A Design-aware Test Code Approach for Code Writing Problem in Java Programming Learning Assistant System

Khin Khin Zaw

Graduate School of Natural Science and Technology,
Okayama University,
Okayama, Japan
E-mail: p8lj1oji@s.okayama-u.ac.jp

Nobuo Funabiki

Graduate School of Natural Science and Technology,
Okayama University,
Okayama, Japan
E-mail: funabiki@okayama-u.ac.jp

Abstract: To advance Java programming educations, we have developed the Web-based *Java Programming Learning Assistant System (JPLAS)* that provides the *code writing problem* among the four type problems with different levels. This problem asks a student to write a Java source code for a given assignment, where the correctness is verified by running the *test code* on *JUnit*. Unfortunately, it is found that even after solving many simple problems, a lot of students cannot solve harder problems that require multiple classes/methods, where the proper code design is necessary. In this paper, we propose a *design-aware test code* approach for the code writing problem in JPLAS. The design-aware test code is generated to test any important method in the *model code* that has the advisable design for the problem. By writing a code that can pass this test code, a student is expected to implement the code with the proper classes/methods in the model code. For evaluations, we asked seven students to write the source code for the *breadth-first-search (BFS)* algorithm of a graph without/with using the design-aware test code. Then, only one student could complete it without it, whereas all of them could do so with it. Besides, the code quality metrics measured by *Metrics plugin* for *Eclipse* showed that the design-aware test code is very helpful for students to write highly qualitative codes. These results were also confirmed in other graph algorithms.

Keywords: JPLAS; Design-aware test code; JUnit; Metric Plugin; BFS.

Reference to this paper should be made as follows: K .K. Zaw, N. Funabiki, 'A Design-aware Test Code Approach for Code Writing Problem in Java Programming Learning Assistant System', *Int. J. Space-Based and Situated Computing*, Vol. x, No. y, pp.aa–bb.

Biographical notes: Khin Khin Zaw received the B.E. degree in information technology from Technological University (HmawBi), Myanmar, in 2006, and the M.E. degree in information technology from Mandalay Technological University, Myanmar, in 2011, respectively. She has been a lecturer in Yangon Technological University, Myanmar, since 2015. She is currently a Ph.D. candidate in Graduate School of Natural Science and Technology at Okayama University, Japan. Her research interests include educational technology and Web application systems. She is a student member of IEICE.

Nobuo Funabiki received the B.S. and Ph.D. degrees in mathematical engineering and information physics from the University of Tokyo, Japan, in 1984 and 1993, respectively. He received the M.S. degree in electrical engineering from Case Western Reserve University, USA, in 1991. From 1984 to 1994, he was with Sumitomo Metal Industries, Ltd., Japan. In 1994, he joined the Department of Information and Computer Sciences at Osaka University, Japan, as an assistant professor, and became an associate professor in 1995. He stayed at University of Illinois, Urbana-Champaign, in 1998, and at University of California, Santa Barbara, in 2000-2001, as a visiting researcher. In 2001, he moved to the Department of Communication Network Engineering (currently, Department of Electrical and Communication Engineering) at Okayama University as a professor. His research interests include computer networks, optimization algorithms, educational technology, and Web technology. He is a member of IEEE, IEICE, and IPSJ.

1 Introduction

Java has been extensively used in industries as a reliable and portable object-oriented programming language, which involves mission critical system for large enterprises and small-sized embedded system. Thus, the cultivation of Java programming engineers has been in high demands amongst industries. A great number of universities and professional schools are offering Java programming courses to meet these needs.

To assist Java programming educations, we have developed the Web-based *Java Programming Learning Assistant System (JPLAS)* N. Ishihara, N. Funabiki, M. Kuribayashi, W.-C. Kao (2010). JPLAS provides the *element fill-in-blank problem* N. Funabiki, Tana, K. K. Zaw (2017), *value trace problem* K. K. Zaw, N. Funabiki, and W.-C. Kao (2015), *statement fill-in-blank problem* N. Ishihara, N. Funabiki, and W.-C. Kao (2015), and *code writing problem* N. Funabiki, Y. Matsushima, T. Nakanishi, N. Amano (2013) that have different difficulties to cover a variety of students at different learning levels. The first two problems are designed for novice students to study Java grammar and code reading by filling in the correct words to the blanks in a given code. The correctness is marked through comparisons with the correct answers. The third problem requests students to fill in the blanked statements in a given code. The correctness is verified by running the *test code* on *JUnit*, JUnit (http://www.junit.org/.) as the *test-driven development (TDD) method* K. Beck (2002). It is designed to fill the gap between the former problems and the last one. The last problem asks students to write a source code for a given specification described in natural language. The correctness is also verified by the test code.

Unfortunately, in the code writing problem, it is found that even after solving many simple problems, most students who are studying Java programming cannot solve harder problems that require longer codes composed of multiple classes/methods. 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. The detailed code design is necessary to help students to find the proper classes and methods in the code.

In this paper, we propose a *design-aware test code* approach for the code writing problem in JPLAS. The design-aware test code tests any important method in each class in the *model source code* for the problem, which describes the detailed code design. For example, for a graph theory algorithm, this test code tests the methods in the class for handling the graph data for a given graph and those in the class for finding the answer. By writing the source code that passes this test code, the student is expected to design and implement the source code using the proper classes/methods.

For evaluations, we first prepared the design-aware test code for the *breadth-first-search (BFS)* algorithm of a simple graph, and asked seven students in our group to write the corresponding code without and with using this test code. Then, only one student could complete it without using the design-aware test code, whereas all of them completed it with using it. After that, we asked three of them to write the codes for *depth-first-search (DFS)*, *Prim*, *Dijkstra*, and *Kruskal* algorithms, and found that all of them could complete the codes even without using design-aware test codes, because they have already become familiar to code implementations of these similar graph algorithms through the BFS algorithm. Furthermore, code quality metrics of these codes were measured by *Metrics plugin* for *Eclipse*, where the results were acceptable. Thus, the design-aware test code is very helpful for students to complete highly qualitative codes.

The rest of this paper is organized as follows: Sections 2 shows related works. Sections 3 and 4 review JPLAS and the TDD method respectively. Section 5 introduces *Metrics plugin for Eclipse.* Section 6 presents the *design-aware test code* approach for the code writing problem. Section 7 evaluates our proposal. Finally, Section 8 concludes this paper with some future works.

2 Related Works

In this section, we discuss some related works.

In Yamamoto (2016.), 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 T. Xue (2017), 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 E. Zhou (2016), 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

Copyright © 201X Inderscience Enterprises Ltd.

Copyright © 201X Inderscience Enterprises Ltd.

several features and user interfaces suitable for the use in a tablet.

In Z. Zhu (2014), 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 design-aware test code for API usage.

In C. Kolassa (2016), 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 H. Grönniger (2008), H. Krahn (2008), H. Krahn (2010), 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 *design-aware 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.

3 JPLAS

In this section, we review *Java Programming Learning Assistant System (JPLAS)* N. Ishihara, N. Funabiki, M. Kuribayashi, W.-C. Kao (2010).

3.1 Software Platform

In JPLAS, *Ubuntu* is adopted for the OS of JPLAS running on *VMware*. *Tomcat* is used as the Web server for *JSP*. *JSP* is a script with embedded Java code within the HTML code, where *Tomcat* can return a dynamically generated Web page to the client. *MySQL* is adopted as the database for managing the data.

3.2 Four Problems in JPLAS

In JPLAS, the four types of problems are provided. For each problem, JPLAS offers service functions for a teacher to generate and register new problems and for a student to answer the problems.

3.2.1 Element Fill-in-blank Problem

This problem requires a student to fill in the blank elements in a given Java code. The correctness of the answer is marked by comparing them with their original elements in the code. Thus, the original elements must be the unique correct answers for the blanks. To help a teacher to generate an element fill-in-blank problem, we have proposed a *blank element selection algorithm.*

3.2.2 Value Trace Problem

This problem is another type of the element fill-in-blank problem that keeps the nature of filling in blanks and marking answers by string matching, but requires much deeper code reading. It questions a student about actual values of important variables in the code that implements a fundamental data structure or algorithm. To generate a value trace problem for a given code, the *blank line selection algorithm* is presented, which blanks the whole data in such a line of the output data from the code execution that at least one data is changed from the previous line.

3.2.3 Statement Fill-in-blank Problem

This problem asks a student to fill in the blank statements in a given Java code. The correctness of the answer is marked by using the test code on *JUnit* as the code writing problem, where the combined code with the blank one and the answer is tested. To help a teacher prepare a statement fill-in-blank problem, we have proposed a *blank statement selection algorithm using the program dependency graph (PDG).*

3.2.4 Code Writing Problem

This problem asks a student to write a whole source code from scratch that satisfies the specifications given by a *test code.* The correctness of the code by a student is marked by using this test code on *JUnit.* A teacher needs to prepare the specification and the test code to register a new assignment in JPLAS.

4 TDD Method

In this section, we review the *test-driven development (TDD) method*.

4.1 JUint

JPLAS adopts *JUnit* as an open-source Java framework to support the TDD method. *JUnit* can assist the automatic unit test of a source code or a class. Since *JUnit* has been designed with the Java-user friendly style, including 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 ”. This paper adopts the “assertEquals” method to compare the execution result of the source code with its expected value.

4.2 Test Code

A test code should be written by using libraries in *JUnit*. Here, 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.

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

Then, the following test code tests the *plus* method in the *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 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 the *plus* method. This test is performed as follows:

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

4.3 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.

5 Metrics Plugin for Eclipse

In this section, we introduce *Metrics plugin for Eclipse* that is used to measure the code quality metrics in this paper.

5.1 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 Y. Higo, A. Saitoh, G. Yamada, T. Miyake, S. Kusumoto, and K. Inoue (2011), T. G. S. Filó and M. A. S. Bigonha (2015).

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*).

5.2 Metrics Plugin for Eclipse

Until now, a lot of software metric measuring tools have been developed. Among them, *Metrics plugin* for *Eclipse* by Frank Sauer is the commonly used open source software plugin for the *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 Metric Plugin (http://metrics.sourceforge.net).

5.3 Adopted Seven Metrics

In this paper, 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 paper:

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:

$$LCOM = \begin{cases} P - Q & (\text{if } P > Q) \\ 0 & (\text{otherwise}) \end{cases} \qquad (1)$$

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.

6 Design-aware Test Code Approach for Code Writing Problem

In this section, we propose the design-aware test code approach for the code writing problem in JPLAS.

6.1 Concept of Design-aware Test Code

The *design-aware test code* helps a student to complete the source code with the high quality for a harder code writing problem by giving the necessary information to implement the code. This information can include the following items for the code:

- important classes and methods
- global variables and their data types used in each class
- arguments in each method
- returning value in each method
- exception handling

6.2 Problem Generation with Design-aware 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 design-aware 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 design-aware 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 *design-aware test code* from the model source code such that any important method in the model source code is tested including the exception handling.

6.3 Example Problem Generation for BFS Algorithm

In this subsection, we describe the details of Steps 1, 2, and 3 using the BFS algorithm BFS (http://www.geeksforgeeks.org/breadth-first-traversal-for-a-graph). 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.

6.3.1 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.

```
node-number node-label
 0  s
 1  r
 2  w
 3  t
 4  x
 5  v
 6  u
 7  y
source-node target-node
 s  r
 s  w
 r  v
 w  t
 w  x
 t  u
 x  y
```

6.3.2 Model Source Code

The model source code 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 BFS can be implemented using the *graph class* for handling the graph data, the *BFS class* for applying the algorithm procedure, and the *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 source codes in them, the teacher can select the best source code for the model one.

6.3.3 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 BFS, it includes the selected edges by the algorithm in the selected order that are described by a pair of two end node labels.

```
selec-node pre-node
 s  -
 r  s
 w  s
 v  r
 t  w
 x  w
 u  t
 y  x
```

6.3.4 Design-aware Test Code

The design-aware 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 JUnit-Tools (http://junit-tools.org/index.php/getting-started). 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.

6.3.5 Design-aware Test Code Example

The following test code 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. By reading the test code carefully and understanding the details, the student can design and implement the source code that contains the same classes/methods as the model source code.

```
import static org.junit.Assert.*;
import java.io.BufferedReader;
import java.io.File;
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 the *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 Evaluations

In this section, we evaluate the design-aware test code approach for the code writing problem through applications to seven students in our group. For evaluations, we prepare the *design-aware test codes* for five well-known graph algorithms, *BFS*, *DFS*, *Prim*, *Dijkstra*, and *Kruskal*.

7.1 Simple Test Code

To compare the solving performance with the design-aware test code, we also prepare the *simple test codes* for them. The following *simple test code* for *BFS* contains only the test methods for the *readFile* method, the *writeFile* method, and the *findBFS* method in the *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.

```
import static org.junit.Assert.*;
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.util.Arrays;
import org.junit.Test;

public class BFSTest {

 @Test
 public void testFindBFS() throws IOException {
     BFS bfs= new BFS();
     File testFileName=new File
                      ("./Graph/graphBFS.txt");
     File OutFileName=new File
                      ("D:/Graph/bfsout.txt");
     String graph=bfs.readFile(testFileName);
     String [] path= bfs.findBFS(graph);
     bfs.writeFile(OutFileName, path);
 }

 @Test
 public void assertReaders() throws IOException {
   BufferedReader expected= new BufferedReader
     (new FileReader("./Graph/expectedbfsout.txt"));
   BufferedReader actual = new BufferedReader
     (new FileReader("D:/Graph/bfsout.txt"));
   String line;
   while ((line = expected.readLine()) != null) {
        assertEquals(line, actual.readLine());
   }
        assertNull("Actual had more lines than
             the expected.", actual.readLine());
        assertNull ("Expected had more lines than
             the actual.", expected.readLine());
 }
}
```

7.2 Code Completion Results

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 *design-aware 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 design-aware 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 *design-aware test codes*, because they have known how to design and implement the codes for the similar graph algorithms from their experiences in *BFS*.

7.3 Metric Results for BFS

The seven software metrics in Section 5.3 were measured for these completed codes of the students using *Metrics plugin* for *Eclipse*. Table 1 shows the measured metric results of the eight source codes for *BFS* by them. In this table, the student $S1$ completed the source codes both with the simple and design-aware 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 design-aware one, as shown in Table 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.

Table 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

Besides, the metric value for LCOM was close to 1 in Table 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 design-aware 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 Metric Plugin (http://metrics.sourceforge.net). Thus, these source codes can be recognized as highly qualitative codes.

7.4 Metric Results for Four Graph Algorithms

Table 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 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.

8 Conclusion

In this paper, we proposed the *design-aware test code* approach for the code writing problem in JPLAS. The design-aware test code tests any important method in the *model source code* that has the advisable design for the problem. By writing a code that can pass this test code, a student is expected to design and implement the code using the proper classes/methods in the model code. For evaluations, seven students were encouraged to solve the code writing problem for the *breadth-first-search (BFS)* algorithm of a graph without/with the design-aware test code. Then, the use of the design-aware test code drastically increased the number of completing students. Besides, the code quality metrics measured by *Metrics plugin* for *Eclipse* showed that the design-aware test code was very helpful to write highly qualitative codes. These results were also confirmed in other graph algorithms. In future works, we will prepare design-aware test codes for other problems and assign them to students in Java programming courses.

References

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.

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.

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, vol. 1, no. 3, pp. 9-18, Sep. 2015.

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,” Inf. Eng. Express, vol. 1, no. 3, pp. 19-28, Sep. 2015.

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

JUnit, http://www.junit.org/.

K. Beck, Test-driven development: by example, Addison-Wesley, 2002.

N. Yamamoto, “An improved group discussion system for active learning using smart-phone and its

Table 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

experimental evaluation," Int. J. Space-Base. Situated Comput., vol. 6, no. 4, pp. 221-227, 2016.

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.

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.

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.

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.

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.

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.

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, Sep. 2010.

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.

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

Metric Plugin, http://metrics.sourceforge.net.

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

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

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

Fundamental Java, http://www.sbcr.jp/books.

E. Zhou, Z.Niibori, S.Okamoto, M. Kamada, 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.

N.Yamamoto, "An improved group discussion system for active learning using smartphone and its experimental evaluation " , Int. J. Space-Based and Situated Computing, Vol. 6, No. 4, pp.221-227, 2016.

Tong Xue, Shi Ying, Qing Wu, Xiangyang Jia, Xiaohui Hu, Xiaoying Zhai, Tao Zhang , " Verifying integrity of exception handling in service-oriented software " , International Journal of Grid and Utility Computing , Vol. 8, No. 1, pp.7-21, 2017.