An Informative Test Code Approach in Code Writing Problem for Three Object-Oriented Programming Concepts in Java Programming Learning Assistant System

Khin Khin Zaw, Win Zaw, Nobuo Funabiki, and Wen-Chung Kao

***Abstract*—To enhance Java programming educations, we have developed a *Java Programming Learning Assistant System (JPLAS)* that offers various types of exercise problems to cover studies at different levels. Among them, the *code writing problem* asks a student to write a source code that passes the given *test code* in the assignment. In Java programming, *encapsulation*, *inheritance*, *polymorphism* are the fundamental *object-oriented programming (OOP)* concepts that every student should master and freely use, which is very hard for novice students. In this paper, we propose the *informative test code approach* in the code writing problem for studying the three OOP concepts. This test code describes the necessary information to implement the source code using the concepts, such as the names, access modifiers, data types of the member variables and methods. Then, a student is expected to learn how to use them by writing a source code to pass the test code. To evaluate the effectiveness of the proposal, we generated informative test codes for 10 assignments using three concepts, and asked eight students who are currently studying Java programming in Myanmar and Japan to solve them. Then, all of them could complete source codes that pass the test codes, where the *quality metrics* measured by *Metrics plugin for Eclispe* were generally acceptable. Unfortunately, due to the insufficiency of test codes, the *coverage metric* by *code coverage tool for Eclipse* was not 100% at some source codes. The informative test code generation by a teacher should be assisted to avoid this problem.**

***Index Terms*—JPLAS, Java programming education, code writing, informative test code, encapsulation, inheritance, polymorphism, metric**

I. Introduction

NOWADAYS, *Java* has been extensively used in practical systems in industries as a reliable, scalable, and portable object-oriented programming language. *Java* involves a lot of mission critical systems for large enterprises and small-sized embedded systems. Then, the cultivation of Java programming engineers has been in high demands amongst industries. As a result, a great number of universities and professional schools are offering Java programming courses to meet these needs.

To enhance Java programming educations, we have developed a *Java Programming Learning Assistant System (JPLAS)* [1]. JPLAS performs excellently not only in reducing the load of a teacher by the function of automatically marking answers from students, but also in advancing the motivation of a student by the immediate response to each answer. It is expected that JPLAS improves Java programming educations in all kinds of institutes around the world. JPLAS has been implemented as a Web application using *JSP/Java* [2]. For the server platform, it adopts the operating system *Linux*, the Web server *Apache*, the application server *Tomcat*, and the database system *MySQL*, as shown in Figure 1. For the browser, it assumes the use of *Firefox* with *HTML*, *CSS*, and *JavaScript*.

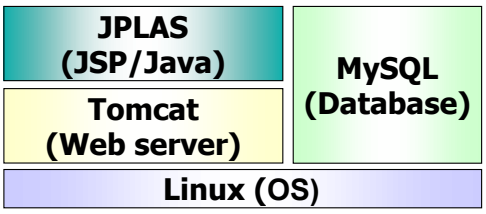

Fig. 1: JPLAS Server Platform.

Currently, JPLAS provides the four types of exercise problems, namely, *element fill-in-blank problem* [3], *value trace problem* [4], *statement fill-in-blank problem* [5], and *code writing problem* [6], to support the long-term self-study of a student at various learning levels. Among them, the *code writing problem* asks a student to write a Java source code that passes the *test code* given in the assignment. The test code will examine the correctness of the specifications and behaviors of the source code through running on *JUnit*, called the *test-driven development (TDD) method* [7]. The test code describes the necessary information to implement the source code, such as names of classes/methods/variables, arguments of methods, and data types of variables. As well, they can help a student completing a complex code that requires multiple classes and methods [8]. It is expected that by writing a code that passes the test code, a student will be able to implement the source code using proper classes/methods.

In Java programming, *encapsulation*, *inheritance*, *polymorphism* are the three fundamental *object-oriented programming (OOP)* concepts that every student should master and freely use. By implementing a source code using them, the advantage of Java programming in reliability and scalability can be realized. However, it is very hard for a novice

Manuscript received Jan 26, 2019; revised March 24, 2019.

K. K. Zaw and W. Zaw are with the Department of Computer Engineering and Information Technology, Yangon Technological University, Yangon, Myanmar, e-mail: thihakhinkhin85@gmail.com.

N. Funabiki is with the Department of Electrical and Communication Engineering, Okayama University, Okayama, Japan, e-mail: funabiki@okayama-u.ac.jp.

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

student to study them. Therefore, JPLAS should adopt the function of supporting study of the three OOP concepts.

In this paper, we propose the *informative test code approach* in the code writing problem in JPLAS for studying the three OOP concepts. The informative test code describes the necessary information to implement the source code using them, such as the names, the access modifiers, and the data types of the member variables and methods. Then, a student is expected to learn how to write a source code by using the three OOP concepts.

To evaluate the effectiveness of our proposal, we generated informative test codes for 10 programming assignments using the three OOP concepts, and asked eight students who are currently studying Java programming in Myanmar and Japan to solve them. Then, all of the students completed the source codes that pass the test codes. Their *quality metrics* were measured by *Metrics plugin for Eclipse* [9], which were generally acceptable. Unfortunately, due to insufficiency of the test codes, the *coverage metric* measured by *code coverage tool for Eclipse* was not 100% for some codes whose quality metrics were different from others. Thus, the generation of complete informative test codes should be supported to avoid this problem.

The rest of this paper is organized as follows: Section II reviews the TDD method, quality metrics, coverage metrics, and the three OOP concepts as preliminary of this paper. Section III presents the informative test code approach to the code writing problem. Section IV shows the evaluation of the proposal. Section V concludes this paper with future studies.

II. Preliminary

In this section, we briefly review the TDD method, quality metrics by *metrics plugin for Eclipse*, coverage metrics by *code coverage tool for Eclipse*, and the three OOP concepts as preliminary for the study in this paper.

A. TDD Method

In this subsection, we review the TDD method [7].

JPLAS adopts *JUnit* as the open-source Java framework to support the TDD method. *JUnit* assists the automatic unit test of a Java source code or a class by running a *test code*. Each test can be performed by using a method in the library whose name starts with "assert". It compares the execution result of the source code with its expected one.

In **source code 1** for *MyMath* class, *plus* method returns the summation of two integer arguments. Then, in **test code 1** for *MyMath* class, *testPlus* method tests *plus* method by comparing the result for 1 and 4 with its expected result 5. 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 *plus* method.

Listing 1: **source code 1**

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

Listing 2: **test code 1**

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

B. Quality Metrics

In this subsection, we introduce *metrics plugin for Eclipse* and adopted seven quality metrics in this paper.

A substantial amount 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 *Eclipse IDE* [9]. That is to say, 23 metrics can be measured by this tool, which can be used for quality assurance testing, software performance optimization, software debugging, process management of software developments such as time or methodology, and estimating the cost or size of a project.

This tool is adopted to measure the following seven metrics to evaluate the quality of source codes from students:

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 conditional statements in the source code. The larger value for VG indicates that the source code is more complex and becomes harder when 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, a value close to 1 indicates the lack of cohesion and suggests that the class might better be split into several classes. *LCOM* can be calculated as follows:
   1) Each pair of two methods in the class are selected.
   2) If they access to a disjoint set of instance variables, P is increased by one. If they share at least one variable, Q is increased by one.
   3) *LCOM* is calculated by:

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

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.

C. Coverage Metrics

In this subsection, we introduce *code coverage tool for Eclipse* and the code coverage metrics.

The *code coverage* or *test coverage* is one of the most important aspects in the unit test to ensure the test quality with respect to functional points. The code coverage measures the completeness of the test suites that verify the correctness of the source code. It shows which lines in the code were or were not executed by the test suites, and provides the percentage of the executed or covered lines by the test suites.

A variety of code coverage tools have been developed for different programming languages to measure the code coverage in the test. In this paper, *EclEmma Java code coverage plug-in* is used as an open source tool to test the code coverage of a Java source code. This tool measures the following four coverage metrics. It counts the number of items that have been executed by the test suites, and reports the percentage of the covered items. It also identifies the items that have not been tested.

1. Function coverage:
   Each function in the code has been called?
2. Statement coverage:
   Each statement in the code has been executed?
3. Branches coverage:
   Each branch of a control structure in the code, such as *if* and *case* statements, has been executed?
4. Condition coverage:
   Each boolean sub-expression has been tested at both of *true* and *false*?

D. Three Fundamental Concepts of Object Oriented Programming

In this subsection, we introduce the three important concepts for the *object-oriented programming (OOP)* in this paper. *OOP* is a methodology or paradigm to design a program using classes and objects, and simplifies the software development and maintenance by providing specific concepts.

1) Encapsulation: The *encapsulation* is the mechanism of wrapping data (variables) and the code parts acting on the data (methods) together as a single unit [10]. By the encapsulation, the variables of a class are hidden from the other classes, and can be accessed merely through the methods implemented in the class. It is also known as the *data hiding*. The 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 variable *name* in class *Student* is encapsulated and can be accessed using *getName* and *setName* methods:

Listing 3: **source code 2**

```
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  }
```

2) Inheritance: The *inheritance* is the mechanism such that 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 the inheritance, the code can be made in the hierarchical order [11]. The following code demonstrates the example of the inheritance, where class *B* inherits class *A* that defines variable *salary*:

Listing 4: **source code 3**

```
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 }
```

3) Polymorphism: The *polymophism* is the ability of an object to take on a plenty of forms. The most common use of *polymophism* occurs when the parent class reference is used to refer to the child class [12]. Two types, *method overloading* and *method overwriting*, exist for the *polymophism*. In the *method overloading*, a class has multiple methods that have the same name but different in parameters. In the *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 the *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 5: **source code 4**

```
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 }
```

III. Informative Test Code Approach for Three OOP Concepts

In this section, we present the *informative test code approach* in the code writing problem for studying the three OOP concepts.

A. Overview of Informative Test Code

The *informative test code* is designed to help a student to study the three OOP concepts of the *encapsulation*, the *inheritance*, and the *polymosphism* by giving the necessary information to implement the source code using them. They include the class names, the method names, the arguments, the member variable names, the access modifiers, and the data types.

B. Source Code for Queue

Queue is an abstract data structure following *First-In-First-Out (FIFO)*. *Queue* is open at both its ends, where one end is used to insert a new data and the other is used to remove an existing data [13].

source code 5 implements the *Queue* data structure using *encapsulation*. It represents the *circular queue* where the last position is connected back to the first position to make a circle. A new element can be inserted until the queue becomes full when the next element even cannot be inserted.

In **source code 5**, five important *member variables*, *MAX_QSIZE*, *content*, *head*, *tail*, and *queSize*, are declared as *private*, so that they are hidden from other classes. *MAX_QSIZE* stores the size of *content* array. *content* stores string or integer values. *head* and *tail* store the index of first and last stored value in *content* respectively. *queSize* stores the number of currently stored values in *content*. It is increased by 1 when a new value is inserted into *content*, and is decreased when a value is removed from it. When *queSize* is 0, the queue is empty, and when it is equal to *MAX_QSIZE*, the queue is full.

Four *methods*, *full*, *empty*, *push*, and *pop*, are declared as *public*, so that they can be accessed from other classes. *full* method returns *true* if *queSize* is equal to *MAX_QSIZE*. *empty* method returns *true* if *queSize* is equal to 0. *push* method inserts an integer or string value at *tail* of *content* after increased by 1 as the *setter method*. *pop* method returns the value at *head* of *content* as the *getter method*.

Listing 6: **source code 5**

```
1  public class QueExample{
2    private final int MAX_QSIZE = 5;
3    private Object content[] = new Object[MAX_QSIZE];
4      private int head = 0;
5      private int tail = −1;
6      private int queSize = 0;
7      public boolean full() {
8        return (queSize==MAX_QSIZE);
9      }
10     public boolean empty() {
11       return (queSize==0);
12     }
13     public Object push(Object data) {
14       if (!full()) {
15         queSize ++;
16         tail = (tail+1) % MAX_QSIZE;
17         content[tail] = data;
18         return data+" is inserted";
19       } else
20         return "Que is full and overflow
           occurs";
21     }
22     public Object pop() {
23       if (!empty()) {
24         queSize −−;
25         Object result = content[head];
26         head = (head+1) % MAX_QSIZE;
27         System.out.println(head);
28         return result+ " is deleted";
29       } else
30         return "Que is empty and underflow
           occurs";
31     }
32   }
33 }
```

C. Informative Test Code for Queue

Then, **test code 2** is generated as the *informative test code* for *Queue*.

In **test code 2**, *variableTest* method tests the names, access modifiers, data types of the five important member variables, and its number. All the access modifiers must be *private*. The data type must be *Object* for *content* and *int* for the others. The number of the member variables must be five, which is tested to avoid defining unnecessary variables.

methodTest method tests the names, access modifiers, returning data types of the four methods. The access modifiers must be *public*. The returning data type of *full* and *empty* must be *Boolean*, and that of *push* and *pop* must be *Object*. The number of the methods must be four.

behaviorTest method tests the behaviors of the four methods. *full*, *empty*, *push*, and *pop*. Initially, five integers, “10”, “20”, “30”, “40”, and “50”, are inserted to the queue. Then, the values of the five variables, *MAX_QSIZE*, *content*, *tail*, *head*, and *queSize* are evaluated.

Listing 7: **test code 2**

```
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 variableTest() throws Exception {
9      QueExample q = new QueExample();
10   //Field
11     Field f1 = q.getClass().getDeclaredField("MAX_QSIZE
         ");
12     Field f2 = q.getClass().getDeclaredField("content");
13     Field f3 = q.getClass().getDeclaredField("tail");
14     Field f4 = q.getClass().getDeclaredField("head");
15     Field f5 = q.getClass().getDeclaredField("queSize");
16   //test the access modifier
17     assertEquals(f1.getModifiers(), Modifier.PRIVATE,
         Modifier.FINAL);
18     assertEquals(f2.getModifiers(), Modifier.PRIVATE);
19     assertEquals(f3.getModifiers(), Modifier.PRIVATE);
20     assertEquals(f4.getModifiers(), Modifier.PRIVATE);
21     assertEquals(f5.getModifiers(), Modifier.PRIVATE);
22   //test the datatype of variables
23     assertEquals(f1.getType(), int.class);
24     assertEquals(f2.getType(), Object[].class);
25     assertEquals(f3.getType(), int.class);
26     assertEquals(f4.getType(), int.class);
27     assertEquals(f5.getType(), int.class);
28   //test the number of variables
29     Field[] f=q.getClass().getDeclaredFields();
30     assertEquals(5, f.length);
31 @Test
32   public void methodTest() throws Exception {
33     QueExample q = new QueExample();
34     Method m1 = q.getClass().getDeclaredMethod("full"
         , null);
35     Method m2 = q.getClass().getDeclaredMethod("empty
         ", null);
36     Method m3 = q.getClass().getDeclaredMethod("push"
         , Object.class);
37     Method m4 = q.getClass().getDeclaredMethod("pop",
         null);
38     assertEquals(m1.getModifiers(), Modifier.PUBLIC);
39     assertEquals(m2.getModifiers(), Modifier.PUBLIC);
40     assertEquals(m3.getModifiers(), Modifier.PUBLIC);
41     assertEquals(m4.getModifiers(), Modifier.PUBLIC);
42     assertEquals(m1.getReturnType(), boolean.class);
```

```
43      assertEquals(m2.getReturnType(), boolean.class);
44      assertEquals(m3.getReturnType(), Object.class);
45      assertEquals(m4.getReturnType(), Object.class);
46      Method[] m = q.getClass().getDeclaredMethods();
47      assertEquals(4, m.length);
48    }
49  @Test
50    public void behaviorTest() throws throws Exception {
51      QueExample q = new QueExample();
52      Field f1 = q.getClass().getDeclaredField("MAX_QSIZE
          ");
53      Field f2 = q.getClass().getDeclaredField("content");
54      Field f3 = q.getClass().getDeclaredField("tail");
55      Field f4 = q.getClass().getDeclaredField("head");
56      Field f5 = q.getClass().getDeclaredField("queSize");
57    //make private variables accessible for tests
58      f1.setAccessible(true);
59      f2.setAccessible(true);
60      f3.setAccessible(true);
61      f4.setAccessible(true);
62      f5.setAccessible(true);
63      int MAX_QSIZE = (int)f1.get(q);
64      Object[] content = (Object[])f2.get(q);
65      int tail = (int)f3.get(q);
66      int head = (int)f4.get(q);
67      int queSize = (int)f5.get(q);
68    //test the initialize value of each variable
69      assertEquals(5, MAX_QSIZE);
70      assertEquals(-1, tail);
71      assertEquals(0, head);
72      assertEquals(0, queSize);
73    //test behaviors of "push" method
74      assertEquals("10 is inserted", q.push(10));
75      assertEquals("20 is inserted", q.push(20));
76      assertEquals("30 is inserted", q.push(30));
77      assertEquals("40 is inserted", q.push(40));
78      assertEquals("50 is inserted", q.push(50));
79      assertEquals("Que is full and overflow
          occurs", q.push(60));
80    //test behaviors of "full " method
81      assertEquals(true, q.full());
82    //test full queue size
83      queSize = (int)f5.get(q);
84      assertEquals(queSize, MAX_QSIZE);
85    //test current values of "content"
86      assertEquals(10, content[0]);
87      assertEquals(20, content[1]);
88      assertEquals(30, content[2]);
89      assertEquals(40, content[3]);
90      assertEquals(50, content[4]);
91    //test current value of "tail"
92      tail = (int)f3.get(q);
93      assertEquals(4, tail);
94    //test behaviors of "pop" method
95      assertEquals("10 is deleted", q.pop());
96      assertEquals("20 is deleted", q.pop());
97      assertEquals("30 is deleted", q.pop());
98      assertEquals("40 is deleted", q.pop());
99      assertEquals("50 is deleted", q.pop());
100   //test current value of "head"
101     head = (int)f4.get(q);
102     assertEquals(0, head);
103   //test behaviors of "push" method again
104     assertEquals("70 is inserted", q.push(70));
105     assertEquals("80 is inserted", q.push(80));
106   //test current values of "content"
107     assertEquals(70, content[0]);
108     assertEquals(80, content[1]);
109   //test current value of "tail"
110     tail = (int)f3.get(q);
111     assertEquals(1, tail);
112   //test behaviors of "pop " method again
113     assertEquals("70 is deleted", q.pop());
114     assertEquals("80 is deleted", q.pop());
115     assertEquals("Que is empty and underflow
          occurs", q.pop());
116   //test current value of "head"
117     head = (int)f4.get(q);
118     assertEquals(2, head);
119   //test current value of "queSize"
120     queSize = (int)f5.get(q);
121     assertEquals(queSize, 0);
122   //test behavior of " empty " method
123     assertEquals(true, q.empty());
124   }
125 }
```

D. Source Code for Stack

Stack is another basic data structure following *Last-In-First-Out (LIFO)*. In *Stack*, the insertion and deletion of data take places at one end called the top of the stack [14]. **source code 6** implements *Stack* using the *inheritance* and *polymorphism*. *Stack* class inherits the five important variables, *MAX_QSIZE*, *content*, *tail*, *head*, and *queSize*, and the three methods, *full*, *empty*, and *push*, from *Que* class. To inherit these variables in *Stack* class, their access modifiers must be changed to *protected* in *Que* class. *pop* method is overwritten in *Stack* class, to retrieve the data at the top of the *content*.

Listing 8: **source code 6**

```
1  public class StackExample extends QueExample {
2    public Object pop() {
3      if (!empty()) {
4        queSize --;
5        Object result = content[tail--];
6        return result+ " is deleted";
7      } else
8       return "Que is empty and underflow
          occurs";
9    }
10 }
```

E. Informative Test Code for Stack

Then, **test code 3** is generated as the *informative test code* for *Stack*.

In **test code 3**, *variableTest* method (*methodTest* method) tests the names, access modifiers, returning data types of variables (methods) defined in the parent *Que* class. *methodTest* method also tests the name, access modifier, returning data type of the overwritten *pop* method, where the access modifier must be *public* and the returning data type must be *Object*, and tests that the number of methods defined in *Stack* class must be one.

behaviorTest method tests the behaviors of *full*, *empty*, *push*, and *pop*. *full*, *empty*, and *push* are tested by called from *Que*, and *pop* is tested by called from *Stack*.

Listing 9: **test code 3**

```
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 variableTest() throws Exception {
9      Stack s = new Stack();
10     Class<?> parentClass = s.getClass().getSuperclass();
11   //test variables defined in parent class.
12     Field f1 = parentClass.getDeclaredField("MAX_QSIZE
         ");
13     Field f2 = parentClass.getDeclaredField("content");
14     Field f3 = parentClass.getDeclaredField("tail");
15     Field f4 = parentClass.getDeclaredField("head");
16     Field f5 = parentClass.getDeclaredField("queSize");
17   //test access modifiers of variables in parent class.
18     assertEquals(f1.getModifiers(), Modifier.PROTECTED,
         Modifier.FINAL);
19     assertEquals(f2.getModifiers(), Modifier.PROTECTED);
20     assertEquals(f3.getModifiers(), Modifier.PROTECTED);
21     assertEquals(f4.getModifiers(), Modifier.PROTECTED);
22     assertEquals(f5.getModifiers(), Modifier.PROTECTED);
23   //test data types of variables in parent class
```

```
24    assertEquals(f1.getType(), int.class);
25    assertEquals(f2.getType(), Object[].class);
26    assertEquals(f3.getType(), int.class);
27    assertEquals(f4.getType(), int.class);
28    assertEquals(f5.getType(), int.class);
29  //test number of variables defined in Stack class
30    Field[] f = s.getClass().getDeclaredFields();
31    assertEquals(0, f.length);
32  @Test
33  public void methodTest() throws Exception {
34    Stack s = new Stack();
35    Class<?> parentClass = s.getClass().getSuperclass();
36  //test methods defined in parent class
37    Method m1 = parentClass.getDeclaredMethod("full",
        null);
38    Method m2 = parentClass.getDeclaredMethod("empty
        ", null);
39    Method m3 = parentClass.getDeclaredMethod("push",
        Object.class);
40    Method m4 = parentClass.getDeclaredMethod("pop",
        null);
41  //test access modifiers of methods in parent class.
42    assertEquals(m1.getModifiers(), Modifier.PUBLIC);
43    assertEquals(m2.getModifiers(), Modifier.PUBLIC);
44    assertEquals(m3.getModifiers(), Modifier.PUBLIC);
45    assertEquals(m4.getModifiers(), Modifier.PUBLIC);
46  //test data types of methods in parent class.
47    assertEquals(m1.getReturnType(), boolean.class);
48    assertEquals(m2.getReturnType(), boolean.class);
49    assertEquals(m3.getReturnType(), Object.class);
50    assertEquals(m4.getReturnType(), Object.class);
51  //test method defined in Stack class as overwrite method
52    Method m = s.getClass().getDeclaredMethod("pop",
        null);
53    assertEquals(m.getModifiers(), Modifier.PUBLIC);
54    assertEquals(m.getReturnType(), Object.class);
55  //test number of methods defined in Stack class
56    Method[] methods=s.getClass().getDeclaredMethods();
57    assertEquals(1, methods.length);
58  }
59  @Test
60  public void behaviorTest() {
61    StackExample s = new StackExample();
62  //test initialize value of each variable
63    assertEquals(5, s.MAX_QSIZE);
64    assertEquals(−1, s.tail);
65    assertEquals(0, s.head);
66    assertEquals(0, s.queSize);
67  //test behaviors of "push" method
68    assertEquals("10 is inserted", s.push(10));
69    assertEquals("20 is inserted", s.push(20));
70    assertEquals("30 is inserted", s.push(30));
71    assertEquals("40 is inserted", s.push(40));
72    assertEquals("50 is inserted", s.push(50));
73    assertEquals("Que is full and overflow
        occurs", s.push(60));
74  //test behaviors of "full " method
75    assertEquals (true, s.full()) ;
76   //test full queue size
77    assertEquals(s.queSize, s.MAX_QSIZE);
78  //test current values of "content"
79    assertEquals(10, s.content[0]);
80    assertEquals(20, s.content[1]);
81    assertEquals(30, s.content[2]);
82    assertEquals(40, s.content[3]);
83    assertEquals(50, s.content[4]);
84  //test current value of "tail"
85    assertEquals(4, s.tail);
86  //test behaviors of "pop" method
87    assertEquals("50 is deleted", s.pop());
88    assertEquals("40 is deleted", s.pop());
89    assertEquals("30 is deleted", s.pop());
90    assertEquals("20 is deleted", s.pop());
91    assertEquals("10 is deleted", s.pop());
92  //test current value of "tail"
93    assertEquals(−1, s.tail);
94  //test "push" method again
95    assertEquals("70 is inserted", s.push(70));
96    assertEquals("80 is inserted", s.push(80));
97    assertEquals("90 is inserted", s.push(90));
98  //test current values of "content"
99    assertEquals(70, s.content[0]);
100   assertEquals(80, s.content[1]);
101   assertEquals(90, s.content[2]);
102 //test current value of "tail"
103   assertEquals(2, s.tail);
104 //test "pop" method again
105    assertEquals("90 is deleted", s.pop());
106   assertEquals("80 is deleted", s.pop());
107   assertEquals("70 is deleted", s.pop());
108   assertEquals("Que is empty and underflow
        occurs", s.pop());
109 //test current value of "tail"
110   assertEquals(−1, s.tail);
111 //test current value of "queSize" when queue is empty
112   assertEquals(true, q.empty());
113   assertEquals(s.queSize, 0);
114  }
115 }
```

IV. Evaluations

In this section, we evaluate the informative test code approach in the code writing problem for studying the three OOP concepts.

A. Evaluation Setup

We generated the informative test codes for 10 programming assignments that require the use of the three OOP concepts, and asked totally 13 students in Myanmar and Japan who are currently studying Java programming to solve them. These assignments include *Queue*, *Stack*, and eight sample codes in Web sites [15]-[19]. First, we asked the eight students to solve those assignments. All of the students completed the source code for any assignment that passes the test code.

Then, we measured the *quality metrics* and the *coverage metric* using the *metric plugin for Eclipse* and the *coverage tool for Eclipse* respectively. Their results were analyzed to examine the quality of the source code using the OOP concepts.

B. Quality Metrics Results

Table I shows the distribution of the measured quality and coverage metric of the sources codes written by the students. Here, “coverage (%)” indicates the coverage rate of the methods. Any quality metric exhibits a good value except for assignment #2.

For **NOC**, most students use the same number of classes as the specified one in the test code for all the assignments. Some students use the larger number of classes for assignments #1, #2, and #4 by producing unnecessary classes. Unfortunately, the current informative test code does not test the existence of them, which will be in future studies.

For **NOM**, some students use the larger number of methods than the specified one in the test code for the assignments #1, #2, #3, #4, #6, #7, #10 by making the *constructor*. The current informative test code does not test the existence of *constructor*. Besides, the student makes unnecessary classes where the informative test code does not test the existence of methods in the unnecessary cl asses. Then, they are counted as the number of methods by **NOM**.

For **VG**, the values for the assignments #1, #2, and #7 are distributed, because they ask more complex codes using the OOP concepts that require conditional statements. The larger number of conditional statements makes the larger value of **VG**. It is noted that **VG** should be less than 20. The informative test code does not test the existence of them.

TABLE I: Quality and Coverage Metrics Results of Answer Source Codes.

metrics	assignment									
	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10
encapsulation	√	−	−	−	√	√	√	−	√	√
inheritance	−	√	√	√	−	−	√	−	−	√
polymorphism	−	√	√	−	−	−	−	√	−	−
NOC	1~3	2~3	2	3~4	1	1	2	1	1	2
NOM	4~14	5~14	2~4	3~6	6	8~9	3~7	6	3	5~12
VG	2~4	2~3	1	1	1	1	1~3	1	1	1
NBD	2~3	2~3	1	1	1	1	1	1	1	1
LCOM	0.2~0.6	0.1~1	0	0	0.5~0.7	0.6~0.7	0.2~0.5	0.5~0.6	0	0.4~0.6
TLC	40~118	44~94	16~25	18~33	26~33	34~40	17~33	23~27	12~21	24~35
MLC	21~56	21~46	2~5	3~9	8~9	11~14	3~12	6~12	3~9	5~15
coverage (%)	58.8~100	72.8~100	63.6~90.9	62.5~100	100	56.5~94.6	63.35~100	100	100	70.0~100

For **NBD**, the values are varied for the assignments #1 and #2, because they require conditional statements. However, **NBD** should not be greater than 5, which is satisfied for any assignment. Again, the informative test code does not give the specification for it.

For **LCOM**, the value is smaller than 1 for any assignment except one answer code in assignments #2. The larger value makes the class less cohesiveness of the class in the code.

For **MLC**, the students use a different number of statements in the methods, including **conditional** statements, **expression** statements, and **declaration** statements.

For **TLC**, the students use a different number of statements in the class including **import statements**, **conditional** statements, **expression** statements, and **declaration** statements.

C. Coverage Metric Results

Table I shows that the **coverage** metric is not always 100% for assignments #1, #2, #3, #4, #6, #7,and #10, due to the insufficiency of the test codes. The test code misses testing specified methods and/or possible paths for conditional statements in the source code. For example, **test code 4** for assignment #1 tests only the “true” condition of the two conditional statements in **source code 7** at lines 10 and 18. From the coverage testing, it can be found that the test code must be improved to cover all the statements in the model source code. Thus, it is important for a teacher to measure the coverage metric of the generated test code for the model source code before assigning it to students.

Listing 10: **test code 4**

```
1  .
2    //test behaviors of "push" method
3      assertEquals("10 is inserted", q.push(10));
4      assertEquals("20 is inserted", q.push(20));
5      assertEquals("30 is inserted", q.push(30));
6      assertEquals("40 is inserted", q.push(40));
7      assertEquals("50 is inserted", q.push(50));
8      assertEquals("Que is full and overflow
         occurs", q.push(60));
9    //test behaviors of "full" method after push values
10     assertEquals(true, q.full());
11 ...
12   //test behaviors of "pop" method
13     assertEquals("70 is deleted", q.pop());
14     assertEquals("80 is deleted", q.pop());
15     assertEquals("Que is empty and underflow
         occurs", q.pop());
16 ...
17   //test behaviours of "empty" method after pop the values
18     assertEquals(true, q.empty());
```

Listing 11: **source code 7**

```
1  ...
2    public boolean full(){
3      if (queSize==MAX_QSIZE) {
4        return true;
5      }
6      else {
7        return false;
8      }
9    }
10   public boolean empty(){
11     if (queSize==0) {
12       return true;
13     }
14     else {
15       return false;
16     }
17   }
18 ...
```

Under current situations, it is difficult or impossible for a teacher to prepare the perfect test code that can avoid the insufficiency. Therefore, an assistant tool to help a teacher generate a complete test code should be developed in JPLAS, which will be in future works.

D. Relationship between Quality and Coverage Metrics

Then, we analyze the relationship between the quality metrics results and the coverage metric. Table II shows the quality metrics values for the source codes by the students whose coverage metric is highest and lowest for each of the seven assignments. This table shows that in general, the source code with the high coverage has better quality metrics values than the code with the low coverage. For example, for assignment #1, NOC and NOM are three times larger for the source code with the lowest coverage than the one with the highest coverage.

The results in Table II suggest that the improvement of the coverage metric in the source code can lead to the improvement of the quality metrics. Using the coverage tool, the uncovered statements in the source code can be easily detected. By requesting the student to reduce the uncovered statements in his/her source code, the quality metrics can be improved at the same time. In future works, we will implement the function of showing the uncovered statement in the answer source code of the student and requesting him/her to remove them, and evaluate it.

TABLE II: Quality Metrics for Student Codes with Highest and Lowest Coverage.

metrics	assignment													
	#1		#2		#3		#4		#6		#7		#10	
	max	min	max	min	max	min	max	min	max	min	max	min	max	min
NOC	1	3	2	3	2	2	3	4	1	1	2	2	2	2
NOM	4	14	5	14	5	4	3	6	9	8	3	7	5	9
VG	2	3	2	3	1	1	1	1	1	1	1	1	1	1
NBD	2	2	2	3	1	1	1	1	1	1	1	1	1	1
LCOM	0.6	0.5	0.5	0.8	0	0	0	0	0.7	0.6	0.5	0.3	0.5	0.6
TLC	41	118	58	94	22	18	18	33	34	35	18	30	24	40
MLC	25	56	21	48	2	4	3	6	11	11	3	10	5	11
coverage (%)	100	58.8	100	72.8	90.0	63.6	100	62.5	94.6	56.5	100	63.35	100	70.0

V. Conclusion

This paper proposed the *informative test code approach* in the code writing problem for studying the three object-oriented programming concepts in JPLAS. The informative test code describes the necessary information for implementing the source code using the concepts. The effectiveness of the proposal was evaluated through applying 10 informative test codes to 13 students in Myanmar and Japan, where all of them could complete source codes using the concepts with sufficient quality and coverage metrics in general. Our future works will include the development of the assistant tool to write a complete test code for a teacher, the implementation of the function in JPLAS to show the uncovered statements in the answer code for a student, the generation of informative tests codes to other complex assignments, and their applications to Java programming courses.

References

[1] N. Funabiki, K. K. Zaw, N. Ishihara, and W.-C. Kao, "Java programming learning assistant system: JPLAS," IAENG transactions on Engineering Sciences Special Issue for the International Association of Engineers Conferences 2016, vol. 2, pp. 517-530, 2016.

[2] N. Ishihara, N. Funabiki, M. Kuribayashi, and W.-C. Kao, "A software architecture for Java programming learning assistant system," International Journal of Computer Software and Engineering, vol. 2, no.1, 2017.

[3] 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 International Journal of Computer Science, vol. 44, no. 2, pp. 247-260, 2017.

[4] 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," Journal of Information Engineering Express, vol. 1, no. 3, pp. 9-18, 2015.

[5] 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," Journal of Information Engineering Express, vol. 1, no. 3, pp. 19-28, 2015.

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

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

[8] K. K. Zaw and N. 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, 2017.

[9] MetricPlugin,http://metrics.sourceforge.net.

[10] Encapsulation, https://www.tutorialspoint.com/java/encapsulation.html

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

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

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

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

[15] OOPs in Java: Encapsulation, Inheritance, Polymorphism, Abstraction, https://beginnersbook.com/2013/03/oops-in-java-encapsulation-inheritance-polymorphism-abstraction/

[16] Basic Grammar, https://www.zealseeds.com/Lang/LangJava/BasicGrammar/InheritanceOfJava/index.html

[17] First Java, http://www1.bbiq.jp/takeharu/java100.html

[18] ITSakura, https://itsakura.com/java-inheritance

[19] GitHubGist, https://gist.github.com/rtoal/1685886e6605fe73b792

Khin Khin Zaw received the B.E. degree in information technology from Technological University (HmawBi), Myanmar, in 2006, the M.E. degree in information technology from Mandalay Technological University, Myanmar, in 2011, and the Ph.D. in communication network engineering from Okayama University, respectively. She is currently a lecturer in the Department of Computer Engineering and Information Technology at Yangon Technological University, Myanmar. Her research interests include educational technology and Web application systems. She is a member of IEICE.

Win Zaw received the B.E. degree in electronics engineering from Mandalay Technological University, Myanmar, in 1998, M.E and the Ph.D. in information technology from National Research Nuclear University, Russia, in 2007, respectively. He is currently a professor and the head of the Department of Computer Engineering and Information Technology, Yangon Technological University, Myanmar. His research interests include computer networks, E-learning, and modeling. He is a member of IEEE.

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.

Wen-Chung Kao received the M.S. and Ph.D. degrees in electrical engineering from National Taiwan University, Taiwan, in 1992 and 1996, respectively. From 1996 to 2000, he was a Department Manager at SoC Technology Center, ERSO, ITRI, Taiwan. From 2000 to 2004, he was an Assistant Vice President at NuCam Corporation in Foxlink Group, Taiwan. Since 2004, he has been with National Taiwan Normal University, Taipei, Taiwan, where he is currently a Professor at Department of Electrical Engineering and the Dean of School of Continuing Education. His current research interests include system-on-a-chip (SoC), flexible electrophoretic display, machine vision system, digital camera system, and color imaging science. He is a senior member of IEEE.