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 Eclipse 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.

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
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         assertEquals(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:

   \[
   LCOM = \begin{cases} \frac{P - Q}{Q} & \text{if } P > Q \\ 0 & \text{otherwise} \end{cases} \]

   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, EcIEmma 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

```java
public class Student {
    private String name;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

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

```java
class A {
    float salary=40000;
}
class B extends A {
    int bonus=100000;
    public static void main (String args[]) {
        B b=new B();
        System.out.println(b.salary);
        System.out.println(b.bonous);
    }
}
```

3) Polymorphism: The polymorphism is the ability of an object to take on a plenty of forms. The most common use of polymorphism 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 polymorphism. 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

```java
public class Animal {
    public void makeNoise() {
        System.out.println("Some sound");
    }
}
class Dog extends Animal {
    public void makeNoise() {
        System.out.println("Bark");
    }
    public void makeNoise(int x) {
        for (int i=0; i<x; i++)
            System.out.println("Bark");
}
```

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 polymorphism 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

```java
public class QueExample {
    private final int MAX_QSIZE = 5;
    private Object content[] = new Object[MAX_QSIZE];
    private int head = 0;
    private int tail = -1;
    private int queSize = 0;

    //empty method
    public boolean empty() {
      return (queSize==0);
    }

    //full method
    public boolean full() {
      return (queSize==MAX_QSIZE);
    }

    public Object push(Object data) {
      if (full()) {
        queSize += 1;
        tail = (tail+1) % MAX_QSIZE;
        content[tail] = data;
      } else {
        return "Que is full and overflow occurs";
      }
    }

    public Object pop() {
      if (empty()) {
        queSize -= 1;
        Object result = content[head];
        head = (head+1) % MAX_QSIZE;
        System.out.println(queSize);
        return result;
      } else {
        return "Que is empty and underflow occurs";
      }
    }
}
```

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

```java
import static org.junit.Assert.*;
import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.lang.reflect.Modifier;
import org.junit.Test;

public class QueTest {  
  @Test
  public void variableTest() throws Exception {
    QueExample q = new QueExample();
    //Field
    Field f1 = q.getClass().getDeclaredField("MAX_QSIZE");
    Field f2 = q.getClass().getDeclaredField("content");
    Field f3 = q.getClass().getDeclaredField("tail");
    Field f4 = q.getClass().getDeclaredField("head");
    Field f5 = q.getClass().getDeclaredField("queSize");
    //test the access modifier
    assertEquals(f1.getModifiers(), Modifier.PRIVATE);
    assertEquals(f2.getModifiers(), Modifier.PRIVATE);
    assertEquals(f3.getModifiers(), Modifier.PRIVATE);
    assertEquals(f4.getModifiers(), Modifier.PRIVATE);
    assertEquals(f5.getModifiers(), Modifier.PRIVATE);
    //test the datatype of variables
    assertEquals(f1.getType(), int.class);
    assertEquals(f2.getType(), Object[].class);
    assertEquals(f4.getType(), int.class);
    assertEquals(f5.getType(), int.class);
    //test the number of variables
    Field[] f = q.getClass().getDeclaredFields();
    assertEquals(f.length, 5);
  }

  @Test
  public void methodTest() throws Exception {
    QueExample q = new QueExample();
    Method m1 = q.getClass().getDeclaredMethod("full", null);
    Method m2 = q.getClass().getDeclaredMethod("empty", null);
    Method m3 = q.getClass().getDeclaredMethod("push", Object.class);
    Method m4 = q.getClass().getDeclaredMethod("pop", null);
    assertEquals(m1.getModifiers(), Modifier.FINAL);
    assertEquals(m2.getModifiers(), Modifier.FINAL);
    assertEquals(m3.getModifiers(), Modifier.FINAL);
    assertEquals(m4.getModifiers(), Modifier.FINAL);
    //test the number of methods
    Method[] m = q.getClass().getDeclaredMethods();
    assertEquals(m.length, 4);
  }
}
```
assertEquals(m2.getReturnType(), boolean.class);
assertEquals(m3.getReturnType(), Object.class);
assertEquals(m4.getReturnType(), Object.class);
Method m = q.getClass().getDeclaredMethods();
assertEquals(4, m.length);
}

@Test
public void behaviorTest() throws Exception {
    QueExample q = new QueExample();
    Field f1 = q.getClass().getDeclaredField("MAX_QSIZE");
    Field f2 = q.getClass().getDeclaredField("content");
    Field f3 = q.getClass().getDeclaredField("tail");
    Field f4 = q.getClass().getDeclaredField("head");
    Field f5 = q.getClass().getDeclaredField("queSize");
    //make private variables accessible for tests
    f1.setAccessible(true);
    f2.setAccessible(true);
    f3.setAccessible(true);
    f4.setAccessible(true);
    f5.setAccessible(true);
    int MAX_QSIZE = (int)f1.get(q);
    Object[] content = (Object[])f2.get(q);
    int tail = (int)f3.get(q);
    int head = (int)f4.get(q);
    int queSize = (int)f5.get(q);
    //test the initialize value of each variable
    assertEquals(5, MAX_QSIZE);
    assertEquals(-1, tail);
    assertEquals(0, head);
    assertEquals(0, queSize);
    //test behaviors of "push" method
    assertEquals("10 is inserted", q.push(10));
    assertEquals("20 is inserted", q.push(20));
    assertEquals("30 is inserted", q.push(30));
    assertEquals("40 is inserted", q.push(40));
    assertEquals("50 is inserted", q.push(50));
    assertEquals("Que is full and overflow occurs", q.push(60));
    //test behaviors of "full" method
    assertEquals(true, q.full());
    //test full queue size
    queSize = (int)f5.get(q);
    assertEquals(queSize, MAX_QSIZE);
    //test current values of "content"
    assertEquals(10, content[0]);
    assertEquals(20, content[1]);
    assertEquals(30, content[2]);
    assertEquals(40, content[3]);
    assertEquals(50, content[4]);
    //test current value of "tail"
    tail = (int)f3.get(q);
    assertEquals(4, tail);
    //test behaviors of "pop" method
    assertEquals("10 is deleted", q.pop());
    assertEquals("20 is deleted", q.pop());
    assertEquals("30 is deleted", q.pop());
    assertEquals("40 is deleted", q.pop());
    assertEquals("50 is deleted", q.pop());
    //test current value of "head"
    head = (int)f4.get(q);
    assertEquals(0, head);
    //test behaviors of "push" method again
    assertEquals("70 is inserted", q.push(70));
    assertEquals("80 is inserted", q.push(80));
    //test current values of "content"
    assertEquals(70, content[0]);
    assertEquals(80, content[1]);
    //test current value of "tail"
    tail = (int)f3.get(q);
    assertEquals(1, tail);
    //test behaviors of "pop" method again
    assertEquals("70 is deleted", q.pop());
    assertEquals("80 is deleted", q.pop());
    assertEquals("Que is empty and underflow occurs", q.pop());
    //test current value of "head"
    head = (int)f4.get(q);
    assertEquals(2, head);
    //test current value of "queSize"
    queSize = (int)f5.get(q);
    assertEquals(queSize, 0);
    //test behavior of "empty" method
    assertEquals(true, q.empty());
}

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

```java
public class StackExample extends QueExample {
    public StackExample() {
        if (empty()) {
            queSize --;
            Object result = content[tail--];
        } else
            return "Que is empty and underflow occurs";
    }
}
```

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

```java
import static org.junit.Assert.*;
import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.lang.reflect.Modifier;
import org.junit.Test;
public class StackTest {
    @Test
    public void variableTest() throws Exception {
        Stack s = new Stack();
        Class<?> parentClass = s.getClass().getSuperclass();
        //test variables defined in parent class.
        Field f1 = parentClass.getDeclaredField("MAX_QSIZE");
        Field f2 = parentClass.getDeclaredField("content");
        Field f3 = parentClass.getDeclaredField("head");
        Field f4 = parentClass.getDeclaredField("tail");
        Field f5 = parentClass.getDeclaredField("queSize");
        //test access modifiers of variables in parent class.
        assertEquals(f1.getModifiers(), Modifier.PROTECTED);
        assertEquals(f2.getModifiers(), Modifier.PROTECTED);
        assertEquals(f3.getModifiers(), Modifier.PROTECTED);
        assertEquals(f4.getModifiers(), Modifier.PROTECTED);
        assertEquals(f5.getModifiers(), Modifier.PROTECTED);
        //test data types of variables in parent class
    }
```
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 classes. 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.

<table>
<thead>
<tr>
<th>metrics</th>
<th>#1</th>
<th>#2</th>
<th>#3</th>
<th>#4</th>
<th>#5</th>
<th>#6</th>
<th>#7</th>
<th>#8</th>
<th>#9</th>
<th>#10</th>
</tr>
</thead>
<tbody>
<tr>
<td>encapsulation</td>
<td>√</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>√</td>
<td>√</td>
<td>-</td>
<td>√</td>
<td>-</td>
<td>√</td>
</tr>
<tr>
<td>inheritance</td>
<td>-</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>-</td>
<td>-</td>
<td>√</td>
<td>-</td>
<td>-</td>
<td>√</td>
</tr>
<tr>
<td>polymorphism</td>
<td>-</td>
<td>√</td>
<td>√</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>√</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NOC</td>
<td>1–3</td>
<td>2–3</td>
<td>2</td>
<td>3–4</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>NOM</td>
<td>4–14</td>
<td>5–14</td>
<td>2–4</td>
<td>3–6</td>
<td>6</td>
<td>8–9</td>
<td>3–7</td>
<td>6</td>
<td>3</td>
<td>5–12</td>
</tr>
<tr>
<td>VG</td>
<td>2–4</td>
<td>2–3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3–7</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>NBD</td>
<td>2–3</td>
<td>2–3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>LCOM</td>
<td>0.2–0.6</td>
<td>0.1–1</td>
<td>0</td>
<td>0</td>
<td>0.5–0.7</td>
<td>0.6–0.7</td>
<td>0.2–0.5</td>
<td>0.5–0.6</td>
<td>0</td>
<td>0.4–0.6</td>
</tr>
<tr>
<td>coverage (%)</td>
<td>58.8–100</td>
<td>72.8–100</td>
<td>63.6–90.9</td>
<td>62.5–100</td>
<td>100</td>
<td>56.5–94.6</td>
<td>63.35–100</td>
<td>100</td>
<td>100</td>
<td>70.0–100</td>
</tr>
</tbody>
</table>

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 methods, including conditional statements.

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.

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.

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.

Listing 10: test code 4

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

Listing 11: source code 7

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

(D'Advance online publication: 12 August 2019)
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 test codes to other complex assignments, and their applications to Java programming courses.

### 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 test codes to other complex assignments, and their applications to Java programming courses.

### REFERENCES


[12] Polymorphism, runtime-polymorphism-in-java


Khin Khin Zaw received the B.E. degree in information technology from Technological University (HmaBw), 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, the M.E. degree 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 NaCam 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.