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

Khaing Hsu Wai, Nobuo Funabiki, Soe Thandar Aung, Xiqin Lu, Yanhui Jing, Htoo Htoo Sandi Kyaw, Wen-Chung Kao

***Abstract*—In order to support the learning of novice students in *Java programming*, the web-based *Java Programming Learning Assistant System (JPLAS)* has been developed. JPLAS offers several types of exercise problems to foster *code reading and writing* skills at different 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's workload during the marking process, we present the *answer code validation program* that verifies all the source codes from a large number of students for each assignment 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 new data, and replaces it for each test data in the test code. Furthermore, to verify the correctness of the implemented procedures 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.**

***Index Terms*—programming learning, Java, JUnit, code writing problem, code validation, test data generation**

I. Introduction

FOR many years, *Java* has been widely employed in various industries as a dependable and adaptable object-oriented programming language [1]. It's utilization has involved critical systems within large enterprises as well as smaller embedded systems. The demand for skilled Java programmers remains high among IT companies, leading to a growing number of academic institutions and professional institutions, which are providing *Java programming* courses to fulfill this need.

To support self-studies of novice students in Java programming, *Java Programming Learning Assistant System (JPLAS)* has been developed. The personal answer platform on *Node.js* [2], which will be distributed to students on *Docker* [3], has been implemented [4]. *JPLAS* provides several types of exercise problems with automatic marking functions that have different learning goals. It is expected that the exercise problems in *JPLAS* will gradually progress the learning stages of students. JPLAS can cover self-studies of Java programming at different levels by novice students.

In the process of studying programming, novice students should start by solving uncomplicated and concise exercise problems that focus on *code reading* studies, enabling them to comprehend and grasp the programming language's grammar and concepts. After they have acquired basic knowledge and skills from *code reading* studies, they should move to *code writing* studies. It is crucial for students to develop the abilities of reading source codes effectively as they directly impact their proficiency in writing source codes correctly.

To support the novice students' progressive programming study, JPLAS provides the following types of exercise problems. By solving these problems in this order, it is expected for the students to gradually advance their programming levels by themselves.

1) Grammar-concept Understanding Problem (GUP) gives questions about the concepts of important keywords, including reserved words and commonly used libraries in the programming language, in the provided source code. It focuses on keywords and libraries in the source code [5].
2) Value Trace Problem (VTP) requires analyzing codes to determine the output values and output messages of the variables in the given source code [6].
3) Mistake Correction Problem (MCP) requests to correct the mistaken element in the source code. It is for the study of code debugging [7].
4) Element Fill-in-blank Problem (EFP) requests to complete the missing elements in the given source code in order to gain the original source code [8].
5) Code Completion Problem (CCP) involves correcting errors and filling in missing elements in the provided source code in order to debug and complete the original source code [9].
6) Phrase Fill-in-blank Problem (PFP) requests to fill in each blank by the original set of elements or the message in the source code [10].

Manuscript received August 29, 2023; revised March 9, 2024.

K. H. Wai is a PhD candidate of Department of Information and Communication Systems, Okayama University, Okayama, Japan (e-mail: khainghsuwai@s.okayama-u.ac.jp).

N. Funabiki is a professor of Department of Information and Communication Systems, Okayama University, Okayama, Japan (e-mail: funabiki@okayama-u.ac.jp).

S. T. Aung is a PhD candidate of Department of Information and Communication Systems, Okayama University, Okayama, Japan (e-mail: soethandar@s.okayama-u.ac.jp).

X. Lu is a PhD candidate of Department of Information and Communication Systems, Okayama University, Okayama, Japan (e-mail: pch55zhl@s.okayama-u.ac.jp).

Y. Jing is a postgraduate student of Department of Information and Communication Systems, Okayama University, Okayama, Japan (e mail: pf709l29@s.okayama-u.ac.jp).

H. H. S. Kyaw is an assistant professor of Department of Computer and Information Science, Tokyo University of Agriculture and Technology, Tokyo, Japan (e-mail: htoohtoosk@go.tuat.ac.jp).

W.-C. Kao is a professor of Department of Electrical Engineering, National Taiwan Normal University, Taipei, Taiwan (e-mail: jungkao@ntun.edu.tw).

7) Code Writing Problem (CWP) requests to write a source code that can pass the given *test code* [11].

By following the course curriculum in Okayama university, the previous paper in [12] reported the course implementation of GUP, VTP, MCP, EFP, CCP, and PFP in *JPLAS* for *code reading* and *partial code writing* studies. The correctness of the answer in these problems is checked through *string matching*. In *JPLAS*, students should solve the *code reading* related problems first to understand the definitions of the keywords and control flows in source codes. Then, they should solve the *code writing* related problems to allow writing full source codes by themselves.

Another previous paper in [13] reported the same course implementation of CWP and the application results of the answer code validation program to the source codes submitted by the students in the course. The correctness of the answer in CWP is checked automatically through *unit testing* [14] on *JUnit* [15].

This paper specifically focuses on the *code writing problem (CWP)* of *JPLAS*. The *CWP* 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 [16]. In *JUnit*, one test can be performed by using one method in the library whose name starts with "assert". This paper adopts the "assertEquals" method to compare the execution result of the source code with its expected value.

Therefore, *test codes* and *source codes* are fundamental components in the code writing problem of JPLAS. Students write the source code with the goal of making it pass all the test cases defined in the test codes. The test cases act as a reference or specification for the expected behavior of the source code. By running the test codes against their source code, students can quickly evaluate the correctness and functionality of their implementation. If the source code passes all the test cases, it is an indication that the code is likely correct and performs as expected.

Among the exercise problems in JPLAS, the *CWP* is crucial for students as it enables them to practice and attain proficiency in writing source codes from scratch. For *CWP*, the *answer platform* has been implemented for students to solve the given CWP assignments efficiently [4]. By implementing the capability of automatically running *unit testing*, 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 run and excute the test codes and the source codes one by one manually at the marking process, although a lot of source codes are usually submitted from students. This load is very 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, we implement the *answer code validation program* to solve the first limitation [13]. This program automatically checks the source codes submitted by students for every assignment and generates a CSV file that provides a count of passing tests for each source code. By reviewing the test result summary for all students, the teacher can quickly assess the correctness of student answers and assign grades according to that test result summary.

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.

In summary, the proposal in this paper will reduce the teachers' workload of marking source codes submitted from the students, and enhance the correctness of marking results.

The rest of this paper is organized as follows: Section II provides an overview of related works in literature. Section III reviews our previous works of *CWP* and the answer platform. Section IV presents the implementation details of the *answer code validation program*. Section V presents the test data generation algorithm and the intermediate state testing. Section VI evaluates the proposal. Finally, Section VII concludes this paper with future works.

II. Literature Review

We would like to discuss literature review in this section.

In [17] and [18], Ala-Mutka et al. and Konecki made contributions by highlighting common challenges faced by novice programmers and discussing existing approaches and discussions on programming teaching. Numerous tools were proposed to assist students to solve programming difficulties. Among them, *ToolPetcha*, which was proposed by Queiros et al., is the tool that serves as a automated programming assistance [19].

In [20], Carbone et al. investigated various factors that can contribute to students discontinuing an basic programming course, encompassing aspects such as motivations and critical thinking abilities.

In [21], Piteira et al. investigated the challenges and difficulties encountered by learners in the process of learning computer programming. This study focused on understanding specific areas or concepts that students find challenging when acquiring programming skills. Through the analysis of the difficulties faced by the learners, the authors provided insights and recommendations for improving programming educations.

In [22], Li et al. presented a learning environment based on games, aimed at assisting novice students in programming educations. The environment employs game creation tasks to simplify the comprehension of fundamental programming

concepts. Additionally, it incorporates visualization techniques that enable students to interact with game objects, facilitating their understanding of crucial programming principles.

In [23], Nguyen et al. discussed the development of an intelligent chatbot designed for educational purposes, particularly in programming courses. It introduces the *Integ-Rela* model, a method for integrating multiple knowledge domains to form a comprehensive knowledge base. This model allows the chatbot to effectively retrieve and provide relevant information across various subjects. The chatbot acts as a virtual tutor, aiding students in learning programming concepts. The effectiveness of this system is demonstrated through experiments, showing its potential as a practical tool in e-learning environments.

In [24], Matsumoto et al. examined the impact of a puzzle-like programming game called, *Algologic*, on learning programming. It focused on the achievement degree of students after learning of programming and reported the analysis result. This study finds a positive relationship between students' performance in the game and their success in learning programming.

In [25], Okonkwo et al. focused on a chatbot called *RevBot*, which was developed to help students practice past exam questions in Python programming. Using the *Snatchbot Chatbot API*, *RevBot* is designed to interact with students, providing questions and answers for revisions. This study included the evaluation of its usefulness, indicating that it can enhance students' performance in Python programming. The paper highlights the potential of *RevBot* as a useful tool in educational contexts, especially for introductory programming courses.

In [26], Ihantola et al. conducted a comprehensive review of recent developments in automated assessment tools designed for programming exercises. The authors examined the significant characteristics and methodologies, such as programming languages, learning in management systems, resubmission restrictions, testing tools, manual assessments, security concerns, distribution mechanisms, and specialized features.

In [27], Staubitz et al. addressed the challenge of providing practical programming exercises and automated assessment in *Massive Open Online Courses (MOOCs)*. They focused on the development of an approach that combines hands-on programming exercises with automated assessment tools to enhance the learning experience in online programming courses.

In [28], Denny et al. introduced the assessment on a web-based tool named *CodeWrite*, which was designed to facilitate drill and practice for Java programming. The tool relies on students to create exercises that can be shared within their class. However, it should be noted that the absence of a testing tool like *JUnit* limits the range of possible variations for program testing.

In [29], Shamsi et al. introduced a grading system named *eGrader*, specifically designed for introductory Java programming courses. The system utilizes a graph-based approach for grading, where dynamic analysis of submitted programs is performed using *JUnit*, and static analysis is conducted based on the program's graph representation. The accuracy of the system was validated through experimental evaluations.

In [30], Mei et al. presented a test case prioritization technique called *JUPTA* that utilizes estimated coverage information obtained from static call graph analysis of test cases in JUnit. The authors demonstrated that test suites prioritized by *JUPTA* exhibited greater effectiveness in terms of fault detection compared to random and untreated test orderings.

In [31], Edwards et al. explored the use of *test-driven development (TDD)* in the classroom. *TDD* is shown to provide students with automatic and concrete feedback on their performance, leading to improved learning outcomes. The paper highlights the benefits of *TDD* in enhancing student engagement, comprehension, and problem-solving skills in computer science education.

In [32], Edwards et al. provided their insights of using *test-driven development (TDD)* with an automated grader. This paper shared the advantages and challenges encountered while implementing *TDD* in the context of computer science education and the evaluation of effectiveness using that automated grading system. Through discussion, the authors highlighted on how *TDD* with automated grader enhances the student learning and gives the valuable comments on their programming assignments.

In [33], Desai et al. presented a survey of evidence regarding the use of *test-driven development (TDD)* in academia. The authors explore existing literature and studies related to *TDD* implementations in educational settings. The paper examines the benefits, challenges, and outcomes associated with *TDD* adoptions in computer science educations. By analyzing the evidence, the authors provide insights into the impact of *TDD* on the student learning, skill development, and overall educational effectiveness.

In [34], Elgendy et al. presented a method using *Genetic Algorithms (GAs)* for automatically generating test data for *ASP.NET* web applications. It introduces new genetic operators designed for the unique structure of web applications, aiming to improve the efficiency and coverage of test data generations. The tool developed in this study uses static analysis to identify crucial data-flow elements in applications, and then, applies GAs to generate test cases that effectively cover these elements. The paper demonstrates the tool's effectiveness through case studies and empirical evaluation, highlighting its utility in enhancing the reliability of *ASP.NET* web applications.

In [35], Kitaya et al. a web-based automatic scoring system designed for Java programming assignments. The system accepts a student's Java application program as input and promptly provides a test result, including a compiler check, *JUnit* testing, and result evaluation. This system shares similarities with the code writing problem in *JPLAS*, where the result test can be incorporated into the existing *JUnit* test.

In [36], Ünal et al. conducted a qualitative study in order to explore students' perceptions of a collaborative learning environment. They developed a learning environment utilizing technology for dynamic web-powered problems. A semi-structured interview method was employed to gather students' perceptions on the learning environment, which facilitated collaborative problem-solving using dynamic web technologies. The findings suggested that incorporating collaborative learning techniques centered around problem-

solving, and leveraging dynamic web technologies can be beneficial for the learning environment in a community college setting.

In [37], Tung et al. discussed the implementation of *Programming Learning Web (PLWeb)*. It can provide an *integrated development environment (IDE)* for teachers to create exercises and a user-friendly editor for students to submit solutions. Features like visualized learning status and a plagiarism detection tool are added in the system to assist the learning and teaching process.

In [38], Szab et al. introduced a Java code grading feature named *MeMOOC*, which automatically evaluates syntactical, semantical, and pragmatic aspects of code. The grading process involves compilation checks, *Checkstyle* analysis, and *JUnit* testing, specifically designed for *MOOC*.

In [39], Zinovieva et al. conducted a comparative assessment of multiple online platforms used for programming education. They specifically selected engaging assignments from *hackerrank.com*, an educational site for students. The study examined user experiences with *online coding platforms (OCP)* and compared the features of different platforms that could be employed for teaching programming to computer science and programming enthusiasts through distance learning. Furthermore, the authors suggested incorporating online programming simulators to enhance computer science instruction, considering the functionality of the simulators as well as students' preparedness levels and expected learning outcomes.

While the previous studies made valuable contributions to programming education, our *Java Programming Learning Assistant System (JPLAS)*, built upon these foundations, introduces novel features and advancements. *JPLAS* offers rich exercise problems for both *code reading* and *code writing* studies, which are essential for understanding and practicing programming concepts and syntax. *JPLAS* incorporates the answer platform built on *Node.js* and *Docker* [3], which allows students to submit their code solutions and receive the automated feedback [4]. This platform includes an automatic marking function that verifies the correctness of student answers using *unit testing* with *JUnit* [15] to validate the functionality of the answer source code. With it, the students can easily check the correctness of their source codes by clicking the corresponding button on the answer platform.

III. Overview of Code Writing Problem

In this section, we discussed an overview of *code writing problem (CWP)* and the answer platform using *Node.js* in JPLAS.

A. Code Writing Problem

The *Code Writing Problem (CWP)* assignment is comprised a statement along with a *test code*, which is given by the teacher. In *CWP*, students are tasked with writing source code that satisfies predefined test cases contained in the test code. Code testing is employed to validate the correctness and accuracy of the students' source code, utilizing *JUnit* to execute the test code alongside with the source code. To ensure the accurate implementation of the source code, the students should follow the detailed specifications provided in the test code.

To generate a new assignment for *CWP*, the teacher needs to perform the following operations.

1) Create the problem statement and prepare the input data for the assignment,
2) Collect the correct answer source code as a model source code for the assignment,
3) Execute the model source code to obtain the expected output data,
4) Prepare the *test code* from the input and output data to make test cases, and describe the required information for implementing the source code, and
5) Register the test code and the problem statement for the new assignment.

B. JUnit

In order to facilitate *code testing*, an open-source Java framework *JUnit* is utilized, aligning with the *test-driven development (TDD)* approach. *JUnit* can help the automatic unit test of a source code or class. Java programmers can use it quite easily because it has been designed with the user-friendly style for Java. With *JUnit*, performing a test is simplified through the usage method in the library whose name starts with "assert". In the case of *CWP*, the test code adopts the "assertEquals" method that compares the output generated by executing the source code with the expected output data for a given set of input data.

C. Test Code

A *test code* is created by using the *JUnit* library. The *BubbleSort* class in Figure 1 [41] 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[] array)" method performs the basic bubble sort algorithm on the input array "array" and returns the sorted array.

```
package p3;
public class BubbleSort {
    public int[] sort(int[] array) {
        int n = array.length;
        for (int i = 0; i < n - 1; i++) {
            for (int j = 0; j < n - i - 1; j
                ++) {
                if (array[j] > array[j + 1])
                    {
                    int temp = array[j];
                    array[j] = array[j + 1];
                    array[j + 1] = temp;
                }
            }
        }
    return array;
    }
}
```

Fig. 1: Example source code for *BubbleSort*.

The *test code* in Figure 2 is designed for testing the *sort* method in the *BubbleSort* class in Figure 1.

The *test code* includes import statements for the *JUnit* packages, which contain the necessary test methods, at lines 2 and 3. At line 5, it also declares the *BubbleSortTest* class.

```
package p3;
import org.junit.Test;
import static org.junit.Assert.*;
import java.util.Arrays;
public class BubbleSortTest {
    @Test
    public void testSort() {
        BubbleSort bSort = new BubbleSort();
        int[] codeInput1 = {8,7,4,1,5,9};
        int[] codeOutput = bSort.sort(
            codeInput1);
        int[] expOutput = {1,4,5,7,8,9};
        try {
            assertEquals("Test1:",Arrays.
                toString(expOutput),Arrays.
                toString(codeOutput));
        } catch (AssertionError ae) {
            System.out.println(ae.getMessage
                ());
        }
    }
}
```

Fig. 2: Example test code for BubbleSort.

The test code contains test methods, annotated with "@Test" in line=6, showing that they are test cases that JUnit, a testing framework, will run to check the output of the *sort* method. This test is performed as follows:

1) Generate the *bSort* object of the *BubbleSort* class in the source code.
2) Call the *sort* method of the *bSort* object with the arguments for the input data.
3) Compare the result of the *sort* method at *codeOutput* with the *expOutput* data using the *assertEquals* method.

D. Answer Platform of CWP

In order to support students solving the CWP assignments, we have developed an answer platform as a web application system using *Node.js*. The software architecture, illustrated in Figure 3, operates on *Linux* or *Windows* operating systems and follows the MVC (Model-View-Controller) model.

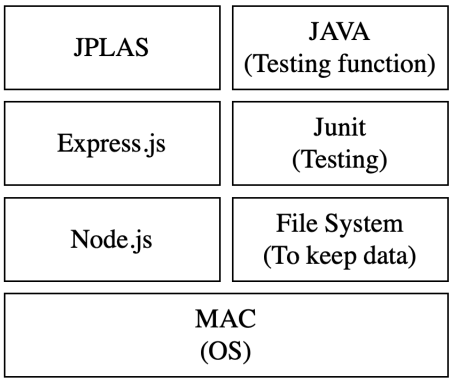

Fig. 3: Software architecture of CWP.

For the *model (M)* part of the MVC model, *JUnit*, a widely-used testing framework for Java, is employed. It facilitates the execution and validation of the test cases specified in the CWP assignments. The platform utilizes the file system for data management, where all the necessary data is provided through files. Students' programs are implemented using Java programming language. In terms of the *view (V)* part, the browser-based interface of the answer platform is designed to provide students with a user-friendly environment for solving CWP assignments. Rather than utilizing the default template engine of *Express.js* (a web application framework for Node.js), the platform utilizes *Embedded JavaScript (EJS)* as the template engine. EJS simplifies the syntax structure and enhances the ease of use for students. For the *control (C)* part, *Node.js* and *Express.js* [40] are adopted as the server-side technologies. *JavaScript* is used to implement the programs that handle the server-side logic and process students' submissions. These technologies enable the platform to handle student requests, manage data, and interact with the model and view components effectively.

The integration of these technologies and frameworks forms the foundation of the CWP answer platform. It provides students with a web-based interface where they can write and submit their source code solutions for CWP assignments. The platform leverages JUnit for testing the submitted code against the specified test cases. The test results are promptly displayed to students, helping them assess the correctness of their solutions.

The web-based answer interface for solving a CWP assignment is illustrated in Figure 4. The interface displays the test code of the assignment on the right side, while the left side provides an input space where students can write their answer source code. To successfully pass all the tests specified in the test code, a student needs to write the corresponding source code while looking at it. Once the student has finished writing the source code, they can click the "Submit" button for submitting it to the system. With submission, the system promptly performs immediate testing by compiling the source code and executing the test code with *JUnit*. The test results are then displayed at the bottom section of the interface.

However, in the current implementation of this platform, the student needs to manually save the source code in a file with a name matching the test code. The teacher must get this source code file in order to verify for the final checking along with the *answer code validation program*. The implementation of automatic file saving for submissions will be our future works. It is noted that Figure 3 and Figure 4 are adopted from the previous paper [13].

IV. Program for Answer Code Validation

The implementation of *answer code validation program* for the *code writing problem* in *JPLAS* is presented in this section. This functionality is achieved by modifying the existing code testing program in the answer platform. Instead of evaluating one source code at a time, the upgraded program now enables automatic testing of all source codes contained within a folder. Therefore, each folder is expected to contain the source codes for the same test code.

A. File System Directory Structure

The folder structure for the *answer code validation program* can be seen in Figure 5, which was adapted from [13]. The "test" folder within the "addon" folder serves as

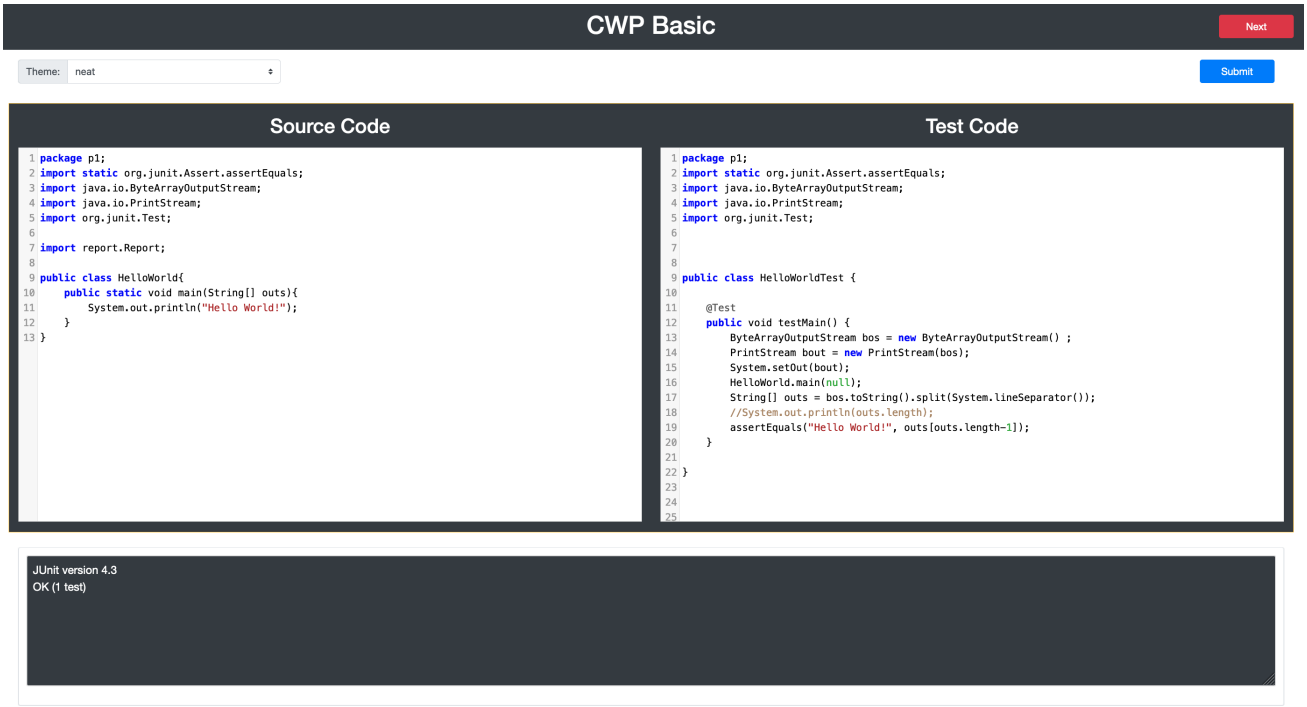

Fig. 4: Answer interface of CWP.

a repository for both the test code and the source code files submitted by students for each CWP assignment. The "codevalidator" folder includes the necessary Java programs to evaluate the source codes and generate the corresponding results. Additional folders and jar files are used in order to execute the code testing program. The "output" folder stores the text files containing the testing results, including JUnit logs. From them, this program will generate the CSV file and save it in the "csv" folder, allowing the teacher to conveniently check the results of all the students in the same file.

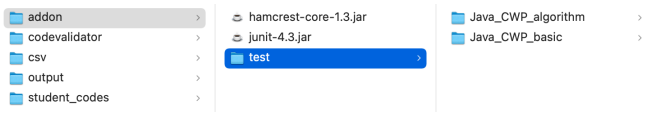

Fig. 5: File system structure for answer code validation program.

B. Procedure of Answer Code Validation

The answer code validation that will check the correctness of all the source codes from the students will be described in the following procedure.

1) The zip file containing the source codes for each assignment using one test code are downloaded. It is noted that a teacher usually uses an e-learning system such as *Moodle* in his/her programming course.
2) The contents of the zip file are extracted and they are stored in the relevant folder under the "student_codes" folder in the project path.
3) The corresponding test codes are then stored under the "addon/test" folder in the project directory.
4) Each source code in the "student_codes" folder is read, the test code is run with the source code using *JUnit*, and the testing results are saved in the text files under the the "output" folder. This process is repeated for every source codes in the folder.
5) The summary of the test results for all the source codes by the CSV file is generated and it is saved in the "csv" folder.

C. Answer Code Validation Example

The example of folder structure and related files are illustrated in Figure 6, which was adapted from [13]. To facilitate the process, the teacher requires to save 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 before running the program. It is noted that in the program, the folder structure for the source codes can be customized by preferences. For example, when using *Moodle*, the source code file for each student can be directly stored in the assignment-student folder.

Then, the test code ("helloWorldTest.java") is executed by the program with every source code sequentially, and the test output is recorded in the corresponding file ("student1_Java_CWP _basic_output.txt") within 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") within the "csv" folder.

D. Advantages and Limitations

The answer code validation program utilizes automated testing techniques to validate the correctness of the student solutions. It leverages *JUnit*, a widely-used testing framework in Java, to execute the *test code* with each *source code* and assess their conformity to the expected output. The program can automatically test a large number of student source codes in a short amount of time. By executing the test code sequentially with each source code, it eliminates

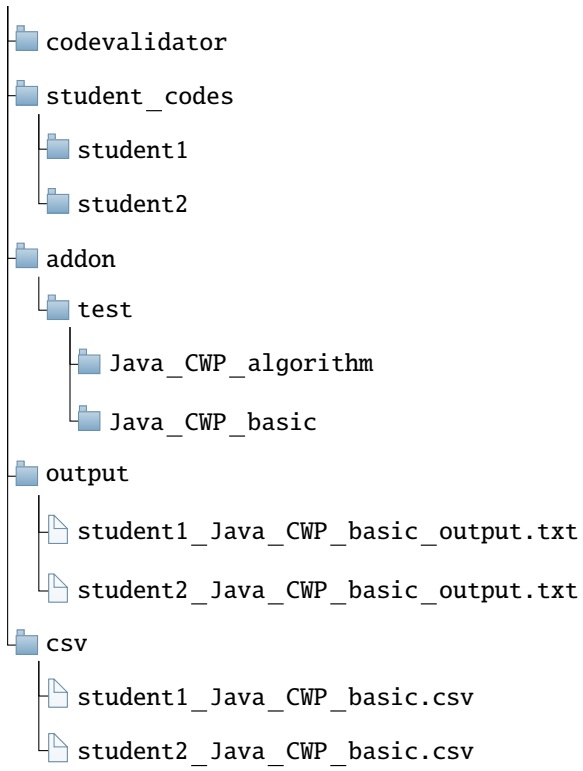

Fig. 6: Example of file structures with folder hierarchy

the need for manual operations, reducing the time and efforts required by the teacher.

While the answer code validation program offers several advantages, it also has some limitations. The program can identify whether a student's solution produces the expected output or not. However, it may not provide detailed insights into the specific errors or issues in the code. Further manual analysis may be required to diagnose and address the exact problems in the students' solutions. The correctness of the validation heavily relies on the quality and coverage of the test cases in the test code. If the test cases do not cover all the possible scenarios, the program may overlook certain errors or inaccurately assess the students' solutions. The program focuses solely on evaluating the output of the students' source codes, using the predefined test cases. It may not fully capture the overall understanding or design aspects of the solutions. The evaluation of subjective or higher-level aspects of the assignments may still require manual assessments by the teacher.

V. 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.

A. Limitation of Current Test Code

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

1) Fixed Data Output: The fixed test data in the test code can lead to 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 2.

```
package p3;
public class BubbleSort {
    public static int[] sort(int[] array) {
        int[] res = {1,4,5,7,8,9};
        return res;
    }
}
```

Fig. 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 generation of test data 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.

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 p3;
import java.util.Arrays;
public class BubbleSort {
    public static int[] sort(int[] a) {
        Arrays.sort(a);
        return a;
    }
}
```

Fig. 8: Example source code for using *library*.

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 p3;
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;
}
```

Fig. 9: Example source code for implementing of *different algorithm*.

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

B. Test Data Generation Algorithm

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 a standard format for describing 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`.

```
package p3;
import static org.junit.Assert.*;
import org.junit.Test;
import java.util.Arrays;
public class BubbleSortTest {
    @Test
    public void testSort() {
        BubbleSort bSort = new BubbleSort();
        int[] codeInput1 = {8,7,4,1,5,9};
        int[] codeOutput = bSort.sort(
            codeInput1);
        int[] expOutput = {1,4,5,7,8,9};
        try {
            assertEquals("Test1:",Arrays.
                toString(expOutput),Arrays.
                toString(codeOutput));
        } catch (AssertionError ae) {
            System.out.println(ae.getMessage
                ());
        }
    }
```

Fig. 10: Example of test code with standard format

1) Generating New Test Data: Once the data types are identified, the algorithm can generate new test data based on each data type. The approach for generating test data can vary depending on the specific data type. Here are some considerations for generating different data types:

- For integer types: Random numbers within a specified range can be generated.
- For floating-point types: Random real numbers within a specified range can be generated.
- For arrays: The algorithm can determine the array size and populate it with random values based on the element type.
- For strings: Various strategies can be used, such as generating random strings, using existing word lists, or incorporating specific patterns or constraints based on the assignment requirements.

The goal is to generate a diverse set of test data that covers different scenarios and edge cases to ensure comprehensive testing.

2) Replacing Test Data: Once the new test data is generated, the algorithm replaces the original test data in the test code with the newly generated test data. This ensures that each test case is executed with different input values. There are also some limitations, for complex data types.

3) Automatic Test Data Generation Procedure: The procedure for the test data generation algorithm is described as follows:

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

C. Intermediate State Testing for Logic and Algorithms

The intermediate state testing checks the randomly selected intermediate state of the important variables during the execution of the logic/algorithm. 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 p3;
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;
    }
}
```

Fig. 11: Example source code for intermediate state testing.

By observing the intermediate states, we can gain insights into how the code progresses and whether it behaves correctly at each step. This approach allows us to detect potential issues, such as incorrect loop conditions, incorrect variable assignments, or improper algorithm implementations. It ensures that the logic or algorithm is correctly implemented and functioning as intended.

The intermediate state testing plays a crucial role in assessing code quality for several reasons. Firstly, it helps identify logical errors or algorithmic flaws that may not be evident from the final output alone. Secondly, it encourages students to think critically about their codes and consider the step-by-step executions of the codes. It promotes a deeper understanding of the code's behaviors and encourages better programming practices. Lastly, by incorporating intermediate state testing into the evaluation process, we can provide more comprehensive and accurate assessments of the students' abilities, as it highlights their understanding of the underlying logic and their attentions to details. Therefore, by examining intermediate values of variables, it provides insights into the step-by-step execution of the code and allows for the detection of logical errors or algorithmic flaws.

```
package p3;
import static org.junit.Assert.*;
import org.junit.Test;
import java.util.Arrays;
public class BubbleSortTest {
    //intermediate state testing
    @Test
    public void testSortIteration() {
        BubbleSort bSort = new BubbleSort();
        int[] codeInput1 = {5,2,8,1,9};
        int codeInput2 = 2;
        int[] codeOutput = bSort.sort(
            codeInput1, codeInput2);
        int[] expOutput = {2,1,5,8,9};
        try {
            assertEquals("Test 1:",Arrays.
                toString(expOutput),Arrays.
                toString(codeOutput));
        } catch (AssertionError ae) {
            System.out.println(ae);
        }
    }
    //final state testing
    @Test
    public void testSort() {
        BubbleSort bSort = new BubbleSort();
        int[] codeInput1 = {8,7,4,1,5,9};
        int[] codeOutput = bSort.sort(
            codeInput1);
        int[] expOutput = {1,4,5,7,8,9};
        try {
            assertEquals("Test 2:",Arrays.
                toString(expOutput),Arrays.
                toString(codeOutput));
        } catch (AssertionError ae) {
            System.out.println(ae.getMessage
                ());
        }
    }
}
```

Fig. 12: Example test code for intermediate state testing.

VI. Evaluation

In this section, we evaluate the proposal through applications to the Java programming course in Okayama University, Japan, in two years. The evaluation was conducted to 1,005 source codes from 83 students to 15 CWP assignments. These source codes were processed using the answer code validation program, which automatically verified the code and generated a report indicating the number of passed tests. The evaluation was also focused on the performance of the students in terms of the pass rates of the test cases.

A. CWP Assignments in Course

The Java programming course is offered to the third-year students in Okayama University, Japan. They have studied *C programming* in the first year. A total of 28 students took this course in 2022, where a total of 55 students did in 2023.

15 CWP assignments were prepared with the two groups for studying *basic grammar* and *fundamental algorithms* topics, considering their levels in Java programming study. The corresponding test codes were made and given to the

students. Then, a total of 260 source codes were submitted from the students in 2022, and a total of 745 codes were in 2023.

Table I shows the group topic, the class name, the number of test cases in the test code, and the number of students who submitted answer source codes in two years for each of the 15 CWP assignments. After submissions, the submitted source codes were verified using the *answer code validation program*. After that, we analyzed the solution results of the students.

TABLE I: CWP assignments.

group topic	ID	class name	# of test cases	# of students 2022	# of students 2023
basic grammar	1	CodeCorrection1	3	28	55
	2	CodeCorrection2	3	28	55
	3	MaxItem	6	28	55
	4	MinItem	7	28	55
	5	ReturnAndBreak	3	28	55
fundamental algorithms	6	BinarySearch	9	12	47
	7	BinSort	5	12	47
	8	BubbleSort	4	12	47
	9	Divide	4	12	47
	10	GCD	5	12	47
	11	HeapSort	4	12	47
	12	InsertionSort	4	12	47
	13	LCM	5	12	47
	14	QuickSort	5	12	47
	15	ShellSort	4	12	47

B. Individual Assignments Results

First, the solution results of the individual CWP assignments are analyzed. Figure 13 shows the class name, and the average pass rate by the test data in the test codes that were given to the students for the two years. When the average pass rate by the test data generated by the proposal is different from that by the original test data, it is also shown with the bracket. The average pass rate is calculated by dividing the number of passed test cases by the total number of test cases in the test code. Moreover, we generated three different sets of random test data, and tested the source codes by them.

By comparing the two average pass rates for the "fundamental algorithms" group, the effectiveness of the proposed test code is confirmed. In the assignment with ID=14, the method in the source code of one student returned the output data in the test case instead of implementing the algorithm, which was found by applying the random test data. It is noted that some source codes cannot pass the test case for the intermediate state of the algorithm, because they use the library method or the different algorithm. For the assignment with ID=6, the library method "Arrays.binarySearch()" is used. For the assignments with ID=7, 8, 11, the library method "Arrays.sort()" is used. Moreover, the enrollment in the Java programming course in 2023 increased compared to 2022. The class is onsite in 2023 while the class was online in 2022. Therefore, we observed that the onsite class had higher engagement levels than the online class. For the upcoming year, we are considering a hybrid model that combines both onsite and online.

C. Individual Students Results

Next, we analyze the solution results of the individual students for the 15 CWP assignments. Table II provides the number of submitted answer codes from the students and the average CPU time for each assignment group for the two years.

TABLE II: Number of students and results in each group.

group topic	# of students 2022	# of students 2023	# of source codes 2022	# of source codes 2023	CPU Time (mins) 2022	CPU Time (mins) 2023
basic grammar	28	55	140	275	2.43	4.77
fundamental algorithms	12	47	120	470	1.95	7.64
total	40	102	260	745	4.38	12.41

Figure 14 and 15 present the solution results of the individual students in 2022 and 2023, respectively. In 2022, all of the 28 students correctly answered to the "basic grammar" assignments, whereas only 12 students answered to the "fundamental algorithms" assignments. It seems that many students did not understand or take the "fundamental algorithms" course that was offered in 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 2023, all of the 55 students answered to the "basic grammar" assignments and 47 students among them tried to answer the "fundamental algorithms" assignments.

D. Reducing Teacher Workload

Then, we evaluate the reduction of workloads of the teacher by the proposal. If the teacher does not use the proposal, he/she needs to repeat the following steps for each of the $1,005$ source codes: 1) open the source code on an IDE such as *VS-Code*, 2) run the test code with *JUnit* on the IDE, and 3) manually record the result in a text file. If the proposal is used, the teacher only needs to run the program. The CPU time was 7.2*min* for the *basic grammar* assignments and 9.59*min* for the *fundamental algorithms* assignments. Hence, the proposal can greatly reduce the workload.

E. Discussion

The evaluation results offer valuable insights into the effectiveness of the proposed test code in assessing the student solutions and reducing the teacher workload. In this section, we will discuss the implications of the results, highlight the strengths and limitations of this study, and suggest potential future improvements.

The analysis of the individual assignments reveals that the proposed test codes generally resulted in similar average pass rates to those by the original test codes. However, in the three assignments for "fundamental algorithms", the average pass rates decreased when the proposed test codes were used. Some students used library methods in their source codes. Thus, the proposal could avoid incorrect marking of the source codes.

Furthermore, when examining the results of the individual students, it becomes apparent that the "basic grammar" assignments were well understood by the majority of students, as evidenced by the high average pass rates. In contrast, the "fundamental algorithms" assignments posed more challenges to them, as evidenced by the lower average pass rates.

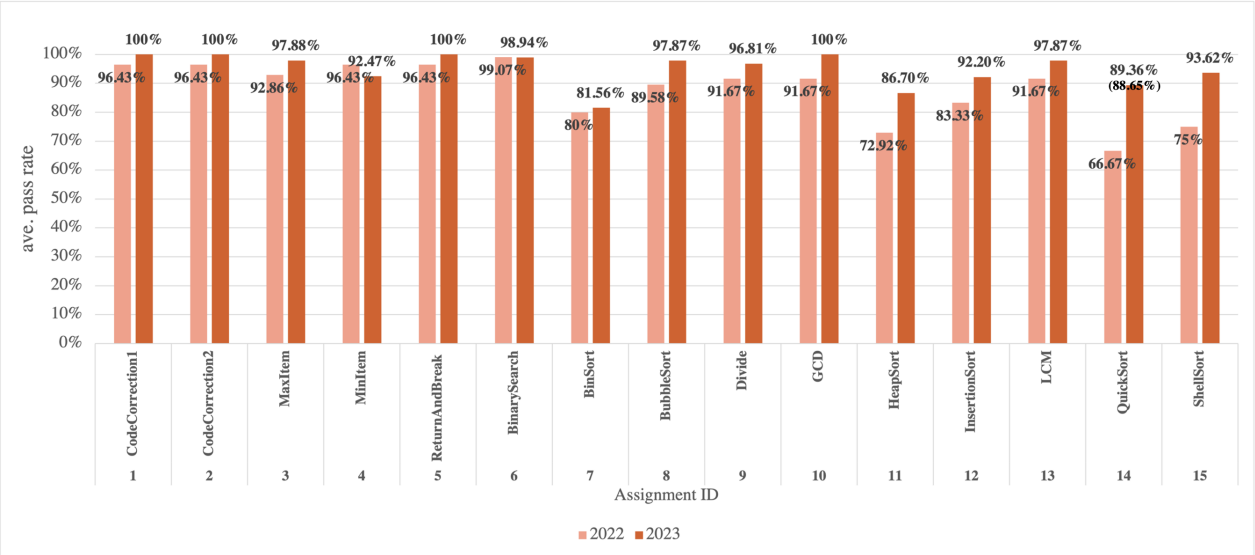

Fig. 13: Results of individual assignments.

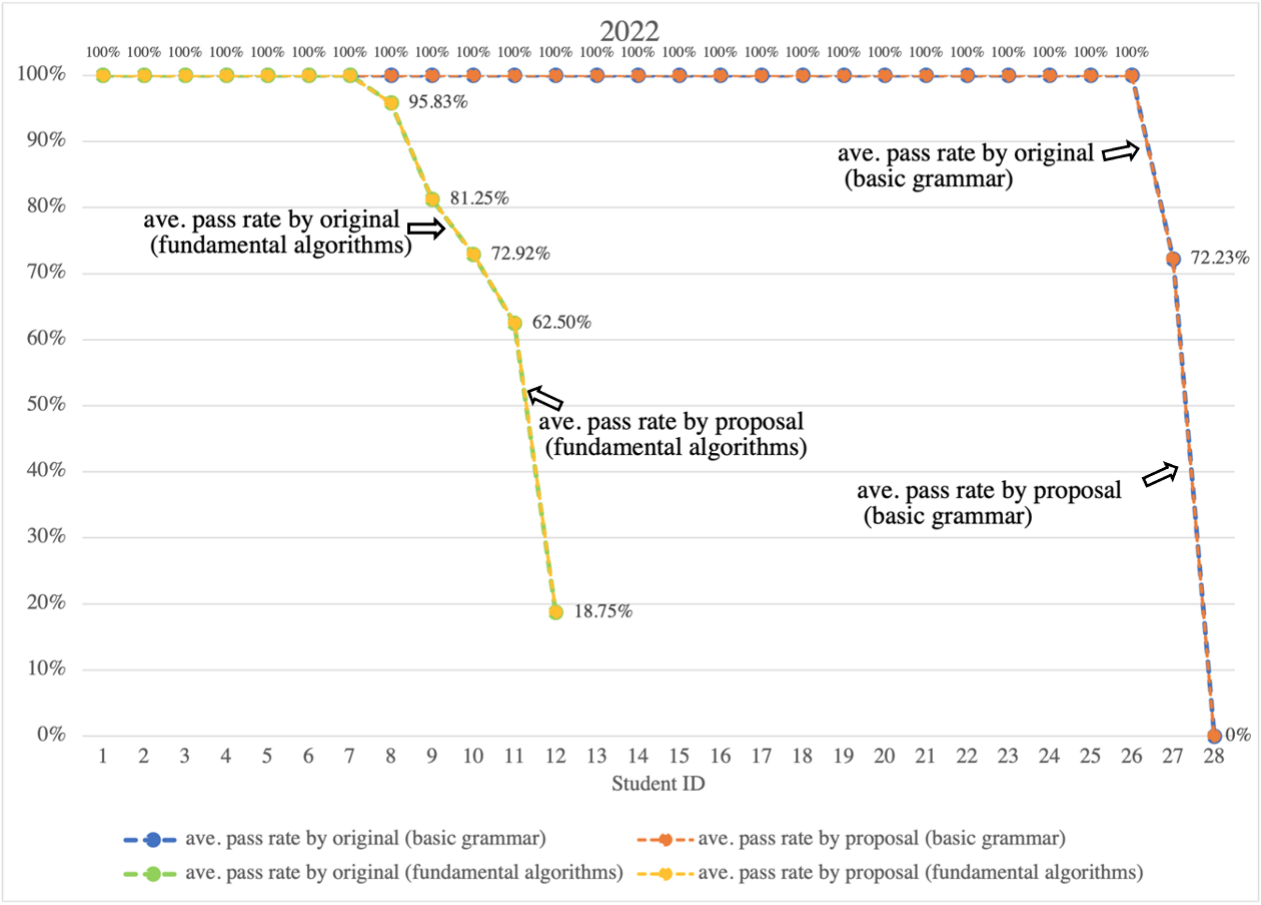

Fig. 14: Solution results for individual students in 2022.

This result highlights the need of emphasizing algorithmic programming skills to students.

With the automatic code validation, the proposed test code eliminates the need of manual assessments and result recording. It could save considerable time and efforts for the teacher, allowing him/her to focus on providing feedback and guidance to students. The reduced workload also opens up the possibility to handle larger classes efficiently. By enabling students to self-assess their understanding of the assignments, the proposal will encourage active learning and promote self-improvements by them.

While the evaluation offered promising results, there are several limitations to be considered. This evaluation was conducted within the specific context of a Java programming course in Okayama University. The findings may not directly be generalized to other educational settings or programming

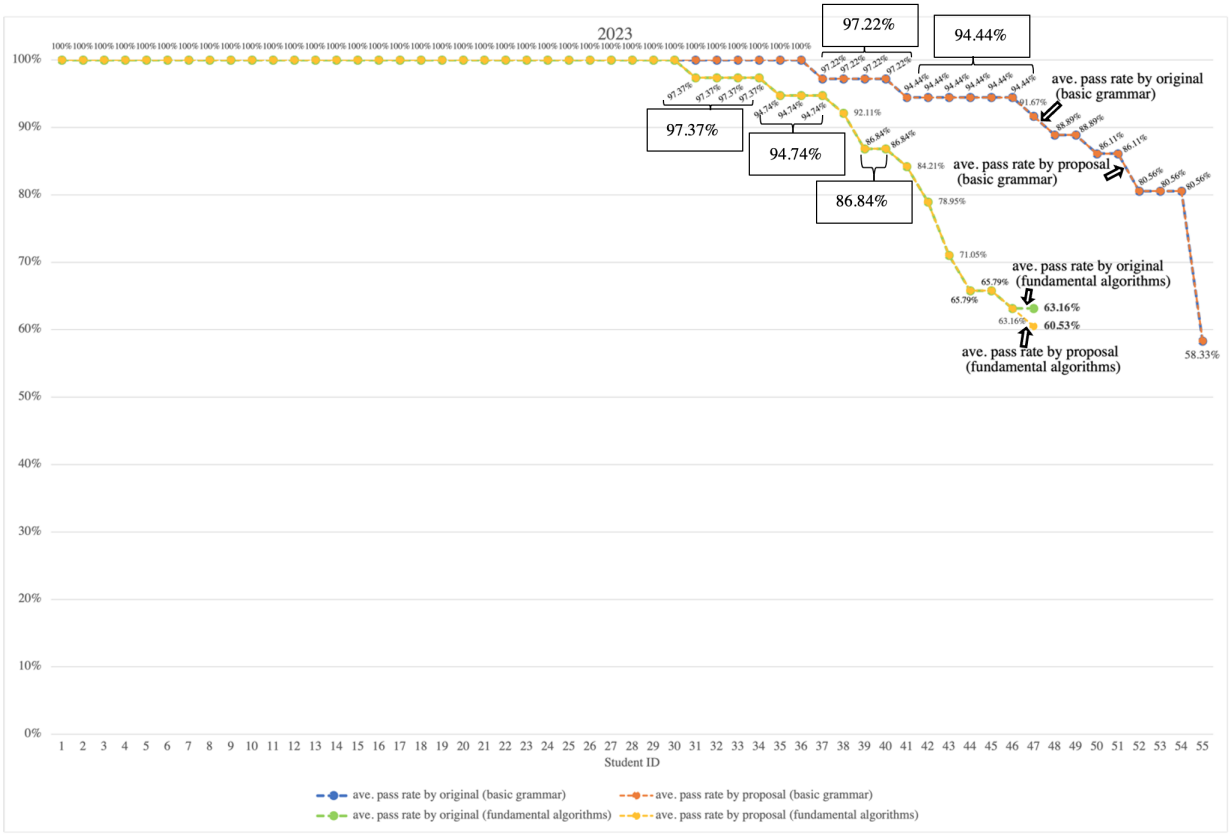

Fig. 15: Solution results for individual students in 2023.

languages. The effectiveness of the proposed solution should be further validated in different contexts for its wider applicability. The evaluation primarily focused on the pass rates of the test cases to assess the validity and effectiveness of the proposal. Other important aspects of the code quality, such as readability, efficiency, and adherence to programming best practices, were not evaluated. Therefore, future studies should incorporate these dimensions for comprehensive assessments of student performances.

VII. 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. By dynamically generating various test data with different data types and replacing them in the test code, our aim was to enhance the validity of CWP assignments and reduce the risk of cheating. Furthermore, the inclusion of *intermediate state testing* in the test code helps to verify the implementation of the requested logic or algorithm, promoting a deeper understanding of logic and algorithmic thinking among students.

For evaluations, the proposal was applied to $1,005$ source codes from 55 undergraduate in the Java programming course in Okayama university, Japan for two years. The analysis of individual assignments reveals that there were a few assignments in the "fundamental algorithms" group where the average pass rate decreased when using the proposed test code. This was observed in cases where students utilized library methods or implemented different algorithms by using the proposed *intermediate state testing*. Moreover, as the CPU time for testing 1,005 source codes was 7.2*min* for *basic grammar* and 9.59*min* for *fundamental algorithms*, the proposal can greatly reduce the teachers' workload. Therefore, the validity and effectiveness of the proposal have been confirmed.

However, it is important to acknowledge certain limitations of this study. Firstly, the evaluation was conducted within the specific context of a Java programming course at Okayama University, which may limit the results to other educational settings or programming languages. Additionally, the evaluation focused primarily on the pass rates of test cases, and other aspects such as code readability, efficiency, and adherence to programming best practices were not explicitly evaluated. Future studies could consider these additional dimensions to provide a more comprehensive assessment of student performance. In future works, we will study test codes with the proposal for other logic 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. Besides, we will study about the coding rule checking program and plagiarism for the readability and efficiency of the codes.

References

[1] Top Programming Languages 2022. IEEE Spectrum (online), https://spectrum.ieee.org/top-programming-languages-2022.

[2] Node.js (online), https://nodejs.org/en.
[3] Docker (online), https://www.docker.com/.
[4] S. T. Aung, N. Funabiki, L. H. Aung, H. Htet, H. H. S. Kyaw, and S. Sugawara, "An implementation of Java programming learning assistant system platform using Node.js," in Proc. ICIET, pp. 47-52, 2022.
[5] S. T. Aung, N. Funabiki, Y. W. Syaifudin, H. H. S. Kyaw, S. L. Aung, N. K. Dim, and W.-C. Kao, "A proposal of grammar-concept understanding problem in Java programming learning assistant system," J. Adv. Inform. Tech., vol. 12, no. 4, pp. 342-350, 2021.
[6] K. K. Zaw, N. Funabiki, and W.-C. Kao, "A proposal of value trace problem for algorithm code reading in Java programming learning assistant system," Inf. Eng. Express, vol. 1, no. 3, pp. 9-18, 2015.
[7] Y. Jing, N. Funabiki, S. T. Aung, X. Lu, A. A. Puspitasari, H. H. S. Kyaw, and W-C. Kao, "A proposal of mistake correction problem for debugging study in C programming learning assistant system," Int. J. Info. Edu. Tech. (IJIET), vol. 12, no. 11, pp. 1158-1163, 2022.
[8] 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.
[9] H. H. S. Kyaw, S. S. Wint, N. Funabiki, and W.-C. Kao, "A code completion problem in Java programming learning assistant system," IAENG International Journal of Computer Science, vol. 47, no. 3, pp. 350-359, 2020.
[10] X. Lu, S. Chen, N. Funabiki, M. Kuribayashi, and K. Ueda, "Aproposal of phrase fill-in-blank problem for learning recursive function in C programming," in Proc. LifeTech, pp. 127-128, 2022.
[11] N. Funabiki, Y. Matsushima, T. Nakanishi, 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.
[12] X. Lu, N. Funabiki, S. T. Aung, Y. Jing and S. Yamaguchi, "An implementation of Java programming learning assistant system in university course", in Proc. ICIET, pp. 215-220, 2023.
[13] K. H. Wai, N. Funabiki, S. T. Aung, K. T. Mon, H. H. S. Kyaw and W.-C. Kao, "An Implementation of Answer Code Validation Program for Code Writing Problem in Java Programming Learning Assistant System," in Proc. ICIET, pp. 193-198, 2023.
[14] Unit-testing (online), https://www.javatpoint.com/unit-testing.
[15] JUnit (online), http://www.junit.org/.
[16] Test-driven development (online), https://testdriven.io/test-driven-development/.
[17] K. Ala-Mutka, "Problems in Learning and Teaching Programming," A literature study for developing visualizations in the Codewitz-Minerva project, pp. 1-13, 2004.
[18] M. Konecki, "Problems in programming education and means of their improvement," DAAAM Int. Sci. Book, pp. 459-470, 2014.
[19] R. A. Queiros, L. Peixoto, and J. Paulo, "PETCHA - a programming exercises teaching assistant," in Proc. ITiCSE, pp. 192-197, 2012.
[20] A. Carbone, I. Mitchell, J. Hurst, and D. Gunstone, "An exploration of internal factors influencing student learning of programming," in Proc. Conf. Res. Pract. Inform. Tech. Ser., pp. 25-34, 2009.
[21] M. Piteira and C. Costa, "Learning computer programming: study of difficulties in learning programming," in Proc. ISDOC, pp. 75-80, 2013.
[22] F. W.-B. Li and C. Watson, "Game-based concept visualization for learning programming," in Proc. ACM MTDL, pp. 37-42, 2011.
[23] H. D. Ngyyen, T.-V. Tuan, X.-T. Pham, A. T. Huynh, V. T. Pham, D. Nguyen, "Design intelligent educational chatbot for information retrieval based on integrated knowledge bases," IAENG International Journal of Computer Science, vol. 49, no. 2, pp. 531-541, 2022.
[24] S. Matsumoto, S. Yamagishi, and T. Kashima, "Relationship Analysis between Puzzle-Like Programming Game and Achievement Result After Learning the Basic of Programming," LNECS Int. Multi. Conf. Eng. Comput. Sci., pp. 168-171, 2018.
[25] C. W. Okonkwo, and A. Ade-Ibijola, "Revision-Bot: A Chatbot for Studying Past Questions in Introductory Programming," IAENG International Journal of Computer Science, vol. 49, no.3, pp. 644-652, 2022.
[26] P. Ihantola, T. Ahoniemi, V. Karavirta, and O. Seppälä, "Review of recent systems for automatic assessment of programming assignments," in Proc. Koli Calling, pp. 86–93, 2010.
[27] T. Staubitz, H. Klement, J. Renz, R. Teusner, and C. Meinel, "Towards practical programming exercises and automated assessment in Massive Open Online Courses," in Proc. TALE, pp. 23-30, 2015.
[28] P. Denny, A. Luxton-Reilly, E. Tempero, and J. Hendrickx, "CodeWrite: supporting student-driven practice of Java," in Proc. SIGCSE, pp. 471-476, 2011.
[29] F. A. Shamsi and A. Elnagar, "An intelligent assessment tool for student's Java submission in introductory programming courses," J. Intelli. Learn. Syst. Appl., vol. 4, pp. 59-69, 2012.
[30] H. Mei, D. Hao, L. Zhang, J. Zhou, and G. Rothermel, "A static approach to prioritizing JUnit test cases," IEEE Trans. Soft. Eng., vol. 38, no. 6, pp. 1258-1275, 2012.
[31] S. H. Edwards, "Using test-driven development in the classroom: Providing students with automatic, concrete feedback on performance," in Proc. EISTA, pp. 421-426, 2003.
[32] S. H. Edwards and M. A. Pérez-Quiñones, "Experiences using test-driven development with an automated grader," *Journal of Computing Sciences in Colleges*, vol. 22, no. 3, pp. 44-50, 2007.
[33] C. Desai, D. Janzen, and K. Savage, "A survey of evidence for test-driven development in academia," *ACM SIGCSE Bulletin*, vol. 40, no. 2, pp. 97-101, 2000.
[34] I. T. Elgendy, M. R. Girgis, and A. A. Sewisy, "A GA-Based Approach to Automatic Test Data Generation for ASP.NET Web Applications," IAENG International Journal of Computer Science, vol. 47, no.3, pp. 557-564, 2020.
[35] H. Kitaya and U. Inoue, "An online automated scoring system for Java programming assignments," *International Journal of Information and Education Technology*, vol. 6, no. 4, pp. 275-279, 2016.
[36] E. Ünal and H. Çakir, "Students' views about the problem based collaborative learning environment supported by dynamic web technologies," *Malaysian Online Journal of Educational Technology*, vol. 5, no. 2, pp. 1-19, 2017.
[37] S. H. Tung, T. T. Lin and Y. H.Lin, "An Exercise Management System for Teaching Programming," *J. Softw.*, vol. 8, no. 7, pp. 1718-1725, 2013.
[38] M. Szab and K. Nehz, "Grading Java code submissions in MeMOOC," in Proc. microCAD Int. Sci. Conf, 2018.
[39] I. S. Zinovieva, V. O. Artemchuk, A. V. Iatsyshyn, O. O. Popov, V. O. Kovach, A. V. Iatsyshyn, Y. O. Romanenko, and O. V. Radchenko, "The use of online coding platforms as additional distance tools in programming education,"*Journal of physics: Conference series,*, vol. 1840, 2021.
[40] Express (online), https://expressjs.com/.
[41] Bubble Sort (online), https://www.javatpoint.com/bubble-sort-in-java.

K. H. Wai received the B.E. and M.E. degrees in information science and technology from University of Technology (Yatanarpon Cyber City), Myanmar, in 2016 and 2020, respectively. She is currently an Ph.D. candidate in Graduate School of Natural Science and Technology, Okayama University, Japan. Her research interests include educational technology and web application systems.

N. 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. 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.

S. T. Aung received the B.E. degree in Information Technology from the University of Technology (Thanlyin), Myanmar, in 2017. She received the M.E. degree in Electronic and Information System Engineering at Okayama University, Japan, in 2023. She is currently a Ph.D student in Department of Information and Communication System Engineering at Okayama University, Japan. She received the OU Fellowship in 2023. Her research interests include educational technology.

X. Lu received the B.S. degree in electronic information engineering from Hubei University of Economics, China, in 2017, and received the M.S degree in electronic information systems from Okayama University, Japan, in 2021, respectively. She is currently a Ph.D. student in Graduate School of Natural Science and Technology, Okayama University, Japan. She received the OU Fellowship in 2021. Her research interests include educational technology.

Y. Jing received the B.S. degree in information management and information system (Japanese-English bilingual strengthening) from Dalian University of Foreign Languages, China, in 2020. She is currently a master student in electronic information systems at Okayama University, Japan. Her research interests include educational technology.

H. H. S. Kyaw received the B.E. and M.E. degrees in information science and technology from University of Technology (Yatanarpon Cyber City), Myamar, in 2015 and 2018, and Ph. D. in information communication engineering from Okayama University, Japan, in 2021, respectively. She is currently an assistant professor in Division of Advanced Information Technology and Computer Science, Tokyo University of Agriculture and Technology, Koganei, Japan. Her research interests include educational technology and web application systems. She is a member of IEICE.

W.-C. Kao received the M.S. and Ph.D. degrees in electrical engineering from National Taiwan University, Taiwan, in 1992 and 1996, respectively. He was at SoC Technology Center, ITRI, Taiwan, from 1996 to 2000, and at NuCam Corporation, Taiwan, from 2000 to 2004. Since 2004, he has been with National Taiwan Normal University (NTNU), Taipei, Taiwan, where he is currently the Research Chair Professor at Department of Electrical Engineering and the Dean of College of Technology and Engineering. His current research interests include system-on-a-chip (SoC) as well as embedded software design, flexible electrophoretic display, machine vision system, digital camera system, and color imaging science. He is a fellow of IEEE.