Article

Answer Code Validation Program with Test Data Generation for Code Writing Problem in Java Programming Learning Assistant System†

Khaing Hsu Wai [1], Nobuo Funabiki [1,*], Soe Thandar Aung [1], Xiqin Lu [1], Yanhui Jing [1], Htoo Htoo Sandi Kyaw [2], Wen-Chung Kao[3]

[1] Graduate School of Natural Science and Technology, Okayama University, Okayama Japan; pjsu9uam@s.okayama-u.ac.jp; funabiki@okayama-u.ac.jp; soethandar@s.okayama-u.ac.jp; pch55zhl@s.okayama-u.ac.jp; pf709l29@s.okayama-u.ac.jp
[2] Department of Computer and Information Science, Tokyo University of Agriculture and Technology, Tokyo, Japan; htoohtoosk@go.tuat.ac.jp
[3] Department of Electrical Engineering, National Taiwan Normal University, Taipei, Taiwan; jungkao@ntnu.edu.tw
* Correspondence: funabiki@okayama-u.ac.jp

† This paper is an extended version of our paper published in 2023 11th International Conference on Information and Education Technology (ICIET), 18-20 March 2023, Fujisawa, Japan.

Abstract: To assist *Java programming* learning of novice students, we have developed the web-based *Java programming learning assistant system (JPLAS)*. JPLAS provides several types of exercise problems to cultivate *code reading/writing* skills at various levels. In JPLAS, the *code writing problem (CWP)* asks a student to write a source code that will pass the *test code* given in the assignment where the correctness is verified by running them on *JUnit*. In this paper, to reduce the teacher load at marking process, we present the *answer code validation program* that verifies all the source codes from a lot of students for each assignment at a time and reports the number of passing tests for each source code in the CSV file. Besides, to test a source code with various input data, we implement the *test data generation algorithm* that identifies the data type, generates a new data, and replaces it for each test data in the test code. Furthermore, to verify the correctness of the implemented procedure in the source code, we introduce the *intermediate state testing* in the test code. For evaluations, we applied the proposal to source codes and test codes in a Java programming course in Okayama university, Japan, and confirmed the validity and effectiveness.

Keywords: programming learning; Java; JUnit; code writing problem; code validation; test data generation

1. Introduction

For decades, *Java* has been extensively used in industries as a reliable and portable object-oriented programming language. The use of *Java* has involved mission critical systems in large enterprises and also small-sized embedded systems. There have been strong demands among IT companies for cultivations of *Java programming* engineers. A great number of universities and professional schools are offering Java programming courses to meet the needs.

To assist self-studies of Java programming by novice students, we have developed the *Java programming learning assistant system (JPLAS)*, and have implemented the personal answer platform on *Node.js* [1], which will be distributed to students on *Docker* [2]. *JPLAS* offers various types of exercise problems with automatic marking functions that will gradually progress the learning stages of students. Therefore, JPLAS can cover self-studies of Java programming at different levels by novice students.

In programming study, novice students should start from solving simple and short exercise problems for *code reading study*, to understand and master grammar and programming concepts of the language. Then, after having some knowledge and skills through *code reading study*, they should move to *code writing study*. If students cannot read source codes, they cannot write them correctly.

To assist this progressive programming study for novice students, JPLAS offers the following exercise problems.

1. The *grammar-concept understanding problem (GUP)* requests to answer the important words in the given source code, such as reserved words and common libraries in the programming language, by giving the questions that describe their concepts [3].
2. The *value trace problem (VTP)* requests to the current values of the important variables and the output messages in the given source code [4].
3. The *element fill-in-blank problem (EFP)* requests to fill in the blank elements in the given source code so that the original source code is gained [5].
4. The *code completion problem (CCP)* requests to correct and complete the given source code that has several blank elements and incorrect ones [6].
5. The *code writing problem (CWP)* requests to write a source code that will pass the tests in the given *test code* [7].

For any exercise problem, the correctness of any answer from a student is verified automatically at the answer platform. For *GUP*, *VTP*, *EFP*, and *CCP*, the correctness is checked through *string matching* with the correct one. For *CWP*, it is checked through *unit testing* on *JUnit* [8].

Among the exercise problems in JPLAS, the *CWP* is essential for students to study and master practically writing source codes from scratch. For *CWP*, we have implemented the *answer platform* for students to solve the given CWP assignments efficiently [2]. By implementing the capability of automatically running *unit tesing*, students can easily check the correctness of their source codes by clicking the corresponding button.

However, the current implementation of *CWP* causes three limitations. First, the teacher needs to manually run the test codes and the source codes one by one at the marking process, although a lot of source codes are usually submitted from students. This load is relative large for the teacher. Second, the test data in each test code is usually only one type. Thus, *unit testing* may pass an incorrect source code that will only output the expected output described in the given test code without implementing the requested procedure. Third, even if the source code implements a different logic or algorithm from the requested one, *unit testing* cannot find it.

In this paper, to solve the first limitation, we present the *answer code validation program* to help the teacher by reducing the load at marking a lot of source codes from students in a Java programming course. This program automatically verifies all the source codes from students for each assignment, and reports the number of passing tests for each source code in the CSV file. By looking at the summary of the test results for all the students, the teacher can easily grasp the correctness of the student answers and grade them.

To solve the second limitation, we implement the *test data generation algorithm* that identifies the data type, randomly generates a new data with this data type, and replaces it for each test data in the test code, so that the source code can be tested with various input data in the test code. By dynamically changing the test data, it is expected to reduce the risk of cheating and enhance the validity of CWP assignments.

To solve the third limitation, we introduce the *intermediate state testing* in the test code that will check the correctness of the important variables in the source code to implement the requested logic or algorithm. If a student implements a different logic or algorithm including the use of a library, this test is not passed. Besides, it is expected that students will gain deeper understanding of the logic or algorithm, improve problem-solving skills, and develop strong foundations in algorithmic thinking.

The rest of this paper is organized as follows: Section 2 discusses related works in literature. Section 3 reviews our previous works of *CWP* and the answer platform. Section 4 presents the implementation of the *answer code validation program*. Section 5 presents the test data generation algorithm and the intermediate state testing. Section 6 evaluates the proposal. Finally, Section 7 concludes this paper with future works.

2. Related Works in Literature

In this section, we discuss related works in literature.

In [9] and [10], the authors suggested the common problems among programming novices, along with existing efforts and discussions of current methods used in teaching programming. Several tools have been proposed to help students to solve programming learning difficulties. Among them, *ToolPetcha* is the example tool that acts as an automated assistant in matters of programming [11].

In [12], the authors explored several factors that can influence students to be dropped out from an introductory programming course, including motivations and problem-solving skills.

In [13], the authors reviewed recent developments of automatic assessment tools for programming exercises, and discussed their major features and approaches, including programming languages, learning management systems, testing tools, limitations on resubmissions, manual assessments, security, distributions, and specialties.

In [14], the authors presented and evaluated a web-based tool providing drill and practice that supports for Java programming called *CodeWrite*. Here, students are responsible for developing exercises that are shared among classmates. Because it does not adopt a testing tool such as *JUnit*, possible variations for program testing are limited.

In [15], the authors proposed a game-based learning environment to assist beginner students learning programming. It uses game creation tasks to make basic programming easier to understand and includes idea visualization approaches to let students manipulate game objects to learn important programming concepts.

In [16], the authors proposed a graph-based grading system for introductory Java programming courses called *eGrader*. The dynamic analysis of a submitted program is done on *JUnit*, and the static analysis is done on the graph representation of the program. The accuracy was confirmed through experiments.

In [17], the authors proposed a test case prioritization approach on JUnit on the estimated coverage information from the static call graph analysis of test cases, called *JUPTA*. They showed that the test suites by *JUPTA* were more effective than those in random and untreated test orders in terms of fault-detection effectiveness.

In [18], the authors proposed a web-based automatic scoring system for Java programming assignments. It receives a Java application program as the answer from a student and returns the test result that consists of compiler check, JUnit test, and result test immediately. This system is similar to the code writing problem in JPLAS where result test can be included in *JUnit* test.

In [19], to investigate students' perceptions of the learning environment, the authors developed a collaborative, learning environment based on the problems powered by the technology for dynamic webs. This research was planned as a qualitative study. A semi-structured interview format was created to get opinions of students about the learning environment that was supported by technologies for dynamic webs and was used for collaboratively solving issues. Their findings implied that collaborative learning techniques can focus on problems and the learning environment at a community college can benefit from technologies for dynamic web pages.

In [20], the authors presented an automatic Java code grading function called *MeMOOC* for checking syntactical, semantical, and pragmatic aspects with compiling, *Checkstyle*, and *JUnit* for *MOOC*.

In [21], the authors made comparative evaluations of several online platforms for teaching programming, and chose engaging assignments from the site used to educate students named *hackerrank.com*. They investigated user experiences with *online coding platforms (OCP)* and contrasted the features of various online platforms that should be utilized to teach programming to aspiring computer scientists and programmers via distance learning. In addition, they suggested use of online programming simulators to enhance computer science instructions, taking into account functionality, as well as students' preparation levels and expected results of learning.

3. Overview of Code Writing Problem

In this section, we review our previous works of the *code writing problem (CWP)* and the answer platform using *Node.js* in JPLAS.

3.1. Code Writing Problem

One assignment in the *code writing problem (CWP)* consists of the statement and the *test code* that should be prepared by a teacher. A student is requested to write the Java source code that passes every test case described in the test code. The correctness of the source code written by the student is verified by running the test code with the source code on *JUnit* for *code testing*. The student should write the source code by referring to the detailed specifications that are described in the test code.

A teacher generates a new assignment for *CWP* through the following procedure:

1. To prepare the problem statement and the input data for the new assignment,
2. To prepare the model source code as the answer code of this assignment,
3. To obtain the expected output data by running the model source code,
4. To write the *test code* using the input data and output data for test cases, and describing the necessary information to implement the source code, and
5. To register the statement and the test code for the new assignment.

3.2. JUnit

For *code testing*, *JUnit* is adopted as an open-source Java framework to support the *test-driven development (TDD)* method. *JUnit* can assist the automatic unit test of a source code or a class. Since it has been designed with the Java-user friendly style, the use is relatively easy for Java programmers. Using *JUnit*, one test can be performed by using the method in the library whose name starts with “assert”. The test code for *CWP* adopts the “assertEquals” method that compares the execution result of the source code with its expected output data for the given input data.

3.3. Test Code

A *test code* is written by using the library in *JUnit*. The *BubbleSort* class in Figure 1 [22] is used to explain how to write the corresponding test code. This *BubbleSort* class contains a method for performing the bubble sort algorithm on an integer array. “sort(int[] a)” method performs the basic bubble sort algorithm on the input array “a” and returns the sorted array.

```
package CWP;
public class BubbleSort {
public static int[] sort(int[] a) {
        int n = a.length;
        int temp = 0;
        for(int i=0; i < n; i++){
            for(int j=1; j < (n-i); j++){
                if(a[j-1] > arr[j]){
                    temp = a[j-1];
                    a[j-1] = a[j];
                    a[j] = temp;
               }
           }
     }
        return a;
    }
}
```

Figure 1. Example source code for *BubbleSort*.

The *test code* in Figure 2 should be made to test the *sort* method in the *BubbleSort* class in Figure 1.

The *test code* imports the *JUnit* packages containing test methods at lines 1 and 2, and declares the *BubbleSortTest* class at line 4. *@Test* at line 4 indicates that the succeeding *testSort* method represents the test case, which describes the procedure for testing the output of the *sort* method. This test is performed as follows:

1. The *bubbleSort* object of the *BubbleSort* class in the source code is generated in the test code.
2. The *sort* method of the *bubbleSort* object is called with the arguments for the input data.
3. The result of the *sort* method at *codeOutput* is compared with the *expOutput* data using the *assertEquals* method.

```
package CWP;
import static org.junit.Assert.*;
import org.junit.Test;
import java.util.Arrays;
public class BubbleSortTest {
    @Test
    public void testSort() {
        BubbleSort bubbleSort = new BubbleSort();
        int[] codeInput1 = {7,5,0,4,1,3};
        int[] codeOutput = bubbleSort.sort(codeInput1);
        int[] expOutput = {0,1,3,4,5,7};
        try {
            assertEquals("1:One input case:",Arrays.toString(expOutput),
                Arrays.toString(codeOutput));
        } catch (AssertionError ae) {
            System.out.println(ae.getMessage());
        }
    }
}
```

Figure 2. Example test code for BubbleSort.

3.4. CWP Answer Platform

To assist students solving the given CWP assignments, we have implemented the CWP answer platform as a web application system using *Node.js*. Figure 3 illustrates the software architecture. It is noted that the OS can be *Linux* or *Windows*. This platform follows the *MVC model*. For the *model (M)* part, *JUnit* is adopted and the *file system* is used to manage the data, because every data is provided by files. *Java* is used to implement the programs. For the *view (V)* part on the browser, *Embedded JavaScript (EJS)* is used instead of using the default template engine of *Express.js* to avoid the complex syntax structure. For the *control (C)* part, *Node.js* and *Express.js* are adopted together. *JavaScript* is used to implement the programs.

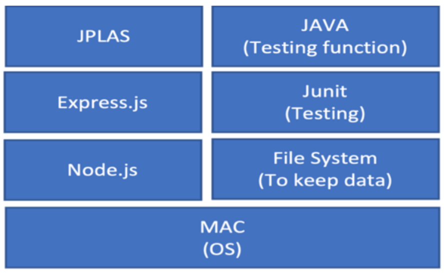
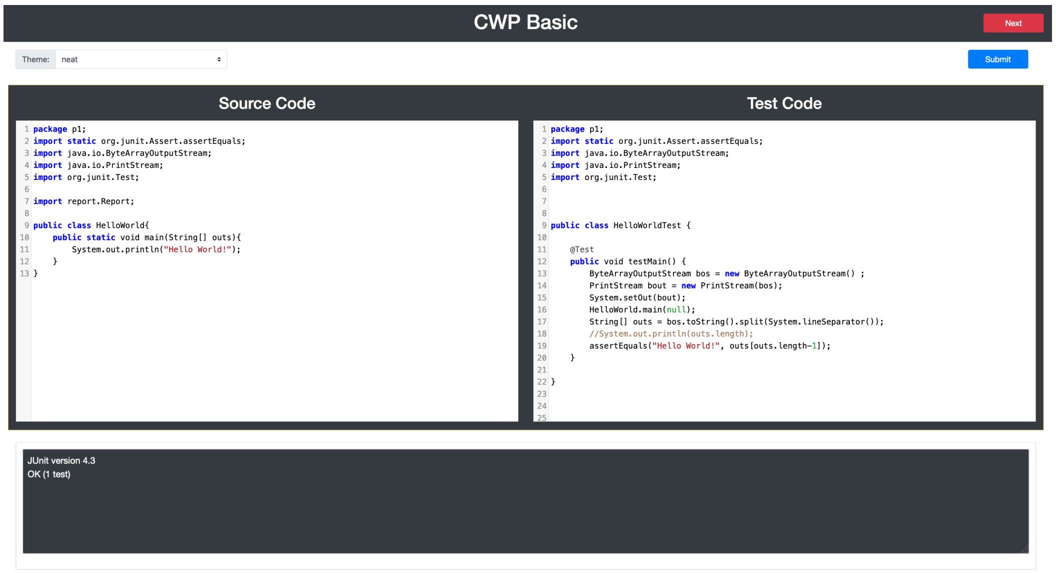


Figure 3. CWP software architecture.

Figure 4 illustrates the answer interface to solve an CWP assignment on a web browser. The right side of the interface shows the test code of the assignment. The left side shows the input space to write the answer source code. A student needs to write the code to pass all the tests in the test code while looking at it. After writing the source code, the student needs to submit it to the system by clicking the “Submit” button. Then, the code testing is applied immediately by compiling the source code and running the test code with it on *JUnit* and returns the test results at the lower side of the interface.

Unfortunately, the student needs to save the source code in the file whose name corresponds to the test code name manually, in the current implementation of this platform. This source code file is necessary to be submitted to the teacher for the final verification using the *answer code validation program*. The implementation of the automatic file saving for submissions will be in future works.



```java
package p1;
import static org.junit.Assert.assertEquals;
import java.io.ByteArrayOutputStream;
import java.io.PrintStream;
import org.junit.Test;

import report.Report;

public class HelloWorld{
    public static void main(String[] outs){
        System.out.println("Hello World!");
    }
}
```

```java
package p1;
import static org.junit.Assert.assertEquals;
import java.io.ByteArrayOutputStream;
import java.io.PrintStream;
import org.junit.Test;



public class HelloWorldTest {

    @Test
    public void testMain() {
        ByteArrayOutputStream bos = new ByteArrayOutputStream() ;
        PrintStream bout = new PrintStream(bos);
        System.setOut(bout);
        HelloWorld.main(null);
        String[] outs = bos.toString().split(System.lineSeparator());
        //System.out.println(outs.length);
        assertEquals("Hello World!", outs[outs.length-1]);
    }

}
```

```
JUnit version 4.3
OK (1 test)
```

Figure 4. CWP answer interface.

4. Answer Code Validation Program 186

In this section, we present the implementation of the *answer code validation program* for the *code writing problem* in *JPLAS*. This function is implemented by modifying the code testing program in the answer platform so that it can test all the source codes in the folder automatically, instead of testing only one source code. Thus, it assumes that each folder contains the source codes for the same test code. 187 188 189 190 191

4.1. Folder Structure in File System 192

Figure 5 shows the folder structure in the file system for the *answer code validation program*. The "test" folder inside the "addon" folder is used to keep the test code file and all the source code files from the students for each CWP assignment. The "codevalidator" folder contains the required Java programs for testing the source codes and reporting the results. The other jar files and folders are used to execute the code testing program. The text files of the testing results including the JUnit logs will be recorded in the "output" folder. From them, this program will produce the CSV file in the "csv" folder, so that the teacher can easily check the results of all the students in the same file. 193 194 195 196 197 198 199

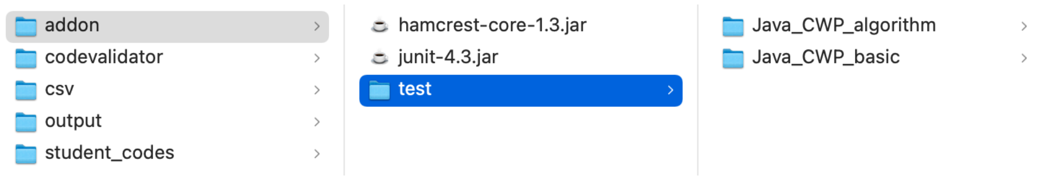


Figure 5. Designated file system for answer code validation program.

4.2. Answer Code Validation Procedure 200

The following procedure describes the answer code validation that will check the correctness of all the source codes from the students. 201 202

1. The folder containing the source codes for each assignment using one test code is downloaded as a zip file. It is noted that a teacher usually uses an e-learning system such as *Moodle* in his/her programming course. 203 204 205
2. The zip file is unzipped and stored in the corresponding folder under the "student_codes" folder inside the this project path. 206 207
3. The corresponding test code is stored in "addon/test" folder. 208
4. The program reads the test code. 209

5. The program reads each source code in the “student_codes” folder, runs the test code with the source code on *JUnit*, and saves the test result in the text file in the “output” folder. This process is repeated until all the source codes in the folder are tested.
6. The program makes the summary of the test results for all the source codes by the CSV file and saves it in the “csv” folder.

4.3. Example of Answer Code Validation

Figure 6 shows the example of the related files with the folder structure. Before running the program, the teacher needs to store the source code (“helloWorld.java” in this example) of each student in the assignment folder (“Java_CWP_basic”) inside the student folder (“student1”) for each assignment. It is noted that in the program, the folder structure for the source codes can be customized by preferences. For example, from *Moodle*, the source code file for each student is directly stored in the assignment-student folder.

Then, the program runs the test code (“helloWorldTest.java) with every source code sequentially, and writes the test output into the corresponding file (“student1_Java_CWP_basic_output.txt”) in the “output” folder. After testing all the source codes in the assignment folder, the program writes all the test outputs in the CSV file (“student1_Java_CWP_basic.csv”) in the “csv” folder.

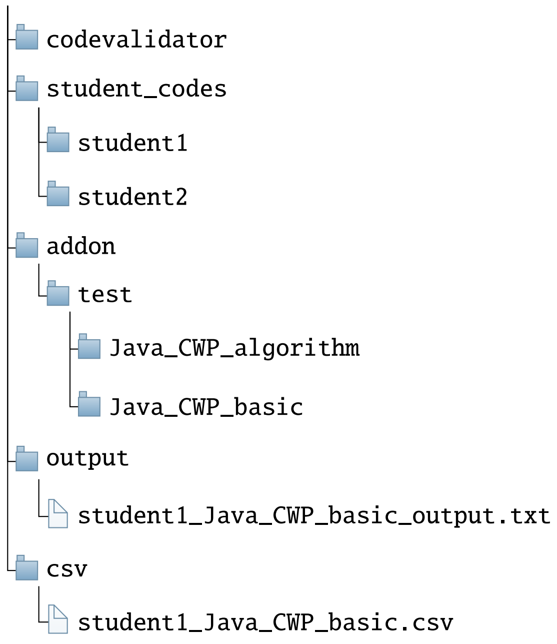


Figure 6. File examples with folder structure.

5. Test Data Generation Algorithm and Intermediate State Testing

In this section, we analyze the limitations of the current test code, and present the test data generation algorithm and the intermediate state testing in the test code to solve them.

5.1. Limitation of Current Test Code

First, we discuss the limitations of current test code for the above example *test code* in Figure 2.

5.1.1. Fixed Data Output

The fixed test data in the test code can lead the issue of cheating, where a student may rely on the limited set of test cases to write the source code without truly understanding the concepts. The following *source code* in Figure 7 shows one of the examples of the limitation of the fixed output data for the above *test code* in Figure 1.

```
package CWP;
public class BubbleSort {
    public static int[] sort(int[] a) {
        int[] res ={0,1,3,4,5,7};
        return res;
    }
}
```

Figure 7. Example source code for *fixed data ouptput*.

In this example, the source code directly returns the output without implementing any logic or algorithm as the test case has the fixed output data. Therefore, the test data generation algorithm should be implemented in order to dynamically change the test data and replace them in the test code. This algorithm will analyze and identify the type of data in the test codes, and will generate the new data to replace the fixed test data. This algorithm can reduce the risk of cheating and can improve the validity and reliability of the programming assignments.

5.1.2. Library Use

Another limitation of the current test code lies in use of a library for implementing the logic or algorithm. A student may use the library without implementing the correct logic/algorithm. For instance, a scenario is considered where it is required to implement a sorting algorithm. Instead of implementing the algorithm from scratch, the student may rely on a library that provides a pre-built sorting function. The student does not understand the fundamentals of the logic/algorithm itself. The following *source code* in Figure 8 shows this example for using a library without implementing the algorithm. The current test code cannot check it.

```
package CWP;
import java.util.Arrays;
public class BubbleSort {
    public static int[] sort(int[] a) {
        Arrays.sort(a);
        return a;
    }
}
```

Figure 8. Example source code for using *library*.

5.1.3. Implementation of Different Logic or Algorithm

The current test code cannot detect the implementation of the different logic or algorithm from the requested one. The following *source code* in Figure 9 shows the example for implementing a different simple sorting algorithm. According to this example, the students may implement selection sort or other simple sorting algorithms instead of bubble sort as the final output sorted result of almost all the sorting algorithms is the same. To find this error, the intermediate state of the important variables, such as the data to be sorted, should be checked, in addition to the final state.

```
package CWP;
public class BubbleSort {
    public static int[] sort(int[] a) {
        int n = a.length;
        for (int i = 0; i < n - 1; i++) {
            int minIndex = i;
            for (int j = i + 1; j < n; j++) {
                if (a[j] < a[minIndex]) {
                    minIndex = j;
                }
            }
            int temp = a[i];
            a[i] = a[minIndex];
            a[minIndex] = temp;
        }
        return a;
}
```

Figure 9. Example source code for implementing of *different algorithm*.

To address the limitations, we propose the test data generation algorithm and the intermediate state testing in the test code in the following subsections. 258 259

5.2. Test Data Generation Algorithm 260

The test data generation algorithm automatically generates and replaces the input data and the expected output data for each test case in the given test code. To achieve this goal, we adopt the standard format to describe them in the test code. Figure 10 shows the test code with the standard format for testing the *sort* method in the *BubbleSort* class in Figure 1. In this standard format, for each test case, the input data to the method under testing is given by `codeInput1`, the output data from the method under testing is by `codeOutput`, and the expected output data is by `expOutput`. 261 262 263 264 265 266

```
package CWP;
import static org.junit.Assert.*;
import org.junit.Test;
import java.util.Arrays;
public class BubbleSortTest {
    @Test
    public void testSort() {
        BubbleSort bubbleSort = new BubbleSort();
        int[] codeInput1 = {7,5,0,4,1,3};
        int[] codeOutput = bubbleSort.sort(codeInput1);
        int[] expOutput = {0,1,3,4,5,7};
        try {
            assertEquals("1:One input case:",Arrays.toString(expOutput),
                Arrays.toString(codeOutput));
        } catch (AssertionError ae) {
            System.out.println(ae.getMessage());
        }
    }
```

Figure 10. Example of test code with standard format

The procedure of the test data generation algorithm is described as follows: 267

1. Read the *input data* from the test code with the standard format. 268
2. Detect the input data by `codeInput1` and find the *data type*. 269
3. Generates the new input data according to the following procedure: 270
    - For *int*, an integer number between 2 and 10 is randomly selected. 271

- For *double* and *float*, a real number between 2.0 and 10.0 is randomly selected.
- For *int[]*, the array size between 5 and 10 is randomly selected at first and an integer number between -99 and 99 is randomly selected.
- For *double[]* and *float[]*, the array size between 5 and 10 is randomly selected at first and a real number between -99 and 99 is randomly selected.
- For the *String* and *String[]*, an English full name is randomly selected by using *names* library. The array size between 5 and 10 is randomly selected for *String[]*. The other data type will be considered in our future works.

4. Replace the input data for `codeInput1` in the test code by the newly generated input data.
5. Run the newly generated *test code* with the correct *source code* on *JUnit*, where the correct *source code* needs to be prepared for each assignment.
6. Find the expected output data from the *JUnit* log.
7. Replace the expected output data for `expOutput` in the test code by this expected output data.

5.3. Intermediate State Testing for Logic and Algorithms

The intermediate state testing checks the randomly selected intermediate state of the important variables during the logic/algorithm execution. Figure 12 shows the test code to check the values of the variables for the sorted data after two iterations are over, in addition to checking the final values. In the test code, the second input data `codeInput2` represents the number of iteration steps to be tested. To pass this test code, a student needs to additionally implement the `sort(int[] a, int iteration)` method by overloading the original `sort` method in the source code. Figure 11 shows the *source code* to pass the test code in Figure 12. A student can easily implement the method for intermediate testing from the original method, where only the `for` loop termination condition needs to be modified. In addition, a student can practice the use of overloading.

```
package CWP;
public class BubbleSort {
    // intermediate state
    public static int[] sort(int[] a, int iteration) {
        int i, j, temp;
        for (i = 0; i < iteration; i++) {
            for (j = 0; j < a.length - 1; j++) {
                if (a[j] > a[j + 1]) {
                    temp = a[j + 1];
                    a[j + 1] = a[j];
                    a[j] = temp;
                }
            }
        }
        return a;

    //final state
    public static int[] sort(int[] a) {
        int temp = 0;
        for(int i=0; i < a.length; i++){
            for(int j=1; j < (a.length - i); j++){
                if(a[j-1] > a[j]){
                    temp = a[j-1];
                    a[j-1] = a[j];
                    a[j] = temp;
                }
            }
        }
        return a;
    }
}
```

Figure 11. Example source code for intermediate state testing.

```java
package CWP;
import static org.junit.Assert.*;
import org.junit.Test;
import java.util.Arrays;
public class BubbleSortTest {
    //intermediate state testing
    @Test
    public void testSortIteration() {
        BubbleSort bubbleSort = new BubbleSort();
        int[] codeInput1 = {5,2,8,1,9};
        int codeInput2 = 2;
        int[] codeOutput = bubbleSort.sort(codeInput1, codeInput2);
        int[] expOutput = {2,1,5,8,9};
        try {
            assertEquals("2:Two input case:",Arrays.toString(expOutput),
                Arrays.toString(codeOutput));
        } catch (AssertionError ae) {
            System.out.println(ae);
        }
    }
    //final state testing
    @Test
    public void testSort() {
        BubbleSort bubbleSort = new BubbleSort();
        int[] codeInput1 = {7,5,0,4,1,3};
        int[] codeOutput = bubbleSort.sort(codeInput1);
        int[] expOutput = {0,1,3,4,5,7};
        try {
            assertEquals("1:One input case:",Arrays.toString(expOutput),
                Arrays.toString(codeOutput));
        } catch (AssertionError ae) {
            System.out.println(ae.getMessage());
        }
    }
}
```

Figure 12. Example test code for intermediate state testing.

6. Evaluation 295

In this section, we evaluate the proposal by applying it to 260 source codes from 28 students to 15 CWP assignments in the Java programming course in Okayama University. 296 297

6.1. CWP Assignments 298

In this course, two groups of the CWP assignments for studying *basic grammar* and *fundamental algorithms* were prepared with the corresponding test codes. These groups are considered to be at different levels of Java programming study. Table 1 shows the group topic, and the program name and the number of test cases in the test code for each CWP assignment. The 15 CWP assignments were assigned to 12-28 students who were taking the Java programming course in Okayama University, and the answer source codes from students were tested by the *answer code validation program*. Then, we discuss the solution performances of the students from the output of the program. 299 300 301 302 303 304 305

6.2. Results of Individual Assignments 306

First, we examine the solution results of the individual CWP assignments. Table 2 shows the results, including the group topic, the assignment ID, the program name, the average pass rate by the original test code, and the average pass rate by the proposed test code. The average pass rate is given by dividing the number of passed test cases by the total number of test cases in each test code. 307 308 309 310

By comparing the two average pass rates for the "fundamental algorithms" group, the effectiveness of the proposed test code is confirmed. The average pass rate by the proposal becomes lower 311 312

Table 1. CWP assignments for evaluations.

group topic	ID	program name	# of test cases
basic grammar	1	CodeCorrection1	3
	2	CodeCorrection2	3
	3	MaxItem	6
	4	MinItem	7
	5	ReturnAndBreak	3
fundamental algorithms	6	BinarySearch	9
	7	BinSort	5
	8	BubbleSort	4
	9	Divide	4
	10	GCD	5
	11	HeapSort	4
	12	InsertionSort	4
	13	LCM	5
	14	QuickSort	5
	15	ShellSort	4

in the four assignments, where either the use of the library method or the different algorithm was observed. For the assignment with ID=6, the library method “Arrays.binarySearch()” was used. For the assignments with ID=7, 8, 11, the library method “Arrays.sort()” was used.

Table 2. Results of individual assignments.

group topic	ID	program name	ave. pass rate by original (%)	ave. pass rate by proposal (%)
basic grammar	1	CodeCorrection1	96.43	96.43
	2	CodeCorrection2	96.43	96.43
	3	MaxItem	92.86	92.86
	4	MinItem	96.43	96.43
	5	ReturnAndBreak	96.43	96.43
fundamental algorithms	6	BinarySearch	100	99.07
	7	BinSort	81.67	80
	8	BubbleSort	91.67	89.58
	9	Divide	91.67	91.67
	10	GCD	91.67	91.67
	11	HeapSort	75	72.92
	12	InsertionSort	83.33	83.33
	13	LCM	91.67	91.67
	14	QuickSort	66.67	66.67
	15	ShellSort	75	75

6.3. Results of Individual Students

Next, we analyze the solution results of the individual students for the 15 CWP assignments. Table 3 presents the results of individual students’ solutions. They are separated by the topic groups, because all of the 28 students answered to the “basic grammar” assignments while only 12 students among them answered to the assignments. Unfortunately, it seems that many students did not take the course on “fundamental algorithms” that was educated at one year before. Therefore, at the beginning of this Java programming course, it will be necessary to encourage students to study “fundamental algorithms” by themselves if they did not take the course, because the algorithm programming is very important for them.

In the “basic grammar” group, all the students except two students passed all the test cases in any test code. This indicates that the students well understand the concepts. One student did not pass the test cases for one assignment and another one student did not try to finish the assignments.

In the "fundamental algorithms" group, the average pass rate is often lower than that in the "basic grammar" group. The average correct rate by the proposal is lower than the rate by the original in three students who used the library methods. The algorithm programming seems more difficult for the students.

Table 3. Results of individual students.

basic grammar			fundamental algorithms		
student ID	ave. pass rate by original (%)	ave. pass rate by proposal (%)	student ID	ave. pass rate by original (%)	ave. pass rate by proposal (%)
1	100	100	1	100	100
2	100	100	2	100	100
3	100	100	3	100	100
4	100	100	4	66.67	62.50
5	100	100	5	100	100
6	100	100	6	72.92	72.92
7	100	100	7	100	100
8	100	100	8	100	100
9	100	100	9	83.33	81.25
10	0	0	10	18.75	18.75
11	100	100	11	97.92	95.83
12	100	100	12	100	100
13	72.73	72.73			
14	100	100			
15	100	100			
16	100	100			
17	100	100			
18	100	100			
19	100	100			
20	100	100			
21	100	100			
22	100	100			
23	100	100			
24	100	100			
25	100	100			
26	100	100			
27	100	100			
28	100	100			

6.4. Teacher Load Reduction

We evaluate the teacher load reduction by using the proposal to test the source codes for the assignments in Table 1. Without the proposal, a teacher needs to repeat to 1) open the source code on an IDE such as VS-Code, 2) run the test code on *Junit*, and 3) copy the result to a text file for each of 260 source codes. On the other hand, with the proposal, a teacher only needs to run the program. The CPU time was 2.43*min* for *basic grammar* and 1.95*min* for *fundamental algorithms*. Thus, the proposal can greatly reduce the teacher load.

7. Conclusion

This paper presented the *answer code validation program* that will automatically verify all the source codes from a lot of students for each *code writing problem (CWP)* assignment in *Java programming learning assistant system (JPLAS)* and report the number of passing tests in the CSV file to help the teacher. To address the limitations in the current test code, we also presented the *test data generation algorithm* and the *intermediate state testing*. For them, we defined the standard format of writing test cases in the test code. For evaluations, the proposal was applied to 260 source codes from 28 undergraduate in the Java programming course in Okayama university, Japan, where the validity and effectiveness was confirmed. In future works, we will study test codes with the proposal for other logics or algorithms in mathematics, physics, and engineering topics, generate new assignments for other Java grammar topics, and apply them to students in Java programming courses.

References

1. Node.js. Available online: https://nodejs.org/en
2. Aung, S. T.; Funabiki, N.; Aung, L.H.; Htet, H.; Kyaw, H. H. S.; Sugawara, S. An implementation of Java programming learning assistant system platform using Node.js. In Proceedings of the International Conference on Information and Education Technology, Matsue, Japan, 9-11 April 2022; pp. 47-52.
3. Aung, S. T.; Funabiki, N; Syaifudin, Y. W.; Kyaw, H. H. S.; Aung, S. L.; Dim, N. K.; Kao, W.-C. A proposal of grammar-concept understanding problem in Java programming learning assistant system. *J. Adv. Inform. Tech.* **2021**, *12(4)*, 342-350.
4. Zaw, K. K.; Funabiki, N; Kao, W.-C. A proposal of value trace problem for algorithm code reading in Java programming learning assistant system. *Inf. Eng. Express.* **2015**, *1(3)*, 9-18.
5. Funabiki, N; Tana; Zaw, K. K.; Ishihara, N.; Kao, W.-C. A graph-based blank element selection algorithm for fill-in-blank problems in Java programming learning assistant system. *IAENG Int. J. Comput. Sci.* **2017**, *44(2)*, 247-260.
6. Kyaw, H. H. S.; Wint, S. S.; Funabiki, N; Kao, W.-C. A code completion problem in Java programming learning assistant system. *IAENG Int. J. Comput. Sci.* **2020**, *47(3)*, 350-359.
7. Funabiki, N; Matsushima, Y.; Nakanishi, T.; Amano, N. A Java programming learning assistant system using test-driven development method. *IAENG Int. J. Comput. Sci.* **2013**, *40(1)*, 38-46.
8. JUnit. Available online: http://www.junit.org/
9. Ala-Mutka, K.; Problems in Learning and Teaching Programming. A literature study for developing visualizations in the Codewitz-Minerva project. 2004; pp. 1-13.
10. Konecki, M.; Problems in programming education and means of their improvement. DAAAM Int. Sci. Book, 2014; pp. 459-470.
11. Queiros, R. A.; Peixoto, L.; Paulo, J. PETCHA - a programming exercises teaching assistant, In Proceedings of ACM annual conference on Innovation and technology in computer science education, Haifa, Israel, 3-5 July 2012; pp. 192-197.
12. Carbone, A.; Mitchell, I.; Hurst, J.; Gunstone, D. An exploration of internal factors influencing student learning of programming. In Proceedings of Australasian Computing Education Conference, Wellington, New Zealand, 2009; pp. 25-34.
13. Ihantola, P.; Ahoniemi, T.; Karavirta, V.; Seppälä, O. Review of recent systems for automatic assessment of programming assignments. In Proceedings of Koli Calling, 2010; pp. 86–93.
14. Denny, P.; Luxton-Reilly, A.; Tempero, E.; Hendrickx, J. CodeWrite: supporting student-driven practice of Java. In Proceedings of ACM Technical Symposium on Computer Science Education, Dallas, USA, 9-12 March, 2011; pp. 471-476.
15. Li, F. W.-B.; Watson, C.; Game-based concept visualization for learning programming. In Proceedings of ACM workshop on Multimedia technologies for distance learning, Scottsdale Arizona, USA, 1 December 2011; pp. 37-42.
16. Shamsi, F. A.; Elnagar, A.; An intelligent assessment tool for student's Java submission in introductory programming courses. *J. Intelli. Learn. Syst. Appl.* **2012**, *4*, 59-69.
17. Mei, H.; Hao, D.; Zhang, L.; Zhou, J.; Rothermel, G.; A static approach to prioritizing JUnit test cases. *IEEE Trans. Soft. Eng.* **2012**, *38(6)*, 1258-1275.
18. Kitaya, H.; Inoue, U.; An online automated scoring system for Java programming assignments. *Int. J. Infrom. Edu. Tech.* **2016**, *6(4)*, 275-279.
19. Ünal, E.; Çakir, H.; Students' views about the problem based collaborative learning environment supported by dynamic web technologies. *Malaysian Online J. Edu. Tech.* **2017**, *5(2)*, 1-19.
20. Szab, M.; Nehz, K. Grading Java code submissions in MeMOOC. In Proceedings of International Multidisciplinary Scientific Conference, 5-6 September 2018.
21. Zinovieva, I. S.; Artemchuk, V. O.; Iatsyshyn, A. V.; Popov, O. O.; Kovach, V. O.; Iatsyshyn, A. V.; Romanenko, Y. O.; Radchenko, O. V. The use of online coding platforms as additional distance tools in programming education. *J. Phys.: Conf. Ser.* **2021**, *1840*.
22. Bubble Sort. Available online: https://www.javatpoint.com/bubble-sort-in-java