functions*. *Teacher service functions* include the problem generation, the assignment generation, and the student performance reference. *Student service functions* include the assignment view, the problem view, the problem solution, and the score reference.

2.2.1 System Utilization Procedure

The utilization procedure for both JPLAS functions by a teacher and a student is given as follows:

1. A teacher generates a new *problem* and registers it to the database.
2. A teacher generates a new *assignment* by selecting proper problems in the database and registers it to the database.
3. A student selects an assignment to be solved.
4. A student selects a problem in the assignment to be solved.
5. A student solves the questions in the problem and submits the answers to the server.
6. The server marks the answers and returns the marking results.
7. A student modifies the incorrect answers and resubmits them to the server, if necessary.
8. A student refers to his/her solution results of the assignments.
9. A teacher refers to the solution results of all the students of the assignments.

2.3 Fill-in-blank Problem in JPLAS

The *element fill-in-blank problem* in JPLAS intends for a student to learn the Java grammar and basic programming through code reading. This problem asks a student to fill the correct elements in the blank in a given Java code. The correctness of the answer is marked by comparing the student answer element with the original element in the code. Thus, the original element must be the unique answer for the blank. To help a teacher to generate an element fill-in-blank problem, the *blank element selection algorithm* has been proposed [1]-[3]. To generate a new problem by using this algorithm, a teacher needs to prepare the high quality source code.

2.3.1 Lexical Analyzer

For the *element fill-in-blank problem*, we adopt open source software *JFlex* [4] and *jay* [5]. *JFlex* is a lexical analyzer generator for a Java code, which is also coded by Java. It transforms a source code into a sequence of lexical units that represent the least meaningful elements to compose the code. It can classify each element in a code into either a reserved word, an identifier, a symbol, or an immediate data. A *reserved word* signifies a fixed sequence of characters that has been defined in 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).

For example, a statement `int value = 123 + 456;` is divided into `int`, `value`, `=`, `123`, `+`, `456`, and `;`. Unfortunately, *JFlex* cannot identify an identifier among a class, a method, or a variable. Thus, *jay* is applied as well. Since, *jay* is a syntactic parsing program based on the LALR method, it can identify an identifier.

2.4 Code Writing Problem in JPLAS

The code writing problem in JPLAS intends for a student to learn writing a source code from scratch. This problem asks a student to write a whole source code from scratch that satisfies the specifications given by the *test code* [6]. The JPLAS function for this problem is implemented based on the *test-driven development (TDD) method* using an open source framework *JUnit* [7]-[8]. It automatically tests the answer code on the server to verify the correctness when submitted from a student. A teacher needs to prepare the specification and the test code to register a new assignment in JPLAS.

2.4.1 TDD Method

Here, we discuss the *test-driven development (TDD) method* [7].

2.4.1.1 JUint

JUnit can assist an automatic unit test of a Java source code or a class. Since *JUnit* has been designed with the Java-user friendly style, including the use of the test code programming, is rather simple for Java programmers. In *JUnit*, one test can be performed by using one method in the library whose name starts with " assert ". For example, " assertEquals " method is used to compare the execution result of the source code with its expected value.

2.4.2 Test Code

A test code should be written by using libraries in *JUnit*. The following *MyMath* class source code is used to introduce how to write a test code. *MyMath* class returns the summation of two integer arguments.

Listing 2.1: Source code for *MyMath* class

```
1 public class Math {
2   public int plus(int a, int b) {
3     return( a + b );
4   }
5 }
```

Then, the following test code tests *plus* method in *MyMath* class.

Listing 2.2: Test code for *MyMath* class

```
1  import static org.junit.Assert.*;
2  import org.junit.Test;
3  public class MyMathTest {
4  @Test
5    public void testPlus(){
6      MyMath ma = new MyMath();
7      int result = ma.plus(1, 4);
8      assertEquals(5, result);
9    }
10 }
```

The test code imports *JUnit* packages containing required test methods at lines 1 and 2, and declares *MyMathTest* at line 3. *@Test* at line 4 indicates that the succeeding method

represents the test method. Then, it describes the procedure for testing the output of *plus* method. This test is performed as follows:

1. An instance *ma* for *MyMath* class is generated.
2. *plus* method for this instance *ma.plus* is called with given arguments.
3. The result *result* is compared with its expected value using *assertEquals* method.

2.4.2.1 Features in TDD Method

In the TDD method, the following features can be observed.

1. The test code represents the specifications of the source code, because it must describe every function which will be tested in the source code.
2. The testing process of a source code becomes efficient, because each function can be tested individually.
3. The refactoring process of a source code becomes easy, because the modified code can be tested instantly.

2.5 Summary

In this chapter, we introduced the outline of *Java programming Learning assistant System (JPLAS)*. The software architecture of the JPLAS implementation, and the two offering problems, the *element fill-in-blank problem* and the *code writing problem* were discussed.

Chapter 3

Extensions of Blank Element Selection Algorithm

In this chapter, we present the extensions of the *blank element selection algorithm* for the *element fill-in-blank problem* in JPLAS.

3.1 Introduction

The *blank element selection algorithm* is extended in three ways to control the difficulty level of the generated problem by changing the number of blank elements. 1) It adds the operators in conditional expressions for blank element candidates. 2) It improves the edge generation method in the constraint graph to increase the number of blanks. 3) It introduces two parameters to change the frequency of selecting blanks.

In this chapter, the overview of the existing blank element selection algorithm is first reviewed. Then, the three extensions are presented sequentially. After them, these extensions are evaluated. Finally, the conclusion is given for this chapter.

3.2 Review of Blank Element Selection Algorithm

In this section, we review the *blank element selection algorithm* in [1]. This algorithm generates the *constraint graph* to describe the constraints in the blank element selection.

3.2.1 Vertex Generation for Constraint graph

In the constraint graph, each vertex represents a candidate element for being blank. The candidate elements or vertices are extracted from the source code through the lexical analysis using *JFlex* [4] and *jay* [5]. Each vertex contains the associated information in Table 3.1 that is necessary for the following edge generation.

3.2.2 Edge Generation for Constraint graph

Then, an edge is generated between any pair of two vertices or elements that should not be blanked at the same time. There are three categories to represent the constraints in selecting blank elements with unique answers.

Table 3.1: Vertex information

item	content
symbol	symbol of element
line	row index of element
column	column index of element
count	number of element appearances
order	appearing order of element in the code
group	statement group index partitioned by { and }
depth	number of { from top

3.2.2.1 Group Selection Category

In the *group selection category*, all the elements related with each other in the code are grouped together. In each group, one vertex is randomly selected first. Then, edges are generated between this vertex and the other vertices so that at least this selected element is not selected for blank. There are five conditions of this category.
(1) Identifier appearing two or more times in the code

The multiple elements representing the same identifier of variable, class and method using the same name, are grouped together. If all such elements are blanked, a student cannot answer the original identifier.
(2) Pairing reserved words composed of three or more elements

The three or more elements representing the reserved words in pairs are grouped together. If all of them are blanked, the unique answers may become too difficult as the following two cases:

- switch-case-default
- try-catch-finally

(3) Data type for variables in equation

The elements representing the data types for variables in one equation are grouped together. For example, in `sum = a + b`, the data types of the three variables, `sum`, `a`, and `b`, must be the same.
(4) Data type for method and its returning variable

The elements representing the data type of a method and its returning variable are grouped together.
(5) Data type for arguments in method

The elements representing the data type of an argument in a method and its substituting variable are grouped together.

3.2.2.2 Pair Selection Category

In the *pair selection category*, the elements appearing in the code in pairs are grouped together. For each pair, an edge is simply generated between the two corresponding vertices so that at least one element is not selected for blank.
(1) Elements appearing continuously in statement

The two elements appearing continuously in the same statement are paired in the code. If both of them are blanked, their unique correct answers may not be guaranteed. The two elements connected with a dot (".") are also paired.
(2) Variables in equation

The elements representing any pair of the variables in an equation are paired. If both are blanked, the unique answers become impossible. For example, for `sum = a + b`, `sum = b + a` is also feasible.
(3) Pairing reserved words

The two elements representing the paring reserved words are paired. If both are blanked, the unique correct answers may not be guaranteed. The following five paring reserved words are considered:

- if-else
- do-while
- class-extends
- interface-extends
- interface-implements

(4) Pairing control symbols

The two elements representing a pair of control symbols, namely "(,)" (bracket) and "{, }" (curly bracket), are paired. The novice students should carefully check them in their codes because they often make mistakes with them.

3.2.2.3 Prohibition Category

In the *prohibition category*, an element is prohibited from the blank selection because it does not satisfy the uniqueness with the high probability. There are four conditions for this category. However, an element in a fixed sequence of elements indicating a specific meaning in a Java code, such as `public static void main` and `public void paint(Graphics g)`, are excluded from this category, because they should be mastered by students.
(1) Identifier appearing only once in code

The selected element representing the identifier in this category appears only once in the code. If it is blanked, a student cannot answer the original identifier.
(2) Operator

The element representing an operator such as the arithmetic operator: `=`, `+`, `-`, `*`, `/`, the comparative operator: `<`, `>`, `<=`, `>=`, `==`, `!=`, and the logical operator: `&`, `|`, `^`, `!` is selected to this category. If an operator is blanked, a student cannot answer the original one unless the proper explanation on the specification related to the operator is given.
(3) Access modifier

The element representing an access modifier for an identifier is selected to this category. If it is blanked, either `public`, `protected`, `private` can often be grammatically correct.
(4) Constant

The element representing a constant is selected to this category. If it is blanked, a student cannot answer the original constant.

3.2.3 Example Constraint Graph

Figure 3.1 illustrates the constraint graph for lines 1-6 in the Java code for *bubbleSort*. Some elements are excluded there by the prohibition category. For example, the four `array` are grouped together by 3.2.2.1 (1), and `static` and `void` are paired by 3.2.2.2 (1), `public` is excluded by 3.2.2.3 (3).

Listing 3.1: Source code for *BubbleSort*

```
1  public static void bubbleSort(int[ ] array){
2    int start=1;
3    for (int i=array.length−1; i>0; i−−)
4      for (int j=0; j<i; j++)
5        if (array[j]>array[j+start]){
6          int tmp= array[j];
7          array[j]=array[j+1];
8          array[j+1]=tmp;
9          for (int k:array){
10           System.out.print(k);
11           System.out.print(",");
12       }
13       System.out.println();
14     }
15   }
16 }
17 }
```

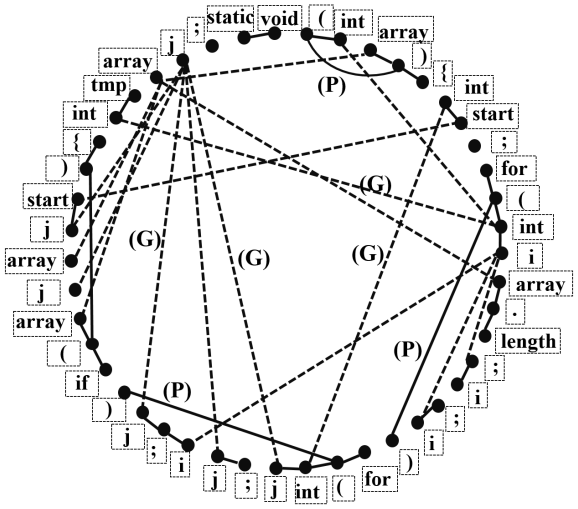

Figure 3.1: Constraint graph for *bubbleSort*

3.2.4 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.

3.2.5 Maximal Clique Extraction of Compatibility Graph

Finally, a maximal clique of the compatibility graph is extracted by a simple greedy algorithm to find the maximal number of blank elements with unique answers from the given Java code. A clique of a graph represents its subgraph where any pair of two vertices is connected by an edge. The procedure for our algorithm is described as follows:

1) Calculate the degree (= number of incident edges) every vertex in the compatibility graph.
2) Select one vertex among the vertices whose degree is maximum. If two or more vertices have the same maximum degree, select one randomly.
3) If the selected vertex is a *control symbol* and the number of selected control symbols exceeds 1/3 of the total number of selected vertices, remove this vertex from the compatibility graph and go to (5).
4) Add the selected vertex for blank, and remove it as well as its non-adjacent vertices from the compatibility graph.
5) If the compatibility graph becomes null, terminate the procedure.

3.2.6 Fill-in-blank Problem Generation

In the maximal clique procedure, 3) is used to sustain the total number of blank control symbols, because a code usually has a lot of control symbols. Here, we examined the average number of blanks for control symbols and other symbols by the algorithm. Then, we empirically selected 1/3 as an appropriate ratio to generate the feasible *fill-in-blank problems* for novice students. However, in these condition, the generated fill-in-blank problems can be solved without reading out the code if students are familiar with Java grammar.

3.3 Extension 1: Additional Blank Candidates

In this section, we present the first extension of the *blank element selection algorithm*. 16 *operators* for conditional expressions in Table 3.2 [9] are added for blank element candidates. To satisfy the uniqueness of the correct answers and avoid the extreme hardness for novice students, the *operators* in the same conditional expression are grouped together as the group selection category, where at least one element in the same group is not selected for blank. In the code for `bubbleSort`, `>` and `--` in line 3 and `<` and `++` in line 4 are grouped together.

3.4 Extension 2: Improvement of Edge Generation

In this section, we present the second extension.

3.4.1 Overview

In our previous studies [10][11], it was found that as the number of blanks increases, the correct answer rates by students decreases. Thus, even from the same source code, the different number of blanks can change the difficulty level of the *element fill-in-blank problem*.

Table 3.2: Operators in conditional expressions for blank

operator	example	operator	example
<	a<b	++	a++
<=	a<=b	--	a--
>	a>b	!	!a
>=	a>=b	+=	a+=b
==	a==b	-=	a-=b
!=	a!=b	*=	a*=b
&&	a&&b	/=	a/=b
||	a||b	%	a%b

In this thesis, to select a larger number of blanks, the following improved edge generation method is adopted for the group selection category in the constraint graph. Here, instead of randomly selecting one vertex in the same group in the previous work, the vertex that has the largest number of incident edges is selected. As a result, other many vertices can be selected for blanks when this vertex is not selected for the blank.

3.4.2 Improved Edge Generation Procedure

The following procedure describes the details:

1) Generate the edge between two vertices for each vertex pair in the pair selection category.
2) Sort the vertex groups for the group selection category in the descending order of the group size.
3) Select one vertex for each group in 2) from the top by the following procedure:
   (1) Calculate the degree of the vertices in the group.
   (2) Select the vertex that has the largest degree. If two or more vertices have the same largest degree, randomly select one among them.
   (3) Generate the edges between the vertex in (2) and the other vertices in the group.

3.4.3 Example

In `bubbleSort`, the four `int` in lines 1, 3, 6 and 9 are grouped by the group selection category. At least one `int` must not be selected as the blank element for the unique correct answer. Using this group, we explain the improved edge generation method for the constraint graph.

Figure 3.2 illustrates the four vertices in this group and their incident vertices are selected by the pair selection category. `int` and `(`, `int` and `[` in line 1, `int` and `(`, `int` and `i` in line 3, `int` and `tmp` in line 6, `int` and `(`, `int` and `k` in line 9 are connected by the pair selection category respectively. These edges are described by the straight lines with (P). Then, `int` in lines 1, 3 and 9 has the same largest degree 2 among the four. Thus, we select `int` in line 1 among the two randomly. Then, we generate the edges between this vertex and the other three vertices for `int` that are described by the dotted lines with (G).

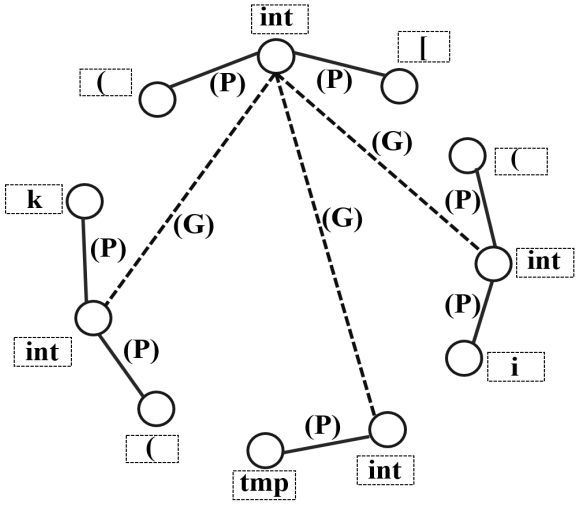

Figure 3.2: Example of improved edge generation method for *bubbleSort*

3.5 Extension 3: Introduction of Two Parameters

In this section, we present the third extension.

3.5.1 Overview

The ratio between the number of blank elements and non-blank ones in the problem code can change the difficulty of the *element fill-in-blank problem*. In general, more blanks makes the problem harder, and less blanks make it easier. To control this ratio, the two new parameters are introduced, namely, BG (blank gap number) and CB (continuous blank number).

3.5.2 Blank Gap Number

The non-blanked elements in a problem code become hints to solve the *element fill-in-blank problem*. As more non-blanked elements exist between blanked elements, it becomes easier. Thus, we try to control the difficulty of the problem by changing the number of non-blanked elements between blanked ones by introducing the *blank gap number* BG. To realize it, for the constraint graph, we generate an edge for each vertex with every vertex in the same statement in the code that exists within its BG neighbors. Here, we note that the previous algorithm actually adopts $BG = 1$ where at least one non-blanked element exists. For example, in the case of $BG = 2$, `bubbleSort` at line 1 has an edge with `static`, `void`, `(`, and `int` so that at least two non-blanked elements exist in the problem code.

3.5.3 Continuous Blank Number

On contrary, as more blanked elements continue in a problem code, it becomes harder. Thus, we also try to control the difficulty by changing the number of continuously blanked elements by introducing the *continuous blank number* CB. Here, we note that when CB is 2 or larger, BG must be set 0. The following procedure describes the extension to realize it:

Table 3.3: Average number of vertices and blanks by extensions 1 and 2

avg.# of vertices		127.39	132.41	127.39	132.41
BG	CB	previous	extension 1	extension 2	both
1	1	34.59	37.15	35.91	37.65
0	∞	52.81	57.41	53.39	58.11

(1) Find a solution by applying the blank element selection algorithm with $BG = 0$.

(2) Change the last blanked element into non-blanked one if the number of continuously blanked elements exceeds CB.

For example, in the case of $CB = 3$, the following blanks can be generated for line 1.

```
1 public static void bubbleSort _1_ _2_ _3_ array)
```

3.6 Evaluations

In this section, we evaluate the three extensions of the *blank element selection algorithm* for the *element fill-in-blank problem* in JPLAS.

3.6.1 Blank Selection Evaluation of Extensions 1 and 2

55 Java source codes for fundamental data structure or algorithms in textbooks and Web sites in [12]-[21] are collected to generate element fill-in-blank problems. First, the effects of *Extensions 1 and 2* are evaluated by applying the extended algorithm with $BG = 1$ and $CB = 1$ in *Extension 3*. Table 3.3 shows the average number of blank candidates or vertices in the constraint graphs and the average number of blanks for the 55 problem codes that are generated by the previous algorithm, the extended algorithm with *Extension 1* only (operators in conditional expressions), the one with *Extension 2* only (improved edge generation method), and the one with both *Extensions 1 and 2*. Table 3.3 indicates that the number of blanks increases by applying each extension and becomes the largest by applying both extensions at the same time. *Extension 1* increases the number of blanks by increasing the number of vertices in the constraint graphs, while *Extension 2* increases it even with the same number of vertices as in the previous algorithm by selecting better edges among them for the constraint graph.

Then, their effects are examined in the extreme case of $BG = 0$ and $CB = \infty$ to remove the limitation of controlling the ratio of blank elements and non-blank ones in each problem code. In this case, the number of blanks can be the largest. Table 3.3 also shows the average number of blanks by the four algorithm cases, where the proposed two extensions can increase the number of blanks.

3.6.2 Blank Selection Evaluation of Extension 3

Then, the effects of *Extension 3* is evaluated by applying the extended algorithm with different values for BG and CB, while *Extensions 1 and 2* are adopted together. Table 3.4

Table 3.4: Average number of blanks for different BG and CB

BG	3	2	1	0	0
CB	1	1	1	2	3
avg.# of blanks	23.35	28.33	37.65	57.61	57.69

Table 3.5: Parameter values for problem generations

	L1	L2	L3
BG	3	1	0
CB	1	1	3

shows the average number of blanks for 55 problem codes when BG and CB are changed from 0 to 3 respectively. It is noted that for $BG \geq 1$, CB must be 1, because at most one blank element can be selected continuously to have BG non-blank elements between blank ones, and for $CB \geq 2$, BG must be 0, because two or more blank elements can be selected continuously. Table 3.4 indicates that the larger BG can decrease the number of blanks and the larger CB can increase it. Thus, it is confirmed that they can control the difficulty of the element fill-in-blank problems.

3.6.3 Solution Performance Evaluation

Then, the solution performances of students are evaluated by the algorithm extensions by applying the generated problems to students.

3.6.3.1 Assigned Element Fill-in-blank Problems

As Java source codes for the assigned element fill-in-blank problems, three codes related to the *RSA algorithm*, namely *Euclid* (calculate the GCD of two arguments using the Euclid method), *TrialDiv* (calculate the GCD using the trial division method), and *ModExp* (calculate the modulo exponentiation of a big integer) [21], are used. For the two parameter values in *Extension 3*, the three sets in Table 3.5 are adopted to examine three different levels, $L1$, $L2$, and $L3$.

Table 3.6 shows the LOC (the number of lines) in the problem code and the number of blanks in each problem with three levels. As LOC is larger, the number of blanks increases for any difficulty level.

3.6.3.2 Problem Assignments to Students

The 33 students are randomly divided into three groups, A, B and C, with the equal number. Then, one level of each problem is assigned to each group, so that every student solves the different problems with the different levels. Thus, any student solved any level in our problem assignment.

Table 3.6: Statistics of generated problems

Java code	LOC	number of blanks		
		L1	L2	L3
Euclid	12	6	9	16
TrialDiv	19	14	19	38
ModExp	13	11	14	20

Table 3.7: Solution results for each group

	Euclid		*TrialDiv*		*ModExp*	
Group	L	correct rate (%)	L	correct rate (%)	L	correct rate (%)
A	1	76 %	2	71%	3	75%
B	2	87%	3	67%	1	73%
C	3	80%	1	78%	2	84%

3.6.3.3 Solution Results

Table 3.7 shows the solving problem level and the percentage of the correctly solved blanks among the students in each group. This table indicates that in each group, this percentage for *TrialDiv* is smaller than the others at any difficulty level.

Table 3.8 shows the average correct rate for each group, problem, and level. First, among the three groups, *Group C* has the highest average correct rate. Thus, *Group C* is the best student group. Then, among the problems, the average correct rate for *TrialDiv* is the lowest, because this problem has the larger LOC and number of blanks than other problems at any difficulty level. Finally, among the three levels, $L3$ has the lowest average correct rate. However, $L2$ has the higher rate than $L1$, because *Group B* solved better than *Group A* for *Euclid* and *Group C* solved better than *Group B* for *ModExp*. It is necessary to further investigate the relationship between the two parameter values for the difficulty level and the average correct rate of students, which will be in our future works.

Table 3.8: Average solution performance

Euclid		TrialDiv		ModExp	
A	73%	*Euclid*	81%	L1	76%
B	71%	*TrialDiv*	70%	L2	79%
C	80%	*ModExp*	77%	L3	71%

3.7 Summary

In this chapter, we presented the three extensions of the *blank element selection algorithm* for the *element fill-in-blank problem* in JPLAS. Firstly, the effectiveness of *Extensions 1-3* was evaluated through applications to 55 Java source codes for fundamental data structure or algorithms. Then, the solution performance of students was evaluated through the generated problems by applying the extended algorithm. The future studies include the improvement of the *blank element selection algorithm* by adopting more sophisticated maximal clique algorithm to further increase the number of blanks and the generation of element fill-in-blank problems using the algorithm to assign them in Java programming course.

Chapter 4

Core Element Fill-in-blank Problem

In this chapter, we present the *core element fill-in-blank problem* in JPLAS.

4.1 Introduction

The *core element fill-in-blank* is proposed to enhance the code reading studies by the novice students. In this problem, a Java code implementing the graph theory or fundamental algorithm is used. In general, such the code is longer and usually includes multiple classes/methods and the blank elements are selected while controlling the importance in terms of algorithm/logic implementations by applying *core blank element selection algorithm*. This algorithm is extended by using the *program dependence graph (PDG)* together from the previous *blank element selection algorithm*. PDG represents the dependency between the statements or lines in the code. Thus, it is used to compute the importance of blank elements with regarding the statements that implement the algorithm/logic in the code. The students need to fill in the blanks by reading and understanding the code structure.

In this chapter, the procedure for the PDG generation is first reviewed. Then, *core blank element selection algorithm* is presented. After them, the proposed problem is evaluated. Finally, the conclusion is given for this chapter.

4.2 Review of PDG Generation

In this section, we review the PDG generation procedure in [24].

4.2.1 Basic PDG Generation for Variable

In the PDG, a vertex represents a statement in the code and an edge represents the dependency between the corresponding two statements. Thus, a statement, or vertex, with a larger degree can have influence to many other statements in the code [25], and can be considered as a core statement. Based on the data flow dependence, an edge is generated between two statements $s1$ and $s2$ in a code when the following conditions are satisfied.

1) A variable $v1$ is defined at $s1$ (definition).
2) $v1$ is referred at $s2$ (reference).

4.2.2 Extended PDG Generation for Object

In [24], the PDG generation method was extended for a Java source code. As an object-oriented language, an object in a Java code should be considered in addition to a variable in a non-objected oriented language. Then, the conditions for the edge are modified:

1) An object is considered as a variable in the data flow dependence. Besides, when a variable inside an object, or a member variable, is accessed, it is regarded that the object itself is accessed there.
2) The following two cases are regarded that an object is defined at the corresponding statement: 1) the object appears at the left side of an assign statement, and 2) the object is called.
3) The following two cases are regarded that an object is referred at the statement: 1) the object appears at the right side of an assignment statement, and 2) the object is used as an argument of a method.

4.2.3 Example of PDG

Figure 4.1 illustrates the PDG for a simple Java code that transfers the data from the input stream “is” to the file “out”. Each node represents a statement or vertex, and each directed edge represents the data dependency between statements. Line 2 is selected as the blank statement because it has the maximum degree of seven.

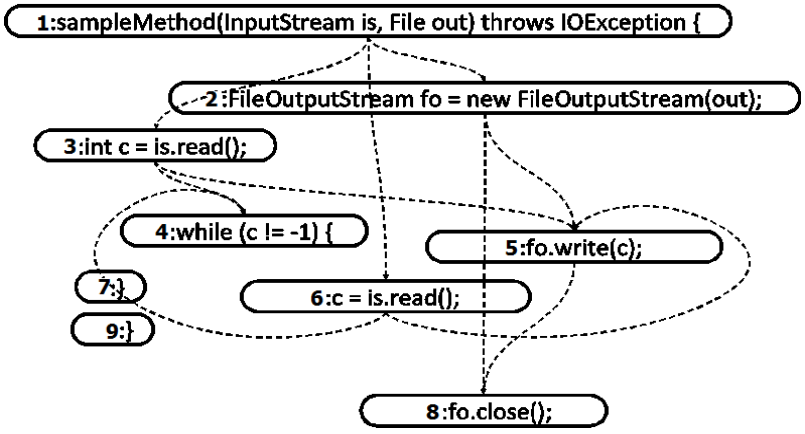

Figure 4.1: Sample PDG

4.2.4 Restrictions of PDG

This thesis only considers the dependency between the statements in the same method to simplify the implementation of the PDG generation. Because our proposal aims to offer Java programming learning environments to novice students, it is important for them to understand the structure of one method first. Besides, the core statements for an algorithm or logic are often described in one method. Then, the statements outside any method are assigned zero for the PDG degrees, and their elements are not selected in this extension at all. The consideration of the dependency between statements beyond a method will be in further works.

4.3 Core Blank Element Selection Algorithm using PDG

In this section, we present the *core blank element selection algorithm* for selecting blank elements from the core statements using PDG of the code.

4.3.1 Overview

The important elements or *core elements* in the source code should be blanked for the *element fill-in-blank problem* to be more effective in studying the related grammar and the code reading. In this thesis, the statements related with many statements in the code are regarded as the core statements that include the core elements. Thus, to find core statements, the PDG of the source code is generated and the statements that have high degrees in the PDG are regarded as the core statements.

4.3.2 Threshold Selection

To find the core statements in a code using the PDG, it is essential to select the proper value for the *threshold* of the PDG degree depending on the code. The following procedure describes the procedure for the threshold selection:

1) The PDG is generated for the given code using the procedure in Section 4.2.
2) The degrees of all the vertices in the PDG are calculated.
3) The PDG degrees are sorted in the descending order to make the sorted list of the PDG degrees.
4) The number of core statements to be extracted, which is named K, is calculated by the following equation:
$$K = NOS * SP/100 \tag{4.1}$$
where NOS represents the total number of statements in the code, and SP does the percentage of the core statements to be extracted in the code that should be properly given by the user. It has been observed that $SP = 25$ is the best choice for the core statement extraction generally from the experimented Java source codes.
5) The K-th degree in the sorted list made in 1) is selected for the threshold. If it is 0, the smallest non-zero PDG degree is selected for the threshold.

4.3.3 Modification of Clique Extraction

To select the blank elements from the core statements only in the code, the clique extraction step in the blank element selection algorithm is modified. Here, the elements in the statements whose PDG degrees are larger than or equal to the *threshold* are considered in the clique extraction.

4.4 Evaluations

In this section, we evaluate the effects of *core element fill-in-blank problem* using four Java codes implementing graph or fundamental algorithms.

Table 4.1: PDG property of adopted Java codes

ID	code	# of vertices	# of edges	edge density	thre-shold	# of cores
P1	Dijkstra	96	132	0.03	3	24
P2	Prim	95	116	0.03	2	24
P3	KMP Search	50	135	0.11	6	13
P4	Knapsack	44	94	0.09	10	11

4.4.1 Java Source Codes

In this evaluation, the four Java source codes implementing *Dijkstra*, *Prim*, *KMP search*, and *Knapsack with dynamic* algorithms are adopted [26][27]. These algorithms have been often taught in corresponding classes in universities. These codes have multiple classes and methods so that they become longer than codes for grammar studies.

Then, the previous *blank element selection algorithm* and the proposed one were applied to the four codes, and generated the corresponding element fill-in-blank problems. For the parameters in the algorithms, $BG = 3$, $CB = 1$, and $SP = 25\%$ were used. Six student in our group who have the better knowledge in Java programming than conventional students in a Java programming course were asked to solve the four problems.

4.4.2 PDG Property of Adopted Codes

First, the property of the generated PDG for each Java code is examined. Table 4.1 shows the number of vertices, the number of edges, and the edge density in the PDG, the threshold for selecting core statements, and the number of core statements obtained by the proposed algorithm for each code. The edge density is calculated by the following equation:

$$density = NOE/[NOV * (NOV - 1)/2] \tag{4.2}$$

where NOE represents the total number of edges and NOV represents the number of vertices in the graph. The results show that the two codes for graph theory algorithms have the larger number of statements (LOC) than the two codes for fundamental algorithms, but have the smaller edge density in the PDG graph. This means that the latter codes have the more complex dependency among statements, and their structure can be more difficult to be understood.

4.4.3 Number of Selected Blanks

Next, the number of blank elements is compared between the previous algorithm and the proposed one. Table 4.2 shows the number of blanks by both algorithms. Because the codes for graph theory algorithms have the larger LOC, the number of selected blank elements is larger for both algorithms. When the number of blank elements is compared between the two algorithms, the proposed one can reduce it by about 30% by limiting the selections from core statements.

Table 4.2: Number of selected elements by two algorithms

ID	code	LOC	# of blanks	
			previous	proposal
P1	Dijkstra	96	72	42
P2	Prim	95	71	42
P3	KMP Search	50	34	22
P4	Knapsack	44	42	18

Table 4.3: Student assigned problem and correct rate

ID	S1	S2	S3	S4	S5	S6
group	A	A	A	B	B	B
P1	2	1	2	1	2	1
P2	1	2	1	2	1	2
P3	2	1	2	1	2	1
P4	1	2	1	2	1	2
correct rate (%)	97%	92%	96%	99%	94%	90%

4.4.4 Solution Performance of Students

Next, the solution performance of the students are evaluated in the problems. To let the same number of students solve each problem and avoid the same student solving the both problems from the same code, the problems are assigned to each student as shown in Table 4.3. In this table, "1" indicates the problem by the previous algorithm and "2" does the problem by the proposed algorithm. Actually, the six students are divided into two groups, A ($S1$, $S2$, $S3$) and B ($S4$, $S5$, $S6$), and assigned the same set of the problems to the three students in each group. For reference, the average correct answer rate for the four problems by each student is also shown there. The student $S4$ shows the best performance and the student $S2$ does the worst one among them.

Table 4.4 shows the average correct answer rate for each problem among the six students. The rate for the problems generated by the previous algorithm is generally higher than the rate by the proposed one except for $P1$, although they have larger number of blanks. This means that the students must understand the algorithm/logic to solve the element fill-in-blank problems by the proposed algorithm. Besides, in both problems, the correct rate for $P3$ is the smallest among the other problems because the edge density is the highest among the codes and it has the complex structure with the highest dependency among the statements.

4.4.5 Questionnaire Evaluations

Finally, we asked the students to answer the four questions in Table 4.5 as the questionnaire. For $Q1$, the approximate time spent to solve each problem is answered by four levels: 1 indicates less than 10 min., 2 does about 20 min., 3 does about 40 min., and 4 does longer

Table 4.4: Solution performance for each problem

ID	LOC	correct rate (%)	
		previous	proposal
P1	96	94%	95%
P2	95	97%	92%
P3	50	91%	90%
P4	44	95%	94%

Table 4.5: Questions in questionnaire

ID	question
Q1	How long did you spend to answer each problem?
Q2	Which one of the previous or the proposal is helpful for implementing the algorithm Java code?
Q3	Which one of the previous or the proposal is easier to solve the fill-in-blank problem?
Q4	Which one of the previous or the proposal is useful for reading the algorithm Java code?

than 40min. Then, for $Q2$-$Q4$, “yes” or “no” questions is answered. Table 4.6 shows the result by each student.

From $Q1$, the student $S2$ spent long time to solve both fill-in-blank problems in graph theory algorithms and fundamental algorithms.

For $Q2$, fours replied that the fill-in-blank problems by the both algorithms are helpful for implementing the algorithm Java codes. On the other hand, the student $S1$ in *Group A* and the student $S4$ in *Group B* replied that the problems by the previous algorithm are not helpful for implementing the Java codes. These two students have higher programming skill among the students, where their correct answer rates are the highest in each group.

Table 4.6: Questionnaire results by students

		S1	S2	S3	S4	S5	S6
Q1	previous	2	4	1	2	3	2
	proposal	1	4	1	2	2	2
Q2	previous	no	yes	yes	no	yes	yes
	proposal	yes	yes	yes	yes	yes	yes
Q3	previous	no	yes	no	no	no	yes
	proposal	yes	no	yes	yes	yes	no
Q4	previous	no	yes	yes	yes	yes	yes
	proposal	yes	yes	yes	yes	yes	yes

For $Q3$, the student $S2$ in *Group A* and the student $S6$ in *Group B* replied that the problems by the previous algorithm are easier than the problems by the proposal. However, it was found that the student $S2$ spent long time to solve these problems. Besides, the correct rates of both students are smaller than the others in each group.

For $Q4$, five students replied that the element fill-in-blank problems by both algorithms are useful for reading algorithm Java codes, and student $S1$ replied that only the problems by the proposal are useful for reading algorithm Java codes.

From these observations, it is concluded that generally, the *core element fill-in-blank problem* is more helpful for reading and implementing algorithm Java source codes, although further investigations through applying a number of problems to the sufficient number of students are necessary to strengthen this conclusion.

4.4.6 Generated Problem Example

For reference, we show an example *core element fill-in-blank problem* generated by the proposed algorithm from the Java code for `Knapsack`. In this code, the lines 17, 18, 20, 21, 26, 27, 28, 30, 34, 35, 36, 37 are selected as the core statements.

Listing 4.1: Core element fill-in-blank problem for *KnapSack*

```
1  class Knapsack {
2    static class Product {
3      public final int size;
4      public final int value;
5      public Product(int size, int value) {
6        this.size = size;
7        this.value = value;
8      }
9    }
10   private static Product[ ] products = {
11     new Product(2, 2), new Product(3, 4),
12     new Product(5, 7), new Product(6, 10),
13     new Product(9, 14) };
14    public static void main(String [ ] args) {
15      int napValue[ ] = new int[products.length + 1];
16      System.out.print("the size of knapsack");
17      _1_ ( int  i = 0; i < products.length; i++) {
18          napValue [ _2_ ] = 0;
19          String num = (i + 1) + "  ";
20          _3_  (num.length() == 2) {
21          _4_  = "  " + num;
22         }
23       System.out.print (num);
24     }
25     System.out.print ("\n\n");
26     for ( _5_  i = 0; i < products.length; i++){
27       _6_  (int j =  _7_ [i].size;  _8_  <
28                     products.length + 1; ++j){
29          _9_  newValue = napValue[ _10_ −
30                       products[ _11_ ].size] +
31                       products[i].value;
32         if (newValue > napValue[j]) {
33           _12_ [j] = newValue;
34         }
35       }
36      System.out.print("Use the goods until"+(i+1));
```

```
37        for ( _13_  j = 1; j < products.length + 1
38                                          ; j++){
39              _14_  num = napValue[ _15_ ] + "  ";
40              _16_  (num.length() == 2) {
41              _17_  = "  " +  _18_ ;
42             }
43           System.out.print(num);
44        }
45       System.out.println();
46     }
47   }
48 }
```

4.5 Summary

In this chapter, we presented the *core element fill-in-blank problem* for enhancing the code reading studies by novice students. The effectiveness of *core element fill-in-blank problem* was evaluated through applications to four Java codes in the graph theory or fundamental algorithms. The future studies include the improvement of PDG generation method to consider the dependency between statements beyond a method, and the generation of fill-in-blank problems by applying the algorithm to assign them to students in Java programming course for evaluations of the educational effects.

Chapter 5

Value Trace Problem

In this chapter, we present the *value trace problem* in JPLAS.

5.1 Introduction

The *value trace problem* is proposed as a new type of *fill-in-blank problem* in JPLAS. This problem asks students to trace the actual values of the important variables in a source code implementing a data structure or an algorithm. To trace the values of the important variables, the whole data in the line of the output data sets that are obtained by executing the code, where at least one data is changed from the previous one, is blanked by applying the *blank line selection algorithm*. Here, it is noted that the *line* intends all the data of the array of the variables at one step or one iteration, where the target algorithm, such as the sorting, usually handle multiple data and change their values iteratively.

In this chapter, some related works of the value trace problem is first introduced. Then, the value trace problem with *insertion sort* and *blank line selection algorithm* are presented respectively. Next, the value trace problems for graph algorithms are studied. After that, our proposal are evaluated. Finally, the conclusion is given for this chapter.

5.2 Related Works

In this section, we briefly introduce some partially related works of the *value trace problem*. However, in our survey, no work has been reported for the same problem.

In [33], Smulders presented the *Annotate Code* project for explaining algorithms in introductory programming courses to students that have not yet developed a mental image of them. It allows teachers to create visualizations based on code stepping. As Web applications, users can submit codes and steps through a browser. Each step can be accompanied by a user-generated drawing to creating a step-by-step animation like a debugger.

In [34], Quinson et al. presented the *Programmer's Learning Machine (PLM)* as an interactive exerciser aimed at learning programming and algorithms. It targets students in (semi-)autonomous settings, using an integrated and graphical environment that provides a short feedback loop. This generic platform also enables teachers to create specific programming microworlds that match their teaching goals. PLM provides two main panels to provide information for students to solve exercises.

In [35], Sykes et al. presented the Web-based *Java Intelligent Tutoring System (JITS)* for students in first programming courses. By bringing together recent developments in intelligent tutoring systems, cognitive science, and AI, it constructs an intelligent tutor to help students learn Java programing.

In [36], Osman et al. introduced a visualized learning system to enhance the education of data structure course. It has the capability to display data structure graphically as well as allow its graphical manipulation for students to observe the execution result and track the algorithm execution.

5.3 Concept of Value Trace Problem

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

5.3.1 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 detail of each step in the value trace problem generation procedure except for step 4), using a Java code for *Insertion sort* [37][38]. *Insertion sort* maintains the sorted data list at the lower positions of the input data list. A new data in the input data list is inserted into the sorted list such that the largest data is located at the last position of the expanded sorted list. Thus, the input data list after k iterations has the property where the first $k+1$ entries are sorted. Here, the following code for *Insertion sort* is adopted.

5.3.2 Step 1): Selection of Java Code for Insertion Sort

In Step 1), the following Java source code for *Insertion Sort* in [39] is selected.

Listing 5.1: Source code for *InsertionSort*

```
1  class InsertionSort{
2        //input data is arr[ ]
3        public static void sort(int[ ] arr){
4              int i, j;
5              int tmp; //item to be inserted
6              //start with 1 (not 0)
7              for (i=1; i<arr.length; i++){
8                    tmp = arr[i];
9                    //smaller values are moving up
10                   for (j=i ; j>0 && arr[j−1]> tmp; j−−){
11                         arr[j] = arr[j−1];
12                   }
13                   arr[j] = tmp;
14             }
15       }
16 }
```

5.3.3 Step 2): Generation of Main Class

Some Java codes may not include *main method* and contain only *class* like the code for *Insertion Sort* in Section 5.3.2. For convenience, it is called *algorithm class*. Then, it is necessary to generate *main class* to contain *main method* to instantiate the class to run the code and read input data of the algorithm as well as write the output data. In the following *main class*, the input data is also described, so that the input file in Step 4) can be skipped.

Listing 5.2: *Insertionsort* main class

```
1  class Insertionsort_Main{
2     public static void main(String args[ ]){
3        int[ ] a= new int[ ] {2,1,3,5,4,7,6,8,9,10};
4        InsertionSort.sort(a);
5     }
6  }
```

5.3.4 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 [40]. 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 becomes necessary to add the functions of writing such variable values into a text file in *main class* and *algorithm class*.

In *Insertion sort*, the values of the variables for the sorted data are essential for understanding the algorithm, and should be traced at each iteration by students. Thus, the function of writing these values of variables into a text file is added as functions to complete the *problem code* in generating a value trace problem as follows:

Listing 5.3: *InsertionSort* code with output function

```
class InsertionSort{
      //input data is arr[ ]
       public static void sort(int[ ] arr){
            int i, j;
            int tmp; //item to be inserted
            //start with 1 (not 0)
            for (i=1; i<arr.length; i++){
                 tmp = arr[i];
                 //smaller values are moving up
                for (j=i ; j>0 && arr[j−1]> tmp; j−−){
                    arr[j] = arr[j−1];
                }
                arr[j] = tmp;
                for(int k:arr){
                    System.out.print(k);
                    System.out.print(",");
                }
               System.out.println();
          }
     }
}
```

5.3.5 Step 5): Obtaining Output Data File

After running the *problem code*, the complete output text file for the value trace problem is given as follows:

Listing 5.4: Output data file for *InsertionSort*

```
1,2,3,5,4,7,6,8,9,10
1,2,3,5,4,7,6,8,9,10
1,2,3,5,4,7,6,8,9,10
1,2,3,4,5,7,6,8,9,10
1,2,3,4,5,7,6,8,9,10
1,2,3,4,5,6,7,8,9,10
1,2,3,4,5,6,7,8,9,10
1,2,3,4,5,6,7,8,9,10
1,2,3,4,5,6,7,8,9,10
```

5.3.6 Step 6): Blanking Values for Problem Generation

To let students trace the data sorting results, the whole line in the output text file is blanked such that at least one data in the line is changed from the previous line, in addition to the first line. Then, the corresponding lines is blanked in the text file as follows:

Listing 5.5: Blanked data file for *InsertionSort*

```
___ , ___ , ___ , ___ , ___ , ___ , ___ , ___ , ___, ___
 1  ,  2  ,  3  ,  5  ,  4  ,  7  ,  6  ,  8  ,  9 ,  10
 1  ,  2  ,  3  ,  5  ,  4  ,  7  ,  6  ,  8  ,  9 ,  10
___ , ___ , ___ , ___ , ___ , ___ , ___ , ___ , ___, ___
 1  ,  2  ,  3  ,  4  ,  5  ,  7  ,  6  ,  8  ,  9 ,  10
___ , ___ , ___ , ___ , ___ , ___ , ___ , ___ , ___, ___
 1  ,  2  ,  3  ,  4  ,  5  ,  6  ,  7  ,  8  ,  9 ,  10
 1  ,  2  ,  3  ,  4  ,  5  ,  6  ,  7  ,  8  ,  9 ,  10
```

9 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10

5.3.7 Step 7): Generating Assignment

After preparing the final Java codes of *main class* and *algorithm class*, the blanked text file, and the correct answer file, all of them are uploaded to the JPLAS server using the existing function. Then, the brief descriptions on the algorithm and the problem can be added to help students to understand the problem.

Figure 5.1 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 source code and the blanked text file.

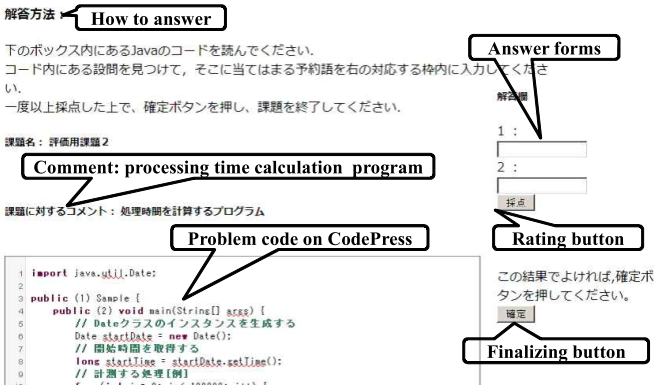

Figure 5.1: Interface for assignment answering

5.4 Blank Line Selection Algorithm

In this section, we present the *blank line selection algorithm* for Step 6).

5.4.1 Idea

In this algorithm, the whole data in one line is blanked in the output text file from the *problem code*. To create a more difficult problem, the line should be selected such that at least one data is changed from the previous line, which is called *change line*. Nevertheless, the number of lines to be blanked or *target lines*, can be specified by the teacher. If the number of *change lines* is smaller than the number of *target lines*, select all the *change lines* and randomly select the remaining number of lines from non *change lines*. If the number of *change lines* is larger, randomly select *change lines* by this number.

5.4.2 Procedure

The procedure for the *blank line selection algorithm* is described as follows:

1. Calculate the number of *target lines* to be blanked ($targetLine$) from the algorithm input parameter ($blankRate$) and the total number of lines in the output text file ($totalLine$) by $targetLine$=$blankRate/100 * totalLine$.

2. Count the number of changed data in each line from the previous one in the output text file.

3. Count the number of *change lines* in the output text file ($changeLine$) such that the number in **2** is not zero.

4. If $changeLine$=$targetLine$, then select all of the *change lines* for blanks.

5. If $changeLine$<$targetLine$, then select ($targetLine - changeLine$) non *change lines* to be blanked by repeating the following procedure:

1) Calculate the selection rate ($selectRate$) by $selectRate$=($targetLine$−$changeLine$) /($totalLine - changeLine$).
2) Initialize the number of the selected blank lines ($selectLine$) by $changeLine$.
3) Repeat the following steps:
   (1) Visit the first line in the output text file.
   (2) If this line has been selected to be blanked, go to (4).
   (3) If $random$<$selectRate$, then select this line to be blanked, and count up by $selectLine++$, where $random$ returns a 0-1 random real number.
   (4) If $selectLine$=$targetLine$, then terminate the procedure.
   (5) If the current line is not the last line in the output text file, then visit the next line and go to (2).
   (6) Go to (1).

6. If $changeLine$>$targetLine$, then select ($targetLine$) *change lines* to be blanked by repeating the following procedure:

1) Calculate the selection rate ($selectRate$) by $selectRate$=$targetLine/changeLine$.
2) Initialize the number of the selected blank lines ($selectLine$) by 0.
3) Repeat the following steps:
   (1) Visit the first line in the output text file.
   (2) If this line has been selected to be blanked, go to (4).
   (3) If $random$<$selectRate$, then select this line to be blanked, and count up by $selectLine++$.
   (4) If $selectLine$=$targetLine$, then terminate the procedure.
   (5) If the current line is not the last line in the output text file, then visit the next line and go to (2).
   (6) Go to (1).

5.4.3 Example Problem for Insertion Sort

For the sample code of Section 5.3.2, this algorithm calculates $targetLine = 5$ (=$50/100 * 9$) by choosing $blankRate = 50$. It first selects the three *change lines*, then randomly selects the two non *change lines*. Then, we obtain the following result:

Listing 5.6: Blanked data file for *InsertionSort* by algorithm

```
1 ---, ---, ---, ---, ---, ---, ---, ---, ---, ---
2  1 ,  2 ,  3 ,  5 ,  4 ,  7 ,  6 ,  8 ,  9 ,  10
3 ---, ---, ---, ---, ---, ---, ---, ---, ---, ---
4 ---, ---, ---, ---, ---, ---, ---, ---, ---, ---
5  1 ,  2 ,  3 ,  4 ,  5 ,  7 ,  6 ,  8 ,  9 ,  10
6 ---, ---, ---, ---, ---, ---, ---, ---, ---, ---
7  1 ,  2 ,  3 ,  4 ,  5 ,  6 ,  7 ,  8 ,  9 ,  10
8 ---, ---, ---, ---, ---, ---, ---, ---, ---, ---
9  1 ,  2 ,  3 ,  4 ,  5 ,  6 ,  7 ,  8 ,  9 ,  10
```

5.5 Value Trace Problem for Dijkstra Algorithm

In this section, we discuss the value trace problem for *Dijkstra algorithm* as the representative *graph theory algorithm*.

5.5.1 Background

In the graph theory, several important algorithms exist to be studied by students. They include *Djkstra algorithm*, *Prim algorithm*, *Breadth first search algorithm (BFS)*, *Depth first search algorithm (DFS)*, and *Maximum flow algorithm*. These algorithms have been used in a lot of important practical applications in computer systems, information systems, and communication networks [15].

5.5.2 Dijkstra Algorithm

It computes a solution to the single source shortest path problem for a weighted graph $G = (V, E)$ where each edge in E has a non-negative weight [16].

1. It starts by assigning initials values for the distances from the starting node s to the other nodes in G.

2. It operates in steps where the shortest distance from node s to another node is improved.

5.5.3 Pseudo Code for Dijkstra Algorithm

The pseudo code for *Dijkstra* algorithm is described as follows:

Listing 5.7: Pseudo code for *Dijkstra*

```
1 begin
2 for each vertex u in V
3 begin
4    dist[v]=infinity
```

```
5     pred[v]=NULL
6   end
7   add all the vertices in V to Q
8   while (!ISEMPTY (Q))
9     begin
10      Extract from Q a vertex u such that dist[u] is minimum
11          remove u from Q
12          for each vertex v adjacent to u do
13          if dist[v]> dist[u]+w(u,v)
14          then
15            begin
16              dist[v]=dist[u]+w(u,v)
17              pred[v]=u
18            end
19      end
20  end
```

For this pseudo code, a connected weighted graph $G = (V, E)$ with a cost function w to an edge, and a source node s are given as the inputs, and a shortest path tree T is generated as the output. In this pseudo code, `dist[v]` represents the minimum weight of the path from s to v, which is initialized by `INFINITY` that represents a larger value that any path weight. The queue `Q` contains all the nodes whose shortest paths have not been found. `pred[v]` represents the parent vertex of v in T.

5.5.4 Java Classes

For the generation of the *value trace problem* for *Dijkstra algorithm*, three Java classes are adopted, namely, *WeightedGraph* class, *PrimMethod* class, and *Main* class from [41].

5.5.4.1 *WeightedGraph* Class

WeightedGraph class defines the necessary procedures to handle the input graph for *Dijkstra algorithm*. It contains the methods of *setLabel*, *getLabel*, *addEdgeWeight*, *getEdgeWeight*, and *neighbors*. *setLabel* method sets the label for the specified vertex. *getLabel* method gets the label of the specified vertex. *addEdgeWeight* method assigns the weight to the edge specified by the incident vertices. *setLabel* and *addEdgeWeight* methods are used to generate a weighted connected graph. *neighbors* method returns the list of the neighbor vertices of the vertex in the argument, which is used for line 12 in the pseudo code in Section 5.5.3. *getEdgeWeight* method returns the weight of the edge in the argument, which is used for lines 13 and 16 in the pseudo code.

Listing 5.8: Source code for *WeightedGraph* class

```
1  public class WeightedGraph{
2    // adjacency matrix
3     public int [ ][ ] edges;
4     public String [ ] lables;
5     public WeightedGraph(int n){
6       edges=int[n][n];
7       lables=new String[n];
8     }
9     public int size(){
10      return labels.length;
11    }
12    public void setLabel(int vertex, String label){
```

```
13      labels[vertex]=label;
14    }
15    public Object getLabel(int vertex){
16      return labels[vertex];
17    }
18    public void addEdgeWeight(int source, int target, int w){
19        edges[source][target] = w;
20    }
21    public int getEdgeWeight(int source, int target){
22      return edges[source][target];
23    }
24    public int [ ] neighbors(int vertex){
25        int count = 0;
26        for(int i=0;i<edges[vertex].length; i++){
27            if (edges[vertex][i]>0)
28                answer[count++]=i;
29        }
30        final int[ ]answer= new int[count];
31        count=0;
32        for(int i=0;i<edges[vertex].length;i++){
33            if (edges[vertex][i]>0)
34                answer[count++]=i;
35      }
36      return answer;
37    }
38  }
```

5.5.4.2 *DijkstraMethod* Class

DijkstraMethod class produces the shortest path tree T, by finding the shortest path from the source node s to another node sequentially in the ascending order of the distance in the graph G [42].

Listing 5.9: Source code for *DijkstraMethod* class

```
1   public class DijkstraMethod{
2     public static int[] dijkstra(WeightedGraph G, int s){
3     final int [ ]dist = new int[G.size()];
4     final int [ ]pred = new int[G.size()];
5     int sum=0;
6     final boolean[ ] visited=new boolean[G.size()];
7       for (int i=0; i<dist.length; i++){
8         //set all distances to maximum value
9         dist[i] = Integer.MAX_VALUE;
10      }
11        //initialize distance of source to zero
12        dist[s] = 0;
13        for (int i=0; i<dist.length; i++){
14            final int u = minVertex(G, dist, visited);
15            visited[u] = true;
16             //each vertex adjacent to u
17          final int [ ] n = G.neighbors(u);
18          for (int j=0; j<n.length; j++){
19              final int v = n[j];
20              final int d = dist[u]+G.getEdgeWeight(u,v);
21                if (dist[v] > d){
22                    dist[v] = d;
23                    pred[v] = u;
```

```
24              }
25          }
26      }
27      return pred; //(ignore pred[s]==0!)
28    }
29    private static int minVertex(WeightedGraph G, int [ ] dist, boolean [ ] v){
30      int x = Integer.MAX_VALUE;
31      //graph not connected
32      int y = -1;
33      for (int i=0; i<dist.length; i++){
34          if (!v[i] && dist[i]<x){
35              y=i;
36              x=dist[i];
37          }
38        }
39      sum = sum + x;
40      System.out.println("selected vertex:"+G.getLabel(y)+", the edge weight: "+x
        );
41      return y;
42    }
43  }
```

5.5.4.3 *Main* Class

Main class generates an input graph to the algorithm using *WeightedGraph* class, and find the shortest path tree using *DijkstraMethod* class.

Listing 5.10: *Dijkstra* main class

```
1  class DijkstraMain{
2    public static void main (String[ ] args) {
3        //initialize the weighted graph
4        final WeightedGraph t = new WeightedGraph(6);
5        // set label for six vertices
6         t.setLabel(0, "v0");
7         t.setLabel(1, "v1");
8         t.setLabel(2, "v2");
9         t.setLabel(3, "v3");
10        t.setLabel(4, "v4");
11        t.setLabel(5, "v5");
12        t.addEdgeWeight(1,0,2);
13        t.addEdgeWeight(2,1,8);
14        t.addEdgeWeight(2,4,7);
15        t.addEdgeWeight(3,1,15);
16        t.addEdgeWeight(3,2,1);
17        t.addEdgeWeight(3,4,3);
18        t.addEdgeWeight(4,5,3 );
19        t.addEdgeWeight(5,0,9);
20        t.addEdgeWeight(5,1,6);
21        DijkstraMethod.dijkstra(t,0);
22        System.out.println("The total least cost"+sum);
23    }
24  }
```

5.5.5 Generated Value Trace Problem

By running the Java class codes in Section 5.5.4, the following output file is obtained:

Listing 5.11: Output data file for *Dijkstra*

```
selected vertex : v0 , the edge weight: 0
selected vertex : v1 , the edge weight: 2
selected vertex : v5 , the edge weight: 8
selected vertex : v2 , the edge weight: 10
selected vertex : v3 , the edge weight: 11
selected vertex : v4 , the edge weight: 11
The total least cost: 42
```

Then, by blanking all the values of the selected vertices and the edge weights, the following value trace problem is generated. Then, the students are asked to fill in the blanks by reading the codes in Section 5.5.4, which are called *problem codes* for convenience.

Listing 5.12: Blanked data file for *Dijkstra*

```
selected vertex:_1_,the edge weight:_2_
selected vertex:_3_,the edge weight:_4_
selected vertex:_5_,the edge weight:_6_
selected vertex:_7_,the edge weight:_8_
selected vertex:_9_,the edge weight:_10_
selected vertex:_11_,the edge weight:_12_
total least cost:_13_
```

5.6 Evaluation for Sorting Algorithms

In this section, we evaluate the effectiveness of value trace problems for sorting algorithms.

5.6.1 Five Value Trace Problems for Sorting Algorithms

In this evaluation, the five value trace problems are generated by using the Java codes for *selection sort*, *insertion sort*, *bubble sort*, *quick sort* [43] and *shell sort* [13]. These algorithms are commonly taught in universities. Table 5.1 shows the problem outlines.

Table 5.1: Five value trace problems for evaluations

ID	algorithm	LOC	# of blanks
P1	Selection sort	27	24
P2	Insertion sort	27	23
P3	Bubble sort	32	10
P4	Quick sort	44	43
P5	Shell sort	37	22

5.6.2 Solution Performances by Students

Then, 10 students in our group who have different skills and knowledge in Java programming, are asked to solve them using JPLAS. After that, they are requested to answer the five questions in Table 5.2 for the questionnaire. For Q1, students should reply with five levels, where 1 is the easiest and 5 is the most difficult. For Q2, they should reply 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. Then, for Q3-Q5, students should reply with *yes* or *no* for all five problems.

Table 5.2: Questions in questionnaire

ID	question
Q1	How difficult is each problem ?
Q2	How long did you spend to answer each problem ?
Q3	Do you understand the algorithm in the code by solving the problems ?
Q4	Do you think the value trace problem is useful for Java code reading ?
Q5	Can you implement the algorithm in Java code by solving the problems ?

Table 5.3 shows the results for the individual problems. Here, the results show the number of students who solved each problem correctly and the average number of their answer submissions, where JPLAS can record the submission numbers. This table indicates that among the five value trace problems, the problem for *Quicksort* is the most difficult since two students were not able to solve it and the average number of submissions as well as the average difficulty and spending time levels are the highest. The reason will be analyzed in Section 5.6.3.

Table 5.3: Solution and questionnaire results

ID	# of solving students	ave. # of submissions	ave. level for Q1	ave. level for Q2
P1	10	2.5	1.3	1.5
P2	9	3.2	1.3	1.6
P3	10	1.8	1.5	1.8
P4	8	12.8	3.2	3.5
P5	10	4.7	2	2.4

Table 5.4 shows the results for Q3-Q5. From Q3 and Q4, nine students among 10 replied that the value trace problem in JPLAS is effective in understanding the algorithm in the Java code and the code reading. However, for Q5, only seven students replied that they have confidence in writing a code for the algorithm even after solving them. From these results, we conclude that the value trace problem is useful and effective for Java code reading, but may not be sufficient for Java code implementations of algorithms.

Table 5.4: Questionnaire results on effectiveness of value trace problem

	Q3	Q4	Q5
yes	9	9	7
no	1	1	3

5.6.3 Difficulty Analysis of Quick Sort

In the previous subsection, the value trace problem for *Quick Sort* is the most difficult. Our analysis on the reason is that *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, called *partitioning*. Then, it recursively applies the same procedure to each sub-list at the left side and the right side of the pivot, until the whole list is sorted [43].

In the following problem for *Quick Sort*, the codes from line 1 to line 38 describe the *algorithm class* with added output functions to a text file, the codes from line 39 to line 44 describe *main class*, while the codes from line 46 to line 60 depicts the problems yet to be solved by students. 43 blanks are prepared for students to fill in the correct values. 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, students should trace the values of p from the first one to the last and the data arrangement results for each p.

Listing 5.13: Value trace problem for *QuickSort*

```
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("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("output: ");
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  <Problem>
46  pivot: _1_
47  output: _2_,_3_ ,_4_ ,_5_,_6_, _7_ ,_8_ ,_9_ ,_10_
48  pivot: _11_
49  output: _12_,_13_,_14_,_15_,_16_,_17_,_18_,_19_,_20_
50  pivot: _21_
51  output: 50 , 45 , 55 , 60 , 65 , 85 , 80 , 75 , 70
52  pivot: _22_
53  output: _23_,_24_,_25_,_26_,_27_,_28_,_29_,_30_,_31_
54  pivot: _32_
55  output: _33_,_34_,_35_,_36_,_37_,_38_,_39_,_40_,_41_
56  pivot: _42_
57  output: 45 , 50 , 55 , 60 , 65 , 70 , 80 , 75 , 85
58  pivot: _43_
59  output: 45 , 50 , 55 , 60 , 65 , 70 , 75 , 80 , 85
```

5.7 Evaluation for Graph Theory Algorithms

Next, we evaluate the effectiveness of value trace problems for graph theory algorithms.

5.7.1 Size of Generated Value Trace Problems

In this evaluation, first, we evaluate the problem size through the generated seven value trace problems for *Dijkstra*, *Kruskal*, *Prim*, *Breath First Search (BFS)*, *Depth First Search (DFS)*, *Traveling Salesman Problem (TSP)*, and *Maximum Flow* as important graph theory algorithms, using their Java codes in [17][18][19][20]. Table 5.5 shows the number of vertices in the adopted graph, *LOC* (the number of lines) in the problem code, and the number of blanks for each value trace problem.

For comparisons, Table 5.6 shows the number of the input data, LOC in the problem code, and the number of blanks for fundamental data structures or algorithms in [45]. We note that the number of blanks is obtained here after applying the blank selection algorithm such that 50% of the output data lines can be blanked. When we compared the results in the two tables, we found that value trace problems for graph theory algorithms have more LOC and a fewer number of blanks. One reason of the larger LOC is the graph generation procedure of giving labels to the vertices and the weights and the incident vertices to the edges in the code. This indicates that reading out the problem code for a graph theory

Table 5.5: Size of value trace problems for graph theory algorithms

ID	algorithm	# of vertices	LOC	# of blanks
P1	Dijkstra	6	97	13
P2	Prim	6	97	13
P3	BFS	8	85	8
P4	DFS	8	79	8
P5	TSP	5	98	13
P6	Kruskal	7	176	19
P7	Maximum flow	6	141	19

Table 5.6: Size of value trace problems for fundamental data structures or algorithms

ID	algorithm	# of data	LOC	# of blanks
P8	Stack	10	33	20
P9	Queue	10	32	20
P10	Selection sort	8	27	24
P11	Insertion sort	6	27	23
P12	Bubble sort	5	32	10
P13	Quick sort	9	44	43
P14	Shell sort	5	37	22
P15	Merge Sort	8	51	24
P16	Heap Sort	6	63	24

algorithm is generally more difficult and takes more time than doing so for a fundamental data structure or algorithm. In other words, the improvement of displaying the problem code to students is very important to reduce the difficulty in solving the problem by them.

5.7.2 Solution Performances by Students

Then, we evaluate the solution performance by asking five students who have sufficient Java programming skills in our group, to solve the value trace problems for *Prim* and *Dijkstra* algorithms. From Table 5.5, the problem codes for them have the same LOC. Actually, their algorithm structures and the codes are very similar. To investigate the difference by using the *one-column* format or the *two-column* format and by using the online form or the PDF file for displaying the problem codes to students, we prepared the *one-column/online* for *Prim* and the *two-column/PDF* for *Dijkstra* in JPLAS.

After solving the two value trace problems correctly, we asked the students to answer the five questions in Table 5.7 as the questionnaire. For Q1, the difficulty of solving each problem is denoted by four levels, where 1 is the easiest and 4 is the most difficult. For Q2, the approximate time spent to solve each problem is denoted by four levels, where 1 is less than $10min.$, 2 is about $15min.$, 3 is about $20min.$, and 4 is longer than $25min.$ Then, for Q3-Q5, we used "yes" or "no" questions to ask the students for these problems.

Table 5.7: Questions in questionnaire

ID	question
Q1	How difficult is each problem?
Q2	How long did you spent to answer each problem?
Q3	Can you easily read and understand the algorithm Java code to solve the problem in JPLAS?
Q4	Can you easily read and understand the algorithm Java code to solve the problem in two column PDF format?
Q5	Do you think the value trace problem for graph theory is useful to improve the java code reading and understanding?

Table 5.8: solutions and questionnaire results

ID	Q1		Q2		Q3	Q4	Q5
	Dijkstra	Prim	Dijkstra	Prim			
S1	3	4	3	4	no	yes	yes
S2	4	4	4	4	yes	no	no
S3	4	4	4	4	yes	no	yes
S4	4	4	4	4	yes	yes	yes
S5	4	4	4	4	yes	no	yes

Table 5.8 shows the results for this questionnaire. For Q1 and Q2, four students replied that the difficulty of the problem is the highest and took long time to solve them. When compared between the *two-column/PDF* for *Dijkstra* and the *one-column/online* for *Prim*, only student S1 replied that the former is easier than the latter, whereas the others replied the same. This indicates the *two-column/PDF* format slightly improves the user interface in displaying the problem code.

For Q3, four students replied that they can easily read and understand the problem code, whereas one student has difficulty to solve the problem. For Q4, two students replied that the *two-column/PDF* format is easy to read and understand the problem code, whereas three students replied that it is not. This indicates that the problem code format still needs to be improved. For Q5, four students replied that value trace problems for graph theory algorithms are useful to improve the Java code reading.

From the results, we conclude that value trace problems for graph theory algorithms are generally useful to improve the Java code reading capability of students, although they can be difficult for Java novice students due to long problem codes. In future works, we need to improve the user interface in displaying the problem code and have their extensive evaluations.

5.8 Summary

In this chapter, we presented the *value trace problem* in JPLAS for studying algorithm Java code reading by students. The effectiveness of the value trace problem was evaluated through the generated problems using Java codes for sorting and graph theory algorithms. The future studies include the improvement of the user interface in displaying the problem code, and their extensive evaluations through the applications to students in Java programming courses.

Chapter 6

Workbook Design for Fill-in-blank Problems

In this chapter, we present the *workbook design* for the *element fill-in-blank problem*, the *core element fill-in-blank problem*, and the *value trace problem* in JPLAS.

6.1 Introduction

In this thesis, a workbook of the three fill-in-blank problems is designed for use in a Java programming course to enhance the self-studies of Java programming by novice students. This workbook consists of 15 categories that are arranged in the conventional learning order of Java programming, and each category has a considerable number of problems. A set of suitable Java source codes is collected from textbooks and Web sites for the introductory Java programming.

In this chapter, the workbook design of the three problems in JPLAS is presented. Then, the generation of element fill-in-blank problems using the workbook and their evaluations are discussed. Finally, the conclusion is given for this chapter.

6.2 Review of Three Fill-in-blank Problems

In this section, we review the three fill-in-blank problems in JPLAS.

6.2.1 Element Fill-in-blank Problem

This problem intends for students to learn the Java grammar and basic programming skills by filling the blank elements in a given code. *An element* is the least unit of a code including a reserved word, an identifier, and a control symbol. *A reserved word* is a fixed sequence of characters that has been defined in the grammar to represent a specified function. *An identifier* is a sequence of characters defined in the code to represent a variable, a class, or a method. *A control symbol* intends other grammar elements such as " . " (dot), " : " (colon), " ; " (semicolon) , " (,) " (bracket), " {, } " (curly bracket).

The difficulty of the *element fill-in-blank problem* can be changed by controlling the ratio between the number of blank elements and non-blank ones in the problem. As more non-blanked elements exist between the blanked ones, the problem becomes easier. As more

blanked elements continue in the problem, it becomes harder. The two parameters, *BG (Blank Gap Number)* and *CB (Continuous Blank Number)* were introduced to control them. *BG* determines the number of non-blanked elements between the blanked ones, and *CB* does the number of continuous blanked elements that are appearing in a statement.

6.2.2 Core Element Fill-in-blank Problem

In the *element fill-in-blank problem*, the students can solve mechanically without understanding the code behaviors, if they are familiar with Java grammar. In programming educations, the students should study not only the grammar but also the codes that implement some algorithms or logics, such as standard input/outputs, data structure, fundamental algorithms, and graph algorithms. Thus, this problem selects the blank elements from the core statements that implement the algorithm/logic in the code, to enhance code reading studies of novice students. The students need to fill in the blanks by reading and understanding the code structure.

The difficulty of the *core element fill-in-blank problem* depends on the ratio of the selected core statements to all the statements in the code. This parameter, *SP (Selection Percentage)*, should be properly given by the teacher. To select core statements, the PDG is generated from the code.

The PDG represents the *data flow dependency* between the statements in the code, where a vertex corresponds to a statement and an edge exists if the two end statements depend on each other. The statement can be considered as the *core* if it depends on many other statements. Thus, the vertices with high degrees in the PDG are selected as the core ones. Actually, in the clique extraction step in the blank element selection algorithm, we limit the elements in the statements whose PDG degrees are larger than or equal to the threshold that is calculated from SP.

6.2.3 Value Trace Problem

This problem intends for students to enhance the code tracing ability. It asks them to fill the blanks suggesting the actual values of important variables in the code that implements a sorting algorithm or a graph theory algorithm. The students can answer them only by correctly tracing the true values of the variables from the initial to final ones at every step in the algorithm procedure.

The difficulty of the *value trace problem* can be changed by the ratio of the blank steps to all the steps. This parameter, *BS (Blank Step Ratio)*, should properly be set by the teacher.

6.3 Workbook Design for Fill-in-blank Problems

In this section, we present a workbook design for the three fill-in-blank problems in JPLAS.

6.3.1 Code Collections

To help teachers to use the fill-in-blank problems in JPLAS, Java source codes are collected from textbooks [47]-[49] and Web sites [13]-[20], [50]-[52] for Java programming. Then, by referring to the contents of the textbooks that have been used in the introductory Java

Table 6.1: Workbook code collection

category ID	code topic	# of Java codes
1	variable	5
2	operator	7
3	conditional statement	6
4	loop, break, continue	15
5	array	11
6	class: field, method, member	4
7	class: overload, constructor, this	4
8	class: library, string, class method	6
9	class: inheritance, superclass, override	6
10	interface	5
11	package, file	3
12	exception	8
13	data structure	2
14	sorting algorithms	4
15	graph algorithms	7

programming course, 16 categories have been selected to classify these codes as in Table 6.1. The first 12 categories (ID=1∼12) are related to Java grammar where each code usually consists of a single class. These codes can be used for element fill-in-blank problems. On the other hand, the remaining four categories (ID=13∼15) are for applications where each code usually consists of multiple classes and methods. Data structure, sorting algorithms, graph algorithms, and fundamental algorithms are selected here, since they are usually educated in the corresponding courses in universities. These codes can be used for both *element fill-in-blank problem* and *value trace problem*.

6.3.2 Programming Course Use

In general, the difficulty is increased in the order of the *element fill-in-blank problem*, the *core element fill-in-blank problem*, and the *value trace problem*. Besides, the difficulty of the two element fill-in-blank problems can be increased by taking the smaller BG and the larger CB (and SP), and the difficulty of the *value trace problem* can be increased by using the larger BS.

Unfortunately, every student in a Java programming course is not highly motivated in studying Java programming. Some students may easily lose motivations of studying it, if they feel difficulty in solving some element fill-in-blank problems in JPLAS. Therefore, they should start from easiest problems using the codes in $ID = 1$ with $BG = 3$ and $CB = 1$. Then, the assigned problems should be gradually more difficult by using the smaller BG and larger CB until $ID = 12$. After that, the core element fill-in-blank problem and the value trace problem can be assigned to students using codes in $ID = 13 \sim 16$.

Table 6.2: Workbook design of element fill-in-blank problems

category ID	code topic	# of problems	ave. # of lines (LOC)	ave. # of blanks (BG, CB)		
				(3, 1)	(1, 1)	(0, 3)
1	variable	5	9.6	8.6	9.4	13.2
2	operator	7	9.43	8.86	9.0	14.14
3	conditional statement	6	21.17	15.33	15.83	27.33
4	loop, break, continue	15	13.33	10.67	11.4	18.0
5	array	11	16.91	16.36	19.91	29.54
6	class: field, method, member	4	16.5	10.25	13.25	20.75
7	class: overload, constructor, this	4	21.0	14.75	19.5	26.25
8	class: library, string, class method	6	17.17	16.0	17.83	26.83
9	class: inheritance, superclass, override	6	20.5	13.67	15.5	23.33
10	interface	5	24.0	16.0	18.2	26.8
11	package, file	3	27.0	16.0	20.0	31.0
12	exception	8	21.0	16.75	17.5	25.0

6.4 Applications of Workbook

In this section, we discuss the generation of *element fill-in-blank* problems in the workbook and their application results by the students.

6.4.1 Generated Problems for Workbook

In this evaluation, we generated the *element fill-in-blank problems* from source codes by applying the extended *blank element selection algorithm* (ID: 1-12). Table 6.2 shows the code topic, the number of problems, the average number of statements (LOC) for one problem, and the average number of blanks with (3, 1), (1, 1), and (0, 3) for (BG, CB) in each category.

The number of selected blanks is gradually increase by the larger LOC until $ID = 12$. Our past results show that as the number of blanks increases, the problem becomes more difficult [11]. Thus, the teacher should carefully select assigned problems depending on performances of students in the course.

6.4.2 Trial Application Results to Novice Students

Then, we selected eight problems related to Java grammar in this workbook and asked four novice students from Indonesia to solve them. These students have studied Java programming for about 10 days, but sufficiently studied C programming before. Table 6.3 shows the category ID, the number of statements (LOC), the adopted values of (BG, CB) for the problem generation, the average number of selected blanks, and the average correct answer rate for each problem. Here, we note that the original source codes for these problems come from [47] and [48].

Table 6.3 indicates that the two problems Q2 and Q8 had lower correct answer rates than the others. **Problem Q2** and **Problem Q8** illustrate their problem codes respectively. As shown there, the problem code for Q2 includes the *object array* at lines 19-22, and the code for Q8 includes *double loops* at lines 3 and 5. It can be considered that they are difficult for the novice students. On the other hand, LOC and the values of (BG, CB) are not sensitive in

Table 6.3: Trial application results for four students

problem ID	category ID	LOC	BG	CB	ave. # of blanks	ave # of corrects	ave. correct rate (%)
Q1	6	32	1	1	22	20.75	94.31
Q2	9	28	3	1	21	18.75	89.29
Q3	4	26	1	1	17	16	94.12
Q4	3	19	0	3	24	23.75	98.96
Q5	5	18	1	1	19	18	94.74
Q6	7	18	1	1	23	22	95.65
Q7	3	12	0	3	20	19.5	97.5
Q8	4	11	0	3	18	16	88.89

solving performances of students, because these codes for Java grammar have many simple short statements. It is necessary to investigate their performance changes by them when students solve fill-in-blank problems using application codes such as sorting algorithms and graph algorithms, which will be in our future studies.

Listing 6.1: Problem Q2

```
1  class _1_ {
2    protected int num;
3    protected double gas;
4    public Car() {
5      _2_ = 0;
6      _3_ = 0.0;
7      System.out.println("generate a car");
8    }
9  }
10 _4_ RacingCar extends Car {
11   private int course;
12   public _5_ () {
13     _6_ = 0;
14     System.out.println("generate a racing car");
15   }
16 }
17 _7_ CodeQ2{
18   public _8_ void main( _9_ [ ] args) {
19     _10_ [ ] cars;
20     cars = _11_ Car[2];
21     _12_ [0] = _13_ Car();
22     _14_ [1] = _15_ RacingCar();
23     _16_ (int i=0; i<cars.length; i++){
24       Class clsName = _17_ [i] _18_ getClass();
25       _19_ .out. _20_ (class of (i+1) +
26         "th object is" + _21_ );
27     }
28   }
29 }
```

Listing 6.2: Problem Q8

```
1  public _1_ CodeQ8 {
2    public _2_ _3_ main( _4_ [ ] args) {
```

```
3       _5_ ( _6_ i = 0; _7_ _8_ 10; _9_ ++) {
4         _10_ .out.print _11_ i + ":");
5         _12_ (int j = 0; j _13_ _14_ ; _15_ ++) {
6           System.out. _16_ ("*");
7         }
8       _17_ .out. _18_ ("");
9       }
10    }
11  }
```

For reference, the problem code for Q4 is illustrated as follows. The average correct rate for Q4 is the highest among the eight problems. It has a simple structure of `if else`.

Listing 6.3: Problem Q4

```
1   _1_ java.io.*;
2   _2_ CodeQ4{
3     public _3_ _4_ main( _5_ [ ] args) _6_ IOException{
4       System. _7_ .println("Please enter an integer");
5       BufferedReader br =_8_ _9_ ( _10_ InputStreamReader( _11_ .in));
6       _12_ str = _13_ .readLine( _14_ ;
7       int res = Integer.parseInt( _15_ );
8       _16_ ( _17_ == 1){
9         System.out. _18_ ("the input is 1");
10      }
11      else _19_ ( _20_ == 2){
12          _21_ .out.println("the input is 2");
13      }
14      _22_ {
15        _23_ .out. _24_ ("Please enter 1 or 2");
16    }
17  }
18  }
```

6.5 Summary

In this chapter, we presented a workbook design of the three fill-in-blank problems in JPLAS by collecting a set of Java source codes from textbooks and Web sites. Then, eight problems in the workbook were assigned to novice students as the preliminary evaluations. The future studies include the generation of fill-in-blank problems for the remaining categories and the verification of the adequacy of this workbook in Java programming educations for novice students by assigning the problems in the workbook to students in Java programming courses.

Chapter 7

Informative Test Code Approach for Code Writing Problem

In this chapter, we present the *informative test code* approach of the *code writing problem* in JPLAS.

7.1 Introduction

In this chapter, the code writing problem is advanced by introducing the *informative test code* approach to help the students to solve harder problems that require multiple classes and methods. The *informative test code* describes the detailed specifications of the names, access modifiers, and data types of the classes, methods, and arguments. This problem asks a student to write a source code with the proper classes/methods that satisfies the specifications given by an informative code.

Generally, the test code can more clearly describe the specifications than a description using natural language. It is expected that the student obtains the information for the class/method names, the data types, and the argument settings by reading the test code, before writing the source code. Because the information in the test code comes from the model source code, the student is able to complete the same qualitative source code as the model code.

In this chapter, some related works of the test code is first reviewed. Then, *Eclipse Metrics Plugin* is introduced. And then, the *informative test code approach* for the code writing problem is presented. Next, three fundamental concepts in the *object-oriented programming (OOP)* are discussed for use of the informative test code approach. After that, the informative test code approach is evaluated. Finally, the conclusion is given for this chapter.

7.2 Related Works

In this section, we review some related works to the proposal in this chapter.

In [56], Yamamoto et al. presented an improved group discussion system for the active learning system (ALS) using mobile devices to increase the examination pass rate. In their previous study, it was found that the proposed ALS could not increase the examination pass rate of the students although the self-learning time was increased. The experimental evaluation of the improved group discussion system showed that it can increase the examination

pass rate. In future works, we will consider implementing the group discussion function with interfaces for mobile devices in JPLAS, so that students can continue studying Java programming with proper advises or hints from other students.

In [57], Xue et al. presented an integrity verification method for exception handling in service-oriented software. In this method, they construct state spaces associated with exception handling, convert the issue of integrity verification into a model of boundedness analysis based on CPN, and reduce the size of state spaces by extending Stubborn Set and Transition Dependency Graph. The experimental results confirmed that the method has good generalization abilities. In future studies, we will study the use of this method for learning exception handling in JPLAS.

In [58], Zhou et al. presented an Android application system using a tablet called *Isaly* to provide visual programming environments for educations. In this proposal, the concept of the state-transition diagram is used to make a program by a student. *Isaly* contains several features and user interfaces suitable for the use in a tablet.

In [59], Zhu et al. presented a system for mining API usage examples from the test code. They found that the test code can be a good source for API usage examples that programmers need to know, like our approach. The test code can provide the information on small units of a code like functions, classes, procedures, and interfaces. The information in the test code is helpful in developing and maintaining a source code, including the knowledge sharing and transfer among programmers. However, the repetitive *API* use in a test code makes it complicated for programmers to read it. To address this issue, they studied the *JUnit* test code and summarized a set of test code patterns. They employed a code pattern based heuristic slicing approach to separate test scenarios in code examples. Then, they cluster similar API usages to remove redundancy and provide recommendations for API usage examples for programmers. In future works, we will study the use of the *informative test code* for API usage.

In [60], Kolassa et al. presented a system based on *JUnit* to test the partial code in a template of a template-based code generator where it is generated by the template engine. It facilitates the partial testing of a code by supporting the code execution in a mocked environment. They adopted *TUnit*, an extension of *JUnit* based on the *MontiCore* language workbench [61][62][63], to support the unit test of an incomplete code in the mocked environment. By using *TUnit*, a code generator template can be tested with mocked contexts such as mocked variables, mocked templates, and mocked help functions that are the inputs to the template. This testing intends to answer the questions: *Is the set of the specified inputs accepted by the code generator template, e.g., the code can be generated?*, *Does the code generator template produce syntactically valid source code?*, and *Are the target language context conditions valid for the generated source code?*

On the other hand, in this paper, the *informative test code* approach is presented for the code writing problem in JPLAS, so that a student can learn how to write a complex source code in a harder assignment that requires multiple classes, by referring the information on the source code described in the test code, such as the names of the classes, the methods, the essential variables, the arguments, the returning data types of the methods, and the exception handling that are intended by the teacher. In JPLAS, we have implemented the interfaces only for a PC browser using a mouse and a keyboard.

7.3 Eclipse Metrics Plugin

In this section, we introduce *Eclipse Metrics Plugin* that is used to measure the code quality metrics.

7.3.1 Software Metrics

Software metrics are used for a variety of purposes including the evaluation of the software quality and the prediction of the development/maintenance cost. Software metrics can be measured from software products such as source codes and documents. Most of software metrics are defined on the conceptual modules of software systems, including files, classes, methods, functions, and data flows. This means that software metrics can be measured in any programming language.

At present, a variety of software metrics exist. They can be classified into *basic metrics*, *complexity metrics*, *CK metrics*, and *coupling metrics*. *CK metrics* indicate features of object-oriented software, and has been widely used [64][65].

Basic metrics include the following metrics:

- number of classes (*NOC*)
- number of methods (*NOM*)
- number of fields (*NOF*)
- number of overridden methods (*NORM*)
- number of parameters (*PAR*)
- number of static methods (*NSM*)
- number of static fields (*NSF*).

Complexity metrics include the following metrics:

- method lines of code (*MLOC*)
- specialization index (*SIX*),
- McCabe cyclomatic complexity (*VG*)
- nested block depth (*NBD*).

CK metrics include the following metrics:

- weighted methods per class (*WMC*)
- depth of inheritance tree (*DIT*),
- number of children (*NSC*)
- lack of cohesion in methods (*LCOM*).

Coupling metrics include the following metrics:

- afferent/efferent coupling (*CA/CE*).

7.3.2 Eclipse Metrics Plugin

Until now, a lot of software metric measurement tools have been developed. Among them, *Eclipse Metrics Plugin* by Frank Sauer is the commonly used open source software plugin for *Eclipse* IDE for the metrics calculation and the dependency analyzer. It can measure various metrics and display the results in the integrated view. Actually, 23 metrics can be measured by this tool, which can be used for the quality assurance testing, the software performance optimization, the software debugging, the process management of software developments such as time or methodology, and the cost/size estimations of a project [66].

7.3.3 Adopted Seven Metrics

In this thesis, we use this tool to measure the necessary metrics to evaluate the quality of source codes from the students that pass the test code on *JUnit*. The following seven metrics are actually adopted in this thesis:

1. Number of Classes (NOC)
   This metric represents the number of classes in the source code.
2. Number of Methods (NOM)
   This metric represents the total number of methods in all the classes.
3. Cyclomatic Complexity (VG)
   This metric represents the number of decisions caused by the conditional statements in the source code. The larger value for VG indicates that the source code is more complex and becomes harder to be modified.
4. Lack of Cohesion in Methods ($LCOM$)
   This metric represents how much the class lacks cohesion. A low value for $LCOM$ indicates that it is a cohesive class. On the other hand, the value close to 1 for $LCOM$ indicates the lack of cohesion and suggests that the class might better be split into several (sub)classes. $LCOM$ can be calculated as follows:

   1) Each pair of methods in the class are selected.
   2) If they access to the disjoint set of instance variables, P is increased by one. If they share at least one variable, Q is increased by one. It is noted that P and Q are initialized by 0.
   3) $LCOM$ is calculated by:

$$LCOM = \begin{cases} P - Q & (\text{if } P > Q) \\ 0 & (\text{otherwise}) \end{cases} \tag{7.1}$$

5 Nested Block Depth (NBD)
   This metric represents the maximum number of nests in the method. It indicates the depth of the nested blocks in the code.
6. Total Lines of Code (TLC)
   This metric represents the total number of lines in the source code, where the comment and empty lines are not included.

7. Method Lines of Code (*MLC*)
   This metric represents the total number of lines inside the methods in the source code, where the comment and empty lines are not included.

7.4 Informative Test Code Approach for Code Writing Problem

In this section, we present the *informative test code* approach for the code writing problem in JPLAS.

7.4.1 Concept of Informative Test Code

The *informative test code* is intended to help a student to complete the source code by offering the code design information to write the high quality code for a harder code writing problem. It gives the necessary information to implement the code, which includes the following items for the code:

- the class names and method names
- the access modifier, and data types for the important member variables
- the argument types
- the returning data types for the methods
- the exception handling

7.4.2 Problem Generation with Informative Test Code

Generally, in a code writing problem, the test code file, the input data file, and the expected output data file should be given to the students by a teacher, in addition to the problem statement in natural language. Then, a student is requested to write the source code that passes every test described in the test code on *JUnit*. The test code represents the detailed specifications of the source code.

The informative test code can be prepared after the qualitative *model source code* for the problem is prepared by the teacher. It is expected that the student completes the qualitative source code for the problem that has the similar structure with the model source code by referring this test code. The following steps describe the generation procedure of the code writing problem using the informative test code:

1. The teacher prepares the statement and the *input data file* for the new problem.
2. The teacher prepares the *model source code* that does not only satisfy every specification of the problem but has the high quality design.
3. The teacher prepares the *expected output data file* by running the model source code. This output file is used for comparison with the output data file of the student code to check the correctness.
4. The teacher generates the *informative test code* from the model source code such that any information in the model source code is tested including the exception handling.

7.4.3 Example Problem Generation for BFS Algorithm

In this subsection, we describe the details of steps 1, 2, and 3 using the *BFS algorithm* [67]. It starts at the *root* node (or arbitrary node of a graph), and explores the neighbor nodes first, before moving to the next level neighbors.

7.4.3.1 Input Data File

To represent a graph, the *input data file* should contain the *index* and the *label* for every vertex, and the *source vertex label* and the *destination vertex label* for every edge. The following example represents a graph with eight vertices and seven edges.

Listing 7.1: Input data file for *BFS*

```
1  node−number node−label
2   0  s
3   1  r
4   2  w
5   3  t
6   4  x
7   5  v
8   6  u
9   7  y
10  source−node target−node
11  s r
12  s w
13  r v
14  w t
15  w x
16  t u
17  x y
```

7.4.3.2 Model Source Code

The *model source code* for a problem should be prepared carefully by using the proper classes and methods, so that the measured metrics of the model source code exist in the desired ranges. For example, the model source code for the BFS algorithm can be implemented using *graph class* for handling the graph data, *BFS class* for applying the algorithm procedure, and *main class* for controlling the whole code. The teacher can obtain the model source code from textbooks or websites. By comparing the measured metrics of the source codes in them, the teacher can select the best source code for the model one.

7.4.3.3 Expected Output Data File

The expected *output data file* can be obtained by running the model source code with the input data file. It describes the expected results of the source code by a student. For the BFS algorithm, it includes the selected edges by the algorithm in the selected order that is described by a pair of two end node labels.

Listing 7.2: Output data file for *BFS*

```
1  selec−node pre−node
2   s  −
3   r  s
4   w  s
```

```
5 v r
6 t w
7 x w
8 u t
9 y x
```

7.4.3.4 Informative Test Code

The *informative test code* should be generated by referring the model source code such that any important method in the model code must be tested in this test code. It is possible to apply an automatic test code generation tool to help the test code generation [68]. Then, the test code is generated from the model source code by the following rules:

1. The class name is given by the *test class name + Test*.
2. The method name is given by the *test + test method name*.
3. The specific values are specified for the arguments in the test code by the teacher.

The test code can more clearly describe the specifications than a description using natural language. It is expected that the student obtains the information for the class/method names, the data types, and the argument settings by reading the test code, before writing the source code. Because the information in the test code comes from the model source code, the student is able to complete the same qualitative source code as the model code.

7.4.3.5 Informative Test Code Example

The following test code is generated from the source code for the BFS algorithm. It contains the necessary information to implement a source code for the BFS algorithm, including the classes, the methods, the important variables and their data type, the exception handling, and returning values of method.

Listing 7.3: Informative test code for *BFS*

```
1  import static org.junit.Assert.*;
2  import java.io.BufferedReader;
3  import java.io.File;
4  import java.io.FileReader;
5  import java.io.IOException;
6  import java.util.Arrays;
7  import org.junit.Test;
8  public class BFSTest {
9     @Test
10    public void testSimpleGraph() {
11          SimpleGraph G = new SimpleGraph (5);
12          boolean a=G.labels instanceof String [ ];
13          boolean b=G.edges instanceof boolean [ ][ ];
14          assertEquals(true, a);
15          assertEquals(true, b);
16          assertEquals(5,G.labels.length);
17          assertEquals(5,G.edges.length);
18          assertEquals(5,G.edges[0].length);
19    }
20    @Test
21    public void testSetLabel(){
```

```
22          SimpleGraph G= new SimpleGraph(2);
23          G.setLabel(1, "a");
24          assertEquals("a",G.labels[1]);
25      }
26      @Test
27      public void testGetLabel(){
28          SimpleGraph G = new SimpleGraph (2);
29            G.setLabel(1, "b");
30            String label=(String)G.getLabel(1);
31            assertEquals("b",label);
32      }
33      @Test
34      public void testAddEdge(){
35          SimpleGraph G = new SimpleGraph (3);
36            G.addEdge(1, 2);
37            assertEquals(true,G.edges[1][2]);
38      }
39      @Test
40      public void testNeighbours(){
41            SimpleGraph  G = new SimpleGraph(3);
42            int [ ] expectedNode = {1,2};
43            G.addEdge(0,1);
44            G.addEdge(0,2);
45            assertTrue(Arrays.equals(expectedNode, G.neighbors(0)));
46      }
47      @Test
48      public void testFindBFS1(){
49            SimpleGraph G = new SimpleGraph(4);
50            BFS bfs = new BFS();
51            G.setLabel(0, "a");
52            G.setLabel(1, "b");
53            G.setLabel(2, "c");
54            G.setLabel(3, "e");
55            G.addEdge(0,1);
56            G.addEdge(0,2);
57            G.addEdge(1,3);
58            String Path[ ]=bfs.findBFS(G, 0);
59            String[ ] expectedPath = {"a a", "b a", "c a", "e b"};
60             assertTrue(Arrays.equals (expectedPath,Path));
61      }
62      @Test
63      public void testFindBFS2() throws IOException {
64            BFS bfs= new BFS();
65            File testFileName=new File ("./Graph/graphBFS.txt");
66            File OutFileName=new File ("D:/Graph/bfsout.txt");
67            String graph=bfs.readFile(testFileName);
68            String [ ] path=bfs.findBFS(graph);
69            bfs.writeFile(OutFileName, path);
70      }
71      @Test
72      public void assertReaders() throws IOException {
73          BufferedReader expected= new BufferedReader (new FileReader("./Graph/
          expectedbfsout.txt"));
74            BufferedReader actual = new BufferedReader (new FileReader("D:/Graph/
          bfsout.txt"));
75          String line;
76          while ((line = expected.readLine()) != null) {
77              assertEquals(line, actual.readLine());
78          }
```

```
79            assertNull("Actual had more lines than the expected.", actual.readLine());
80            assertNull ("Expected had more lines than the actual.", expected.readLine
      ());
81    }
82  }
```

- Lines from 10 to 19 describe the test method for two important variables, *labels* and *edges*, in *SimpleGraph* class. *labels* has the *String* data type and one dimensional array. *edges* has the *Boolean* data type and two dimensional array.
- Lines from 21 to 25 describe the test method for *setLabel* method in *SimpleGraph*, which accepts two arguments with integer and string data types, namely *index* and *label*, and inserts the information to *labels*.
- Lines from 27 to 32 describe the test method for *getLabel* method, which accepts one argument with integer data type and returns the corresponding label from *labels*.
- Lines from 34 to 38 describe the test method for *addEdge* method, which accepts two arguments with integer data types, namely *source* and *target*, and inserts the information to *edges*.
- Lines from 40 to 46 describe the test method for *neighbours* method, which accepts one argument with integer data type, namely *index*, and returns the integer array which includes the indexes are neighboring to the input *index*.
- Lines from 48 to 61 describe the first test method for *findBFS* method in *BFS* class, which accepts two arguments with the Graph object and the integer data type and returns a string array which includes the labels from *labels* for the selected indexes and the previous index from them by *BFS*. Here, *setLabel* and *addEdge* methods in *SimpleGraph* class are also described here.
- Lines from 63 to 70 describe the second test method for *findBFS* method, which accepts one argument of the string data type and returns the string array which includes the labels from *labels* for the selected indexes and the previous index from them by *BFS*. Here, *readFile* and *writeFile* methods in *BFS* class are also described. *readFile* method accepts one argument of *File* object and returns the string that includes the *index* and *labels* for the graph to be applied to *findBFS* method. *writeFile* method accepts two arguments of the *File* object and the string data type array, and writes the input string array, which includes the labels from *labels* for the selected indexes and the previous index from them by *BFS*, to the output file and generate it. This test method throws *IOException* whenever an input or output operation is failed or interrupted when the program is executed.
- Lines from 72 to 81 describe the test method that is used to compare the expected output data file with the output data file from the source code of the student.

7.4.3.6 Simple Test Code Example

Then, to evaluate the solving performances of students using the informative test codes, we also prepare the *simple test codes* for them. The following *simple test code* for *BFS* contains

only the test methods for *readFile* method, *writeFile* method, and *findBFS* method in *BFS* class. This simple test code only tests the input data file reading and output data file writing functions in the source code, where it does not test the internal functions of the code.

Listing 7.4: Simple test code for *BFS*

```
1  import static org.junit.Assert.*;
2  import java.io.BufferedReader;
3  import java.io.File;
4  import java.io.FileReader;
5  import java.io.IOException;
6  import java.util.Arrays;
7  import org.junit.Test;
8  public class BFSTest {
9      @Test
10     public void testFindBFS() throws IOException {
11         BFS bfs= new BFS();
12           File testFileName=new File ("./Graph/graphBFS.txt");
13           File OutFileName=new File  ("D:/Graph/bfsout.txt");
14           String graph=bfs.readFile(testFileName);
15           String [ ] path= bfs.findBFS(graph);
16           bfs.writeFile(OutFileName, path);
17     }
18     @Test
19     public void assertReaders() throws IOException {
20         BufferedReader expected= new BufferedReader (new FileReader("./Graph/
       expectedbfsout.txt"));
21         BufferedReader actual = new BufferedReader (new FileReader("D:/Graph/
       bfsout.txt"));
22        String line;
23        while ((line = expected.readLine()) != null) {
24           assertEquals(line, actual.readLine());
25        }
26           assertNull("Actual had more lines than the expected.", actual.readLine());
27           assertNull ("Expected had more lines than the actual.", expected.readLine
       ());
28     }
29 }
```

7.5 Informative Test Code for Three Fundamental Concepts

In this section, we study the informative test code for three fundamental concepts.

7.5.1 Overview of Three Fundamental Concepts

OOP is a methodology or paradigm to design a program using classes and objects, and simplifies the software development and maintenance by providing some concepts such as *encapsulation*, *inheritance* and *polymorphism*. They are hard concepts for novice students to understand how to use them.

7.5.1.1 Encapsulation

Encapsulation is the mechanism of wrapping data (variables) and the code parts acting on the data (methods) together as a single unit [69]. By *encapsulation*, the variables of a class are hidden from the other classes, and can be accessed only through the methods implemented in the class. It is also known as the *data hiding*. *Encapsulation* can be realized as follows in Java:

1) to declare the variables in the class as *private*, and
2) to provide the *public setter and getter methods* to modify and view the values of them.

The following code shows the example of the encapsulation, where the variable *name* in class *Student* is encapsulated and can be accessed using *getName* and *setName* methods:

Listing 7.5: Example source code for *Encapsulation*

```
1 public class Student {
2   private String name;
3   public String getName() {
4     return name;
5   }
6   public void setName(String name) {
7     this.name = name;
8   }
9 }
```

7.5.1.2 Inheritance

Inheritance is the mechanism where the object for the child class or *subclass* acquires all the properties and behaviors of the object for its parent class or *superclass*. It represents the *IS-A* relationship, also known as the *parent-child relationship*. By adopting *inheritance*, the code can be made in the hierarchical order [70].

The following code shows the example of the inheritance, where class *B* inherits class *A* that defines the variable *salary*:

Listing 7.6: Example source code for *Inheritance*

```
1  class A {
2    float salary=40000;
3  }
4   class B extends A {
5     int bonous=100000;
6     public static void main (String args[ ]){
7       B b=new B();
8       System.out.println(b.salary);
9       System.out.println(b.bonous);
10    }
11  }
```

7.5.1.3 Polymorphism

Polymophism is the ability of an object to take on many forms. The most common use of *polymophism* occurs when the parent class reference is used to refer to the child class [71].

Two types, *method overloading* and *method overwriting*, exist for *polymophism*. In *method overloading*, a class has multiple methods have same name but different in parameters. In *method overwriting*, the subclass has the same method as declared in the parent class and it is used for run time.

The following code shows the example of *polymorphism*, where *makeNoise* method is first defined in class *Animal*, and is redefined in class *Dog* in the two ways depending on the argument:

Listing 7.7: Example source code for *Polymorphism*

```
1  public class Animal {
2    public void makeNoise(){
3        System.out.println("Some sound");
4    }
5  }
6  class Dog extends Animal{
7    public void makeNoise(){
8        System.out.println("Bark");
9    }
10   public void makeNoise(int x){
11       for (int i=0; i<x; i++)
12        System.out.println("Bark");
13   }
14 }
```

7.5.2 Example Informative Test Code Generation for Three Concepts

For the generation of the informative test code for the three concepts, the source codes for *Queue* and *Stack* are adopted.

7.5.2.1 Source Code for Queue

Queue is an abstract data structure following *First-In-First-Out*. *Queue* is open at both its ends, where one end is always used to insert a new data (*enqueue*) and the other is used to remove an existing data (*dequeue*) [72].

The following source code implements *Queue* data structure using *encapsulation*. The three important variables, *content*, *tail*, and *head*, are declared as *private* and are hidden from the other class. *content* stores the string and integer values. *tail* and *head* store the first and last index number of the stored values in *content*. They can be accessed only through three methods in the class, *push*, *pop*, and *empty*, declared as *public*. *empty* method checks the index number of *content*. *push* inserts the integer and string values into the bottom of *content*, *pop* retrieves the integer and string values from the bottom of *content*. *empty* returns *true* if *content* is empty.

Listing 7.8: Source code for *Queue*

```
1  class Que{
2    private Object content[ ]=new Object[1000];
3    private int tail=0;
4    private int head=0;
5    public boolean empty() {
6      return (tail==head)?true:false;
```

```
7     }
8     public void push(Object num) {
9       content [tail++]=num;
10    }
11    public Object pop() {
12      return content [head++];
13    }
14  }
```

7.5.2.2 Informative Test Code for Queue

Then, the following *informative test code* is generated from the source code for *Queue*. *test1* method first tests the names, the access modifiers, the data types of the three important variables, *content*, *tail*, and *head*. The access modifiers of *content*, *tail* and *head* must be *private*. The data types of *content*, *tail*, and *head* must be *Object*, *int*, and *int* respectively.

Then, it tests the names, the access modifiers, the returning data types of three important methods, *push*, *pop*, and *empty*. The access modifiers of them must be *public*. The returning data types of *push*, *pop*, and *empty* must be *void*, *Object*, and *Boolean* respectively.

test2 method tests the behaviors of *push*, *empty*, and *pop*. *push* stores the integer or string argument in *content* as the setter method where no value is returned. *empty* and *pop* do not accept the argument as the getter method. *empty* returns a boolean value, and *pop* returns an integer or string value in *content*.

Listing 7.9: Informative test code for *Queue*

```
1  import static org.junit.Assert.*;
2  import java.lang.reflect.Field;
3  import java.lang.reflect.Method;
4  import java.lang.reflect.Modifier;
5  import org.junit.Test;
6  public class QueTest {
7   @Test
8   public void test1() throws NoSuchFieldException, SecurityException,
       NoSuchMethodException{
9     //Field
10    Field f1=Que.class.getDeclaredField("content");
11    Field f2=Que.class.getDeclaredField("tail");
12    Field f3=Que.class.getDeclaredField("head");
13    //check Modifier
14    assertEquals(f1.getModifiers(), Modifier.PRIVATE);
15    assertEquals(f2.getModifiers(), Modifier.PRIVATE);
16    assertEquals(f3.getModifiers(), Modifier.PRIVATE);
17     //data type for each variable
18    assertEquals(f1.getType(),Object[].class);
19    assertEquals(f2.getType(),int.class);
20    assertEquals(f3.getType(),int.class);
21     //Method
22    Method m1=Que.class.getDeclaredMethod("empty", null);
23    Method m2=Que.class.getDeclaredMethod("push", Object.class);
24    Method m3=Que.class.getMethod("pop", null);
25     //check Modifier
26    assertEquals(m1.getModifiers(), Modifier.PUBLIC);
27    assertEquals(m2.getModifiers(), Modifier.PUBLIC);
28    assertEquals(m3.getModifiers(), Modifier.PUBLIC);
29    //data type for each Method
30    assertEquals(m1.getReturnType(),boolean.class);
```

```
31      assertEquals(m2.getReturnType(),void.class);
32      assertEquals(m3.getReturnType(),Object.class);
33  }
34   @Test
35   public void test2() {
36      Que q = new Que();
37      q.push(1);
38      q.push("a");
39      if (!q.empty()) {
40         assertEquals(1, q.pop());
41         assertEquals("a", q.pop());
42      }
43    }
44  }
```

7.5.2.3 Source Code for Stack

Stack is a basic data structure following *Last-In-First-Out*. *Stack* allows the operations at one end only, where the insertion and deletion of data take places at one end called the top of the stack [73].

The following source code implements *Stack* data structure using *inheritance* and *polymorphism*. *Stack* class inherits the three important variables, *content*, *tail*, and *head*, and two methods, *empty* and *push*, from *Que* class. To inherit those variables from *Que* where they are declared as *private*, the access modifier must be changed to *protected*. Then, *pop* method is overwritten in *Stack* from that in *Que*, to retrieve the data at the top of the *content*.

Listing 7.10: Source code for *Stack*

```
1  class Stack extends Que{
2     public Object pop() {
3         return content [−−tail];
4     }
5  }
```

7.5.2.4 Informative Test Code for Stack

Then, the following *informative test code* is generated from the source code for *Stack*. *test1* method tests the name, the access modifier, the returning data type of *pop* method. The access modifier must be *public* and the returning data type must be *Object*.

test2 method tests the behaviors of *empty*, *push*, and *pop*. *empty* and *push* are tested by calling them from *Que*, and *pop* is tested by calling it from *Stack*. *push* stores the integer or string argument in *content* and returns no value. *empty* and *pop* do not accept the argument. *empty* returns a boolean value and *push* returns an integer or string value in *content*.

Listing 7.11: Informative test code for *Stack*

```
1  import static org.junit.Assert.∗;
2  import java.lang.reflect.Field;
3  import java.lang.reflect.Method;
4  import java.lang.reflect.Modifier;
5  import org.junit.Test;
6  public class StackTest {
7  @Test
```

```
8     public void test1() throws NoSuchFieldException, SecurityException,
        NoSuchMethodException {
9       //Method
10      Method m1=Stack.class.getMethod("pop", null);
11      //check Modifier
12      assertEquals(m1.getModifiers(), Modifier.PUBLIC);
13      //data type for each Method
14      assertEquals(m1.getReturnType(), Object.class);
15    }
16  @Test
17   public void test2() {
18      Stack q = new Stack();
19      //inherit from Queue
20      q.push(1);
21      q.push("a");
22      if (!q.empty()) {
23        assertEquals("a", q.pop());
24        assertEquals(1, q.pop());
25      }
26    }
27  }
```

7.6 Evaluations

In this section, we evaluate the *informative test code approach* for the code writing problem in JPLAS.

7.6.1 Evaluation for Five Graph Algorithms

First, we evaluate the *informative test code approach* for five well-known five graph algorithms, *BFS*, *DFS*, *Prim*, *Dijkstra*, and *Kruskal*.

7.6.1.1 Code Completion Results

In this evaluation, first, we asked the seven students to write the source code for *BFS* using the *simple test code*, where it was found that only one student could complete it within one week. After that, we gave them the *informative test code* to do the same thing. Then, all of them could complete it. The students tested source codes by using the given test code on *JUnit* from *Eclipse*.

After every student completed the source code for *BFS* using the informative test code, we selected three students who solved it in the shortest time. Then, we asked them to write the source codes for *DFS*, *Prim*, *Dijkstra*, and *Kruskal* algorithms using the *simple test codes*, where all of them could complete them. This time, they did not need *informative test codes*, because they have known how to design and implement the codes for the similar graph algorithms from their experiences in *BFS*.

7.6.1.2 Metric Results for BFS

The seven software metrics in Section 7.3.3 were measured for these completed source codes of the students using *Eclipse Metrics Plugin*. Table 7.1 shows the measured metric results of

Table 7.1: Comparison of metric values for BFS algorithm using proposal

	Metrics	$S1$ (simple)	$S1$	$S2$	$S3$	$S4$	$S5$	$S6$	$S7$
1	NOC	1	2	2	3	2	5	2	2
2	NOM	7	11	10	11	11	19	9	9
3	VG	18	4	5	4	5	2	7	6
4	NBD	3	4	4	4	3	2	4	4
5	$LCOM$	0.9	0.37	0.5	0.5	0.375	0.7	0.5	0.5
6	TLC	142	120	143	137	144	157	114	121
7	MLC	102	87	104	93	102	88	81	88

the eight source codes for *BFS* by them. In this table, the student $S1$ completed the source codes both with the simple and informative test codes.

Actually, the student $S1$ has studied the Java programing only for three months in our group, where the other students have studied it for at least one year. $S1$ has never made similar graph theory programs that require multiple classes/methods. In this experiment, $S1$ spent one week to complete this programming task. The skill of $S1$ is supposed to be lower than the others. Thus, the source code by $S1$ using the simple test code uses only one class where the procedures of the graph data handling and the search algorithm are implemented together.

When the metric values are compared between the two source codes of $S1$, VG, $LCOM$, TLC, and MLC are much worse for the simple test code than those for the informative one, as shown in Table 7.1. Particularly, the metric value for VG becomes very large. It means that this source code is very complex and becomes hard to be modified or extended.

Besides, the metric value for $LCOM$ was close to 1 in Table 7.1, because the member variables (public attributes) and methods in the class were used without being shared with other classes. This class should be split into two or more classes. On the other hand, the seven source codes using the informative test codes have good metrics where VG is 2-7, and $LCOM$ is 0.3-0.7. It has been known that the desired VG should be less than 20, and $LCOM$ should not be close to 1 [66]. Thus, these source codes can be recognized as highly qualitative codes.

7.6.1.3 Metric Results for Four Graph Algorithms

Table 7.2 shows the measured metric results of the 12 source codes for the remaining four algorithms by three students. In *DFS*, every code has good metrics where VG is 2-4 and $LCOM$ is 0-0.7 respectively. It is noted that *DFS* is the most similar to *BFS* among them.

However, in the remaining algorithms, VG for $S3$ is always larger than that for other students, and $LCOM$ by $S3$ is always zero. The reason is that $S3$ implements the source codes using only one class, which results in no cohesion between classes and becomes complex and hard to be modified. In this case, it is necessary to redesign the code with multiple classes by using the design-aware test code.

In each algorithm, VG for $S5$ is always smaller than that for $S3$ and $S4$, whereas $LCOM$ for $S5$ is larger than that for $S3$ and $S4$. From NOC and NOM, $S5$ uses more classes such

Table 7.2: Metric values for four algorithms without using proposal

Metrics	DFS			Prim			Dijkstra			Kruskal		
	$S3$	$S4$	$S5$	$S3$	$S4$	$S5$	$S3$	$S4$	$S5$	$S3$	$S4$	$S5$
NOC	2	2	6	1	4	8	1	4	6	1	3	7
NOM	5	9	23	3	15	35	3	14	23	3	8	29
VG	3	4	2	15	6	2	11	10	3	20	9	2
NBD	3	3	2	5	4	2	4	5	3	6	4	2
$LCOM$	0	0.5	0.7	0	0.5	0.7	0	0.33	0.7	0	0.5	0.7
TLC	74	124	189	108	205	299	107	195	203	123	141	250
MLC	49	58	109	93	114	190	91	109	121	105	64	154

as the node class, the edge class, their subclasses, and the encapsulated class for *Encapsulation*, and more methods than other students. As a result, the cohesion between classes are necessary.

Encapsulation is a technique to protect the important attributes from any unauthorized access. These attributes can be hidden from the other classes, and can be accessed through the public methods defined in the class containing them. In *Encapsulation*, the important attribute or data member to be protected is defined as *private* so that it can only be accessed within the same class. No outside class can access to this private data member. Then, the public getter and setter methods are defined in the class so that it can be read or updated from the outside class.

7.6.2 Evaluation for Three OOP Concepts

Then, we evaluate the effectiveness of the *informative test code* approach for the code writing problem to study the three fundamental concepts of the object-oriented programming.

In this evaluation, first, we asked students to write source codes using *encapsulation*, *inheritance*, and *polymorphism* for *Queue* and *Stack* with the informative test codes. It was found that all of them could complete them. Then, the seven metrics in Section 7.3.3 were measured for these complete codes using *Eclipse Metrics Plugin*. The metrics are compared between the source codes by $S1$, $S2$, and the textbook. Table 7.3 shows the metric results of the three source codes for each assignment.

In *Queue*, any source code is implemented using one class and has good metrics. VG is 1-3, NBD is 1-2, and $LCOM$ is 0.5 except $S2$. It is noted that desired VG is less than 20, NBD is less than or equal to 5, and $LCOM$ should not be close to 1 [66]. However, $LCOM$ by $S2$ is 1 because the important variables for *Queue* are declared by using different names from the ones specified in the test code. To pass the test code, the specified variables are declared only and not used in the code. Our current test code cannot detect it because of the *private* access modifier. Thus, we need to improve the *informative test code* to avoid it, which will be in future studies. TLC and MLC by $S1$ are the larger than the others, because $S1$ implements the source code using more methods than the expected in the test code.

In *Stack*, any source code is also implemented using one class, and has good metrics where VG is 1-2, NBD is 1-2, and $LCOM$ is 0. Unfortunately, although the code by $S2$ does not inherit the variables and methods from *Que*, it can pass the test code assuming the

Table 7.3: Metric values of source codes

Metrics	Queue			Stack		
	textbook	$S1$	$S2$	textbook	$S1$	$S2$
NOC	1	1	1	1	1	1
NOM	3	9	3	1	1	3
VG	2	3	1	1	2	1
NBD	1	2	1	1	2	1
$LCOM$	0.5	0.5	1	0	0	0
TLC	14	53	18	5	15	14
MLC	3	25	3	1	10	3

inheritance and polymorphism. Thus, we need to improve the test code to avoid it, which will also be in future studies.

7.7 Summary

In this chapter, we presented the *informative test code* approach for the code writing problem in JPLAS. The effectiveness of this approach was evaluated by generating informative test codes for graph algorithms and three concepts and applying them to students in our group. The future studies include the improvement of the informative test code to avoid the drawbacks found in the evaluations, the generations of informative test codes for other programming assignments, and their applications to students in Java programming courses.

Chapter 8

Conclusions

In this thesis, we presented the five advancements of the exercise problems for *Java programming learning assistant system (JPLAS)*.

Firstly, we presented the three extensions of the *blank element selection algorithm* for the *element fill-in-blank problem in* JPLAS. We evaluated the effectiveness of these extensions through applications of the generated problems by this extended algorithm with various Java codes to students.

Secondly, we presented the *core element fill-in-blank problem* in JPLAS for enhancing code reading studies by novice students. We evaluated the effectiveness through applications to students of the problems using four Java codes for the graph theory or fundamental algorithms.

Thirdly, we presented the *value trace problem* as the new type of the *fill-in-blank problem* in JPLAS for studying algorithm Java code reading by students. We evaluated the effectiveness through applications to students of the problems using Java source codes for sorting and graph theory algorithms.

Fourthly, we presented the *workbook design* for the three *fill-in-blank problems* in JPLAS by collecting various Java source codes from Java programming textbooks and Web sites. We evaluated the effectiveness of this workbook design through applications of some problems in the generated workbook to novice students.

Finally, we presented the *informative test code* approach for the *code writing problem* in JPLAS for studying code writing in harder assignments. We evaluated the effectiveness through applications to students of the problems for the graph algorithms and the three fundamental concepts for the object-oriented programming.

In future studies, we will further improve the *blank element selection algorithm*, improve the *program dependence graph (PDG)* generation method, generate the different element fill-in-blank problems using these algorithms for a workbook, improve the user interface for displaying the problem code, improve the informative test code to avoid the drawbacks, prepare test codes for other problems, and assign the generated problems to students in Java programming courses.

Bibliography

[1] N. Funabiki, Tana, K. K. Zaw, N. Ishihara, and W.-C. Kao, "A graph-based blank element selection algorithm for fill-in-blank problems in Java programming learning assistant system," IAENG Int. J. Computer Science, vol. 44, no. 2, pp. 247-260, May 2017.

[2] Tana, N. Funabiki, and N. Ishihara, "A proposal of graph-based blank element selection algorithm for Java programming learning with fill-in-blank problem, Proc. IMECS2015," pp. 448-453, March 2015.

[3] Tana, N. Funabiki, and N. Ishihara, "Practices of fill-in-blank problems in Java programming course," Proc. ICCE-TW 2015, pp. 120-121, June 2015.

[4] JFlex, `http://jflex.de/`.

[5] jay, `http://www.cs.rit.edu/~ats/projects/lp/doc/jay/package-summary`.

[6] 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, February 2013.

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

[8] JUnit, `http://www.junit.org/`.

[9] N. Funabiki, S. Sasaki, Tana, and W.-C. Kao, "An operator fill-in-blank problem for algorithm understanding in Java programming learning assistant system," Proc. GCCE 2015, pp. 346-347, October 2015.

[10] T. Ogawa, N. Funabiki, T. Nakanishi, N. Ishihara, Tana, and N. Amano, "A difficulty estimation method of fill-in-blank problems for Java programming learning assistant system," IEICE Tech. Report, ET2013-99, pp. 41-46, March 2013.

[11] N. Funabiki, Tana, N. Ishihara, and W.-C. Kao, "Analysis of fill-in-blank problem solution results in Java programming course," Proc. GCCE 2016, pp. 479-481, October 2016.

[12] Java program samples, `http://www7a.biglobe.ne.jp/~java-master/samples/`.

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

[14] Sorting, http://faculty.simpson.edu/lydia.sinapova/www/cmsc250/ LN250 Weiss/L12-ShellSort.htm.

[15] S. K. Chang, "Data Structures and Algorithms," World Scientific Pub., USA, October 2003.

[16] Dijkstra Algorithm, http://www.ifp.illinois.edu/~angelia/ge330fall09_dijkstra_l18.pdf.

[17] Prim Java, http://cs.fit.edu/~ryan/java/programs/graph/Prim-java.html.

[18] Graph Java, http://www.sanfoundry.com/java-program.

[19] Breadth First Search, https://en.wikipedia.org/wiki/Breadth-first_search.

[20] Depth First Search, https://en.wikipedia.org/wiki/Depth-first_search.

[21] Security, http://www.morikita.co.jp/books/book/2214.

[22] K. K. Zaw, N. Funabiki, and M. Kuribayashi, "Extensions of blank element selection algorithm for Java programming learning assistant system," IEICE Tech. Report, ET2016-15, pp. 41-46, June 2016.

[23] K. K. Zaw, N. Funabiki, and M. Kuribayashi, "A proposal of three extensions in blank element selection algorithm for Java programming learning assistant system," Proc. GCCE 2016, pp. 3-6, October 2016.

[24] N. Ishihara, N. Funabiki, and W.-C. Kao, "A proposal of statement fill-in-blank problem using program dependence graph in Java programming learning assistant system," Inform. Eng. Express, vol. 1, no. 3, pp. 19-28, September 2015.

[25] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," ACM Trans. Program. Lang. Syst., vol. 9, no. 3, July 1987.

[26] Graph Java, http://www.sanfoundry.com/java-program.

[27] Fundamental Java, http://www.sbcr.jp/books.

[28] K. K. Zaw and N. Funabiki, "A core blank element selection algorithm for code reading studies by fill-in-blank problems in Java programming learning assistant system", Proc. 7th Int. Conf. Science and Eng., pp. 204-208, December 2016.

[29] 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, August 2013.

[30] 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, December 2013.

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

[32] Data Structures Tutorials, http://cs-fundamentals.com/data-structures/data-structures-tutorials.php.

[33] B. Smulders, "Annotate Code, introducing a system for code-stepping based visualization," Master Thesis, Leiden Univ., August 2014.

[34] M. Quinson and G. Oster, "A teaching system to learn programming: the programmer's learning machine," Proc. ITiCSE '15, July 2015.

[35] E. R. Sykes and F. Franek, "An intelligent tutoring system prototype for learning to program Java," Proc. Int. Conf. Adv. Learn. Tech., 2003.

[36] W. I. Osman and M. M. Elmusharaf, "Effectiveness of combining algorithm and program animation: a case study with data structures courses," Issue. Inform. Sci. Inform. Tech., vol. 11, 2014, pp. 155-168.

[37] Insertion, http://interactivepython.org/courselib/static/pythonds/SortSearch/TheInsertionSort.html.

[38] InsertionSort, http://mycodinglab.com/insertion-sort-algorithm/.

[39] Java code, http://www.journaldev.com/585/insertion-sort-in-java-algorithm-and-code-with-example.

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

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

[42] Dijkstra Java, http://cs.fit.edu/~ryan/java/programs/graph/Dijkstra-java.html.

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

[44] K. K. Zaw and N. Funabiki, "A concept of value trace problem for Java code reading education," Proc. Int. Cong. Adv. Appl. Inform., pp. 253-258, July 2015.

[45] K. K. Zaw, N. Funabiki, and W.-C. Kao, "A proposal of value trace problem for algorithm code reading in Java programming learning assistant system," Inf. Eng. Express, pp. 9-18 , September 2015.

[46] K. K. Zaw and N. Funabiki, "A blank line selection algorithm for value trace problem in Java programming learning assistant system," IEICE Society Conf., BS-6-2, pp. S19-S20, September 2015.

[47] H. Yuki, Java programming lesson, Softbank Creative, 2012, http://www.hyuki.com/jb/#download.

[48] M. Takahashi, Easy Java, Softbank Creative, 2013, http://homepage3.nifty.com/~mana/yasaj.html.

[49] Y. Kondo, Algorithm and data structure for Java programmers, Softbank Creative, 2011.

[50] ITSenka, http://torialspoint.com/java/index.htm.

[51] tutorialspoint, http://www.tutorialpoint.com/java/index.htm

[52] L. Sinapova, Lecture Notes, http://faculty.simpson.edu/lydia.sinapova/www/cmsc250/LN250Weiss/Contents.htm..

[53] K. K. Zaw, N. Funabiki and M. Kuribayash, "Element fill-in-blank problems in Java programming learning assistant system," IEICE General Conf., BS-1-21, pp. S40-S41, March 2017.

[54] N. Funabiki, M. Dake, K. K. Zaw, W.-C. Kao, "A workbook design for fill-in-blank problems in Java programming learning assistant system," Proc. Int. Conf. Broad. Wireless Comput., Commun. Appl. (BWCCA2016), pp. 331-342, November 2016.

[55] N. Ishihara, N. Funabiki, M. Kuribayashi, and W.-C. Kao, "A proposal of software architecture for Java programming learning assistant system," Proc. AINA-2017, pp. 64-70, March 2017.

[56] N. Yamamoto, "An improved group discussion system for active learning using smartphone and its experimental evaluation," Int. J. Space-Base. Situated Comput., vol. 6, no. 4, pp. 221-227, 2016.

[57] T. Xue, S. Ying, Q. Wu, X. Jia, X. Hu, X. Zhai, and T. Zhang, "Verifying integrity of exception handling in service-oriented software," Int. J. Grid. Utility Comput., vol. 8, pp. 17-21, 2017.

[58] E. Zhou, Z. Niibori, S. Okamoto, M. Kamada, and T. Yonekura, "IslayTouch: an educational visual programming environment for tablet devices", Int. J. Space-Based and Situated Computing, vol. 6, no. 3, pp. 183-197, 2016.

[59] Z. Zhu, Y. Zou, B. Xie, Y. Jin, Z. Lin, and L. Zhang, "Mining API usage examples from test code," Proc. IEEE Int. Conf. Soft. Mainte. Evo., pp. 301-310, 2014.

[60] C. Kolassa, M. Look, K. Müller, A. Roth, D. Rei, and B. Rumpe, "TUnit unit testing for template-based code generators," Proc. Modellierung Conf., pp. 221-236, 2016.

[61] H. Grönniger, H. Krahn, B. Rumpe, M. Schindler, S. Völkel, "MontiCore: A framework for the development of textual domain specific languages," Proc. Int. Conf. Soft. Eng. (ICSE), 2008.

[62] H. Krahn, B. Rumpe, S. Völkel, "MontiCore: modular development of textual domain specific languages," Proc. Int. Conf. Model. Tech. Tool. Comp. Perform. Evaluation, pp. 297-315, 2008.

[63] H. Krahn, B. Rumpe, S. Völkel, "MontiCore: a framework for compositional development of domain specific languages," Int. J. Software Tool. Tech. Transfer, vol. 12, no. 5, pp. 353-372, September 2010.

[64] Y. Higo, A. Saitoh, G. Yamada, T. Miyake, S. Kusumoto, and K. Inoue, "A pluggable tool for measuring software metrics from source code," Proc. IWSM-MENSURA, pp. 2-12, 2011.

[65] T. G. S. Fil and M. A. S. Bigonha, "A catalogue of thresholds for object-oriented software metrics," Proc. SOFTENG, pp. 48-55, 2015.

[66] Metric Plugin, http://metrics.sourceforge.net.

[67] BFS, http://www.geeksforgeeks.org/breadth-first-traversal-for-a-graph.

[68] JUnit-Tools, http://junit-tools.org/index.php/getting-started.

[69] Encapsulation, https://www.tutorialspoint.com//java_encapsulation.htm

[70] Inheritance, https://www.javatpoint.com/inheritance-in-java

[71] Polymorphismm, https://www.javatpoint.com/runtime-polymorphism-in-java

[72] Queue, https://www.tutorialspoint.com/data_structures_algorithms/dsa_queue.htm

[73] Stack, https://en.wikibooks.org/wiki/Data_Structures/Stacks_and_Queues

[74] KMP and KnapSack Algorithms, http://www.sbcr.jp/books/

[75] K. K. Zaw and N. Funabiki, "A desing-aware test code approach for code writing problem in Java programming learning assistant system", Int. J. Spaced-Based and Situated Computing, vol. 7, no. 3 pp. 145-154, September 2017.

[76] K. K. Zaw, and N. Funabiki, "An informative test code approach for code writing problem in Java programming learning assistant system," IEICE Tech. Report, SS-2017-10, pp. 31-36, October 2017.