A Study of Java Answer Code Validation Program and JavaScript Code Modification Problems

September, 2024

Khaing Hsu Wai

Graduate School of
Natural Science and Technology

(Doctor's Course)
Okayama University

Dissertation submitted to
Graduate School of Natural Science and Technology
of
Okayama University
for
partial fulfillment of the requirements
for the degree of
Doctor of Philosophy in Engineering

Written under the supervision of

Professor Nobuo Funabiki

and co-supervised by
Professor Satoshi Denno
and
Professor Yasuyuki Nogami

OKAYAMA UNIVERSITY, September 2024.

To Whom It May Concern

We hereby certify that this is a typical copy of the original doctor thesis of
Ms. Khaing Hsu Wai

Signature of the Supervisor	Seal of
Prof. Nobuo Funabiki	Graduate School of Natural Science and Technology

Abstract

Programming education is crucial in fostering critical thinking, problem-solving abilities, and creativity of students. These skills empower them to explore a broad range of career paths after graduations. Consequently, *computer programming* has become a pivotal subject in universities and professional schools. As a result, many universities and professional schools are offering programming courses to train future programming engineers.

To support self-studies of novice students in *Java* programming, we have developed a web-based *Java Programming Learning Assistant System (JPLAS)* and implemented the personal answer platform on *Node.js* that will be distributed to students via *Docker*. *JPLAS* provides several types of exercise problems that have different learning goals for *code reading* and *code writing* skills. 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.

In *JPLAS*, the *code writing problem (CWP)* asks a student to write a *source code* to pass the given *test code* where the correctness is verified by running them on *JUnit*. Previously, we implemented the *answer platform* for students to solve CWP assignments with the automatic runs. However, the teacher needs to run the test codes and the source codes one by one manually at the marking process. This load is very large for the teacher. Therefore, in this thesis, I present the study of *answer code validation program* for *CWP* in *JPLAS*.

As the first contribution of the thesis, I implement the *answer code validation program* to help teacher works in assigning a lot of CWP assignments to students in a Java programming course in a university or professional school. This program automatically tests and verifies all the source codes that are made to pass the tests in a test code, and reports the number of tests that each source code could pass with the CSV file. By looking at the summary of the test results of all the students, the teacher can easily grasp the progress of students and grade them.

As the second contribution of the thesis, I propose the *intermediate state testing* in the *test code* for *data structure and algorithms* assignments. Against the assignment request, a student may use the library without implementing the correct logic/algorithm in the source codes. If a student implements a different logic or algorithm including the use of library, the conventional *test code* cannot find it. To improve problem-solving skills and develop strong foundations in algorithmic thinking, the *intermediate state testing* can check the randomly selected intermediate state of the important variables during the execution of the logic/algorithm.

As the third contribution of the thesis, I implement the *test data generation algorithm*. The fixed test data in the test code may 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 *test data generation algorithm* 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.

As the fourth contribution of the thesis, I implement the *naming rules checking function* in the *answer code validation program* for *CWP* in *JPLAS* for novice students, to master writing *readable codes* using proper names for variables, classes, and methods in *Java programming*. To master writing *readable codes* using proper names for variables, classes, and methods is crucial in *Java programming*, to improve *understandability* and *maintainability* for the novice students. The *naming rules checking function* finds the naming errors in the source code. It is also implemented in the answer platform so that a student can write a code while checking the rules. The students will master in writing *readable codes* using proper names for variables, classes, and methods at the early stage of programming studies.

In current societies, *web application systems* take central roles in computer systems. Thus, *web client programming* using *JavaScript* has increased values to add dynamic features and functions in web pages by well working with *HTML* and *CSS*. However, due to the fact that most web pages are written with the combination of *JavaScript*, *HTML*, and *CSS*, the current type of exercise problem available in *JSPLAS* may not be suitable for studying web client programming. Therefore, in this thesis, I present the study of *code modification problem (CMP)* as the new exercise problem type in *JSPLAS* for effective self-study of web client-side and server-side programming using JavaScript.

As the fifth contribution of the thesis, I propose a *code modification problem (CMP)* as a new type of exercise problem in *JavaScript Programming Learning Assistant System (JSPLAS)*, to study *web client-side* and *server-side programming* using *JavaScript*. The goal of *CMP* is for the students to carefully read the source code and comprehend how to use the components and functions through modifying parameters, values, or messages. The CMP instance gives a source code using the functions to be studied and the screenshot of the web page generated by it. Then, it requests to modify the code to generate another web page given by the screenshot. The correctness of any answer is checked through *string matching* with the correct one.

In future works, we will study test codes for other logic or algorithms in mathematics, physics, and engineering topics, generate new assignments for other Java grammar topics, and apply naming rules checking function to students in Java programming courses for the readability and efficiency of the codes. Besides, we will study CMP for other topics and investigate the effectiveness.

Acknowledgements

I would like to express my heartfelt gratitude to all who supported and guided me throughout the completion of this thesis at Okayama University, Japan. To all who have been part of this journey, I will just say that you are the greatest blessing in my life.

First and foremost, I owe my deepest gratitude to my honorable supervisor, Professor Nobuo Funabiki for his excellent supervision, meaningful suggestions, persistent encouragements, and other fruitful help at every stage of my Ph.D. study. His thoughtful comments and guidance helped me to complete my research papers and present them in productive ways. Besides, he was always patient and helpful whenever his guidance and assistance were needed in both of my academic and daily life in Japan. Indeed, his guidance has been nothing less than a gift. Needless to say, it would not be possible to complete this thesis without his guidance and active support.

I extend my heartfelt thanks to my Ph.D. co-supervisors, Professor Satoshi Denno and Professor Yasuyuki Nogami, for their continuous support, guidance, thoughtful suggestions, and proofreading of this thesis. I am also grateful to Professor Wen-Chung Kao from National Taiwan Normal University and all my course instructors for their enlightening knowledge and discussions. I also would like to acknowledge to all my respect teachers in Univeristy of Techonlogy (Yatanarpon Cyber City) for guiding a lot of valuable knowledge.

I appreciate all members of the FUNABIKI Lab for their support in my studies and Ms. Keiko Kawabata for her administrative support throughout my Ph.D.

I would like to acknowledge the IUCHI Scholarship, Monbukagakusho Honors Scholarship (JASSO Scholarship) and SGU (MEXT) Scholarship for financially supporting my Ph.D study.

My special thanks to Dr. Htoo Htoo Sandi Kyaw and Dr. San Haymar Shwe, who gave me valuable advice and supported me to start my Ph.D study. My sincere thanks also go to Professor Ye Kyaw Thu, Dr. Thazin Myint Oo, Dun Dun and my six best friends (Hwan Khun, Yun, Chit Pont, Thandar, Ohnmar and Phyu Sin). Your support during challenging times and the shared thoughts and experiences mean a lot to me.

Last but not least, I am eternally grateful to my beloved family, Phay Phay (father), May May (mother) and Nge Lay (sister) for their unconditional love, support, patience, and confidence in me, which have been my greatest motivation and reward. I am proud and blessed to have you all in my life.

Khaing Hsu Wai
Okayama University, Japan
September, 2024

List of Publications

Journal Paper

1. **Khaing Hsu Wai**, Nobuo Funabiki, Khin Thet Mon, May Zin Htun, San Hay Mar Shwe, Htoo Htoo Sandi Kyaw, and Wen-Chung Kao, "A proposal of code modification problem for self-study of web client programming using JavaScript," Advances in Science, Technology and Engineering Systems Journal (ASTESJ), vol. 7, no. 5, pp. 53-61, September 2022. DOI: 10.25046/aj070508

2. **Khaing Hsu Wai**, Nobuo Funabiki, Soe Thandar Aung, Xiqin Lu, Yanhui Jing, Htoo Htoo Sandi Kyaw, and Wen-Chung Kao, "Answer code validation program with test data generation for code writing problem in Java programming learning assistant system," IAENG Engineering Letters, vol. 32, no. 5, pp. 981-994, May 2024.

International Conference Paper

3. **Khaing Hsu Wai**, Nobuo Funabiki, Khin Thet Mon, San Hay Mar Shwe, Htoo Htoo Sandi Kyaw, and Khin Sandar Lin, "A proposal of code modification problem for Web client programming using JavaScript," The Ninth International Symposium on Computing and Networking (CANDAR), pp. 196-202, November 23 - 26, 2021. DOI: 10.1109/CANDAR53791.2021.00035

4. **Khaing Hsu Wai**, Nobuo Funabiki, Huiyu Qi, Yanqi Xiao, Khin Thet Mon, and Yan Watequlis Syaifudin, "Code modification problems for multimedia use in JavaScript-based web client programming," 14th International Workshop on Virtual Environment and Network-Oriented Applications (VENOA-2022), CISIS 2022, LNNS 497, pp. 548-556, June 29 - July 1, 2022. DOI: https://doi.org/10.1007/978-3-031-08812-4_53

5. **Khaing Hsu Wai**, Nobuo Funabiki, Shune Lae Aung, Soe Thandar Aung, Yan Watequlis Syaifudin, and Wen-Chung Kao, "An investigation of code modification problem for learning server-side JavaScript programming in web application system," 2022 IEEE 11th Global Conference on Consumer Electronics (GCCE), pp. 886-887, October 18-21, 2022. DOI: 10.1109/GCCE56475.2022.10014232

6. **Khaing Hsu Wai**, Nobuo Funabiki, Soe Thandar Aung, Xiqin Lu, Yanhui Jing, Htoo Htoo Sandi Kyaw, Wen-Chung Kao, "Code writing problems for basic object-oriented programming study in Java programming learning assistant system," 2023 IEEE 12th Global Con-

ference on Consumer Electronics (GCCE 2023), pp. 5-6, October 10-13, 2023. DOI: 10.1109/GCCE59613.2023.10315469

7. **Khaing Hsu Wai**, Nobuo Funabiki, Soe Thandar Aung, Khin Thet Mon, Htoo Htoo Sandi Kyaw, Wen-Chung Kao, "An implementation of answer code validation program for code writing problem in Java programming learning assistant system," 2023 11th International Conference on Information and Education Technology (ICIET), pp. 193-198, March 18-20, 2023. DOI: 10.1109/ICIET56899.2023.10111392

8. **Khaing Hsu Wai**, Nobuo Funabiki, Soe Thandar Aung, Ryo Hashimoto, Daiki Yokoyama, Wen-Chung Kao, "Analysis of solution results of code writing problems for basic object-oriented programming study in university Java programming course," 2024 12th International Conference on Information and Education Technology (ICIET), pp. 87-92, March 18-20, 2024. DOI: 10.1109/ICIET60671.2024.10542814

Other Papers

9. **Khaing Hsu Wai**, Nobuo Funabiki, Mustika Mentari, Soe Thandar Aung, Wen-Chung Kao, "Implementation of naming rules checking function in code validation program for code writing problem in Java programming learning assistant system," to appear in FIT Conference, September 2024.

10. Ei Ei Htet, **Khaing Hsu Wai**, Soe Thandar Aung, Nobuo Funabiki, Xiqin Lu, Htoo Htoo Sandi Kyaw, and Wen-Chung Kao, "Code plagiarism checking function and its application for code writing problem in Java programming learning assistant system," Analytics, vol. 3, no. 1, pp. 46-62, January 2024. DOI: https://doi.org/10.3390/analytics3010004

List of Figures

2.1 JPLAS architecture. 5
2.2 CWP answer interface. 9

3.1 CWP software architecture. 12
3.2 CWP answer interface. 13
3.3 Designated file system for answer code validation program. 13
3.4 Example of file structures with folder hierarchy 15
3.5 Solution results for individual students (2022). 19
3.6 Solution results for individual students (2023). 20
3.7 Solution results for individual assignments (2022). 21
3.8 Solution results for individual assignments (2023). 21

4.1 Example source code for *BubbleSort*. 24
4.2 Example test code for BubbleSort. 25
4.3 Example source code for using *library*. 25
4.4 Example source code for implementing of *different algorithm*. 26
4.5 Example test code for intermediate state testing. 27
4.6 Example source code for intermediate state testing. 28
4.7 Results of individual assignments. 30
4.8 Solution results for individual students. 31

5.1 Example source code for *BubbleSort*. 33
5.2 Example test code for BubbleSort. 33
5.3 Example source code for *fixed data output*. 34
5.4 Example of test code with standard format 35
5.5 Results of individual assignments. 38
5.6 Solution results for individual students in 2022. 39
5.7 Solution results for individual students in 2023. 39

6.1 Designated file system for naming checking function. 43
6.2 Results for *basic grammar*. 47
6.3 Results for *data structure*. 47
6.4 Results for *object oriented programming*. 48
6.5 Results for *fundamental algorithms*. 48
6.6 Results for *final exam*. 49
6.7 Results for *TI 1C basic programming*. 49
6.8 Results for *TI 1I basic programming*. 50
6.9 Results for *MI 3D algorithm and data structure*. 50
6.10 Results for *Test1*. 51

6.11 Results for *Test2*. 52

7.1 Original source code for CMP instance #19. 56
7.2 Web page by source code for CMP instance #19. 56
7.3 Modified source code for CMP instance #19. 57
7.4 Modified web page for CMP instance #19. 57
7.5 Answer interface for CMP instance #19. 59
7.6 Results for each student. 61
7.7 Solution results for individual CMP instances. 62
7.8 Sample *Timer* page for assignment #1. 63
7.9 Sample *Calculator* page for assignment #2. 64

8.1 Answer interface for example CMP instance. 68
8.2 Solution results for individual students. 70
8.3 Solution results for individual CMP instances. 70

List of Tables

2.1 Files for distribution in JPLAS. 6

3.1 CWP assignments for 2022. 17
3.2 CWP assignments for 2023. 18
3.3 Number of submitted students and average passing rate in each group. 19
3.4 Load reduction results. 22

4.1 CWP assignments. 29
4.2 Number of students and results in each group. 30

5.1 CWP assignments. 37

6.1 Naming rules test result. 45
6.2 Course name and topics for evaluations. 46
6.3 Summary of application results. 52

7.1 Generated CMP instances. 60

8.1 CMP instances and solution results. 69

Contents

Abstract i

Acknowledgements iii

List of Publications iv

List of Figures vii

List of Tables viii

1 Introduction 1
- 1.1 Background for Java Study 1
- 1.2 Background for JavaScript Study 2
- 1.3 Contributions 3
- 1.4 Contents of Dissertation 3

2 Overview of Java Programming Learning Assistant System (JPLAS) 5
- 2.1 System Architecture 5
  - 2.1.1 Operation Flow 6
  - 2.1.2 Distributed Files 6
  - 2.1.3 Cheating Prevention 7
- 2.2 Implemented Problem Types 7
- 2.3 Code Writing Problem (CWP) 7
  - 2.3.1 JUnit 8
  - 2.3.2 Test Code 8
  - 2.3.3 Answer Interface of CWP 9
- 2.4 Summary 9

3 Answer Code Validation Program 10
- 3.1 Introduction 10
- 3.2 Previous Works of Code Writing Problem 10
  - 3.2.1 Code Writing Problem 10
  - 3.2.2 JUnit for Unit Testing 11
  - 3.2.3 Test Code 11
  - 3.2.4 CWP Answer Platform for Students 12
- 3.3 Answer Code Validation Program for Teachers 13
  - 3.3.1 Folder Structure in File System 13
  - 3.3.2 Procedure of Answer Code Validation 14

3.3.3 Example of Answer Code Validation Procedure . . . 14
3.3.4 Advantages and Limitations . . . 14
3.4 Evaluation . . . 15
3.4.1 CWP Assignments . . . 16
3.4.2 Solution Results . . . 18
3.4.2.1 Solution Results of Individual Students . . . 19
3.4.2.2 Results of Individual Assignments . . . 20
3.4.2.3 Reducing Teacher Workload . . . 21
3.5 Summary . . . 22

4 Intermediate State Testing for Fundamental Algorithm Assignments 23
4.1 Introduction . . . 23
4.2 Test Code . . . 23
4.2.1 JUnit . . . 23
4.2.2 Test Code . . . 24
4.3 Intermediate State Testing for Logic and Algorithms . . . 24
4.3.1 Limitation of Current Test Code . . . 25
4.3.1.1 Library Use . . . 25
4.3.1.2 Implementation of Different Logic or Algorithm . . . 25
4.3.2 Intermediate State Testing for Logic and Algorithms . . . 26
4.4 Evaluation . . . 28
4.4.1 CWP Assignments in Course . . . 29
4.4.2 Individual Assignments Results . . . 29
4.4.3 Individual Students Results . . . 30
4.5 Summary . . . 31

5 Dynamic Test Data Generation Algorithm 32
5.1 Introduction . . . 32
5.2 Test Code . . . 32
5.2.1 JUnit . . . 32
5.2.2 Test Code . . . 33
5.3 Test Data Generation Algorithm . . . 34
5.3.1 Limitation of Current Test Code . . . 34
5.3.1.1 Fixed Data Output . . . 34
5.3.2 Test Data Generation Algorithm . . . 35
5.3.2.1 Generating New Test Data . . . 35
5.3.2.2 Replacing Test Data . . . 36
5.3.2.3 Automatic Test Data Generation Procedure . . . 36
5.4 Evaluation . . . 36
5.4.1 CWP Assignments in Course . . . 37
5.4.2 Individual Assignments Results . . . 37
5.4.3 Individual Students Results . . . 38
5.5 Summary . . . 40

6 Naming Rules Checking Function in Code Validation Program **41**
6.1 Introduction 41
6.2 Previous Works of Code Writing Problem in JPLAS 41
6.2.1 Overview of Code Writing Problem (CWP) 41
6.2.2 Answer Code Validation Program for Teachers 42
6.3 Naming Rule Checking Function 42
6.3.1 File Structures with Folder Hierarchy 43
6.3.2 Four Naming Rules 43
6.3.3 Procedure of Naming Checking Function 44
6.3.4 Example of Testing Result 44
6.4 Analysis of Application Results 45
6.4.1 Courses and Topics 45
6.4.2 Analysis Results of Individual Groups in Okayama University 46
6.4.2.1 Results for Basic Grammar 46
6.4.2.2 Results for Data Structure 47
6.4.2.3 Results for Object Oriented Programming 47
6.4.2.4 Fundamental Algorithms 48
6.4.2.5 Final Exam 48
6.4.3 Analysis Results of Individual Groups in Malang State Polytechnic 49
6.4.3.1 TI 1C Basic Programming 49
6.4.3.2 TI 1I Basic Programming 50
6.4.3.3 MI 3D Algorithm and Data Structure 50
6.4.4 Analysis Results of Individual Groups in Yamaguchi University 51
6.4.4.1 Test1 51
6.4.4.2 Test2 51
6.4.5 Percentage Analysis Results of Naming Convention in Each Course 52
6.5 Summary 53

7 Code Modification Problem for Client-side Programming Using JavaScript **54**
7.1 Introduction 54
7.2 Proposal of Code Modification Problem (CMP) 54
7.2.1 Definition of CMP 54
7.2.2 Design Goal of CMP 55
7.2.3 CMP Instance Generation Procedure 55
7.3 Example of CMP Instance Generation 56
7.3.1 Original Source Code 56
7.3.2 Original Web Page 56
7.3.3 Modified Source Code 57
7.3.4 Modified Web Page 57
7.3.5 Answer Interface 57
7.4 Two-Level Answer Marking 58
7.4.1 First-Level Marking 58
7.4.2 Second-Level Marking 58
7.5 Evaluation 60
7.5.1 Generated CMP Instances 60
7.5.2 Solution Results 60
7.5.2.1 Results of Individual Students 61

7.5.2.2 Results of Individual Instances 61
7.6 Project Assignment for Learning Effect Evaluation 62
7.6.1 Overview 62
7.6.2 Project Assignment #1 63
7.6.2.1 Problem Statement 63
7.6.2.2 Hint 63
7.6.2.3 Result and Discussion 63
7.6.3 Project Assignment #2 64
7.6.3.1 Problem Statement 64
7.6.3.2 Hint 64
7.6.3.3 Result and Discussion 65
7.7 Summary 65

8 Code Modification Problem for Server-side Programming Using JavaScript 66
8.1 Introduction 66
8.2 Overview of Code Modification Problem (CMP) 66
8.3 CMP Instance Generation Procedure 67
8.4 Example of CMP Instance Generation 67
8.4.1 Concept Overview 67
8.4.2 Source Code 68
8.4.3 Code Modification Requests 68
8.4.4 Answer Interface 68
8.5 Evaluation 69
8.5.1 Generated CMP Instances 69
8.5.2 Solution Results 69
8.5.2.1 Solution Results of Individual Students 69
8.5.2.2 Solution Results of Individual Assignments 69
8.6 Summary 71

9 Related Works in Literature 72
9.1 Programming Education and Learning 72
9.2 Code Writing in Programming Study 73
9.3 Code Readability and Maintenance 74
9.4 JavaScript Programming Study 74

10 Conclusion 76

References 78

Chapter 1

Introduction

1.1 Background for Java Study

Programming education is crucial in fostering critical thinking, problem-solving abilities, and creativity of students. These skills empower them to explore a broad range of career paths after graduations. Consequently, *computer programming* has become a pivotal subject in universities and professional schools. As a result, many universities and professional schools are offering programming courses to train future programming engineers.

For decades, *Java* has been widely employed in various industries as a dependable and adaptable object-oriented programming language [1]. Its 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 it by solving uncomplicated and concise exercise problems that focus on *code reading* studies. It enables 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].

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

For any exercise problem of them, the correctness of the answer from a student is verified automatically. The correctness of the student answer is checked through *string matching* with the correct one for *GUP*, *VTP*, *MCP*, *EFP*, *CCP* and *PFP*, and through *unit testing* for *CWP*. 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.

In *JPLAS*, the *code writing problem (CWP)* asks a student to write a *source code* to pass the given *test code* where the correctness is verified by running them on *JUnit*. Previously, we implemented the *answer platform* for students to solve CWP assignments with the automatic runs. However, the teacher needs to run the test codes and the source codes one by one manually at the marking process. This load is very large for the teacher.

1.2 Background for JavaScript Study

Nowadays, *JavaScript* has become popular in web programming using *Node.js*, since it can be used on both client and server sides to make web pages interactive in a web application system [12]. *JavaScript* is adopted in 97% websites in the world, making it the most popular client-side and server-side scripting language for *web client programming*. By combining the common web technology by the *Hyper Text Markup Language (HTML)* and *Cascading Style Sheets (CSS)* , *JavaScript* can provide dynamic features on a *web page*. The structure and meaning of the page are provided in *HTML*. The layout, background colors, and fonts used in *HTML* content are described in *CSS*. Then, the *document object model (DOM)* is used for *JavaScript* interactions with them. Therefore, *web client programming using JavaScript* has increased values to add dynamic features and functions in web pages by well working with *HTML* and *CSS*. To support self-studies of *JavaScript programming*, *JavaScript Programming Learning Assistant System (JSPLAS)* has studied by modifying JPLAS for *Java programming*.

However, due to the fact that most web pages are written with the combination of *JavaScript*, *HTML*, and *CSS*, any type of the exercise problem currently available in JSPLAS may not be suitable for studying *web programming*. After studying each language separately, students must be able to relate them in the source code in *client-side programming* and *server-side programming*.

1.3 Contributions

Motivated by the above-mentioned problems, this thesis presents the answer code validation for code writing problem in Java with three features and code modification problems in JavaScript programming.

As the first contribution of the thesis, I implement the *answer code validation program* to help teacher works in assigning a lot of CWP assignments to students in a Java programming course in a university or professional school. This program automatically tests and verifies all the source codes that are made to pass the tests in a test code, and reports the number of tests that each source code could pass with the CSV file. By looking at the summary of the test results of all the students, the teacher can easily grasp the progress of students and grade them.

As the second contribution of the thesis, I propose the *intermediate state testing* in the *test code* for *fundamental algorithms* assignments. Against the assignment request, a student may use the library without implementing the correct logic/algorithm in the source codes. If a student implements a different logic or algorithm including the use of library, the conventional *test code* cannot find it. To improve problem-solving skills and develop strong foundations in algorithmic thinking, the *intermediate state testing* can check the randomly selected intermediate state of the important variables during the execution of the logic/algorithm.

As the third contribution of the thesis, I implement the *test data generation algorithm*. The fixed test data in the test code may 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 *test data generation algorithm* 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.

As the fourth contribution of the thesis, I implement the *naming rules checking function* in the *answer code validation program* for *CWP* in *JPLAS* for novice students, to master writing *readable codes* using proper names for variables, classes, and methods in *Java programming*. To master writing *readable codes* using proper names for variables, classes, and methods is crucial in *Java programming*, to improve *understandability* and *maintainability* for the novice students. The *naming rules checking function* finds the naming errors in the source code. It is also implemented in the answer platform so that a student can write a code while checking the rules. The students will master in writing *readable codes* using proper names for variables, classes, and methods at the early stage of programming studies.

As the fifth contribution of the thesis, I propose a *code modification problem (CMP)* as a new type of exercise problem in *JavaScript Programming Learning Assistant System (JSPLAS)*, to study *web client-side* and *server-side programming* using *JavaScript*. The goal of *CMP* is for the students to carefully read the source code and comprehend how to use the components and functions through modifying parameters, values, or messages. The CMP instance gives a source code using the functions to be studied and the screenshot of the web page generated by it. Then, it requests to modify the code to generate another web page given by the screenshot. The correctness of any answer is checked through *string matching* with the correct one.

1.4 Contents of Dissertation

The remaining part of this thesis is organized as follows.

Chapter 2 reviews the overview the web-based *Java Programming Learning Assistant System (JPLAS)*.

Chapter 3 presents the answer code validation program in JPLAS.

Chapter 4 presents the intermediate state testing for fundamental algorithm assignments.

Chapter 5 presents the dynamic test data generation algorithm for test codes.

Chapter 6 presents the naming rules checking function in code validation program.

Chapter 7 proposes the code modification problem (CMP) for client-side programming using JavaScript in JSPLAS.

Chapter 8 proposes the code modification problem (CMP) for server-side programming using JavaScript in JSPLAS.

Chapter 9 presents previous works related to this thesis.

Finally, Chapter 10 concludes this thesis with some future works.

Chapter 2

Overview of Java Programming Learning Assistant System (JPLAS)

This chapter provides an overview of the web-based *Java Programming Learning Assistant System (JPLAS)*.

2.1 System Architecture

JPLAS allows educators to provide programming exercises to the students, as depicted in Figure 2.1. This web-based architecture includes the *teacher support functions* and the *student support functions*. The teacher support functions encompass the practice problem generation, the assignment generation, and the student learning performance reference. The student support functions include the assignment view, the problem view, the problem solution, the hint reference, and the score reference by automatic marking.

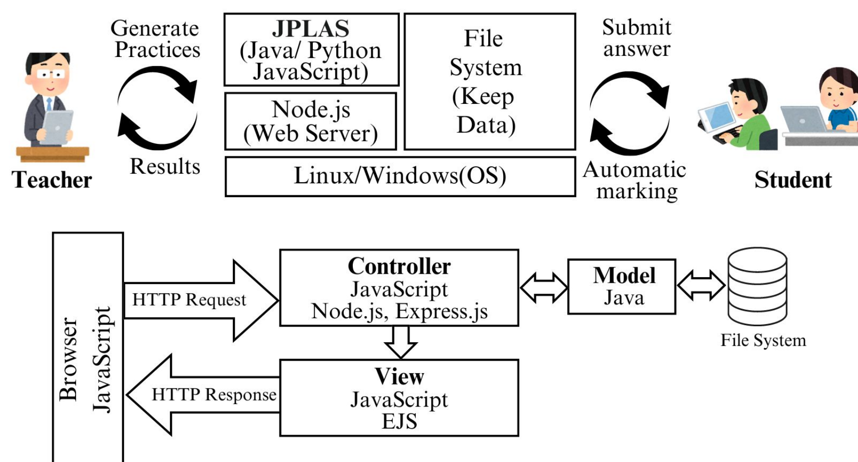


Figure 2.1: JPLAS architecture.

The software architecture of *JPLAS* follows the *MVC model* as the common architecture of the web application system. It is noted that the OS can be *Linux*, *Windows* or *Mac*. For the *model*

(M) part of the MVC model, *Java* is used to run *JUnit* for testing the answer source codes from students. The *file system* is used to manage the data where every data is provided by a file. For the *view (V)* part, *Embedded JavaScript (EJS)* is used as the template engine, rather than the usual template engine of *Express.js* (a web application framework for *Node.js*). For the *control (C)* part, *Node.js* and *Express.js* are adopted as the server-side technologies, where *JavaScript* is used to implement the programs.

2.1.1 Operation Flow

The operation flow of the answering function in JPLAS involves the following steps:

1. Assignment generation: a teacher collects Java source codes for exercises, and generate the files for new assignments by running the corresponding generator or manually.

2. Assignment distribution: the teacher distributes the assignment files to the students by using a file server.

3. Assignment answering: a student downloads the JPLAS Docker image from Docker Hub, installs it on his/her PC, and answers the problem instances in the assignments using a web browser without the Internet connections.

4. Answering result submission: a student submits the final answers to the teacher using the file server or an email.

5. Answering result upload: the teacher stores the answers from the students at the corresponding folders in his/her PC for managements.

2.1.2 Distributed Files

Table 2.1 outlines the files to be distributed to the students for assignment answering. These files are necessary for the problem view, the answer marking, and the answer storage.

Table 2.1: Files for distribution in JPLAS.

File name	Outline
css	CSS file for Web browser
index.html	HTML file for Web browser
page.html	HTML file for correct answers
jplas2015.js	js file for reading the problem list
distinction.js	js file for checking the correctness of answer
jquery.js	js file for use of jQuery
sha256	js file for use of SHA256
storage.js	js file for Web storage

2.1.3 Cheating Prevention

To prevent disclosing the correct answers to the students, their hash values by SHA256 [13] are distributed. In addition, to avoid generating the same hash values for the same correct answers, the assignment ID and the problem ID are concatenated with each correct answer before hashing. This ensures that identical correct answers for different blanks are converted to different hash values, maintaining independence between the blanks.

2.2 Implemented Problem Types

JPLAS provides various types of practices to cover different learning stages of Java programming:

- *Grammar-concept Understanding Problem (GUP)* asks a student to find the corresponding keyword appearing in the source code based on the provided question, where each question describes the meaning of the keyword.
- *Value trace problem (VTP)* questions the output values of the given source code that can be related to important variables and output messages.
- *Element Fill-in-blank Problem (EFP)* asks a student to fill the missing element in the blank with the appropriate words.
- *Code completion problem (CCP)* asks a student to fill several missing elements in the blank like EFP but does not specify the locations. Then, each question requests to find the location of a missing element in the code and fill in it with the proper one.
- *Code Writing Problem (CWP)* asks a student to write a source code that passes the test code given in the assignment where the correctness is verified by running them on *JUnit*.

For any exercise problem, the correctness of the answer from a student is verified automatically. The correctness of the student answer is checked through *string matching* with the correct one for *GUP*, *VTP*, *EFP*, and *CCP*, and through *unit testing* for *CWP*.

2.3 Code Writing Problem (CWP)

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.

2.3.1 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 for a 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.

2.3.2 Test Code

A *test code* is written by using the library in *JUnit*. The following *myAdd* class source code is used to explain how to write a test code. This *myAdd* class returns the summation of two integer arguments.

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

The following *test code* should be written to test the *plus* method in the *myAdd* class.

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

The *test code* imports the *JUnit* packages containing test methods at lines 1 and 2, and declares the *myAddTest* class at line 3. *@Test* at line 4 indicates that the succeeding *testPlus* method represents the test case, which describes the procedure for testing the output of the *plus* method. This test is performed as follows:

1. The *ma* object for the *myAdd* class in the source code is generated in the test code.

2. The *plus* method of the *ma* object is called with the arguments for the input data 1 and 4.

3. The result of the method *result* is compared with the expected output data 5 using the *assertEquals* method.

2.3.3 Answer Interface of CWP

Figure 3.2 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 completing 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.

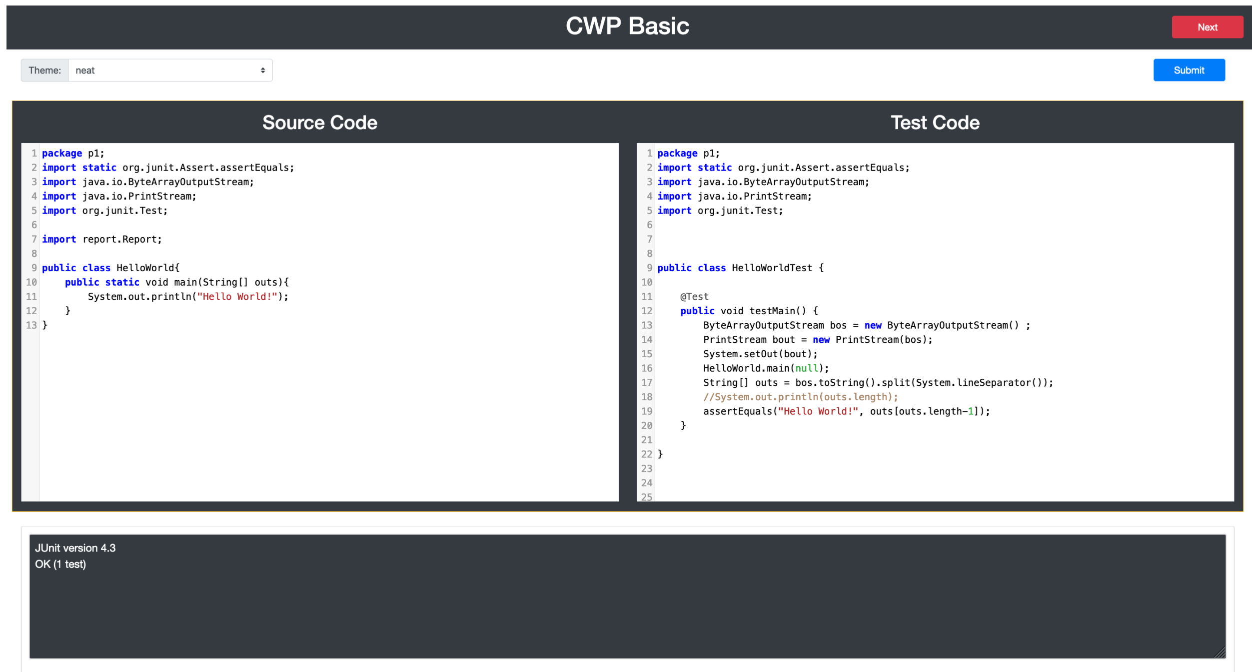


Figure 2.2: CWP answer interface.

2.4 Summary

This chapter overviewed the *Java programming Learning Assistant System (JPLAS)*. It discussed the software architecture of the JPLAS implementation, the answering function in JPLAS, several types of exercise problems, and the details of the *Code Writing Problem (CWP)*.

Chapter 3

Answer Code Validation Program

This chapter presents the implementation of the *answer code validation program* for the *code writing problem (CWP)* in *Java Programming Learning Assistant System (JPLAS)* [14].

3.1 Introduction

To assist *Java programming* learning of novice students, our group has developed the web-based *Java Programming Learning Assistant System (JPLAS)*. *JPLAS* provides various exercise problems at various levels to cultivate *code reading* and *code writing* skills of students. Among them, the *code writing problem (CWP)* asks a student to write a source code to pass the given *test code* where the correctness is verified by running them on *JUnit*.

Previously, we have implemented the *answer platform* to help students to solve CWP assignments effeciently by implementing the automatic runs of *code tesing* [4]. However, the teacher needs to manually run the test code and the source code one by one to verify a lot of source codes from students.

3.2 Previous Works 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*.

3.2.1 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.

3.2.2 JUnit for Unit Testing

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.

3.2.3 Test Code

A *test code* is written by using the library in *JUnit*. The following *myAdd* class source code is used to explain how to write a test code. This *myAdd* class returns the summation of two integer arguments.

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

The following *test code* should be written to test the *plus* method in the *myAdd* class.

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

The *test code* imports the *JUnit* packages containing test methods at lines 1 and 2, and declares the *myAddTest* class at line 3. *@Test* at line 4 indicates that the succeeding *testPlus* method represents the test case, which describes the procedure for testing the output of the *plus* method. This test is performed as follows:

1. The *ma* object for the *myAdd* class in the source code is generated in the test code.

2. The *plus* method of the *ma* object is called with the arguments for the input data 1 and 4.

3. The result of the method *result* is compared with the expected output data 5 using the *assertEquals* method.

3.2.4 CWP Answer Platform for Students

To assist students solving CWP assignments efficiently, we have implemented the CWP answer platform as a web application system using *Node.js*. Figure 3.1 illustrates the software architecture. It is noted that 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 instead of the database, 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.

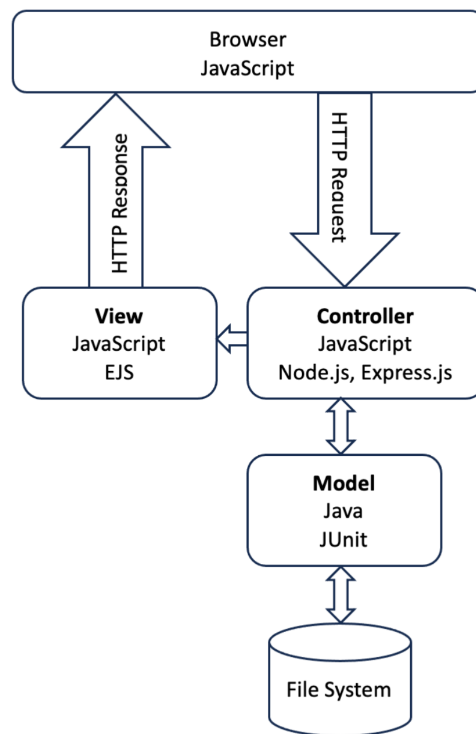


Figure 3.1: CWP software architecture.

Figure 3.2 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.

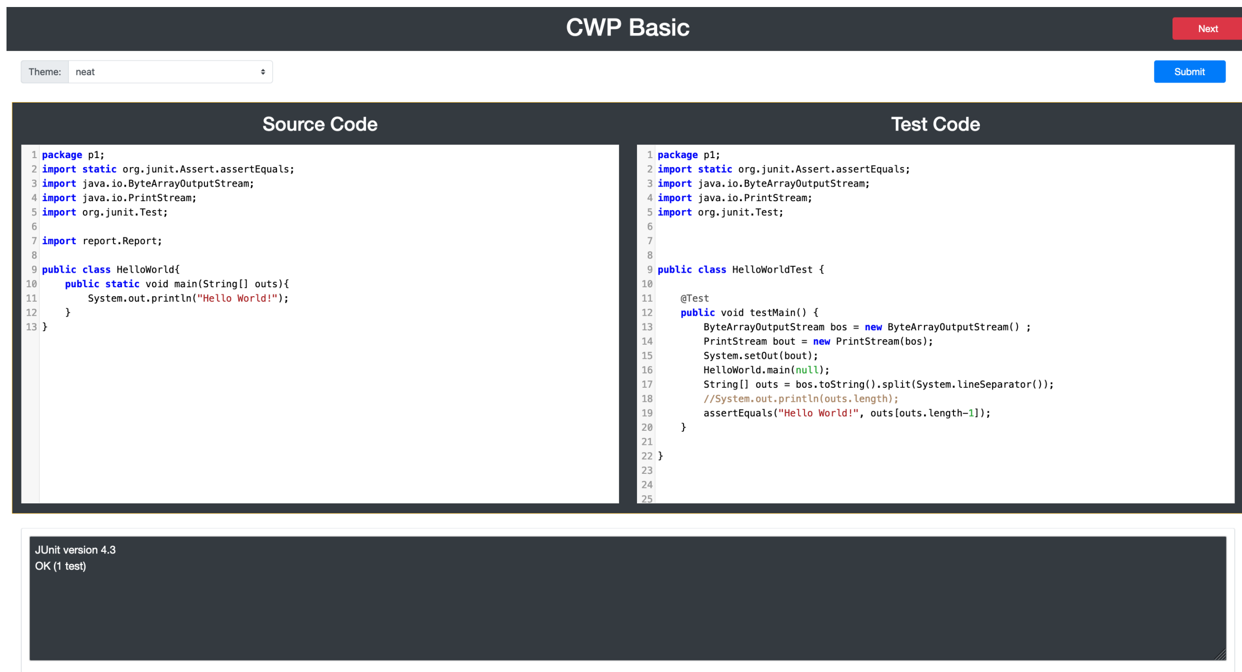


Figure 3.2: CWP answer interface.

3.3 Answer Code Validation Program for Teachers

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

3.3.1 Folder Structure in File System

Figure 3.3 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 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 answer code testing results including the JUnit logs will be recorded in the “output” folder. From them, the 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 CSV file.

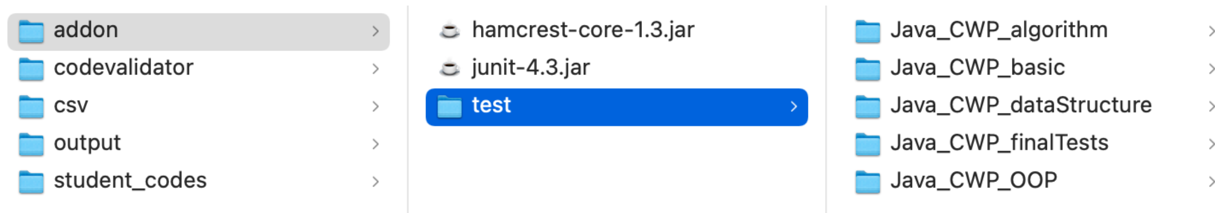


Figure 3.3: Designated file system for answer code validation program.

3.3.2 Procedure of Answer Code Validation

The following procedure is used for the answer code validation to check the correctness of the source codes from the students to help the teacher.

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.
2. The zip file is unzipped and stored in the corresponding folder under the "student_codes" folder inside the this project path.
3. The corresponding test code is stored in "addon/test" folder.
4. The program reads the test code.
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.

3.3.3 Example of Answer Code Validation Procedure

The example of folder structure and related files are illustrated in Figure 3.4. 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.

The summary of test results of an example assignment and the directory/file sample can be seen as follows.

No.	Question Type	Question Name	Total Tests	Correct	Failure
1	Java_CWP_basic	HelloWorld	2	2	0

3.3.4 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

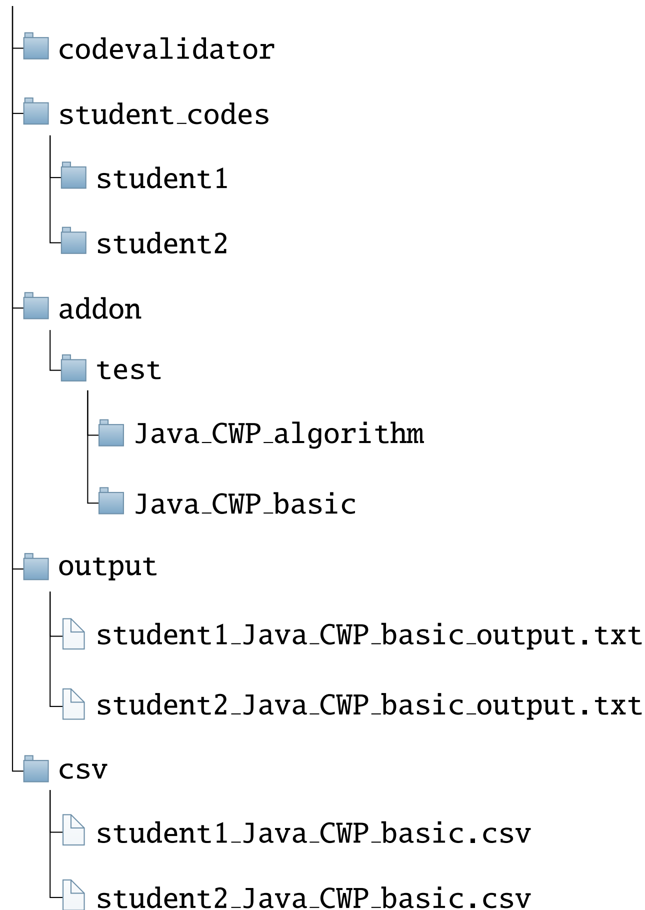


Figure 3.4: Example of file structures with folder hierarchy

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 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.

3.4 Evaluation

In this section, we evaluate the *answer code validation program* through applications to the Java programming course in Okayama University, Japan, in two years. The evaluation was conducted

to $2,861$ source codes from $12 - 55$ students, who were taking Java programming cource for 2022 and 2023. 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.

3.4.1 CWP Assignments

In this course, four groups of CWP assignments were prepared with test codes. The programming topics of them are *basic grammar*, *data structure*, *object oriented programming*, and *fundamental algorithms*. They are considered to be at different levels of Java programming study. In addition, two CWP assignments for the final examination in this course were prepared with the test codes. The topic of each group, and the program name and the number of tests in the test code for each CWP assignment can be seen in Table 3.1 for 2022 and Table 3.2 for 2023 in Java programming course at Okayama University.

Table 3.1: CWP assignments for 2022.

group topic	ID	program name	# of test cases
basic grammar	1	CodeCorrection1	3
	2	CodeCorrection2	3
	3	EscapeUsage	2
	4	HelloWorld	2
	5	Hexadecimal	2
	6	IfAndSwitch	4
	7	MaxItem	6
	8	MessageDisplay	2
	9	MinItem	7
	10	OctalNumber	2
	11	ReturnAndBreak	3
data structure	12	ArrayListImp	7
	13	HashMapDemo	6
	14	LinkedListDemo	7
	15	Que	5
	16	Stack	5
	17	TreeSetDemo	6
object oriented programming	18	Animal	3
	19	Animal1	3
	20	AnimalInterfaceUsage	3
	21	Author	3
	22	Book	3
	23	Book1	4
	24	BookData	5
	25	Car	4
	26	Circle	3
	27	GamePlayer	4
	28	MethodOverloading	3
	29	PhysicsTeacher	3
	30	Student	3
fundamental algorithms	31	BinarySearch	9
	32	BinSort	5
	33	BubbleSort	4
	34	BubbleSort1	5
	35	Divide	4
	36	GCD	5
	37	HeapSort	4
	38	InsertionSort	4
	39	LCM	5
	40	QuickSort1	5
	41	QuickSort2	4
	42	QuickSort3	5
	43	ShellSort	4
final examination	44	MakeArray	5
	45	PrimeNumber	2

Table 3.2: CWP assignments for 2023.

group topic	ID	program name	# of test cases
basic grammar	1	CodeCorrection1	3
	2	CodeCorrection2	3
	3	EscapeUsage	2
	4	HelloWorld	2
	5	Hexadecimal	2
	6	IfAndSwitch	4
	7	MaxItem	6
	8	MessageDisplay	2
	9	MinItem	7
	10	OctalNumber	2
	11	ReturnAndBreak	3
data structure	12	ArrayListImp	7
	13	HashMapDemo	6
	14	LinkedListDemo	7
	15	Que	5
	16	Stack	5
	17	TreeSetDemo	6
object oriented programming	18	Addition	4
	19	Animal	4
	20	InterfaceOOP	3
	21	Person	5
	22	PersonName	3
	23	PolymorphismOverride	4
	24	Vehicle	3
fundamental algorithms	25	BinarySearch	6
	26	BinSort	3
	27	BubbleSort	2
	28	Divide	4
	29	GCD	5
	30	HeapSort	4
	31	InsertionSort	3
	32	LCM	5
	33	QuickSort	3
	34	ShellSort	3
final examination	35	DayofWeek	4
	36	Rectangle, Circle	8

3.4.2 Solution Results

The CWP assignments were assigned to 12-55 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.

3.4.2.1 Solution Results of Individual Students

First, we analyze the solution results of the students individually. Table 4.2 provides the number of students to submit answer codes and the average rate of passed tests among all the tests for each assignment group. The table shows that the number of students who submitted answer codes and the average rate are both varied among the assignment groups. This is because the difficulty levels and learning stages are much different among them, whereas every student has to answer the final examination.

Table 3.3: Number of submitted students and average passing rate in each group.

group topic	2022		2023	
	# of students	ave. rate (%)	# of students	ave. rate (%)
basic grammar	39	82.34	55	96.31
data structure	20	72.78	51	89.00
object oriented programming	16	84.38	53	99.64
fundamental algorithms	12	81.35	47	94.29
final examination	43	62.79	55	97.84

Figure 3.5 shows the average rate of passing the tests (%) for the CWP assignment group by each student in 2022. Most of the students achieved over the 90% rate for each assignment group. The total of 36 cases among 130 could not reach 50% because of the insufficiency of programming knowledge and skills of the students.

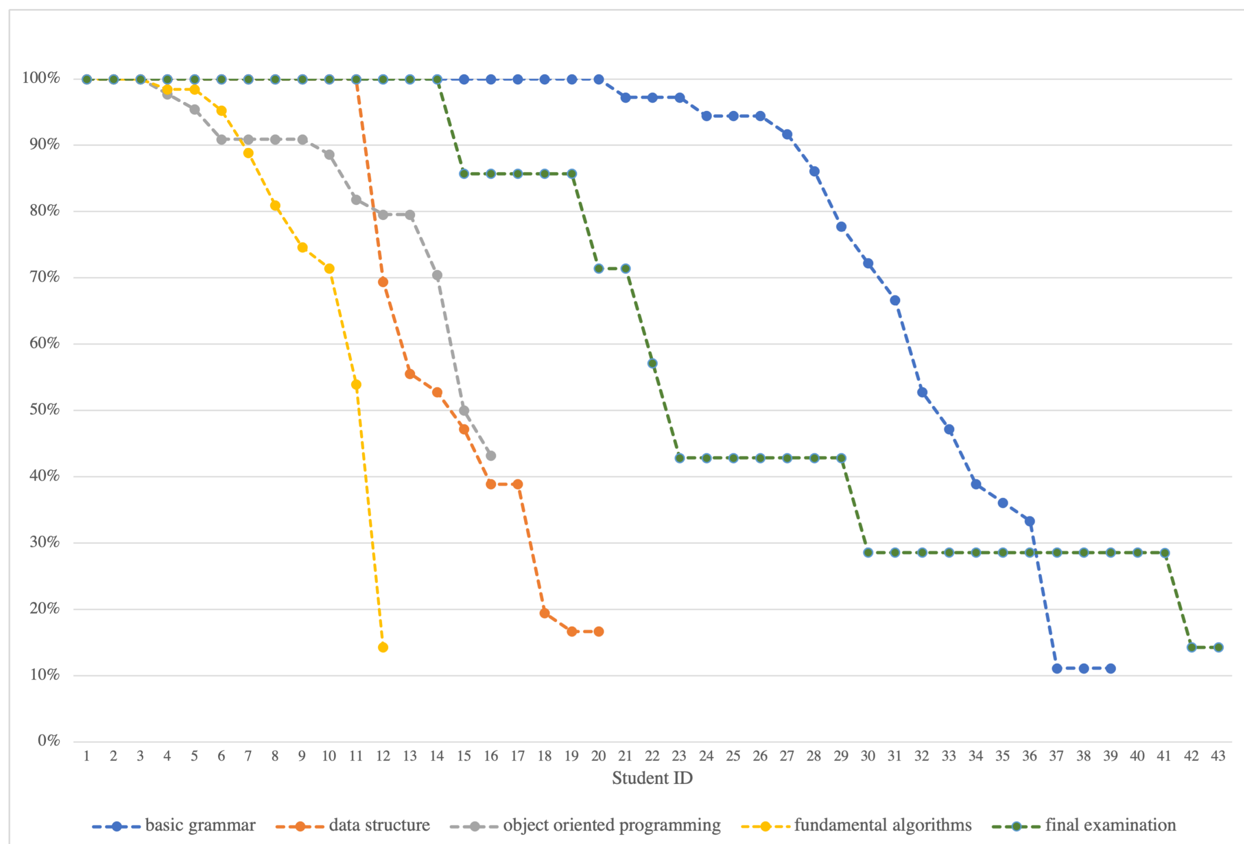


Figure 3.5: Solution results for individual students (2022).

Figure 3.6 shows the average rate of passing the tests (%) for the CWP assignment group by each student in 2023. Most of the students achieved 100% correct rate for each assignment group. Only 4 students could not reach 50%.

Therefore, the generated CWP assignments are appropriate levels for self-study by novice students.

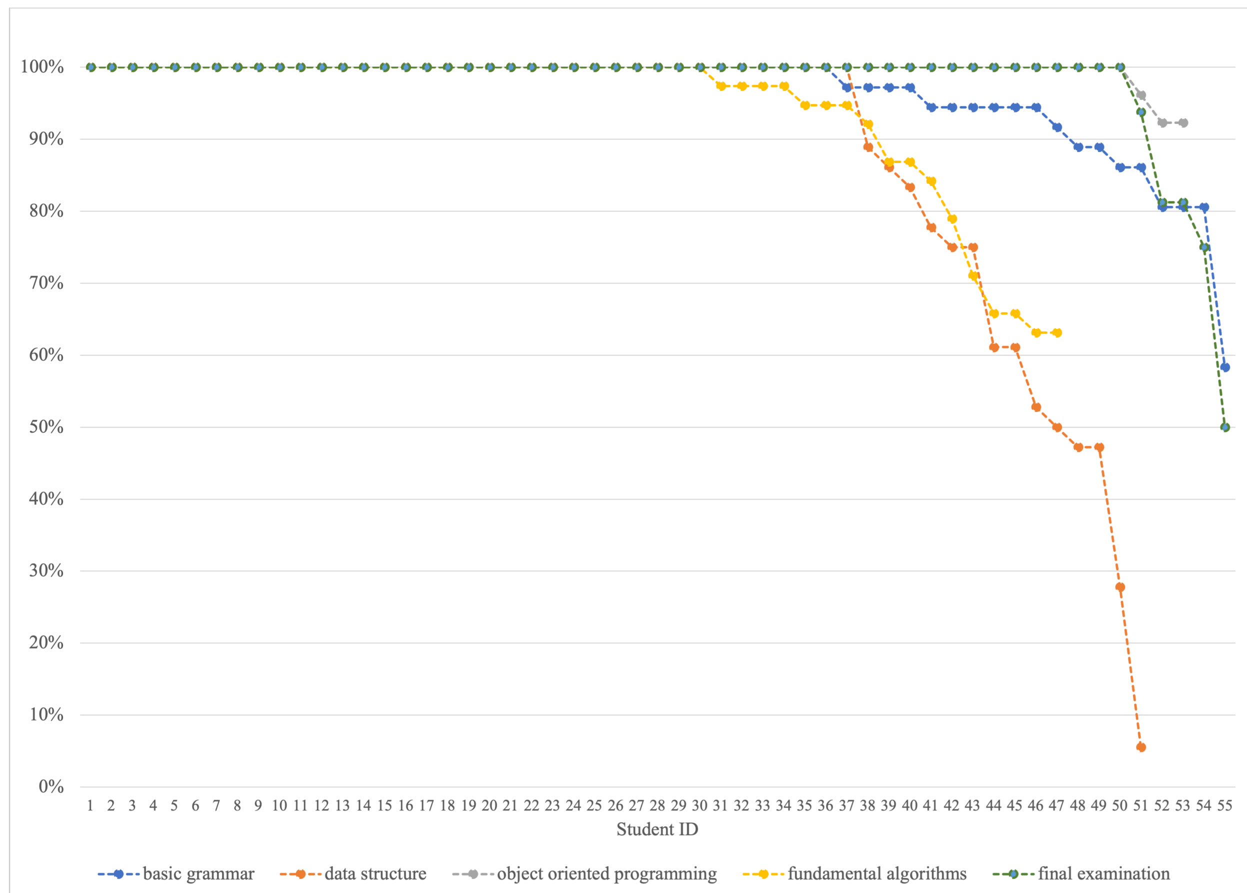


Figure 3.6: Solution results for individual students (2023).

3.4.2.2 Results of Individual Assignments

Next, we examine the solution results of the individual CWP assignments.

Figure 3.7 shows the rate of passing tests (%) for each of the 45 CWP assignments in 2022 Java programming course. According to the results, the assignment at ID=24 resulted in the lowest correct rate of 48.75%, and the assignments at ID=4, ID=20, and ID=31 achieved the highest correct rate of 100%. The reason of the high correct rate is that the test cases in the test codes for these assignments are simple and easy. Thus it is also easy to write the source codes for the students. On the other hand, the reason of the low correct rate is that the test code is a little bit long where the test cases can be difficult for the students. Therefore, to improve the correct rate, it will be necessary to modify the test cases as simple as possible, which will be in our future works.

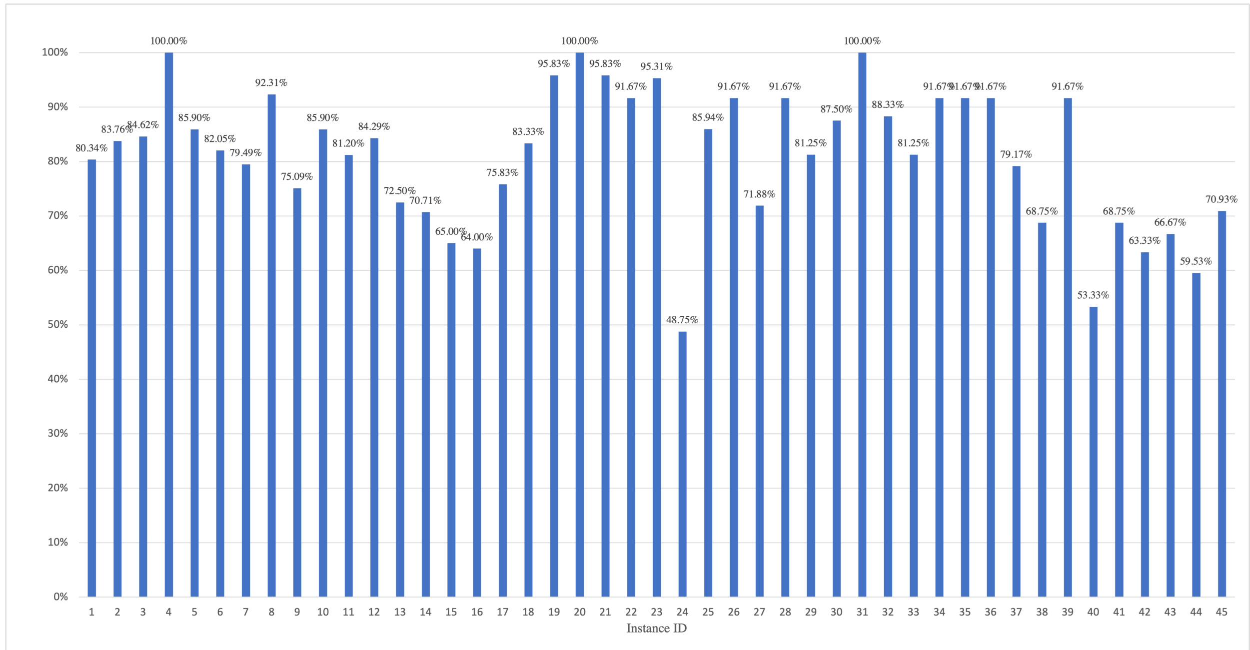


Figure 3.7: Solution results for individual assignments (2022).

Figure 3.8 shows the rate of passing tests (%) for each of the 36 CWP assignments in 2023 Java programming course. According to the results, 10 assignments achieved the highest correct rate of 100% and most of the assignments achieved the correct rate of over 90%. Therefore, the students did really well in the class of 2023 Java programming course.

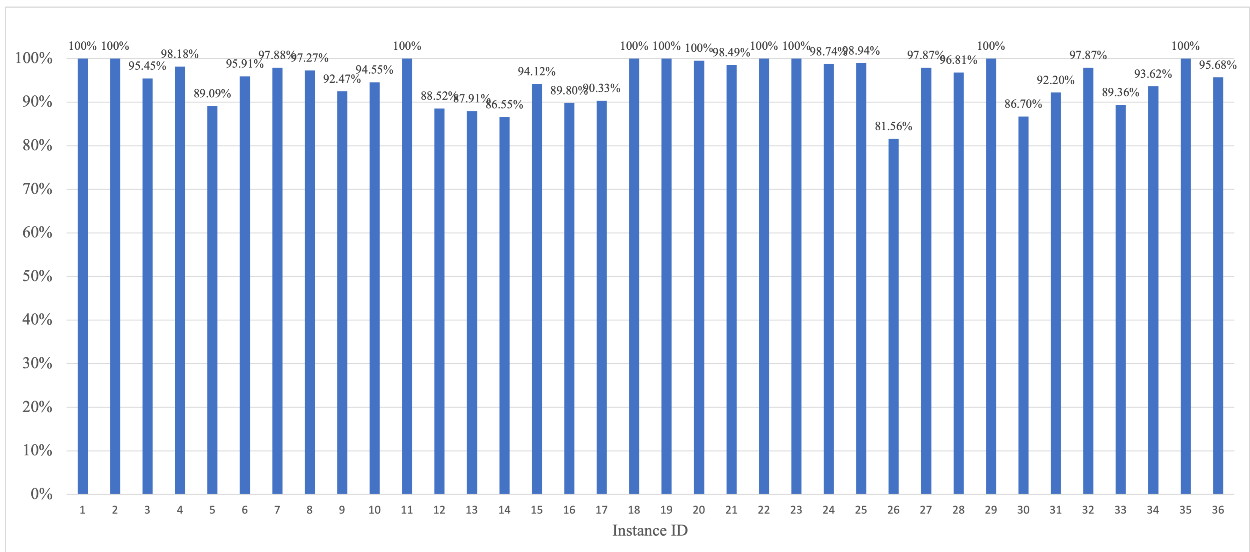


Figure 3.8: Solution results for individual assignments (2023).

3.4.2.3 Reducing Teacher Workload

We would like to discuss the load reduction results to examine the learning effectiveness of the proposal in this section. Previously, the teacher needs to repeat the following steps for each of the 2,861 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.

However, with the code validation program, the teachers need to paste the student source codes in the project folder. By running the answer code validation program, the teachers can easily check the correctness of the source codes from the students. The number of click operations can also be reduced and Table 3.4 shows the CPU time of the proposal for each topic to show the effectiveness of the answer code validation program. Therefore, the advantage of the proposal is to reduce manual operations by a teacher.

Table 3.4: Load reduction results.

group topic	2022		2023	
	# of source codes	CPU Time (mins)	# of source codes	CPU Time (mins)
basic grammar	429	7.46	605	10.53
data structure	120	2.27	306	5.80
object oriented programming	208	3.82	371	6.81
fundamental algorithms	156	2.53	470	7.62
final examination	86	0.87	110	1.2
total	999	16.95	1862	31.96

3.5 Summary

This chapter presented the implementation of the *answer code validation program* to help the teacher in marking a lot of assignments for the *code writing problem* in the *Java programming learning assistant system (JPLAS)*. This program tests all the source codes from the students at once for each assignment automatically, and reports the number of tests that each source code passed with the CSV file. For evaluations, this program was applied to $2,861$ source codes from 12-55 students to CWP assignments in the Java programming course in Okayama University for two years. The results confirms the validity and effectiveness of the proposal.

Chapter 4

Intermediate State Testing for Fundamental Algorithm Assignments

This chapter presents the *intermediate state testing* for *fundamental algorithms* assignments in *code writing problem (CWP)* of *Java Programming Learning Assistant System (JPLAS)* [15].

4.1 Introduction

In *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.

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 given test code. The test case acts as a reference or specification for the expected behavior of the source code. By running the test code against the source code, a student can quickly evaluate the correctness and functionality of the source code implementation. If the source code passes all the test cases, it is an indication that the code is likely correct and performs as expected.

However, the test data in a test code is usually only one type and fixed. 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.

4.2 Test Code

In this section, we review the *test code* for *code writing problem (CWP)* in *JPLAS.*

4.2.1 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 test method

in the library whose name starts with "assert". In the case of *CWP*, the test code often 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.

4.2.2 Test Code

A *test code* is created by using the *JUnit* library. The *BubbleSort* class in Figure 5.1 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.

```
1  package p3;
2  public class BubbleSort {
3     public int[] sort(int[] array) {
4        int n = array.length;
5        for (int i = 0; i < n − 1; i++) {
6           for (int j = 0; j < n − i − 1; j++) {
7              if (array[j] > array[j + 1]) {
8                 int temp = array[j];
9                 array[j] = array[j + 1];
10                array[j + 1] = temp;
11             }
12          }
13       }
14    return array;
15    }
16 }
```

Figure 4.1: Example source code for *BubbleSort*.

The *test code* in Figure 5.2 is designed for testing the *sort* method in the *BubbleSort* class in Figure 5.1.

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. 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.

4.3 Intermediate State Testing for Logic and Algorithms

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

```
1  package p3;
2  import org.junit.Test;
3  import static org.junit.Assert.*;
4  import java.util.Arrays;
5  public class BubbleSortTest {
6     @Test
7     public void testSort() {
8        BubbleSort bSort = new BubbleSort();
9        int[] codeInput1 = {8,7,4,1,5,9};
10       int[] codeOutput = bSort.sort(codeInput1);
11       int[] expOutput = {1,4,5,7,8,9};
12       try {
13          assertEquals("Test1:",Arrays.toString(expOutput),Arrays.toString(codeOutput));
14       } catch (AssertionError ae) {
15          System.out.println(ae.getMessage());
16       }
17    }
18 }
```

Figure 4.2: Example test code for BubbleSort.

4.3.1 Limitation of Current Test Code

First, we discuss the limitations of the above example *test code* in Figure 5.2.

4.3.1.1 Library Use

The limitation of the current test code lies in use of a library for implementing the logic or algorithm. A student may use the library class/method 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 the 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 4.3 shows this example for using a library without implementing the algorithm. The current test code cannot check it.

```
1  package p3;
2  import java.util.Arrays;
3  public class BubbleSort {
4     public static int[] sort(int[] a) {
5        Arrays.sort(a);
6        return a;
7     }
8  }
```

Figure 4.3: Example source code for using *library*.

4.3.1.2 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 4.4 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 is the same for any sorting algorithm. 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.

```
1  package p3;
2  public class BubbleSort {
3     public static int[] sort(int[] a) {
4        int n = a.length;
5        for (int i = 0; i < n − 1; i++) {
6           int minIndex = i;
7           for (int j = i + 1; j < n; j++) {
8              if (a[j] < a[minIndex]) {
9                 minIndex = j;
10             }
11          }
12          int temp = a[i];
13          a[i] = a[minIndex];
14          a[minIndex] = temp;
15       }
16       return a;
17 }
```

Figure 4.4: Example source code for implementing of *different algorithm*.

To address these limitations, we propose the *intermediate state testing* in the test code in the following subsections.

4.3.2 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 4.5 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 4.6 shows the *source code* to pass the test code in Figure 4.5. 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.

```
1  package p3;
2  import static org.junit.Assert.*;
3  import org.junit.Test;
4  import java.util.Arrays;
5  public class BubbleSortTest {
6     //intermediate state testing
7     @Test
8     public void testSortIteration() {
9        BubbleSort bSort = new BubbleSort();
10       int[] codeInput1 = {5,2,8,1,9};
11       int codeInput2 = 2;
12       int[] codeOutput = bSort.sort(codeInput1, codeInput2);
13       int[] expOutput = {2,1,5,8,9};
14       try {
15          assertEquals("Test 1:",Arrays.toString(expOutput),Arrays.toString(codeOutput));
16       } catch (AssertionError ae) {
17          System.out.println(ae);
18       }
19    }
20    //final state testing
21    @Test
22    public void testSort() {
23       BubbleSort bSort = new BubbleSort();
24       int[] codeInput1 = {8,7,4,1,5,9};
25       int[] codeOutput = bSort.sort(codeInput1);
26       int[] expOutput = {1,4,5,7,8,9};
27       try {
28          assertEquals("Test 2:",Arrays.toString(expOutput),Arrays.toString(codeOutput));
29       } catch (AssertionError ae) {
30          System.out.println(ae.getMessage());
31       }
32    }
33 }
```

Figure 4.5: Example test code for intermediate state testing.

```
34  package p3;
35  public class BubbleSort {
36      // intermediate state
37      public static int[] sort(int[] a, int iteration) {
38          int i, j, temp;
39          for (i = 0; i < iteration; i++) {
40              for (j = 0; j < a.length − 1; j++) {
41                  if (a[j] > a[j + 1]) {
42                      temp = a[j + 1];
43                      a[j + 1] = a[j];
44                      a[j] = temp;
45                  }
46              }
47          }
48          return a;
49
50      //final state
51      public static int[] sort(int[] a) {
52          int temp = 0;
53          for(int i=0; i < a.length; i++){
54              for(int j=1; j < (a.length − i); j++){
55                  if(a[j−1] > a[j]){
56                      temp = a[j−1];
57                      a[j−1] = a[j];
58                      a[j] = temp;
59                  }
60              }
61          }
62          return a;
63      }
64  }
```

Figure 4.6: Example source code for intermediate state testing.

By observing the intermediate states, we can gain insights into whether the code 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 the 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 source 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, where 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.

4.4 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 590 source codes from

59 students to 10 CWP assignments. These source codes were processed using the *answer code validation program*, which automatically verified the source 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.

4.4.1 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. 10 CWP assignments were prepared for studying *fundamental algorithms* topic, considering their levels in Java programming study. The corresponding test codes were made and given to the students. Then, a total of 120 source codes were submitted from the students in 2022, and a total of 470 codes were in 2023.

Table 5.1 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 10 CWP fundamental algorithms 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 4.1: CWP assignments.

group topic	ID	class name	# of test cases	# of students	
				2022	2023
fundamental algorithms	1	BinarySearch	9	12	47
	2	BinSort	5	12	47
	3	BubbleSort	4	12	47
	4	Divide	4	12	47
	5	GCD	5	12	47
	6	HeapSort	4	12	47
	7	InsertionSort	4	12	47
	8	LCM	5	12	47
	9	QuickSort	5	12	47
	10	ShellSort	4	12	47

4.4.2 Individual Assignments Results

First, the solution results of the individual CWP assignments are analyzed. Figure 5.5 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.

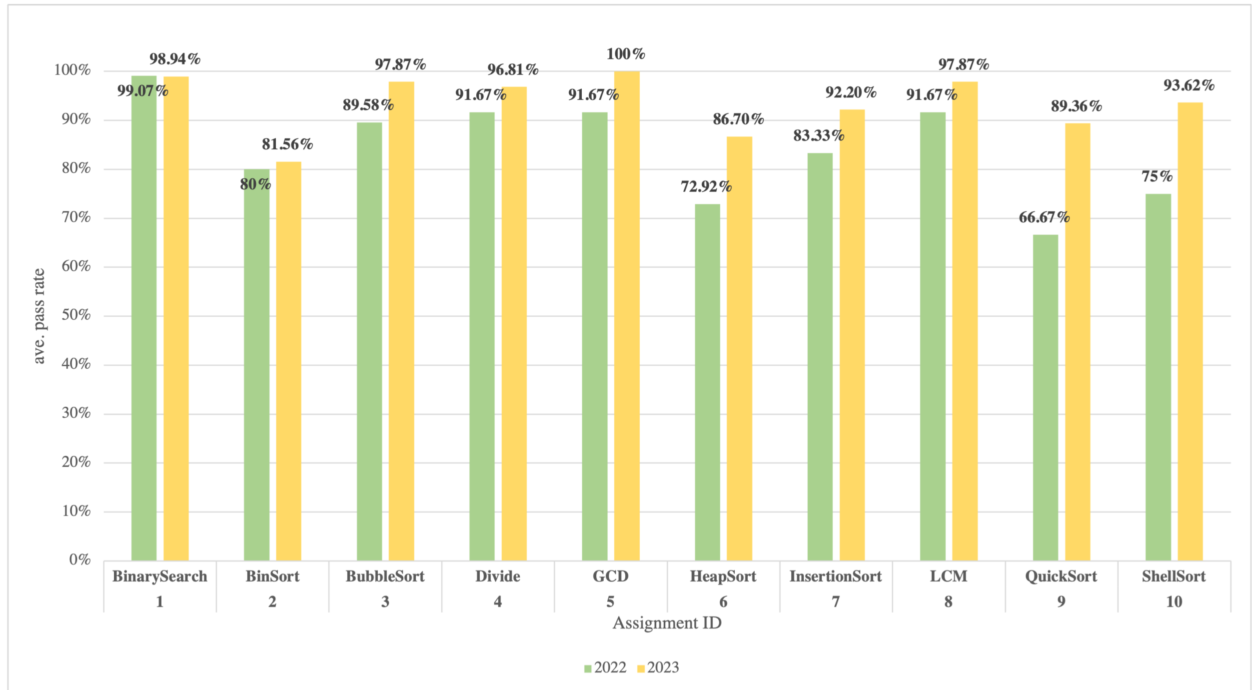


Figure 4.7: Results of individual assignments.

By comparing the two average pass rates for the "fundamental algorithms" group, the effectiveness of the proposed test code is confirmed. 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=1, the library method "Arrays.binarySearch()" is used. For the assignments with ID=2, 3, 6, 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.

4.4.3 Individual Students Results

Next, we analyze the solution results of the individual students for the 10 CWP assignments. Table 4.2 provides the number of submitted answer codes from the students and the average CPU time for each assignment group for the two years.

Table 4.2: Number of students and results in each group.

group topic	# of students		# of source codes		CPU Time (mins)	
	2022	2023	2022	2023	2022	2023
fundamental algorithms	12	47	120	470	1.95	7.64

Figure 4.8 presents the solution results of the individual students in 2022 and 2023. In 2022, only 12 students answered to the "fundamental algorithms" assignments. It seems that many stu-

dents 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, 47 students among them tried to answer the "fundamental algorithms" assignments.

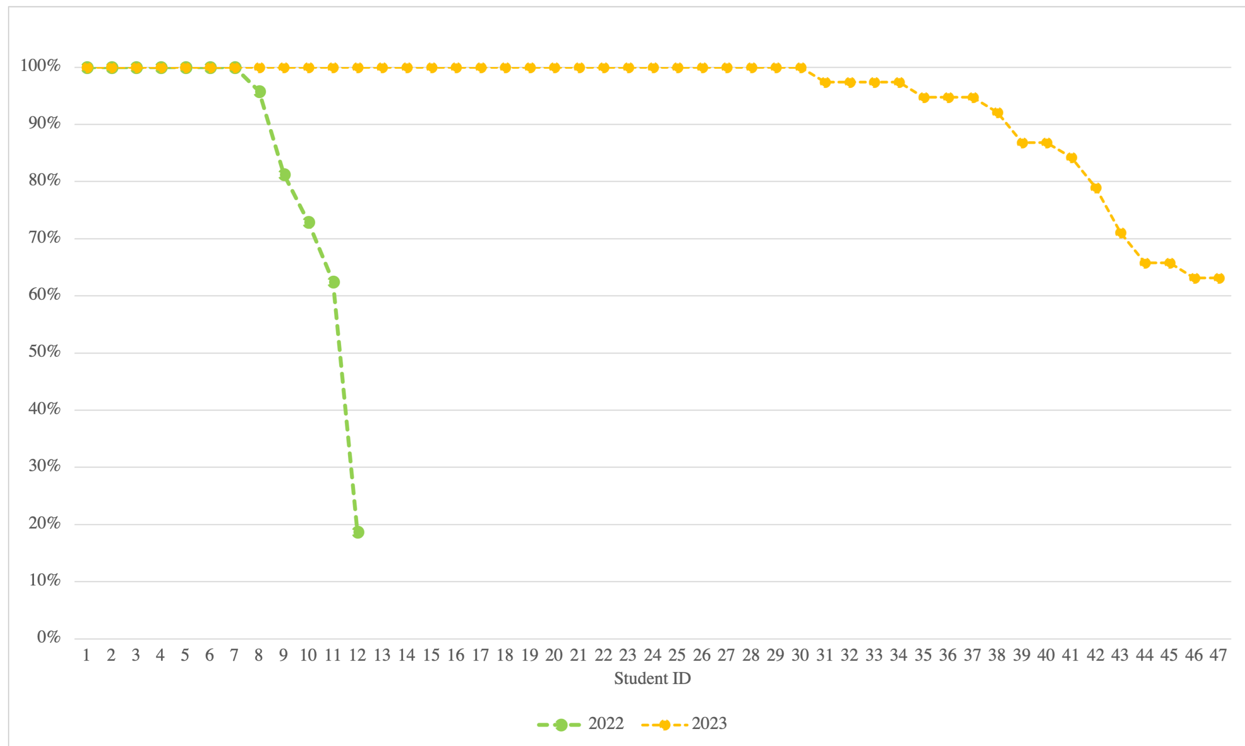


Figure 4.8: Solution results for individual students.

When examining the results of the individual students, the "fundamental algorithms" assignments posed more challenges to them, as evidenced by the lower average pass rates. This result highlights the need of emphasizing algorithmic programming skills to students.

4.5 Summary

This chapter presented the *intermediate state testing* for fundamental algorithms assignments. If a student implements a different logic or algorithm including the use of library, the conventional test code cannot find it. To improve problem-solving skills and develop strong foundations in algorithmic thinking, the intermediate state testing can check the randomly selected intermediate state of the important variables during the execution of the logic/algorithm. The results confirms the validity and effectiveness of the proposal.

Chapter 5

Dynamic Test Data Generation Algorithm

This chapter presents the *dynamic test data generation algorithm* for *code writing problem (CWP)* in *Java Programming Learning Assistant System (JPLAS)* [15].

5.1 Introduction

In *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*. This *test code* approach is 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.

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* described in the given test code. A test case acts as a reference or specification for the expected behavior of the source code. By running the *test code* with the *source code* on *JUnit*, a student can quickly verify the correctness and functionality of the implemented source code. If the source code passes all the test cases, the code is likely correct and performs as expected.

However, the test data in a test code is usually only one case and fixed. 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.

5.2 Test Code

In this section, we review the *test code* for *code writing problem (CWP)* in *JPLAS.*

5.2.1 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. Using *JUnit*, performing a test is simplified through the test method in the library whose name starts with "assert". In the case of *CWP*, the test code often adopts

the "assertEquals" method that compares the output data that will be generated by executing the source code with the expected output data for a given set of input data.

5.2.2 Test Code

A *test code* is created by using the *JUnit* library. The *BubbleSort* class in Figure 5.1 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.

```
1  package p3;
2  public class BubbleSort {
3     public int[] sort(int[] array) {
4        int n = array.length;
5        for (int i = 0; i < n − 1; i++) {
6           for (int j = 0; j < n − i − 1; j++) {
7              if (array[j] > array[j + 1]) {
8                 int temp = array[j];
9                 array[j] = array[j + 1];
10                array[j + 1] = temp;
11             }
12          }
13       }
14    return array;
15    }
16 }
```

Figure 5.1: Example source code for *BubbleSort*.

The *test code* in Figure 5.2 is designed for testing the *sort* method in the *BubbleSort* class in Figure 5.1.

```
1  package p3;
2  import org.junit.Test;
3  import static org.junit.Assert.∗;
4  import java.util.Arrays;
5  public class BubbleSortTest {
6     @Test
7     public void testSort() {
8        BubbleSort bSort = new BubbleSort();
9        int[] codeInput1 = {8,7,4,1,5,9};
10       int[] codeOutput = bSort.sort(codeInput1);
11       int[] expOutput = {1,4,5,7,8,9};
12       try {
13          assertEquals("Test1:",Arrays.toString(expOutput),Arrays.toString(codeOutput));
14       } catch (AssertionError ae) {
15          System.out.println(ae.getMessage());
16       }
17    }
18 }
```

Figure 5.2: Example test code for BubbleSort.

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 declares the *BubbleSortTest* class. 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.

5.3 Test Data Generation Algorithm

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

5.3.1 Limitation of Current Test Code

First, we discuss the limitations of the current test code.

5.3.1.1 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 5.3 shows one of such examples of the fixed output data for the above *test code* in Figure 5.2.

```
1 package p3;
2 public class BubbleSort {
3    public static int[] sort(int[] array) {
4       int[] res = {1,4,5,7,8,9};
5       return res;
6    }
7 }
```

Figure 5.3: Example source code for *fixed data output*.

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 the 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 data type of the test data in the test code, and will generate the new data to replace the fixed test data. This algorithm can reduce the risk of cheating and improve the validity and reliability of the programming assignments.

5.3.2 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 5.4 shows the test code with the standard format for testing the *sort* method in the *BubbleSort* class in Figure 5.1. 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`.

```
65 package p3;
66 import static org.junit.Assert.*;
67 import org.junit.Test;
68 import java.util.Arrays;
69 public class BubbleSortTest {
70    @Test
71    public void testSort() {
72       BubbleSort bSort = new BubbleSort();
73       int[] codeInput1 = {8,7,4,1,5,9};
74       int[] codeOutput = bSort.sort(codeInput1);
75       int[] expOutput = {1,4,5,7,8,9};
76       try {
77          assertEquals("Test1:",Arrays.toString(expOutput),Arrays.toString(codeOutput));
78       } catch (AssertionError ae) {
79          System.out.println(ae.getMessage());
80       }
81    }
```

Figure 5.4: Example of test code with standard format

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

5.3.2.2 Replacing Test Data

Once a new test data is generated, the algorithm replaces the original test data in the test code with this newly generated test data. This ensures that each test case is executed with different input values. There are some limitations for complex data types.

5.3.2.3 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.

5.4 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.

5.4.1 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 for the two groups of *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 5.1 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 5.1: CWP assignments.

group topic	ID	class name	# of test cases	# of students	
				2022	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

5.4.2 Individual Assignments Results

First, the solution results of the individual CWP assignments are analyzed. Figure 5.5 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. Here, 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 was 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

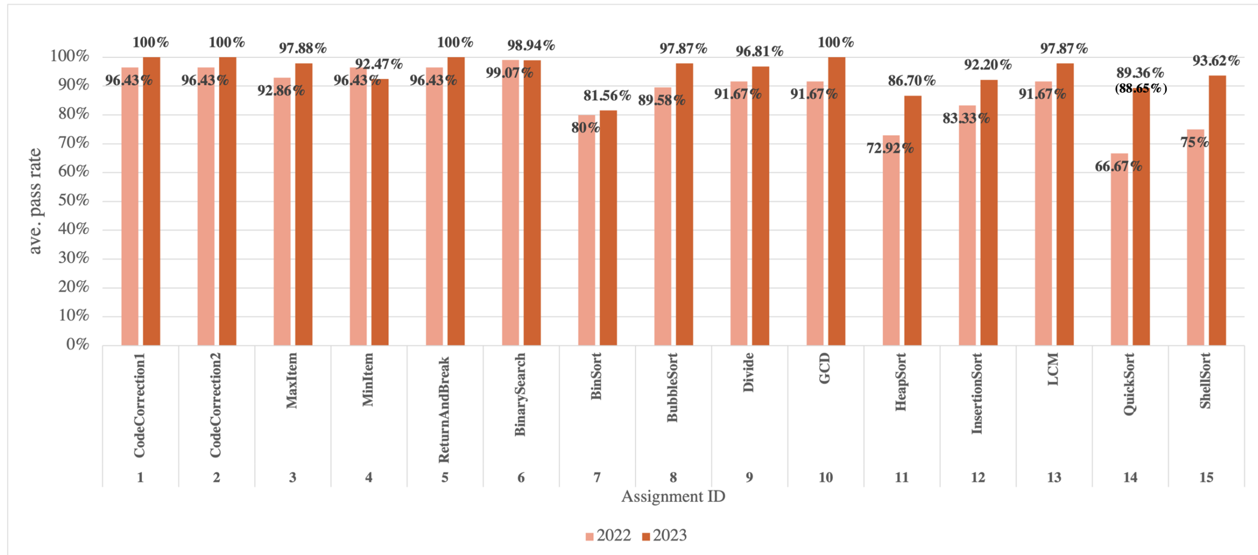


Figure 5.5: Results of individual assignments.

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.

5.4.3 Individual Students Results

Next, we analyze the solution results of the individual students for the 15 CWP assignments.

Figure 5.6 and 5.7 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.

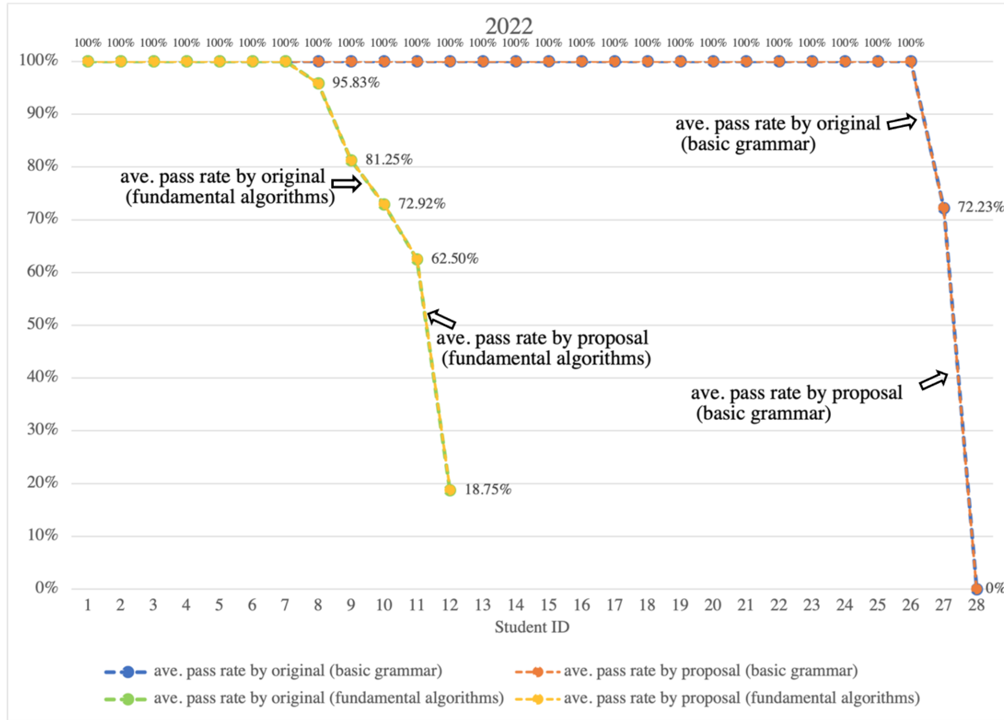


Figure 5.6: Solution results for individual students in 2022.

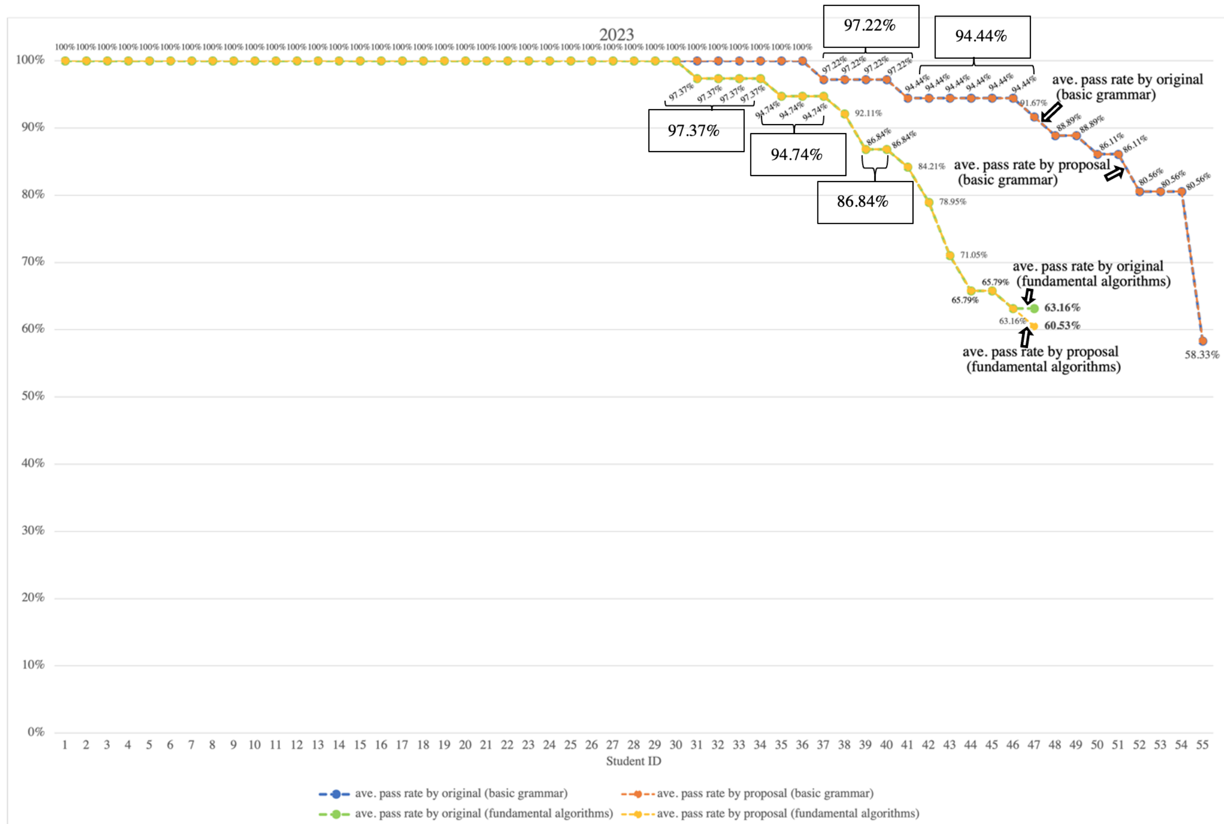


Figure 5.7: Solution results for individual students in 2023.

5.5 Summary

This chapter presented the *dynamic test data generation algorithm* for code writing problem (CWP) assignments. 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, we enhanced the validity of CWP assignments and reduced the risk of cheating. We evaluated 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. The results confirmed the validity and effectiveness of the proposal.

Chapter 6

Naming Rules Checking Function in Code Validation Program

This chapter presents the *naming rules checking function* in the *code validation program* for *Java Programming Learning Assistant System (JPLAS)* [17].

6.1 Introduction

Generally, to master *readable codes* is critical at the programming learning stage in order to develop good coding habits and skills. A readable code can be realized by following *coding rules*, which may be composed of *naming rules*, *coding styles*, and *potential problems*. *Coding rules* [18] represent a set of rules or conventions for producing high quality source codes. By following *coding rules*, the uniformity of the code can be maintained, which enhances the *readability*, *maintainability*, and *scalability*. Developing good coding habits and skills are important not only for academic settings but also for professional software development environments. Especially, *naming conventions checks* are crucial in *Java programming* educations. Therefore, to foster *code writing* abilities of novice students, it is essential to master how to write *readable codes* using proper names for variables, classes, and methods. They can improve *understandability* and *maintainability* of source codes which will help fast and easy completions of programming assignments by students.

Although a teacher usually explains the naming conventions during classes in the *Java programming* courses, students may fail to follow them and make codes far from readable ones. However, if the teacher wants to check checking naming errors from the submitted source codes, he/she needs to check them manually, which is time-consuming.

6.2 Previous Works of Code Writing Problem in JPLAS

In this section, we review our previous works of the *code writing problem (CWP)* and *answer code validation program* for teachers in *JPLAS*.

6.2.1 Overview of Code Writing Problem (CWP)

In *code writing problem (CWP)*, the student needs to write *Java programming* source code that can satisfy the predefined test cases contained in the test code. *Source code testing* is employed to

validate the accuracy and correctness of the source codes by utilizing *JUnit* to execute the test code alongside with the source code. The students should follow the detailed specifications provided in the test code in order to ensure the accurate implementation of the source code.

The following steps show the generation steps to generate a CWP instance for the teacher:

1. Create the problem statement with the input data for the new assignment
2. Collect the model source code as the solution for this assignment
3. Run the model source code to obtain the expected output
4. Write the test code using the input and output data to make sure the code works properly and include the necessary details to write the source code for the students
5. Register the test code with the statement and test cases for the new assignment

6.2.2 Answer Code Validation Program for Teachers

The *answer code validation program* for CWP in *JPLAS* has been implemented to help teachers in checking the correctness of the source codes submitted from the students. This program facilitates the automatic testing of all the student source codes stored in a folder for a specific assignment using the same test code. The process involves the following steps:

1. The zip file containing the source codes for each assignment using one *test code* is downloaded. It is noted that a teacher usually uses an e-learning system such as *Moodle* in the programming course.
2. The zip file is then unzipped and stored in the corresponding folder under the "student_codes" directory within the project directory.
3. The corresponding test code is stored in the "addon/test" folder.
4. The program proceeds to read each source code in the "student_codes" folder. It then runs the test code with the source code on *JUnit*, saving the test result in a text file within the "output" folder. This process repeats until all the source codes in the folder are tested.
5. Finally, the program generates a summary of the test results for all the source codes in a CSV file and saves it in the "csv" folder.

6.3 Naming Rule Checking Function

In this section, we present the implementation of the *naming rule checking function* to help the teacher in finding the naming errors from a lot of source codes from students. We consider four naming rules to write a *readable code* and implement the *naming rule checking function* to check whether the given code follows them. Basically, this function is implemented by modifying the naming rules testing program in [18] so that it can check all the source codes in a folder automatically.

6.3.1 File Structures with Folder Hierarchy

Figure 6.1 shows the folder structure in the file system for the *naming rule checking function*. The "src" folder contains the required Java programs necessary for checking the source codes and generating the result reports. Within the "student_codes" directory, the course folders contains various assignments such as Basic Grammar, with subfolders for each student (e.g., "Student1"), which hold their respective *Java* files (e.g., "helloWorld.java"). The function stores the text files of the naming rule checking results, including the student ID and assignment name in the "output" folder. From them, the function can produce the CSV files, which provide a summary of the results for all students and individual assignments, enabling the teachers to easily check the results, into the "csv" folder.

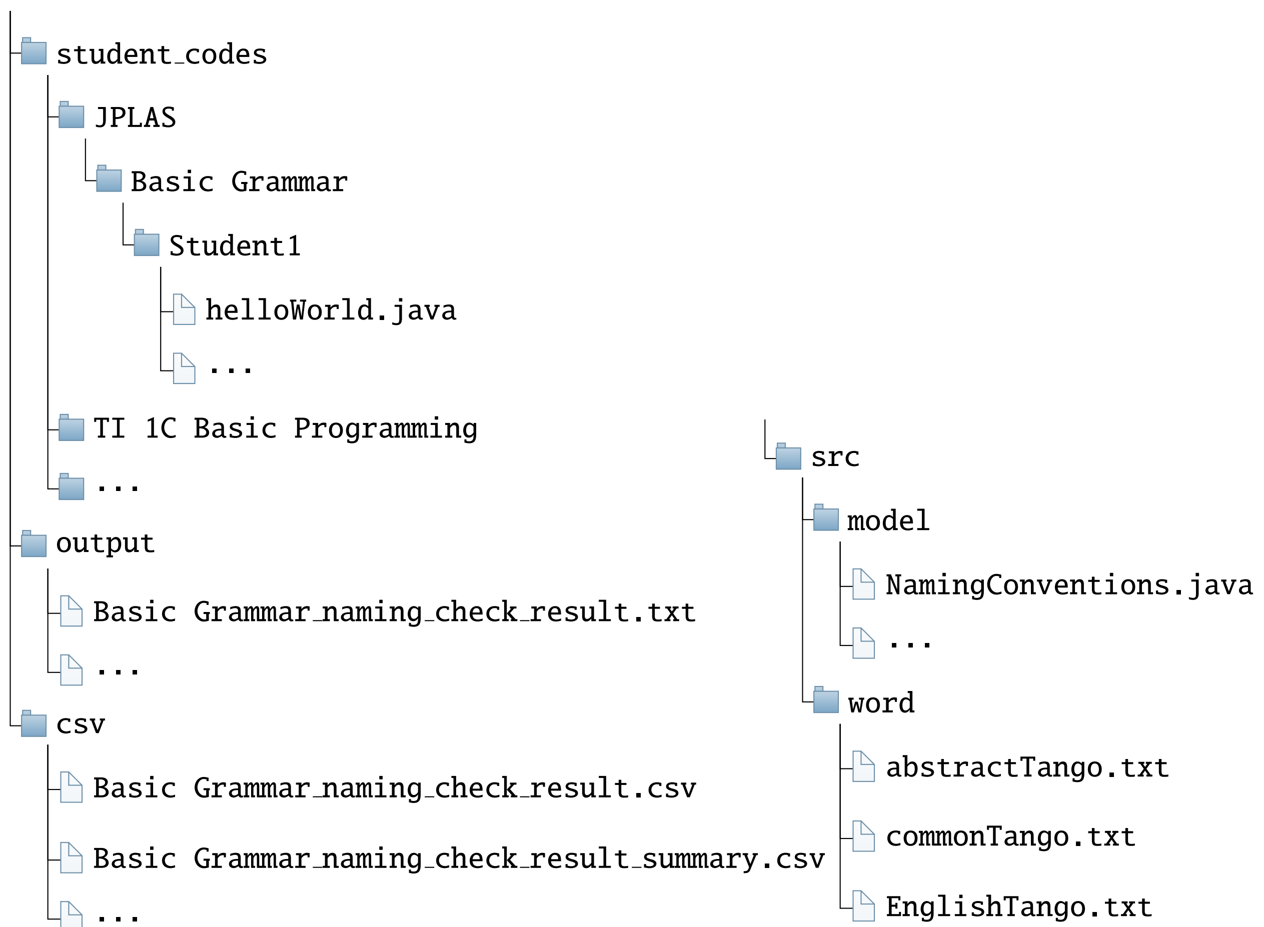


Figure 6.1: Designated file system for naming checking function.

6.3.2 Four Naming Rules

The naming rule checking function finds the naming errors from a source code based on the following *four rules*:

1. **Camel Case:** Every identifier name must use *Camel case*. It is checked using *regular expressions*.

2. **English Dictionary:** Each identifier name must be a valid English word found in an English dictionary. If a name uses multiple words in *Camel case*, it is split into individual words. This check helps prevent spelling mistakes and ensures words are understandable.

3. **Abstract Word:** Identifiers should avoid using abstract words such as *data*, *size*, and *make*. A specific list of such abstract words is prepared for this check.

4. **Word Length:** The length of any identifier name should be between 3 and 17 characters, except for common short words used in programming like `i` and `j` for loop pointers.

6.3.3 Procedure of Naming Checking Function

The *naming rule checking function* that will check the naming errors of source code files and generate a CSV file containing the results, will be described in the following procedure.

1. Read a source code from each student.
2. Check the *four naming rules* of the code.
3. Record the results in a text file and save it in the *output* folder.
4. Repeat the procedure for every source code in the folder.
5. Generate the CSV file in the *csv* folder and save the summary result for all the codes there.

6.3.4 Example of Testing Result

In this subsection, we exhibit an example of testing result in the proposed naming rule checking function using the source code for *BubbleSort* class in **Listing** 6.1.

Listing 6.1: **Example source code**

```
1  class BubbleSort {
2     public static int[] bubble_sort(int[] a) {
3        int size = a.length;
4        int t = 0;
5        for(int i=0; i < size; i++){
6           for(int j=1; j < (size−i); j++){
7              if(a[j−1] > arr[j]){
8                 t = a[j−1];
9                 a[j−1] = a[j];
10                a[j] = t;
11             }
12          }
13       }
14       return a;
15    }
16 }
```

The example naming rule test results are shown in Table 6.1. In the example results, the method name `bubble_sort` does not follow the *Camel case*, which should be corrected to `bubbleSort`. The variable names `a` and `t` are too short, which should be corrected to `array` and `temp` for example, respectively. They are more descriptive and commonly used. The variable name `size` is abstract, which should be replaced with a more specific name like `numArray` to clearly represent the array size.

Table 6.1: Naming rules test result.

Identifier name	Error
bubble_sort	not Camel case
a, t	length
size	abstract word
i, j	proper name

6.4 Analysis of Application Results

In this section, we applied the *naming rule checking function* to a total of $2,908$ source codes submitted from students for assignments in *Java* programming courses in Okayama University, Japan, Malang State Polytechnic, Indonesia, and Yamaguchi University, Japan, and analyzed the application results.

6.4.1 Courses and Topics

The assignments in the courses can be categorized into six groups, namely, *JPLAS* in Okayama University, *TI 1C Basic Programming*, *TI 1I Basic Programming*, and *MI 3D Algorithm and Data Structure* in Malang State Polytechnic and *Test1* and *Test2* in Yamaguchi University. Basically, these groups have different levels and different number of students. Table 6.2 shows the university name, the course name, the group topic, and the number of total source codes for each group. It is noted that the *TI 1I Basic Programming* course in Malang State Polytechnic, is offered in English as it is intended for the International students.

Table 6.2: Course name and topics for evaluations.

university	course	ID	group topic	# of source codes
Okayama University	JPLAS	1	Basic Grammar	547
		2	Data Structure	293
		3	Object Oriented Programming	465
		4	Fundamental Algorithms	370
		5	FinalExam1	55
		6	FinalExam2	110
Malang State Polytechnic	TI 1C Basic Programming	7	Quiz1	34
		8	Quiz2	49
		9	midExam	49
		10	finalExam	18
	TI 1I Basic Programming	11	Quiz1	35
		12	Quiz2	29
		13	midExam	24
		14	finalExam	30
	MI 3D Algorithm and Data Structure	15	Quiz1	49
		16	Quiz2	60
		17	midExam	40
		18	finalExam	74
Yamaguchi University	Test1	19	Test1_1	98
		20	Test1_2	99
		21	Test1_3	100
	Test2	22	Test2_1	95
		23	Test2_2	95
		24	Test2_3	90

6.4.2 Analysis Results of Individual Groups in Okayama University

First, we analyze the solution results of the individual groups by the students in *JPLAS* in Okayama University, Japan.

6.4.2.1 Results for Basic Grammar

Figure 6.2 shows the instance ID, the number of proper names, and the number of errors in identifier names found by the function for each rule for *basic grammar*. It is noted that a *proper name* indicates the one following the rules. It indicates that most students used proper identifier names in their source codes whereas a small number of students made errors in the naming rules.

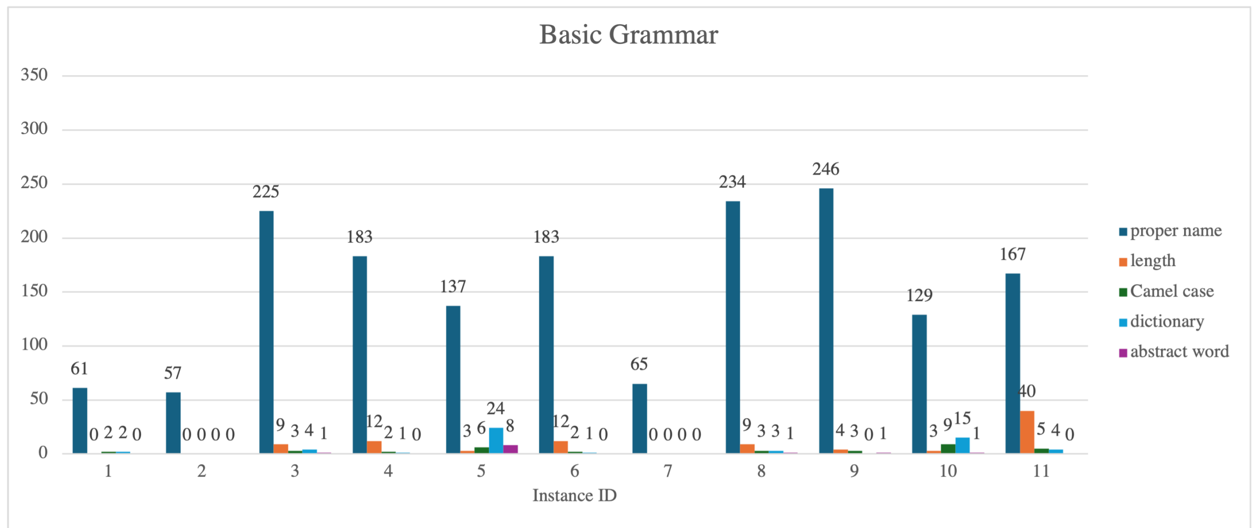


Figure 6.2: Results for *basic grammar*.

6.4.2.2 Results for Data Structure

Figure 6.3 shows the instance ID, the number of proper names, and the number of errors in identifier names found by the function for each rule for *data structure*. It indicates that most students used proper identifier names in their source codes whereas some students misused abstract words, such as *data* in the identifier names.

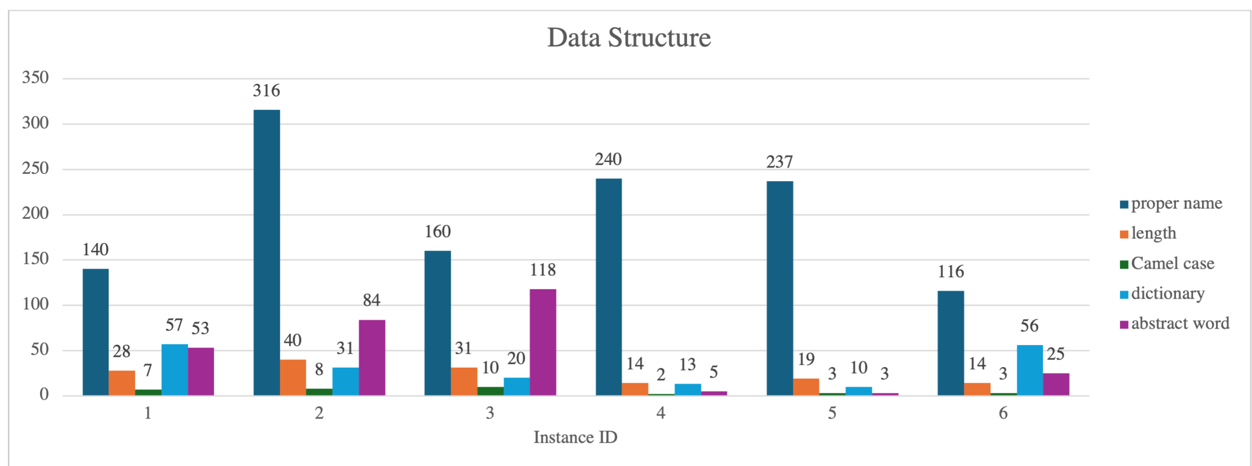


Figure 6.3: Results for *data structure*.

6.4.2.3 Results for Object Oriented Programming

Figure 6.4 shows the instance ID, the number of proper names, and the number of errors in identifier names found by the function for each rule for *object oriented programming*. It indicates that most students used proper identifier names in their source codes. The students follow the Camel case style in their source codes for this group.

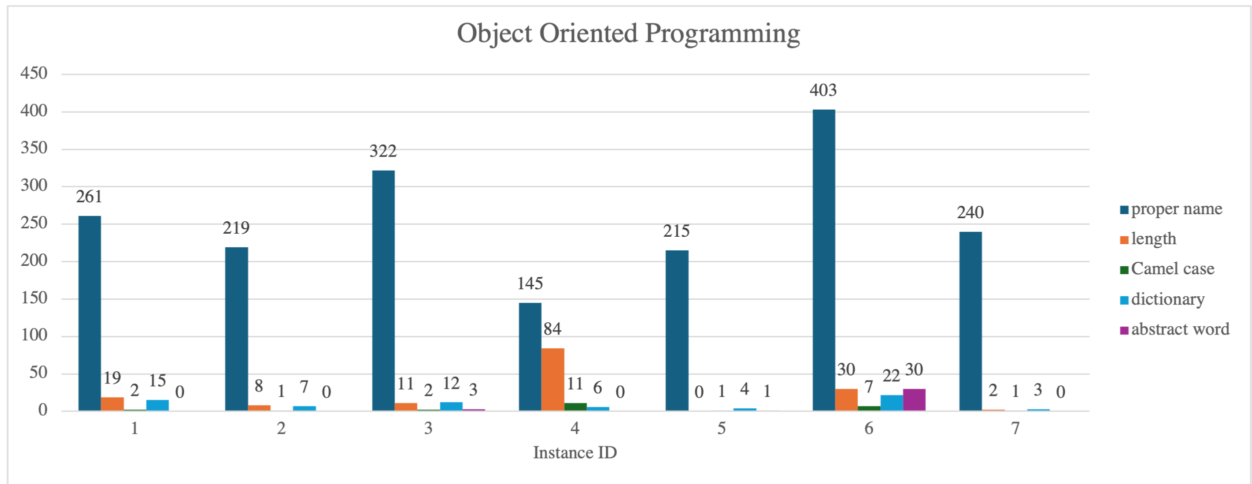


Figure 6.4: Results for *object oriented programming*.

6.4.2.4 Fundamental Algorithms

Figure 6.5 shows the instance ID, the number of proper names, and the number of errors in identifier names found by the function for each rule for *fundamental algorithms*. It indicates that most students used proper identifier names in their source codes whereas a small number of students made errors in the naming rules. Here, we can see that the students used higher number of identifiers for this topic than other three topics.

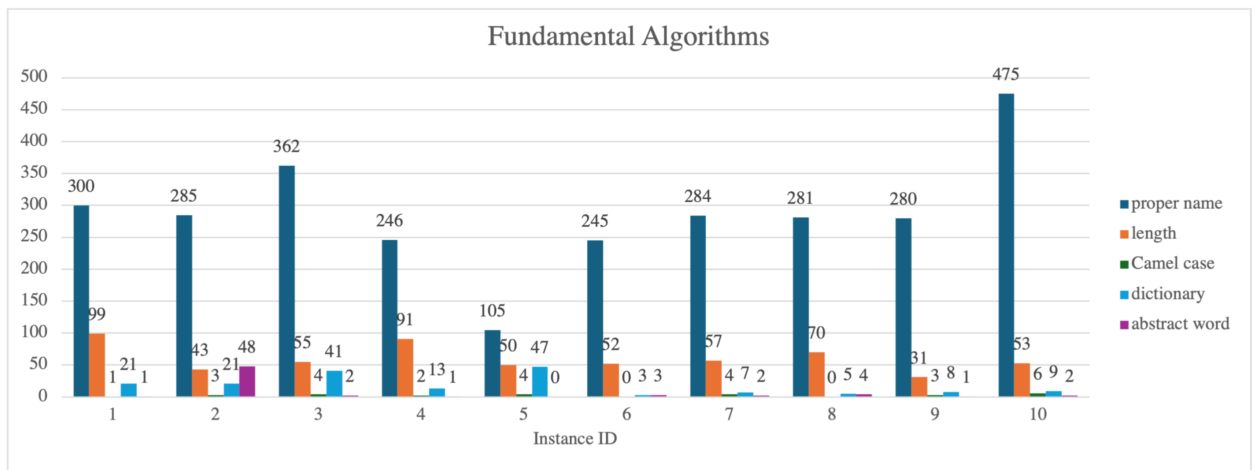


Figure 6.5: Results for *fundamental algorithms*.

6.4.2.5 Final Exam

Figure 6.6 shows the instance ID, the number of proper names, and the number of errors in identifier names found by the function for each rule for *final exam*. It indicates that most students used proper identifier names in their source codes whereas a small number of students made errors in the naming rules. For *FinalExam1*, the length errors are most commonly found in *a*, *b*, and *d* in the source codes.

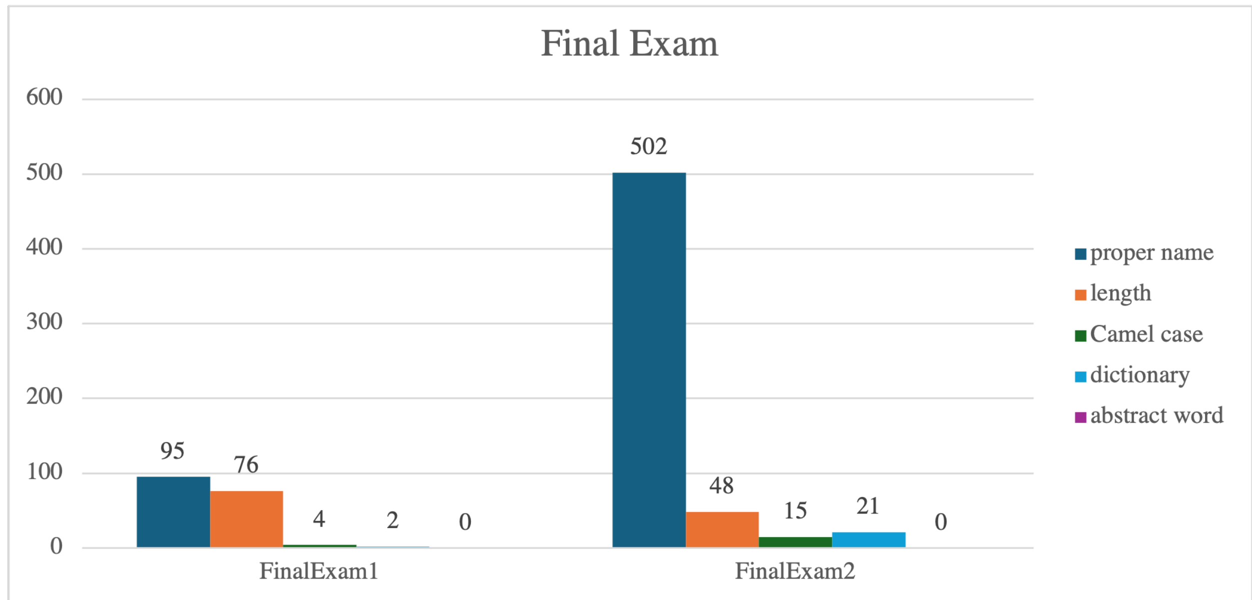


Figure 6.6: Results for *final exam*.

6.4.3 Analysis Results of Individual Groups in Malang State Polytechnic

Next, we analyze the solution results of the individual groups by the students in *Malang State Polytechnic*, Indonesia.

6.4.3.1 TI 1C Basic Programming

Figure 6.7 shows the group topic, the number of proper names, and the number of errors in identifier names found by the function for each rule for *TI 1C basic programming* course. It indicates that most students did not use English words for identifier names, although they used the English alphabet. The alphabet is used as the symbol in the Indonesian language. Moreover, depending on the fixed identifier names in the questions, most of them are in Indonesian.

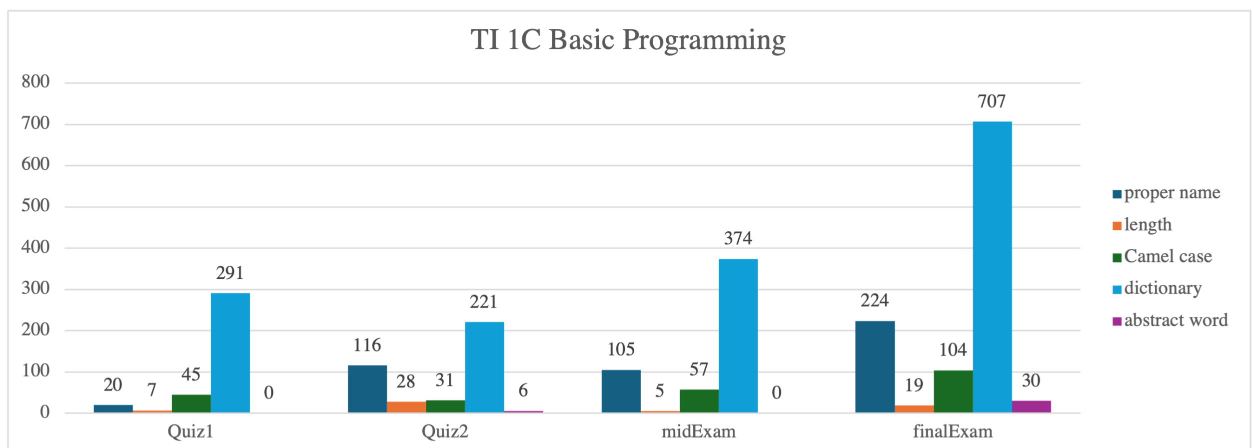


Figure 6.7: Results for *TI 1C basic programming*.

6.4.3.2 TI 1I Basic Programming

Figure 6.8 shows the group topic, the number of proper names, and the number of errors in identifier names found by the function for each rule for *TI 1I basic programming* course. As this course is conducted in English, the analysis result indicates that most students used proper identifier names in their source codes, while some Indonesian students did not use English words for identifier names.

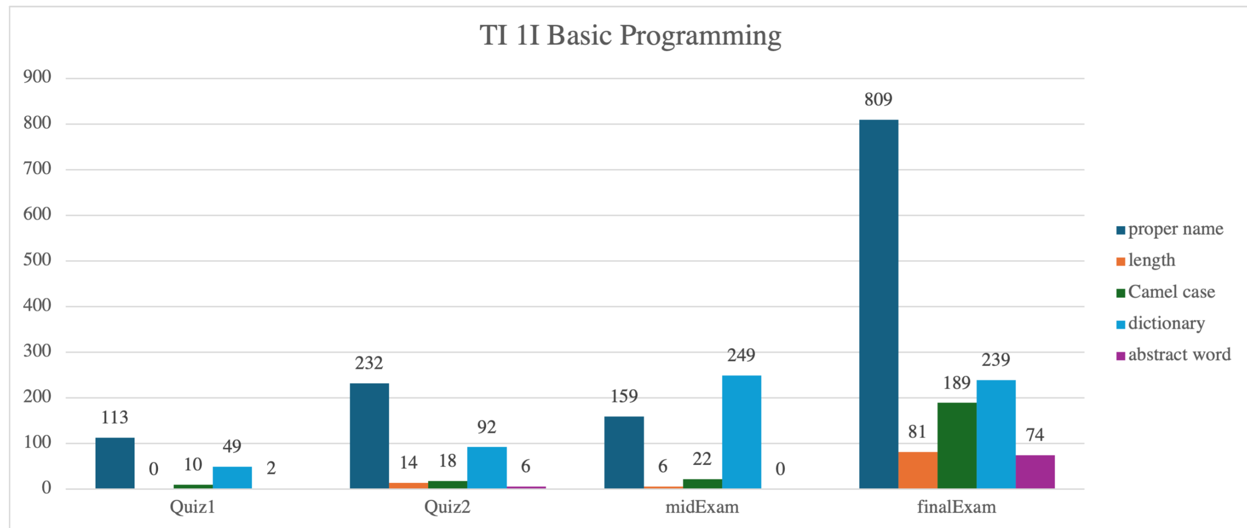


Figure 6.8: Results for *TI 1I basic programming*.

6.4.3.3 MI 3D Algorithm and Data Structure

Figure 6.7 shows the group topic, the number of proper names, and the number of errors in identifier names found by the function for each rule for *TI 1C basic programming* course. It also indicates that most students did not use English words for identifier names, where some of the identifier names in the questions were written in Indonesian language.

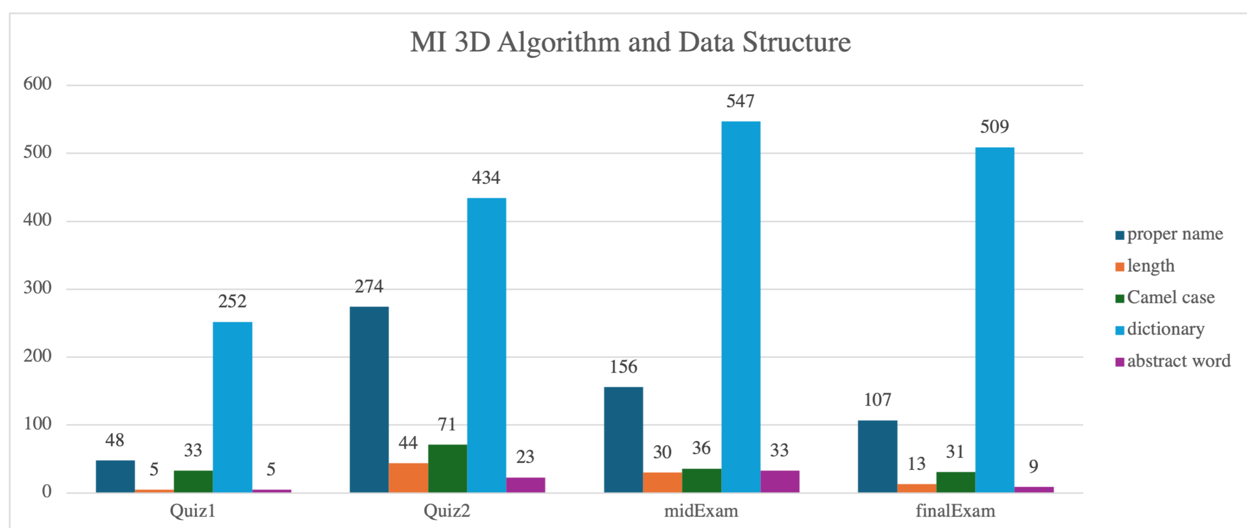


Figure 6.9: Results for *MI 3D algorithm and data structure*.

6.4.4 Analysis Results of Individual Groups in Yamaguchi University

Next, we analyze the solution results of the individual groups by the students in *Yamaguchi University*, Japan.

6.4.4.1 Test1

Figure 6.10 shows the group topic, the number of proper names, and the number of errors in identifier names found by the function for each rule for *Test1* course. It indicates that most students used proper identifier names in their source codes. However, most of the students did not use English words for identifier names, although they used the English alphabet in Test1_1. The alphabet is used as the symbol in the Japanese language, which is known as *Romaji*. Moreover, depending on the fixed identifier names in the questions, most of them are in *Romaji*.

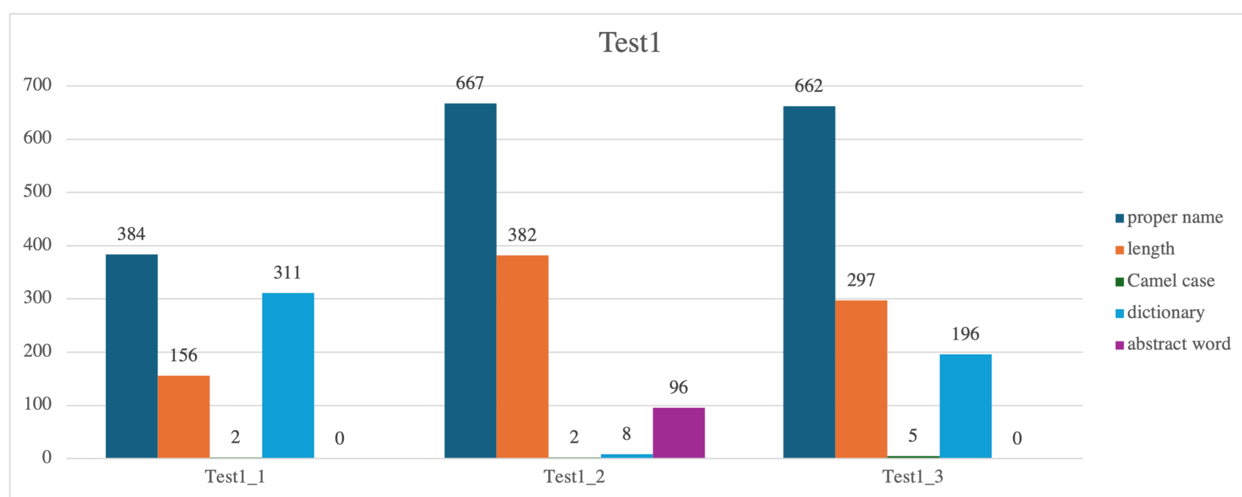


Figure 6.10: Results for *Test1*.

6.4.4.2 Test2

Figure 6.11 shows the group topic, the number of proper names, and the number of errors in identifier names found by the function for each rule for *Test2* course. It indicates that most students used proper identifier names in their source codes whereas a small number of students made errors in the naming rules. For *Test2_1 and Test2_3*, the dictionary errors are most commonly found in *bn*, *bs*, and *sn* in the source codes.

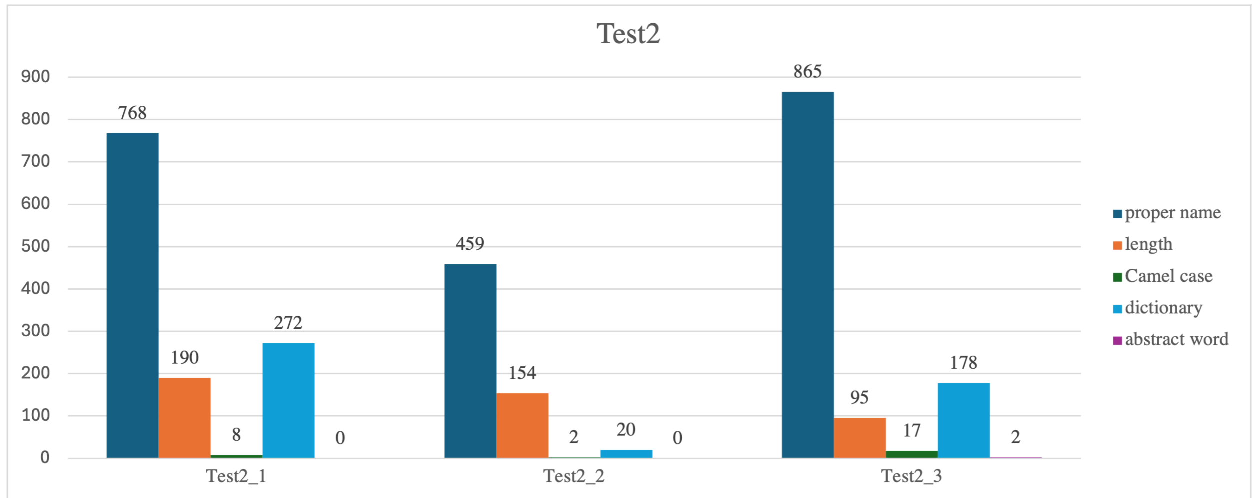


Figure 6.11: Results for *Test2*.

6.4.5 Percentage Analysis Results of Naming Convention in Each Course

Table 6.3 shows the group topic, the usage of proper names, and the usage of errors in identifier names found by the function for each rule in percentage (%). It is noted that a *proper name* indicates the one following the rules.

Table 6.3: Summary of application results.

course	group topic	proper name	length	Camel case	dictionary	abstract word
JPLAS	Basic Grammar	**90%**	5%	2%	3%	1%
	Data Structure	**65%**	8%	2%	10%	15%
	Object Oriented Programming	**62%**	34%	1%	2%	1%
	Fundamental Algorithms	**77%**	16%	1%	5%	2%
	FinalExam1	**54%**	43%	2%	1%	0%
	FinalExam2	**86%**	8%	3%	4%	0%
TI 1C Basic Programming	Quiz1	6%	2%	12%	**80%**	0%
	Quiz2	29%	7%	8%	**55%**	1%
	midExam	19%	1%	11%	**69%**	0%
	finalExam	21%	2%	10%	**65%**	3%
TI 1I Basic Programming	Quiz1	**65%**	0%	6%	28%	1%
	Quiz2	**64%**	4%	5%	25%	2%
	midExam	36%	1%	5%	**57%**	0%
	finalExam	**58%**	6%	14%	17%	5%
MI 3D Algorithm and Data Structure	Quiz1	14%	1%	10%	**73%**	1%
	Quiz2	32%	5%	8%	**51%**	3%
	midExam	19%	4%	4%	**68%**	4%
	finalExam	16%	2%	5%	**76%**	1%
Test1	Test1_1	**45%**	18%	0%	36%	0%
	Test1_2	**58%**	33%	0%	1%	8%
	Test1_3	**57%**	26%	0%	17%	0%
Test2	Test2_1	**62%**	15%	1%	2%	2%
	Test2_2	**72%**	24%	0%	3%	0%
	Test2_3	**75%**	8%	1%	15%	0%

The analysis results revealed that (1) most students in Okayama University used appropriate

identifier names in *source codes*, as they were required to adhere to the specifications described in *test codes*; (2) most students in Malang State Polytechnic did not use English words for identifier names, although they used the English alphabet, since the alphabet is used in the Indonesian language; (3) latter students used a significantly higher number of identifiers than former students, since they did not use *test codes* to guide them; (4) most students in Yamaguchi University also did not use English words for identifier names due to the fixed identifier names in the questions and *Romaji*.

These results confirmed the effectiveness and validity of the proposal in checking the coding rules of a lot of source codes from students, and the efficacy of the use of *test codes* in code writing assignments for correcting names. We will encourage the use of *test codes* for code writing assignments in Malang State Polytechnic, Indonesia, and Yamaguchi University, Japan.

6.5 Summary

This chapter presented the implementation of *naming rule checking function* to check whether the given code follows the *four naming rules* to write a *readable code*. For evaluations, we applied the proposal to 2,908 source codes submitted from students in *Java programming* courses in Okayama University, Japan, Malang State Polytechnic, Indonesia, and Yamaguchi University, Japan. The results confirms the validity and effectiveness of the proposal.

Chapter 7

Code Modification Problem for Client-side Programming Using JavaScript

This chapter presents the *code modification problem (CMP)* for client-side programming using JavaScript in *JavaScript Programming Learning Assistant System (JSPLAS)* [19].

7.1 Introduction

Nowadays, *JavaScript* has become popular in web programming using *Node.js*, since it can be used on both client and server sides to make web pages interactive in a web application system [12]. *JavaScript* is adopted in 97% websites in the world, making it the most popular client-side and server-side scripting language for *web client programming*. By combining the common web technology by the *Hyper Text Markup Language (HTML)* and *Cascading Style Sheets (CSS)*, *JavaScript* can provide dynamic features on a *web page*. The structure and meaning of the page are provided in *HTML*. The layout, background colors, and fonts used in *HTML* content are described in *CSS*. Then, the *document object model (DOM)* is used for *JavaScript* interactions with them. Therefore, *web client programming using JavaScript* has increased values to add dynamic features and functions in web pages by well working with *HTML* and *CSS*. To support self-studies of *JavaScript programming*, *JavaScript Programming Learning Assistant System (JSPLAS)* has been studied by modifying *JPLAS* for *Java programming*.

However, due to the fact that most web pages are written with the combination of *JavaScript*, *HTML*, and *CSS*, any type of the exercise problem currently available in JSPLAS may not be suitable for studying *web programming*. After studying each language separately, students need to relate them in the source code in *client-side programming* and *server-side programming*.

7.2 Proposal of Code Modification Problem (CMP)

In this section, the *code modification problem (CMP)* for self-study of *JavaScript* based *web client programming* in PLAS is discussed.

7.2.1 Definition of CMP

In a CMP instance, a source code containing the HTML/CSS elements and the functions to be studied here, and a pair of the screenshot of the original web page generated by the code and the

screenshot of the slightly altered web page are provided to students. These web pages could have different parameters, functionalities, or variables. Then, the students are requested to modify the source code to generate the altered web page. By solving the given CMP instances, it is expected that the students can master the basic concepts of *web client programming* and understand the interacted use of HTML, CSS, and JavaScript in the source code. *String matching* is used to check the correctness of any answer in the source code.

7.2.2 Design Goal of CMP

CMP is designed with the following goals:

1. Source codes of various kinds are provided along with full forms in order to assist novice students with learning *web client programming using JavaScript*.
2. By answering the questions in CMP instances correctly, students can learn how to generate web pages using various JavaScript functions.
3. As a result of using *string matching*, the student feedback is immediately provided when the answers are automatically marked.
4. A novice student who has never studied *web client programming* can answer the questions without encountering serious difficulties.

7.2.3 CMP Instance Generation Procedure

The following procedure can be used to generate a new CMP instance:

1. From a website or book that contains the library functions to be examined in this instance, choose a proper source code with HTML, CSS, and JavaScript to develop the web page.
2. Save the screenshot of the web page after running the source code in a web browser.
3. Identify and determine the parts of the source code that students should modify to better understand the intended functionality. This step is manually carried out currently. The automatic processing will be studied in future works.
4. Save the screenshot of the altered web page after running the modified source code in a web browser.
5. Due to the fact that HTML tags are not visible on web browsers, replace the HTML tags with the HTML entities according to their numbers and names.
6. Make the input text file for the newly created CMP instance that consists of the problem statement, the original source code, and the modified source code as the correct solution.
7. Using this text file, generate the CMP instance files, which are combined with the HTML, CSS, and JavaScript, for the answer interface on a web browser using the *instance generation program*. It should be noted that this program has already been implemented and used for GUP, VTP, EFP, and CCP.
8. Insert the screenshots of the original and modified web pages into the HTML file for the CMP instance.

7.3 Example of CMP Instance Generation

Next, an example web page source code will be used to discuss the details of the CMP instance generation.

7.3.1 Original Source Code

The original source code should include the fundamental JavaScript library functions for *web client programming* to be studied in this instance. The example source code in Figure 7.1 shows the web page that displays the *alert box* after clicking *Submit button* from accepting the user name in the input form.

```
01 <html>
02 <body>
03 <p>Submit the Form with alert box</p>
04 <form name="myForm" onsubmit="myFunction()">
05         Enter name: <input type="text" name="fname">
06         <input type="submit" value="Submit">
07 </form>
08 <script>
09 function myFunction() {
10         var x = document.forms["myForm"]["fname"].value;
11         alert("Hi, "+x+" .The form was submitted");
12 }
13 </script>
14 </body>
15 </html>
```

Figure 7.1: Original source code for CMP instance #19.

7.3.2 Original Web Page

The screenshot of the original source code for CMP instance #19 in Figure 7.2 shows the web page that was created using the source code in Figure 7.1. This generated web page is offered as a reference for understanding the source code.

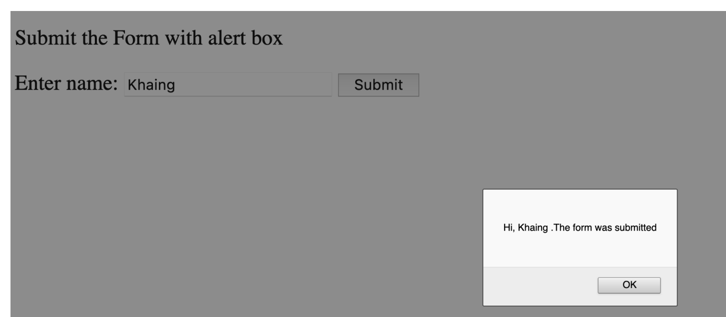


Figure 7.2: Web page by source code for CMP instance #19.

7.3.3 Modified Source Code

The original source code can be transformed into the modified source code by changing a few parameters and functions. It is important for students to understand the connections among the corresponding elements of HTML, CSS, and JavaScript when creating web pages. Figure 7.3 displays the source code for the correct answer. In this example instance, it is necessary to change the text messages in the alert box and the button, and the type and the name in the "input" tag.

```
01 <html>
02 <body>
03 <p>Submit the Form with alert box</p>
04 <form name="myForm" onsubmit="myFunction()">
05         Password: <input type="password" name="passwrd">
06         <input type="submit" value="Create Password">
07 </form>
08 <script>
09 function myFunction() {
10         var x = document.forms["myForm"]["passwrd"].value;
11         alert("Your password is created.");
12 }
13 </script>
14 </body>
15 </html>
```

Figure 7.3: Modified source code for CMP instance #19.

7.3.4 Modified Web Page

Figure 7.4 shows the screenshot of the altered web page by the modified source code.

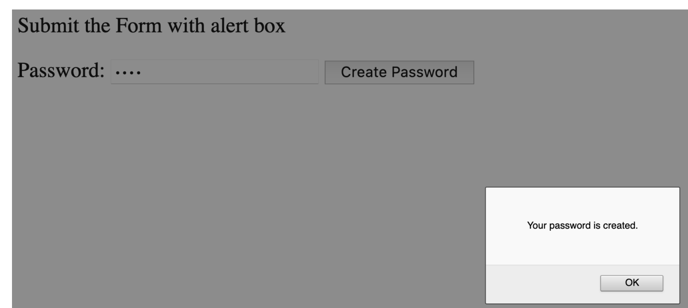


Figure 7.4: Modified web page for CMP instance #19.

7.3.5 Answer Interface

The answer interface is implemented using HTML, CSS, and JavaScript, where the JavaScript program handles the answer marking for the CMP instance. This allows both the online use and the offline use of the answer interface. In order to prevent students from cheating, the correct answers in any CMP instance are encrypted using *SHA256* [9].

Figure 7.5 illustrates the answer interface for the CMP instance that requests the change of the form for the alert box. The source code lines in the input forms, which are indicated by red backgrounds, must be modified by students to complete this CMP instance. To better understand the source code, students can also download and run the source code, if they want to see how the web page is created and how it works.

After completing all the required changes in the input fields, the student can click the "Answer" button to confirm the validity of the answers. The relevant input form's background color is changed to *red* if the answer on the line is incorrect. Otherwise, it turns *white*. The student can submit their answers again until every answer is verified as correct.

7.4 Two-Level Answer Marking

When a student clicks the answer button in the interface, the two-level marking is applied for marking the student answers. The answer is marked through the *string matching* of the whole statement using the Java program at the web server for online, or using the JavaScript program at the student browser for offline. The statement in the student answer and the corresponding statement in the original code are compared.

This marking process is executed at two levels in our implementation. The first level marking examines the statements after removing the spaces and tabs from the answers. To avoid confusions of novice students on use of spaces or tabs, this first level marking does not consider the spaces and tabs in the *string matching*.

However, to encourage a student to be aware of a readable code, the correct insertions of tabs and spaces are important in the source code. Also, the indentation is of utmost importance in *HTML*, *CSS*, and *JavaScript programming*, and plays a vital role in determining the syntax, structure, and readability of the code. Therefore, we need to carefully consider spaces, tabs, and indentations in program codes. As the result, the second level marking marks the statements including the spaces and tabs. If the answer statement contains a missing tab or space, or extra one, the warning message will be returned to the student in the answering interface.

7.4.1 First-Level Marking

First, the first-level marking is applied to the answer source code. Here, after every space and tab is removed from the answer code, each statement is compared with the correct one without a space or tab. If they are different, the corresponding input form of the statement is highlighted by the pink background to suggest that at least one character in the answer statement is different from the correct one. Otherwise, the following second-level marking is applied.

7.4.2 Second-Level Marking

In the second-level marking, the whole statement, including the spaces and tabs, will be compared between the answer code and the original code. If they are different, it is highlighted by the yellow background. Otherwise, the form is not highlighted at all.

Submit the Form with alert box

Enter name: Submit

Submit the Form with alert box

Password: Creat Password

Submit the Form with alert box

Enter name: Khaing Submit

Submit the Form with alert box

Password: .... Create Password

Click Here To Download Source Code

Modify the code to create the password. Use "password" for the type and passwrd" for the name in 'input' tag.

Source Code

```
01:<html>
02:<body>
03:<p>Submit the Form with alert box</p>
04:<form name="myForm" onsubmit="myFunction()">
05:     Enter name: <input type="text" name="fname">
06:     <input type="submit" value="Submit">
07:</form>
08:<script>
09:function myFunction() {
10:     var x = document.forms["myForm"]["fname"].value;
11:     alert("Hi, "+x+" .The form was submitted");
12:}
13:</script>
14:</body>
15:</html>
```

the output

```
01 <html>
02 <body>
03 <p>Submit the Form with alert box</p>
04 <form name="myForm" onsubmit="myFunction()">
05     Enter name: <input type="text" name="fname">
06     <input type="submit" value="Submit">
07 </form>
08 <script>
09 function myFunction() {
10     var x = document.forms["myForm"]["fname"].value;
11     alert("Hi, "+x+" .The form was submitted");
12 }
13 </script>
14 </body>
15 </html>
```

Answer

Answer File Save

Figure 7.5: Answer interface for CMP instance #19.

7.5 Evaluation

This section evaluates the proposed *code modification problem (CMP)* for *web client programming using JavaScript*.

7.5.1 Generated CMP Instances

In this evaluation, 25 CMP instances are generated, to examine fundamental concepts and functions in *web client programming* for performing dynamic behaviors of web pages using JavaScript, HTML, and CSS. The topic/function, the number of lines in the source code, and the number of elements that need to be modified for each CMP instance are displayed in Table 7.1. Other topics like using media devices will be studied in our upcoming works.

Table 7.1: Generated CMP instances.

ID	topic/function	# of lines		# of modified elements	
		HTML/CSS	JavaScript	HTML/CSS	JavaScript
1	JavaScript object	6	4	1	4
2	JavaScript class	13	10	2	2
3	JavaScript math	7	3	3	1
4	click button	10	3	2	1
5	disable button	12	20	2	3
6	circle drawing	5	14	0	5
7	unordered list	19	7	6	3
8	radio button	11	4	5	1
9	checkbox	8	12	1	1
10	information and color change	15	3	5	1
11	color form type	10	12	2	2
12	element position change	27	10	2	2
13	rotating image	14	7	1	3
14	clickable image map	23	5	3	1
15	image button with counter	22	7	3	3
16	background image position change	21	6	4	2
17	text transformation from text area	37	11	7	1
18	file input type with alert box	25	5	8	1
19	input form with alert box	9	6	2	2
20	numbers addition/subtraction from input form	33	8	3	2
21	progress bar with clicking button	23	23	4	3
22	slider control with range input type	39	9	10	1
23	bar chart drawing from input value	49	14	5	3
24	table row insertion with button	41	9	8	3
25	table column deletion with button	66	16	32	2

7.5.2 Solution Results

A total of 37 first-year master students in Okayama University, Japan, who have not taken any formal course of JavaScript programming, were assigned these 25 CMP instances. Prior to this assignment, we only offered a few websites as references to them.

7.5.2.1 Results of Individual Students

The results of the 37 individual students are first analyzed in our evaluations. The correct answer percentage (%) and the number of submissions in average among the 25 CMP instances are shown for each student in Figure 7.6.

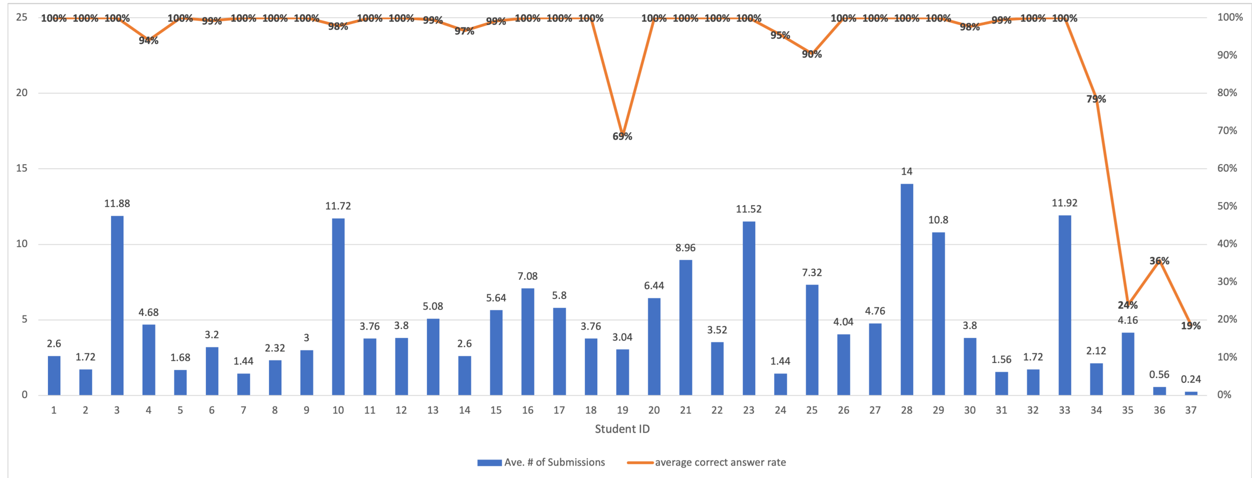


Figure 7.6: Results for each student.

According to the results, out of the 37 students, 22 students correctly answered the questions and achieved the 100% correct rate, and 10 students scored above 90%. Only five students could not reach 90%. Among these five students, three are considered giving up answering the problems as zero answer submission attempts, and two students had not solved and submitted all the problems. Therefore, in general, the novice students can solve the generated CMP instances through self-study of *web client programming using JavaScript*.

Since each instance requires at least one submission of the answer, 25 is the least number to solve the 25 CMP instances. There are 131.4 submission times on average. These students generally submitted answers 2 – 14 times to solve one CMP instance. As a result, they carefully prepared and checked their answers before submitting them.

7.5.2.2 Results of Individual Instances

Next, we examine the solution results of individual 25 CMP instances. The correct answer percentage (%) and the number of submissions on average are shown for each of the 25 CMP instances among the 37 students in Figure 8.3.

According to the results, the instance at ID=18 gets the lowest correct rate, which is 90.8%, and the instance at ID=7 achieves the highest correct rate, which is 100%. The maximum number of submissions is 10 for the instances at ID=18 and ID=23 and the minimum ones are 3.3 for the CMP instances of the instances at ID=2, ID=17 and ID=19. Regarding the proportion of the correct answers, 30 elements/functions are needed to be changed in the instance at ID=18. In this case, we can assume that students looked at the screenshots rather than reading the instructions. Some students had not tried to solve this problem from the answer results. In contrast, the source code of the instance at ID=7 was simple for students where they just need to change the colors and the lists.

As for the number of submission, the instances at ID=2, ID=17, and ID=23 are simple to find the necessary modifications. On the other hand, the instances at ID=18 and ID=23 require several

modifications. Thus, students submitted their answers 10 times on average. In general, the CMP instances are simple to be solved. However, some instances contain long source codes, where students took time to read and understand them, and made higher submission times.

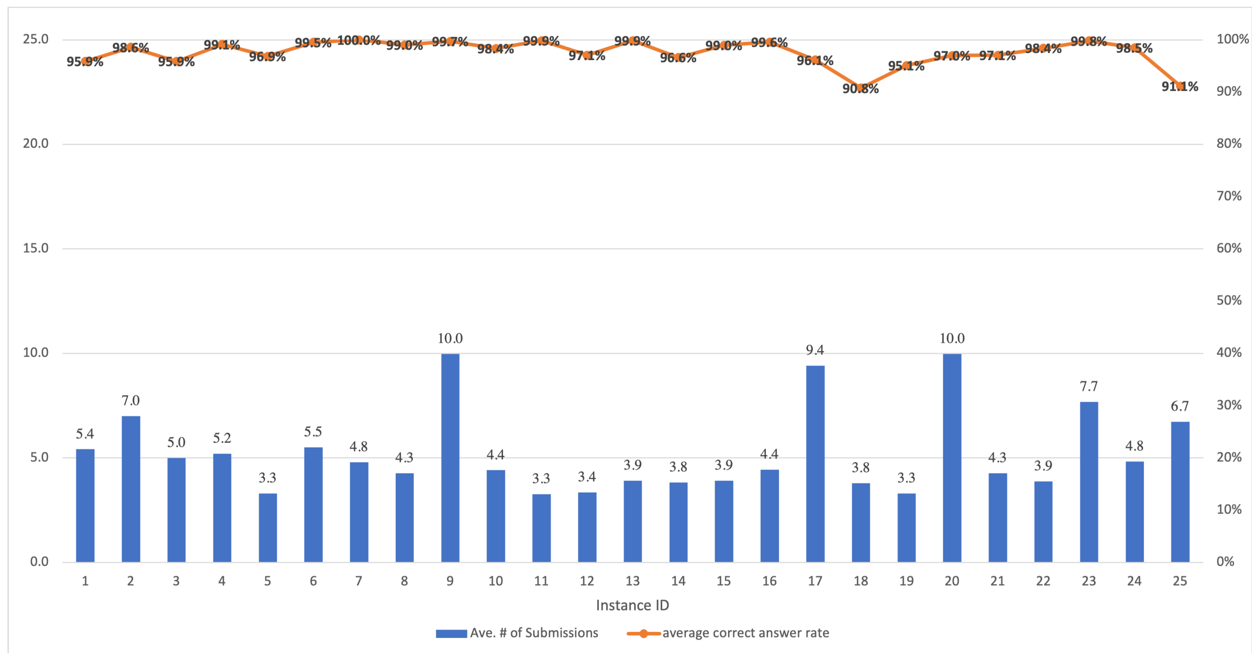


Figure 7.7: Solution results for individual CMP instances.

7.6 Project Assignment for Learning Effect Evaluation

In this section, we present the *project assignments* to evaluate learning effects of the *code modification problem (CMP)* in *JavaScript-based web-client programming*.

7.6.1 Overview

The proposed CMP is designed for students to read a source code carefully by asking changes of some parameters, values, or messages there, and to understand how to use the HTML/CSS elements and the JavaScript library functions appearing in the code for *web client programming*. However, the CMP does not ask writing source codes by students, which will be a weak point of this approach. Thus, we will evaluate learning effects in code writing to students after solving the CMP instances.

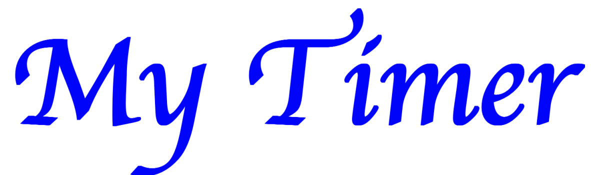
For this purpose, we prepared two *project assignments*, requesting to design the web pages and complete the source codes by referring to some given CMP instances. We made the corresponding answer interface to the project assignment that contains the *problem statement* on the requirements, the *input form* for the answer source code, the *web page output area* by running the code, and the *hint* including the sample page layout, the related CMP instances, and the short instruction video. The web page by the code is shown so that students can easily test their source codes. The project assignments were assigned to the same students after solving the 25 CMP instances.

7.6.2 Project Assignment #1

The first project assignment is *Timer* that requires to implement the text input form, the start, stop, and clear buttons, and the time counting funcion by using the interval function. Figure 7.8 illustrates the sample web page.

7.6.2.1 Problem Statement

In this assignment, the problem statement is given by:
"Create and develop simple "Timer" using HTML, CSS and JavaScript. The timer will count the minutes by using *setInterval ( )* function. You can see the hints by clicking the 'Hint' button below. You can also try to write and run the code in the 'Code' area and see how your code works in the 'Output' area. If your code is OK, you can save your project with your student ID."



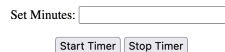


Figure 7.8: Sample *Timer* page for assignment #1.

7.6.2.2 Hint

In this assignment, the hints are given by:
"For *Timer*, you will need the input form to set the minutes and two buttons: start and stop for setting the timer based on the user input value. You can also see Problem #19 and #21 for the reference.
For the *layout interface*, you can create your free design for the simple minute count timer. In the following example video, the timer will start as soon as the user inputs the minutes value and starts the timer button. The timer will be stopped and clear the interval when the user click the stop timer button.
The *setInterval( )* method executes a function or evaluates an expression at specified intervals for *JavaScript library functions* (in milliseconds). Until *clearInterval()* is invoked or the window is closed, the *setInterval( )* method will keep running the procedure. The ID value returned by setInterval( ) is used as the parameter for the *clearInterval( )* method."

7.6.2.3 Result and Discussion

Among the 37 students, only seven students completed this project assignment by making the source codes satisfying the requirements on the interface and functions. Five students among them achieved the 100% correct rate in solving the 25 CMP instances.

Feedbacks from other students who could not complete this assignment suggested that they found difficulties in combined use of HTML, CSS, and JavaScript, which usually results in long source codes. They also found difficulties in debugging source codes, because the web browser does not show any error message at running them. The use of an advanced IDE returning the error messages should be recommended to edit the source code in *web-client programming*.

Then, it was observed that the web page designs of the submitted source codes are unique from each other. Some pages show the timer with minutes and seconds. Some additionally show hours, and even years, months, and days. It is concluded that these seven students well studied *web-client programming using JavaScript*.

7.6.3 Project Assignment #2

The second project assignment is *Calculator* that requires to implement the table layout, the result area, the text input form, the calculation button, and the four arithmetic operations of addition, subtraction, multiplication, and division. Figure 7.9 illustrates the sample web page. Numbers (0-9), and operators (+, −, ∗, /, %, =) should appear in the table layout and the result area.

7.6.3.1 Problem Statement

In this assignment, the problem statement is given by:
"Create and develop simple "Calculator" using HTML, CSS, and JavaScript. The calculator may contain numbers (0-9), simple basic calculations (+, −, ∗, /, %, =) and result area to show the calculation answers. The hints can be seen by clicking the 'Hint' button below. It is also possible to write and run codes in the 'Code' area and see how the code work in the 'Output' area. If the code is OK, it can be saved with your student ID."

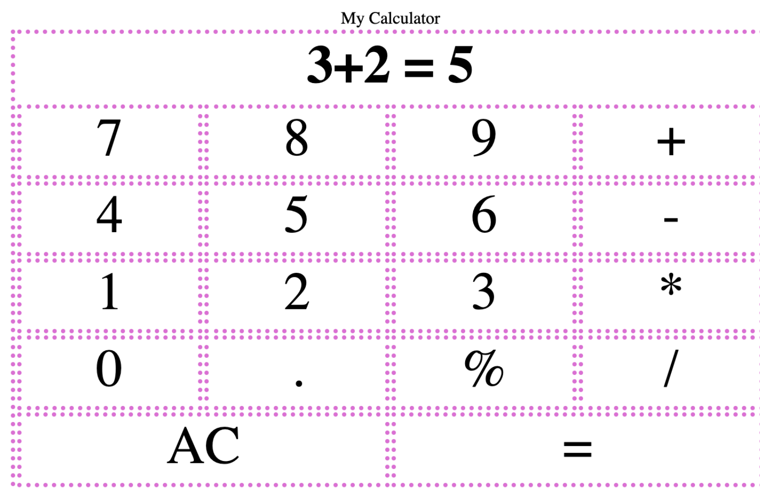


Figure 7.9: Sample *Calculator* page for assignment #2.

7.6.3.2 Hint

In this assignment, the hints are given by:
"For *Calculator*, you will need numbers, operators, and display area for the results.
For the *layout interface*, you can create your free design for your calculator. In the following

example video, table layout and onclick function is used to getting the inputs. You can also see Problem #25 for the reference.
For *JavaScript library functions*, you can use any arithmetic function for calculation in JavaScript. Here, the simple *eval( )* function is used in the example. But, using *eval( )* in real-world applications is far more dangerous. We used it here for keeping our project simple."

7.6.3.3 Result and Discussion

The same seven students completed this project assignment using the table layout design. Their implementations of the arithmetic operations are similar but not the same among them as in the first project assignment.

7.7 Summary

This chapter presented the *code modification problem (CMP)* in *programming learning assistant system (PLAS)* for self-study of *web client programming using JavaScript*. The proposed CMP is designed for students to read a source code carefully by asking changes of parameters, values, or messages there, and to understand how to use the HTML/CSS elements and JavaScript library functions appearing in the code. Students can submit the answers until achieving the correct answers. *String matching* is used to check the correctness of any answer. Based on the results of the students' solutions, the validity of the proposal has been confirmed. Additionally, two simple project assignments were assigned to the students to evaluate learning effects in code writing abilities.

Chapter 8

Code Modification Problem for Server-side Programming Using JavaScript

This chapter presents the *code modification problem (CMP)* for server-side programming using JavaScript in *JavaScript Programming Learning Assistant System (JSPLAS)* [20].

8.1 Introduction

Nowadays, *JavaScript* has become popular in developments of web application systems using *Node.js*, since it can be used on both client and server sides to make interactive web pages [12]. *JavaScript* is adopted in 97% websites in the world, making it the most popular scripting language for *web client programming*. By combining the common web technology by the *Hyper Text Markup Language (HTML)* and *Cascading Style Sheets (CSS)*, *JavaScript* can provide dynamic features on a *web page*. The structure and meaning of a page are provided in *HTML*. The layout, colors, and fonts used in *HTML* contents are described in *CSS*. The *document object model (DOM)* is used for their interactions with *JavaScript* programs. *Web client programming using JavaScript* has increased values to add dynamic features and functions in web pages by well working with *HTML* and *CSS*. To support self-studies of *JavaScript programming*, *JavaScript Programming Learning Assistant System (JSPLAS)* has been studied by modifying *JPLAS* for *Java programming*.

Most web pages are made with the combination of *JavaScript*, *HTML*, and *CSS*. As a suitable exercise problem type for them, the *code modification problem (CMP)* has been adopted in *JSPLAS* for *web programming*. After studying each language separately, students will study how to relate them in the source code through CMP.

8.2 Overview of Code Modification Problem (CMP)

In a CMP instance for *server-side* programming, an *overview* of the concept to be studied, a *source code* using it, and a set of *code modification requests* are given to a student. Then, he/she needs to modify the source code to satisfy the code modification requests by carefully reading the source code and finding the parts to be modified. The correctness of the answer is checked through *string matching* between the answer code and the correctly modified code. Unlike for *client-side programming*, no screenshots are given because the source code is related to logic.

The design goals of CMP are described by:

1. Source codes of various kinds are provided along with full forms in order to assist novice students with learning *web server-side programming using JavaScript*.
2. By answering the questions in CMP instances correctly, students can learn how to generate web pages using various JavaScript functions.
3. As a result of using *string matching*, the student feedback is immediately provided when answers are automatically marked.
4. A novice student who has never studied *web client programming* can answer the questions without encountering serious difficulties.

8.3 CMP Instance Generation Procedure

A new CMP instance can be generated by the following procedure:

1. Select a source code from a website or a textbook that is suitable for the current topic study.
2. Find important parts (functions, variables, parameters, etc.) in the source code to be modified.
3. Prepare another source code by replacing the modified parts from the original source code.
4. Put together the source code and the modified source code for the correct answers into one text file.
5. Run the program with the text file as the input to generate the CMP instance with HTML/C-SS/JavaScript files for the answer interface.
6. Add the descriptions and instructions of the original and modified source into the HTML file in the CMP instance files.

8.4 Example of CMP Instance Generation

Here, we explain an example CMP instance to study the *HTTP module* for responding a text/html file.

8.4.1 Concept Overview

The following *overview* of the *HTTP module* concept is given:

> *HTTP module* can create an *HTTP server* that listens to the server ports and gives a response back to the client. If the response from the *HTTP server* is supposed to be displayed as an *HTML* file, it includes an *HTTP header* with the correct content type: `res.writeHead(200, {\Content-Type": \text/html"})`, where 3-digit *HTTP status* codes allow us to know whether a specific *HTTP request* has been successfully completed.

8.4.2 Source Code

The following *source code* of using *HTTP module* for responding a text/html file is in this instance:

```
1 var httpServer = require("http");
2 httpServer.createServer(function (request, response) {
3   response.writeHead(201, {"Content-Type": "text/plain"});
4   response.write("Hello World!");
5   response.end();
6 }).listen(3000);
```

8.4.3 Code Modification Requests

The following *code modification requests* are given to modify the *source code* in this instance, where the intension of each request is noted in the brackets:

- Change the “response.writeHead(201, {“Content-Type”: “text/plain”});” to “response.writeHead(201, {“Content-Type”: “text/html”});”. (how to change the text format in the web page from plain to html)
- Change the status code ”201 (created)” to ”200 (all response is OK)”. (how to change the status code)
- Change “Hello World!” to “NodeJS response as HTML type”. (how to change the output message)
- Change the ”port 3000” to ”port 8080”. (how to change the port number of this application)

8.4.4 Answer Interface

The two-column CMP answer interface for *server-side programming* is made by slightly modifying the one for *client-side programming* in [19]. Since the screenshots are not given here, the code modification requests in the instance are described at the right side of the interface. Figure 8.1 illustrates the answer form for this instance. By clicking the “Answer” button, the correctness of the answer is checked line-by-line. If one answer line is not correct, the corresponding background color becomes red. The student can submit answers again until all the answers become correct.

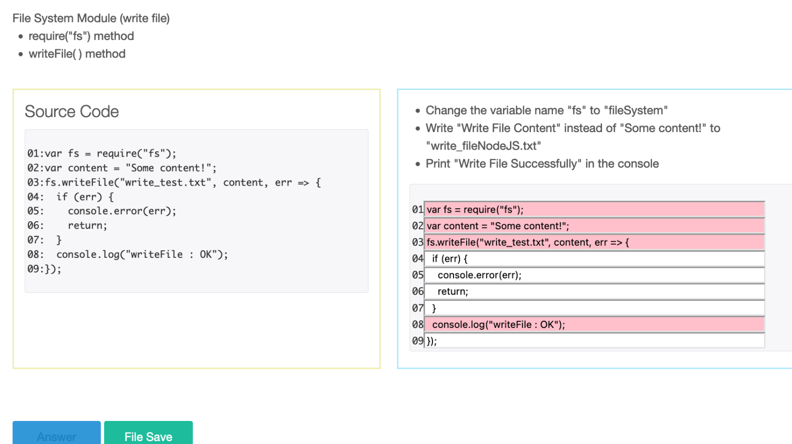


Figure 8.1: Answer interface for example CMP instance.

8.5 Evaluation

This section evaluates the proposed *code modification problem (CMP)* for *server-side programming using JavaScript*.

8.5.1 Generated CMP Instances

In this section, we generated 10 CMP instances for basic topics in *server-side programming*, and assigned them to 64 students to evaluate the feasibilty. Table 8.1 shows the instance ID, the topic, the number of lines in the source code, the number of elements to be modified, the average number of answer submissions by the students, and the correct answer rate by them in each instance.

Table 8.1: CMP instances and solution results.

ID	topic	# of lines	# of elements to be modified
1	HTTP (as message)	5	8
2	HTTP (as text/html)	6	4
3	File System (write)	9	5
4	File System (append)	6	6
5	File System (read)	8	7
6	File System (delete)	8	4
7	URL	8	11
8	NPM	8	7
9	Express	13	8
10	Event	18	10

8.5.2 Solution Results

Then, we assigned the instances to 64 students in Okayama University, Japan, and State Polytechnic of Malang, Indonesia, who have not studied *JavaScript programming* formally. Besides, before this assignment, we did not give any lecture on *server-side programming* using *JavaScript* to them. Instead, we only gave some useful web sites on for their references.

8.5.2.1 Solution Results of Individual Students

First, we analyze the solution results of the 64 students individually. Figure 8.2 shows the average correct answer rate (%) and the average number of submission times among the 10 instances for each student. The average rate is 98%. 49 students achieved the 100% rate and 11 did over 90%. Only four students could not reach 90% but just over 80%. The average number of submission times is 13.1. Each student solved one instance by submitting answers 2 – 14 times. They carefully checked their answers before submissions.

8.5.2.2 Solution Results of Individual Assignments

Next, we analyze the solution results of the 10 CMP instances individually. Figure 8.3 shows the average correct answer rate (%) and the average number of submission times among the 64 students

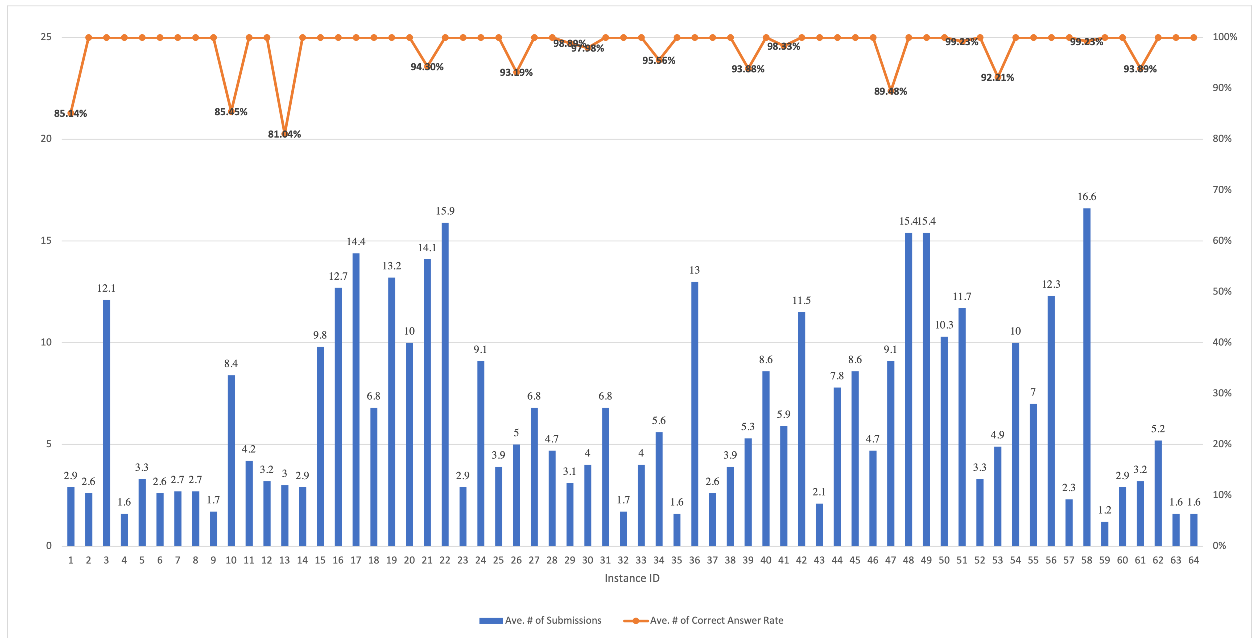


Figure 8.2: Solution results for individual students.

for each instance. All the instances achieved over 96% correct rate. The highest submission number is 12.5 for $ID = 10$ and the lowest one is 3.5 for $ID = 5$. In $ID = 4$, which achieved the 96.09% correct rate, some elements are needed to be modified, where the students could answer them correctly using the instruction in this instance. However, some students missed modifying the parameter and variable in the instance. Thus, the instruction should be improved. In $ID = 10$, as the code is a little bit long, many students needed to submit answers many times. The results show that the generated CMP instances are at appropriate levels for self-study by novice students.

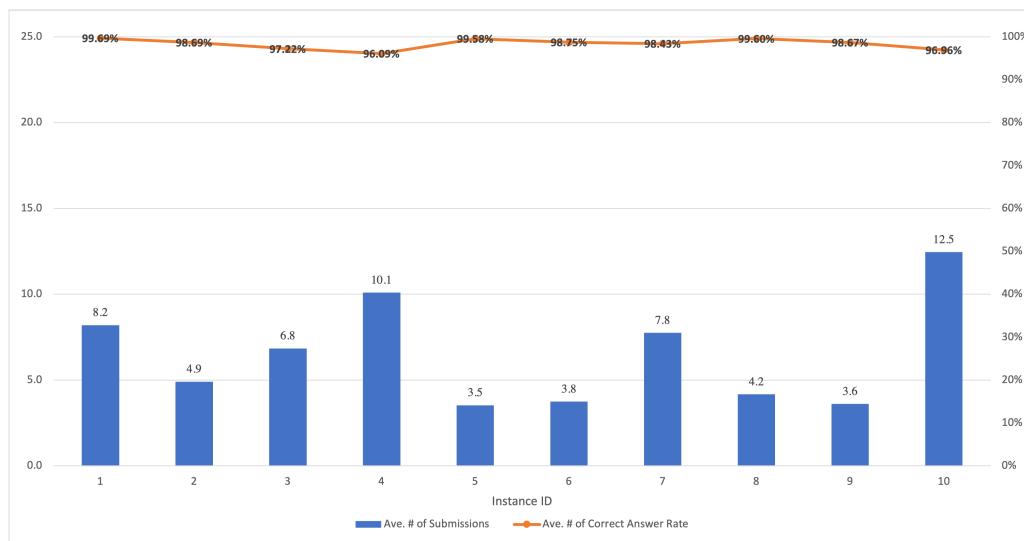


Figure 8.3: Solution results for individual CMP instances.

8.6 Summary

This chapter presented the *code modification problem (CMP)* for self-studies of *web server-side JavaScript programming*. The proposed CMP is designed for students to understand the basic concepts of *Node.js* to create the web applications using *JavaScript*. Since the server-side programming needs to use various *JavaScript* library functions together, reading and understanding sample source codes using them will be an effective way to study. Students can submit the answers until achieving the correct answers where *string matching* is used to check the correctness of any answer. Based on the results of the students' solutions, the validity of the proposal has been confirmed.

Chapter 9

Related Works in Literature

In this section, we introduce some related works to this thesis.

9.1 Programming Education and Learning

In [21] [22], 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 [23].

In [24], 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 [25], Li et al. presented a learning environment based on games, aiming 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 [26], Nguyen et al. discussed the development of an intelligent chatbot designed for educational purposes, particularly in programming courses. It introduced the *Integ-Rela* model, a method for integrating multiple knowledge domains to form a comprehensive knowledge base. The chatbot acts as a virtual tutor, aiding students in learning programming concepts. The effectiveness of this system was demonstrated through experiments, showing its potential as a practical tool in e-learning environments.

In [27], 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 were added in the system to assist the learning and teaching process.

In [28], Matsumoto et al. examined the impact of a puzzle-like programming game called *Algologic* to learning programming. It focused on the achievement degree of students after learning of programming and reported the analysis result. This study found a positive relationship between students' performance in the game and their success in learning programming.

In [29], 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* was designed to interact with students, providing questions and answers for revisions. The paper highlighted the potential of *RevBot* as a useful tool in educational contexts, especially for introductory programming courses.

In [30], 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 [31], 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.

9.2 Code Writing in Programming Study

In [32], 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 [33], 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 [34], 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 [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], 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 [37], 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 [38], 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 imple-

menting *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 [39], Desai et al. presented a survey of evidence regarding the use of *test-driven development (TDD)* in academia. The authors explored existing literature and studies related to *TDD* implementations in educational settings. The paper examined the benefits, challenges, and outcomes associated with *TDD* adoptions in computer science educations. By analyzing the evidence, the authors provided insights into the impact of *TDD* on the student learning, skill development, and overall educational effectiveness.

In [40], Elgendy et al. presented a method using *Genetic Algorithms (GAs)* for automatically generating test data for *ASP.NET* web applications. It introduced new genetic operators designed for the unique structure of web applications, aiming to improve the efficiency and coverage of test data generations. The paper demonstrated the tool's effectiveness through case studies and empirical evaluation, highlighting its utility in enhancing the reliability of *ASP.NET* web applications.

9.3 Code Readability and Maintenance

In [41], Oliveira et al. investigated the impact of code readability and legibility on software maintenance. They reviewed human-centric studies to understand developers' perceptions of code readability and proposed a framework assessing both subjective and objective aspects of code legibility. This research is crucial for improving practices in software maintenance and evolution.

In [42], Boswell et al. provided strategies aimed at improving code clarity and understandability. They emphasized the importance of writing simple, easily readable, and maintainable code, offering practical tips applicable across various programming languages. This guide is valuable for developers seeking to enhance code legibility and software quality.

In [43], Whalley et al. examined the relationship between code readability and the ease of writing. By analyzing the challenges faced by beginners, they aimed to quantify task difficulty and provide insights for designing more effective educational tools for teaching programming.

In [44], Sedano explored the effects of code readability testing on software development. Through tests among software engineers, the author assessed how readability impacts code quality and maintenance. The findings suggested that readability testing can significantly improve software development efficiency, providing crucial insights for both practitioners and educators.

9.4 JavaScript Programming Study

In [45], Maskeliunas et al. proposed an interactive serious programming game for JavaScript programming course at their university. The gamification pattern-based method was used to create this game, together with the *Technology Acceptance Model (TAM)*, and the *Technology-Enhanced Training Effectiveness Model (TETEM)*. Using pre-test and post-test knowledge evaluations, *TAM*, and *TETEM*, they presented the game's evaluation findings.

In [46], Arawjo et al. proposed an educational game approach called *Reduct* to instruct novice students on fundamental JavaScript programming principles, such as functions, Booleans, equivalence, conditional expressions, and mapping functions onto sets. The designs used theories of progression design and skill learning to scaffold concepts and motivate players to create accurate

mental models of the codes. The current objective of this paper is to teach up to the level of advanced and fundamental functional programming in JavaScript.

In [47], Appleton described a prototype system to aid students in learning the web language JavaScript. They discussed how portable intelligent exercises activities were implemented and tested them in the web programming course. According to survey assessment data, they demonstrated that the system may assist students in learning of JavaScript, the web-based programming language.

In [48], Uehara proposed the *JavaScript development environment (JDE)* for the purpose of supporting programming education. The *JDE* offers a setting in which JavaScript programming may be done anywhere, at any time. Additionally, the *JDE* may offer comprehensive snippet features and makes it possible to write code using a limited number of actions. It is appropriate for usage on smartphones because it is an environment based on a browser. The *JDE* can edit HTML, CSS, and JavaScript-enabled web pages.

In [49], Vostinar presented contributions of an interactive e-learning course for teaching web technologies including JavaScript and HTML as a component of the *Moodle* platform. This course may have been taught using traditional methods of instruction, or an alternative, utilizing the *Scrum* agile software development methodology and the *EduScrum* teaching methodology.

Chapter 10

Conclusion

In this thesis, I presented studies of the *answer code validation program* for the code writing problem in Java programming and the *code modification problem* in JavaScript programming.

Firstly, I presented the *answer code validation program* to help a teacher in assigning a lot of CWP assignments to many students in a Java programming course in a university or professional school. This program automatically tests and verifies all the source codes from students running the test code on *JUnit*, and reports the number of tests that each source code could pass with the CSV file. By looking at the summary of the test results of all the students, the teacher can easily grasp the progress of students and grade them.

Secondly, I presented the *intermediate state testing* in the *test code* for *data structure and algorithms* assignments. Against the assignment request, a student may use the library without implementing the correct logic/algorithm in the source codes. If a student implements a different logic or algorithm including the use of library, the conventional *test code* cannot find it, since it only checks the final state of the logic/algorithm. To improve problem-solving skills and develop strong foundations in algorithmic thinking, the *intermediate state testing* can check the randomly selected intermediate state of the important variables during the execution of the logic/algorithm.

Thirdly, I presented the *test data generation algorithm*. The fixed test data in the test code may lead to the issue of cheating, where a student relies on the limited set of test cases to write the source code without truly understanding the concepts. The *test data generation algorithm* 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.

Fourthly, I presented the *naming rules checking function* in the *answer code validation program* for *CWP* in *JPLAS* for novice students, to master writing *readable codes* using proper names for variables, classes, and methods in *Java programming*. The *naming rules checking function* finds the naming errors in the source code. The students will master in writing *readable codes* using proper names for variables, classes, and methods at the early stage of programming studies.

Finally, I presented the *code modification problem (CMP)* as a new type of exercise problem in *JavaScript Programming Learning Assistant System (JSPLAS)*, to study *web client-side* and *server-side programming* using *JavaScript*. The goal of *CMP* is for the students to carefully read the source code and comprehend how to use the components and functions through modifying parameters, values, or messages. The CMP instance gives a source code using the functions to be studied and the screenshot of the web page generated by it. Then, it requests to modify the code to generate another web page given by the screenshot. The correctness of any answer is checked

through *string matching* with the correct one. I evaluated the effectiveness through applications to university students and the application results confirmed the validity and effectiveness of the proposed contributions.

In future works, we will study test codes for other logic or algorithms in mathematics, physics, and engineering topics, generate new assignments for other topics, and apply naming rules checking function in courses in Java programming. Besides, we will study the *code modification problem (CMP)* for other topics and investigate the effectiveness in JavaScript programming.

Bibliography

[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 Int. J. Comp. Sci., 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 Int. J. Comp. Sci., 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 Int. J. Comp. Sci., vol. 40, no.1, pp. 38-46, 2013.

[12] S.H. Jensen, A. Moller, P. Thiemann, "Type analysis for JavaScript," International Static Analysis Symposium, 238–255, 2009, doi:10.1007/978-3-642-03237-0_17.

[13] "SHA-256 Cryptographic Hash Algorithm," Internet: http://www.movable discretionary type.co.uk/scripts/sha256. html/, Access June 20, 2023

[14] K. H. Wai, N. Funabiki, S. T. Aung, K. T. Mon, H. H. S. Kyaw, 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.

[15] K. H. Wai, N. Funabiki, S. T. Aung, X. Lu, Y. Jing, H. H. S. Kyaw, W.-C. Kao, "Answer code validation program with test data generation for code writing problem in Java programming learning assistant system," IAENG Eng. Let., vol. 32, no. 5, pp. 981-994, 2024.

[16] Test-driven development (online), `https://testdriven.io/test-driven-development/`.

[17] K. H. Wai, N. Funabiki, M. Mentari, S. T. Aung, W,-C. Kao, "Implementation of naming rules checking function in code validation program for code writing problem in Java programming learning assistant system," in Proc. FIT2024, 2024. (submitted)

[18] N. Funabiki, T. Ogawa, N. Ishihara, M. Kuribayashi, W.-C. Kao, "A proposal of coding rule learning function in Java programming learning assistant system," in Proc. VENOA, pp. 561-566, 2016.

[19] K. H. Wai, N. Funabiki, K. T. Mon, M. Z. Htun, S. H. M. Shwe, H. H. S. Kyaw, W.-C. Kao, "A proposal of code modification problem for self-study of web client programming using JavaScript," Adv. in Sci. Tech. and Eng. Sys. J. (ASTESJ), vol. 7, no. 5, pp. 53-61, September 2022.

[20] K. H. Wai, N. Funabiki, S. L. Aung, S. T. Aung, Y. W. Syaifudin, W.-C. Kao, "An investigation of code modification problem for learning server-side JavaScript programming in web application system," in Proc. GCCE, pp. 886-88, 2022.

[21] K. Ala-Mutka, "Problems in Learning and Teaching Programming," A literature study for developing visualizations in the Codewitz-Minerva project, pp. 1-13, 2004.

[22] M. Konecki, "Problems in programming education and means of their improvement," DAAAM Int. Sci. Book, pp. 459-470, 2014.

[23] R. A. Queiros, L. Peixoto, and J. Paulo, "PETCHA - a programming exercises teaching assistant," in Proc. ITiCSE, pp. 192-197, 2012.

[24] M. Piteira and C. Costa, "Learning computer programming: study of difficulties in learning programming," in Proc. ISDOC, pp. 75-80, 2013.

[25] F. W.-B. Li and C. Watson, "Game-based concept visualization for learning programming," in Proc. ACM MTDL, pp. 37-42, 2011.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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," J. of Phys., vol. 1840, 2021.

[32] P. Denny, A. Luxton-Reilly, E. Tempero, and J. Hendrickx, "CodeWrite: supporting student-driven practice of Java," in Proc. SIGCSE, pp. 471-476, 2011.

[33] 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.

[34] 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.

[35] H. Kitaya and U. Inoue, "An online automated scoring system for Java programming assignments," Int. J. Info. Edu. Tech. (IJIET), vol. 6, no. 4, pp. 275-279, 2016.

[36] M. Szab and K. Nehz, "Grading Java code submissions in MeMOOC," in Proc. microCAD Int. Sci. Conf, 2018.

[37] 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.

[38] S. H. Edwards and M. A. Pérez-Quiñones, "Experiences using test-driven development with an automated grader," J. of Comp. Sci. in Col., vol. 22, no. 3, pp. 44-50, 2007.

[39] 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.

[40] 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.

[41] D. Oliveira, R. Bruno, F. Madeiral, F. Castor, "Evaluating code readability and legibility: An examination of human-centric studies," in Proc. ICSME, pp. 348-359, 2020.

[42] D. Boswell, T. Foucher, "The Art of Readable Code: Simple and Practical Techniques for Writing Better Code," O'Reilly Media. Inc, 2011.

[43] J. Whalley, N. Kasto, "How difficult are novice code writing tasks? A software metrics approach," in Proc. ACE2014, pp. 105-112, 2014.

[44] T. Sedano, “Code readability testing, an empirical study,” in Proc. CSEET, pp. 111-117, 2016.

[45] R. Maskeliūnas, A. Kulikajevas, T. Blažauskas, R. Damaševičius, J. Swacha, “An interactive serious mobile game for supporting the learning of programming in JavaScript in the context of eco-friendly city management,” Computers, vol. 9, no. 4, pp. 1-18, 2020.

[46] I. Arawjo, C.-Y. Wang, A.C. Myers, E. Andersen, F. Guimbretière, “Teaching programming with gamified semantics,” in Proc. CHI, pp. 4911-4923, 2017.

[47] J. Appleton, “Introducing intelligent exercises to support web application programming students,” in Proc. ICICTE, pp. 216-225, 2017.

[48] M. Uehara, “JavaScript development environment for programming education using smartphones,” in Proc. CANDARW, pp. 292-297, 2019.

[49] P. Vostinar, “Interactive course for JavaScript in LMS Moodle,” in Proc. ICETA, pp. 810-815, 2019.