A Study of Informative Test Code Approach for Code Writing Problem in Java Programming Learning Assistant System

September, 2020

Ei Ei Mon

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.

Written under the supervision of

Professor Nobuo Funabiki

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

Okayama University, September 2020.

To Whom It May Concern

We hereby certify that this is a typical copy of the original doctor thesis of
Ms. Ei Ei Mon

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

Abstract

Nowadays, the objected oriented programming language *Java* has been widely used in various practical applications in societies due to the high reliability, portability, and scalability. Therefore, there have been strong demands from industries for Java programming educations. To advance them, we have studied a *Java Programming Learning Assistant System (JPLAS)*. To allow the use of *JPLAS* in various environments, both the Web-based online version and the desktop-based offline version have been implemented for *JPLAS*.

JPLAS has several types of problems to cover a variety of students at different learning levels. Among them, the code writing problem asks a student to write a source code to satisfy the specification of a given assignment. The correctness of the student answer is evaluated through the software test with the *test-driven development (TDD) method* on an open source framework *Junit* using the test code. The test code can describe the detailed information for the source code implementation, which can help a student to write a complex code. Thus, we have studied this informative test code approach. As the first contribution of the thesis, we present the informative test code generation to help a teacher preparing the test code for the code writing problem. At the early stage of the Java programming study, a student should master how to write a source code that contains the *standard input/output with exception handling*. Thus, the proposed method focuses on generating an informative test code using the test code template and the reference source code containing the procedure of the *standard input/output with exception handling*.

As the second contribution of the thesis, we present the *informative test code approach for Java collections framework (JCF)*. A student should master how to write a source code using *JCF* that is a strong and useful architecture to store or control a group of objects. *JCF* covers wrapper classes such as Integer, Long, or Double. It can enlarge or shrink the size automatically when an object is added or removed. In this thesis, informative test codes for *List*, *Set*, and *Map* are generated and evaluated as the most useful interfaces in *JCF*.

As the third contribution of the thesis, we implement one teacher service function in *Desktop-version Java Programming Learning Assistant System (D-JPLAS)* to summarize the student answers that have been submitted in text files without database or any server and to analyze them for assessing their performances and giving feedbacks to them. To offer *JPLAS* study environments without the Internet, *D-JPLAS* has been developed.

In future works, I will continue studying *informative test codes* for other Java programming topics and implement another teacher service function to detect illegal copies in student answers for *D-JPLAS*.

Acknowledgements

It is my great pleasure to thank those people who have supported and encouraged me throughout this Ph.D. study in Okayama University, Japan. It would not be possible to complete this thesis without their helps. I want to say many things, but I can hardly find the proper words. Therefore, I will just say that you are the greatest blessing in my life.

I owe my deepest gratitude to my supervisor, Professor Nobuo Funabiki, who has supported me throughout my thesis with his patience and knowledge. I am greatly indebted to him, whose encouragements, advices, and supports from the beginning to the end enabled me to proceed this study, not only in scientific issues but also in life. He gave me wonderful advices, comments, and guidance when formulating problems, implementing codes, conducting experiments, writing papers, and presenting them. Thanks for making me who I am today.

I am deeply grateful to my co-supervisors, Professor Satoshi Denno and Professor Yasuyuki Nogami, for their continuous supports, guidance, mindful suggestions, and proofreading of this work.

I wish to express my sincere gratitude to Associate Professor Minoru Kuribayashi for his valuable suggestions during my research and for the future. I also want to express my gratitude to the course teachers during my Ph.D. study for enlightening me with wonderful knowledge.

I would like to acknowledge the Ministry of Education, Culture, Sports, Science, and Technology of Japan (MEXT) for financially supporting my Ph.D. study, and the Ministry of Education, Myanmar, and Technological University (Monywa), Myanmar, for giving this precious opportunity to study in Japan.

I would like to thank my friends and colleagues who helped me in this study, including Dr. Khin Khin Zaw, Dr. Nobuo Ishihara, Ms. Su Sandy Wint, Mr. Hein Htet, Mr. MuneneKWENGA Ismael, Ms. Htoo Htoo Sandy Kyaw, Ms. Soe Thandar Aung, Mr. RAHMAN Md.Mahbubur and all the FUNABIKI Lab's members. Thank you for your supports at my tough time during this study and thank you for sharing the thoughts and experiences with me.

Finally, I am eternally grateful to my beloved family and teachers, who always encourage and support me throughout my life. Your supports and understanding gave me the strength and inspirations to overcome any difficulty in my life.

List of Publications

Journal Paper

1. **E. E. Mon**, N. Funabiki, R. Kusaka, K. K. Zaw, and W.-C. Kao, “ A test code generation method for coding standard input/output with exception handling in Java programming learning assistant system,” Adv. in Sci., Tech, and Eng. Sys. J., (ASTESJ), vol. 3, no. 1, pp. 30-37, Jan. 2018.

2. **E. E. Mon**, N. Funabiki, M. Kuribayashi, and W.-C. Kao, “ An informative test code approach in code writing problem for Java collections framework in Java Programming Learning Assistant System,” J. of Software, vol. 14, no. 5, pp. 200-208, May 2019.

International Conference Paper

3. N. Funabiki, K. K. Zaw, **E. E. Mon**, and W.-C. Kao, “ An improved informative test code approach for code writing problem in Java programming learning assistant system,” Proc. 6th Int. Conf. on Emerging Internet, Data and Web Tech. (EIDWT-2018), pp. 687-698, Mar. 15 - 17, 2018.

4. K. K. Zaw, N. Funabiki, **E. E. Mon**, and W.-C. Kao, “ An informative test code approach for studying three object-oriented programming concepts by code writing problem in Java programming learning assistant system,” Proc. 7th IEEE Global Conf. on Consumer Electronics (GCCE2018), pp. 592-596, Oct. 2018.

Other Papers

5. **E. E. Mon**, N. Funabiki, and K. K. Zaw, “ Generations of informative test codes for studying encapsulation, inheritance, and polymorphism in Java programming learning assistant system,” JSiSE Student Conf. in Chugoku, pp. 229-230, Mar. 2018.

6. **E. E. Mon**, N. Funabiki, and M. Kuribayashi, “ An informative test code approach to studying collections framework by code writing problems in Java programming learning assistant system,” 2018 IEEE Hiroshima Section Stu. Sym. (HISS2018), pp. 73-76, Nov. 2018.

7. **E. E. Mon**, N. Funabiki, S. S. Wint, and M. Kuribayashi, “ Implementation of student answer analyzing function for desktop-version Java programming learning assistant system,” IEICE General Conf., BS-4-34, pp. S74-S75, Mar. 2019.

8. **E. E. Mon**, S. S. Wint, N. Funabiki and M. Kuribayashi, “ Extension of student answer analyzing function to five problem types in desktop-version java programming learning assistant system,” Proc. of FIT-2019, pp.309-310, Sept. 2019.

List of Figures

List of Tables

List of Codes

Contents

Chapter 1

Introduction

1.1 Background

In 1995, the objected oriented programming language *Java* was first released with the Java Runtime Environment (JRE) from Sun Microsystems [1]. The JRE consists of the Java Virtual Machine (JVM), Java platform core classes, and supporting Java platform libraries. The JRE is the runtime portion of a Java code on a Web browser, where the Java plug-in software is a component of the JRE. The JRE allows *applets* written in Java to run on various Web browsers. Moreover, it allows programmers to create new classes that share some of the attributes of existing classes. Besides, they can use the same method name with different number of parameters and different data types.

Nowadays, *Java* has been widely used in various practical applications like web applications, desktop applications, mobile applications, scientific applications, enterprise applications in societies due to the high reliability, portability, and scalability. Therefore, the strong demands for Java programming education have appeared from IT industries and professional schools. Correspondingly, a plenty of universities and professional schools are currently offering Java programming courses to meet these challenges. A typical Java programming course consists of grammar instructions in the class and programming exercises in computer operations.

To enhance Java programming education, we have developed a *Java Programming Learning Assistant System (JPLAS)* [2] which provides teacher service functions and student service functions. *JPLAS* inspires students by offering sophisticated learning environments via quick responses to their answers for self-studies. At the same time, it supports teachers by reducing loads of evaluating the codes.

To allow the use of *JPLAS* in various environments, both the Web-based online version and the desktop-based offline version have been implemented for *JPLAS*. The online *JPLAS* adopts *Ubuntu* for the operating system, *Tomcat* for the Web application server, *JSP* for the application programs with *HTML*, and *MySQL* for the database for handling the problem descriptions and the students' data. The user can access to *JPLAS* through a Web browser. Since the online *JPLAS* may not be suitable for students in poor Internet access environments, the offline *JPLAS* called the *Desktop-version JPLAS (D-JPLAS)* has been implemented without using the online database and the Web server [3]. In *D-JPLAS*, the assignment files and the answer text files can be shared between the teacher and the students using USB memories, a file server or email.

JPLAS has several types of exercise problems to cover a variety of students at different learning levels, such as code understanding problem (CUP), value trace problem (VTP), element fill-in-blank problem (EFP), code completion problem (CCP), code correction problem (CRP), statement fill-in-blank problem (SFP), and code writing problem (CWP). In Java programming studies using

JPLAS, a student is expected to solve the exercises along this order of problem types.

Among them, the *code writing problem (CWP)* asks a student to write a source code to satisfy the specification of a given assignment. In *JPLAS*, a teacher first registers a Java programming assignment with the problem statement, the model source code, and the test code. Then, a student writes a source code by reading the statement and the test code. The correctness of the student answer code is marked through the software test using the test code on an open source framework *JUnit* [4]. This software test approach is called the *test-driven development (TDD) method* [5]. A student can keep modifying the code until completing the correct one.

We have observed that the test code can describe the *detailed information* on the code implementation, such as access modifiers, class names, variable names, method names, parameter types, and return data types. They can help a student to write a complex code. Thus, we have studied this *informative test code approach* for the code writing problem. Previously, we confirmed the effectiveness of this approach in studying the three fundamental object-oriented programming concepts in Java programming, namely, *encapsulation*, *polymorphism*, and *inheritance*.

1.2 Contributions

This thesis presents studies of *the informative test code approach* for the code writing problem in two important Java programming topics for novice students, namely, *standard input/output with exception handling* and *Java collections framework (JCF)*. It also presents the implementation of *Desktop-version Java Programming Learning Assistant System (D-JPLAS)*.

The first contribution of the thesis is the *informative test code generation method* for *standard input/output with exception handling*. This method can help a teacher preparing complex test codes required for this topic study. At the early stage of the Java programming study, a student should master how to write source codes containing *standard inputs* and *standard outputs*, where the handling of *exceptions* must be studied together.

The proposed method generates the informative test code from the test code template and the model source code through the following steps: 1) a test code template is provided by our proposal, 2) a set of standard inputs to be tested are made by a teacher, 3) by running the model code with each input, the corresponding standard output is extracted, and 4) this pair of the standard input and the standard output are embedded into the test code template. By repeating steps 3) and 4) for every testing standard input, the test code can be completed. To run the source code using the test code on *JUnit*, it introduces the classes to handle the standard input/output functions as the memory access functions [6].

For evaluations, we applied the proposal to 97 source codes in Java programming textbooks and Web sites that contain the standard input/output, and found that any generated test code could correctly verify the corresponding source code, except for one code using a random generator. Then, we generated the test codes from three source codes using the proposal, and asked five students who are currently studying Java programming to solve the CWP instances using them. The results showed that all the students completed the codes that can pass the test code, although the use of exception handling functions was sometimes insufficient or incorrect.

The second contribution of the thesis is the application of the informative test code approach to *Java collections framework (JCF)*. *JCF* is a library of providing a strong and useful architecture for storing and manipulating a group of objects. *List*, *Set*, and *Map* are the most useful interfaces in *JCF*. *JCF* can cover wrapper classes such as Integer, Long, or Double, and can enlarge or shrink the size automatically when an object is added or removed.

Therefore, a student should master how to write source codes using *JCF*. For evaluations, we generated five informative test codes for *JCF*, and asked 19 students from Japan, Myanmar, China, and Indonesia to solve the CWP instances using them. We also calculated the software metrics to confirm the quality of their codes. The results showed that all of them successfully completed the source codes.

The third contribution of the thesis is the implementation of *Desktop-version Java Programming Learning Assistant System (D-JPLAS)*. To offer *JPLAS* study environments even without the Internet, *D-JPLAS* provides both the teacher service functions and the student service functions in *JPLAS*. Here, we implemented one teacher service function that summarizes the student answers submitted in text files and analyzes them for assessing their performances and giving feedbacks to them.

1.3 Contents of This Dissertation

The remaining part of this thesis is organized as follows: Chapter 2 reviews the Java Programming Learning Assistant System (JPLAS), where we discuss the sever platform, the software architecture, the problem types, the user service functions, and the Desktop-version JPLAS. Chapter 3 reviews the informative test code approach for the code writing problem including the concepts and examples and introduce the software metrics to evaluate the quality of generated source codes of students. Chapter 4 presents the informative test code generation method for the code writing problem that asks implementing a source code containing the standard input/output with exception handling. Chapter 5 presents the informative test code approach for studying Java Collections Framework (JCF). Chapter 6 presents the implementation of the student answer analyzing function in D-JPLAS. Chapter 7 presents previous works related to this thesis. Finally, Chapter 8 concludes this thesis with some future works.

Chapter 2

Review of JPLAS

In this chapter, we review the *Java Programming Learning Assistant System (JPLAS)*.

2.1 Overview of JPLAS

First, we overview the system architecture, implemented problem types, and user service functions in *JPLAS*.

2.1.1 Server Platform

Originally, *JPLAS* has been implemented as a Web application system. Figure 2.1 illustrates the software platform for the server [7]. Here, *Ubuntu-Linux* is adopted for the operating system, *Tomcat* is used as the Web server to run *JSP* source codes, and *MySQL* is adopted as the database for managing the data in *JPLAS*. *JSP* is a script language that can embed Java codes within HTML codes. *Tomcat* returns the dynamically generated Web page to the client. The current system is running on *VMware* for the portability.

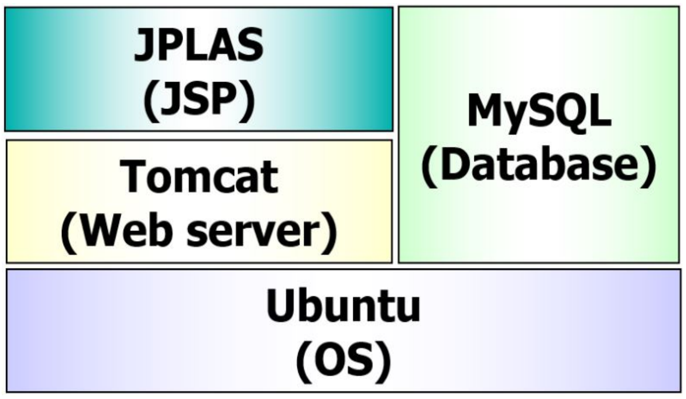


Figure 2.1: Server platform for JPLAS.

2.1.2 Software Architecture

The software architecture of *JPLAS* closely follows the *MVC model* as the common architecture for Web application systems. It basically uses *Java* for the *model (M)*, *HTML/CSS/JavaScript* for the *view (V)*, and *JSP* for the *controller (C)*. Here, it is emphasized that *Servlet* is not used in this case to avoid the possible redundancy that could happen between Java codes and Servlet codes where the same functions may be implemented. A design pattern called *responsibility chain* is adopted to handle marking functions of the student answers, and the specific functions for the database access are implemented such that the controller does not handle them. In the view, the user interface is dynamically controlled with *Ajax*, to reduce the number of JSP files.

2.1.3 Problem Types

JPLAS has several types of problems to accommodate a variety of students at different learning levels. Among them, the *element fill-in-blank problem (EFP)* requires students to fill in correct elements in the blanks in a given Java code [8]. The *value trace problem (VTP)* requires students to answer the actual values of important variables in the code [9]. The *statement fill-in-blank problem (SFP)* makes students write whole statements that are blank in the code [10]. The *code writing problem (CWP)* requires students to write whole Java codes to satisfy the given specifications [11]. Among them, this thesis focuses on the CWP.

In the first two problems, the answers of students are marked by string matching with correct ones. In the latter two problems, the answers are marked by unit testing using test codes on *JUnit*. The difficulty level of the problems is designed to increase in this order of the four problems.

2.2 Code Writing Problem

In this section, we discuss the details of the *code writing problem (CWP)* [11]. The CWP is based on the *test-driven development (TDD) method* [5]. It asks a student to write the source code that satisfies the specification given in the test code on *JUnit*. A student can easily repeat the cycle of writing, testing, and modifying the source code by himself/herself. The test code is prepared by a teacher. **test code 1** shows a sample test code for **source code 1**. "*CityTest* class" at line 4 represents the test class to test "*City* class" in the source code. "*addDataTest* method" at line 6 runs as the test case. "*assertEquals* method" at line 10 tests whether the value at the first index in the list is "Okayama".

test code 1

```
1  import static org.junit.Assert.*;
2  import java.util.List;
3  import org.junit.Test;
4  public class CityTest {
5      @Test
6      public void addDataTest() {
7          City obj = new City();
8          List<String> list = obj.addData();
9          assertEquals(1, list.size());
10         assertEquals("Okayama", list.get(0));
11     }
12 }
```

source code 1

```
import java.util.ArrayList;
import java.util.List;
public class City {
    List<String> addData() {
        List<String> list = new ArrayList<>();
        list.add("Okayama");
        return list;
    }
}
```

2.3 Test-driven Development Method

In this section, we overview the *test-driven development (TDD)* method [5] along with its features.

2.3.1 Outline of TDD Method

In the *TDD* method, the test code should be written before or while the source code is implemented, so that it can verify whether the current source code satisfies the required specifications during its development process. The basic cycle in the *TDD* method is as follows:

1) to write the test code to test each required specification,

2) to write the source code, and

3) to repeat modifications of the source code until it passes each test using the test code.

2.3.2 JUnit

JPLAS adopts *JUnit* as an open-source Java framework to support the *TDD* method. *JUnit* can assist the unit test of a Java source code unit or a *class*. Because *JUnit* has been designed with the Java-user friendly style, its use including the test code programming is less challenging for Java programmers. In *JUnit*, a test is performed by using a given method whose name starts from *assert*. **Test code 2** adopts the *assertThat* method to compare the execution result of the source code with its expected value.

2.3.3 Test Code

A test code should be written using libraries in *JUnit*. Here, by using the following **source code 2** for *MyMath* class, we explain how to write a test code. *MyMath* class returns the summation of two integer arguments.

source code 2

```
public class MyMath{
    public int plus (int a, int b){
        return (a+b);
    }
}
```

Then, the following **test code 2** can test the *plus* method in the *MyMath* class.

test code 2

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

The names in the test code should be related to those in the source code so that their correspondence becomes clear:

- The class name is given by the *test class name* + *Test*.
- The method name is given by the *test* + *test method name*.

The test code imports *JUnit* packages containing test methods at lines 1 and 2, and declares *MathTest* at line 3. *@Test* at line 4 indicates that the succeeding method represents the test method. Then, it describes the test method.

The test code performs the following functions:

1) to generate an instance for the *MyMath* class,

2) to call the method in the instance in 1) using the given arguments,

3) to compare the result with its expected value for the arguments in 2) using the *assertThat* method, where the first argument represents the expected value and the second one does the output data from the method in the source code under test.

2.3.4 Features in TDD Method

In the *TDD* method, the following features can be observed:

1) The test code can represent the specifications of the source code, because it must describe the function tested in the source code.

2) The test process for a source code becomes efficient, because each function can be tested individually.

3) The refactoring process of a source code becomes effective, because the modified code can be tested instantly.

Therefore, to study the *TDD* method and writing a test code is useful even for students, where the test code is equivalent to the source code specification. Besides, students should experience the software test that has become important in software companies.

2.4 User Service Functions

The user functions of *JPLAS* consist of the *teacher service functions* and the *student service functions*. The utilization procedure for both *JPLAS* functions by a teacher and a student is given below.

2.4.1 Teacher Service Functions

Teacher service functions include the registration of courses, the registration and management of assignments, and the verification of source codes that are submitted by students.

To register a new assignment, a teacher needs to input an assignment title, a problem statement, a reference (model) source code, and a test code. After the registration, they are disclosed to the students except for the source code. It is noted that the test code must be able to test the model source code correctly, where we will present the method to automatically generate the test code from the model source code in Section 2.3.3.

To evaluate the difficulty of assignments and the comprehension of students, a teacher can refer to the number of submissions for answer code testing from each student. If a teacher finds an assignment with plenty of submissions, it can be considered as quite difficult for the students, and should be changed to an easier one. If a teacher finds a student who submitted answers many times, this student may require additional assistance from the teacher.

2.4.2 Student Service Functions

Student service functions include the view of the assignments and the submission of source codes for the assignments. A student should write a source code for an assignment by referring the problem statement and the test code. It is requested to use the class/method names, the types, and the argument setting specified in the test code. All submitted source codes will be stored in the database on the server as a reference for students.

2.5 Desktop-version JPLAS

As mentioned before, *JPLAS* has been developed as a Web application system. However, it has been found that the use of *JPLAS* can often be difficult, particularly in developing countries, where the Internet may not be stable due to the weak network infrastructure and the frequent power shortage. Besides, skilled persons to manage the Web server may not be available.

To avoid those difficulties of the online *Web-version JPLAS*, we have implemented the offline *Desktop-version JPLAS (D-JPLAS)* as an efficient solution for schools and homes with the poor Internet access. [3]. Unlike to the online *JPLAS*, *D-JPLAS* runs on the client PC only, without the server access through the Internet. It keeps all the programs and data including the problems and the student answers in the file system of the user's PC, where it does not use the database. Basically, the students submit their answers to the teacher using USB memories. Then, the teacher will analyze the answers on his/her own PC and give feedbacks to the students. In this thesis, we implement the function for analyzing the student answers in *D-JPLAS*.

2.6 Summary

In this chapter, we reviewed the *Java Programming Learning Assistant System (JPLAS)*. On *JPLAS*, we discussed the sever platform, the software architecture, the implemented problem types, the code writing problem, the test-driven development method, the user service functions, and the *Desktop-version JPLAS(D-JPLAS)*.

Chapter 3

Review of Informative Test Code Approach in Code Writing Problem

In this chapter, we review the *informative test code approach* for the code writing problem in JPLAS.

3.1 Informative Test Code Approach in Code Writing Problem

3.1.1 Concept of Informative Test Code

The *informative test code* helps a student to complete the source code by offering the necessary code design information in the hard code writing problem that requires the use of multiple classes/methods, and/or the adoption of advanced concepts of the object-oriented programming such as *encapsulation*, *inheritance*, and *polymorphism*. It may give the following information to implement the code:

- the names for class, methods and member variables,
- the access modifiers for them,
- the data types for the important variables and arguments,
- the returning data types for the methods, and
- the exception handling.

3.1.2 Assignment Generation with Informative Test Code

Generally, for generating an assignment in code writing problem, a teacher prepares the test code file, the input data file, and the expected output data file, in addition to the problem statement in the natural language. Then, a student is requested to write the source code that passes every test described in the test code on *JUnit*. This means that a student writes the source code by referring to the detailed specifications in the test code.

The *informative test code* can be generated after the teacher had prepared the qualitative *model source code* for the problem. It is expected that the student completes the qualitative source code for the problem that has the similar structure with the model source code by referring to this test code. For the teacher, the generation procedure of the code writing problem using the informative test code contains the following steps:

1. The teacher prepares the statement and the *input data file* for the new assignment.
2. He/she prepares the *model source code* that not only satisfies every specification of the problem but also designs the code structure for high quality.
3. He/she prepares the *expected output data file* by running the model source code, where this output file is used to compare the output data file of the student code for checking the correctness.
4. He/she generates the *informative test code* describing the necessary information to implement the source code by a student.

3.1.3 Example Assignment Generation for BFS Algorithm

In this subsection, we describe the details of Steps 1, 2, and 3 using the *BFS algorithm* [12]. This algorithm starts at the *root* node (or arbitrary node of a graph), and explores the neighbor nodes first, before moving to the next level neighbors.

3.1.3.1 Input Data File

To represent a graph, the *input data file* should contain the *index* and the *label* for every vertex, and the *source vertex label* and the *destination vertex label* for every edge. The following example represents a graph with eight vertices and seven edges.

Listing 3.1: Input data file for *BFS*

```
node-number node-label
0 s
1 r
2 w
3 t
4 x
5 v
6 u
7 y
source-node target-node
s r
s w
r v
w t
w x
t u
x y
```

3.1.3.2 Model Source Code

The *model source code* for the code writing problem should be carefully prepared by using the proper classes and methods, so that the measured metrics of the model source code exist in the desired ranges.

For example, the model source code for the BFS algorithm can be implemented using *graph class* for handling the graph data, *BFS class* for applying the algorithm procedure, and *main class* for controlling the whole code. The teacher can obtain the model source code from textbooks or websites. By comparing the measured metrics of the source codes in them, the teacher can select the best source code for the model one.

3.1.3.3 Expected Output Data File

The expected *output data file* can be obtained by running the model source code with the input data file. It describes the expected results of the source code by a student. For the BFS algorithm, it includes the selected edges by the algorithm in the selected order that are described by a pair of two end node labels.

Listing 3.2: Output data file for *BFS*

```
select−node pre−node
s −
r s
w s
v r
t w
x w
u t
y x
```

3.1.3.4 Informative Test Code

The *informative test code* should be generated by referring the model source code such that any important method in the model code must be tested in this test code. It is possible to apply an automatic test code generation tool to help the test code generation [13]. Then, the test code is generated from the model source code by the following rules:

1. The class name is given by the *test class name + Test*.
2. The method name is given by the *test + test method name*.
3. The specific values are specified for the arguments in the test code by the teacher.

The test code can more clearly describe the specifications than a description using natural language. It is expected that the student obtains the information for the class/method names, the data types, and the argument settings by reading the test code, before writing the source code. Because the information in the test code comes from the model source code, the student is able to complete the same qualitative source code as the model code.

3.1.3.5 Informative Test Code Example

The following test code is generated from the source code for the BFS algorithm. It contains the necessary information to implement a source code for the BFS algorithm, including the classes, the methods, the important variables and their data type, the exception handling, and returning values of method.

Listing 3.3: Informative test code for *BFS*

```
import static org.junit.Assert.*;
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.util.Arrays;
import org.junit.Test;
public class BFSTest {
      @Test
      public void testSimpleGraph() {
            SimpleGraph G = new SimpleGraph (5);
            boolean a=G.labels instanceof String [ ];
            boolean b=G.edges instanceof boolean [ ][ ];
            assertEquals(true, a);
            assertEquals(true, b);
            assertEquals(5,G.labels.length);
            assertEquals(5,G.edges.length);
            assertEquals(5,G.edges[0].length);
      }
      @Test
      public void testSetLabel(){
            SimpleGraph G= new SimpleGraph(2);
            G.setLabel(1, "a");
            assertEquals("a",G.labels[1]);
      }
      @Test
      public void testGetLabel(){
            SimpleGraph G = new SimpleGraph (2);
            G.setLabel(1, "b");
            String label=(String)G.getLabel(1);
            assertEquals("b",label);
      }
      @Test
      public void testAddEdge(){
            SimpleGraph G = new SimpleGraph (3);
            G.addEdge(1, 2);
            assertEquals(true,G.edges[1][2]);
      }
      @Test
      public void testNeighbours(){
            SimpleGraph G = new SimpleGraph(3);
            int [ ] expectedNode = {1,2};
            G.addEdge(0,1);
            G.addEdge(0,2);
            assertTrue(Arrays.equals(expectedNode, G.neighbors(0)));
      }
      @Test
      public void testFindBFS1(){
            SimpleGraph G = new SimpleGraph(4);
            BFS bfs = new BFS();
            G.setLabel(0, "a");
            G.setLabel(1, "b");
            G.setLabel(2, "c");
            G.setLabel(3, "e");
            G.addEdge(0,1);
            G.addEdge(0,2);
            G.addEdge(1,3);
            String Path[ ]=bfs.findBFS(G, 0);
            String[ ] expectedPath = {"a a", "b a", "c a", "e b"};
            assertTrue(Arrays.equals (expectedPath,Path));
      }
      @Test
      public void testFindBFS2() throws IOException {
            BFS bfs= new BFS();
65          File testFileName=new File ("./Graph/graphBFS.txt");
66          File OutFileName=new File ("D:/Graph/bfsout.txt");
67          String graph=bfs.readFile(testFileName);
68          String [ ] path=bfs.findBFS(graph);
69          bfs.writeFile(OutFileName, path);
70      }
71      @Test
72          public void assertReaders() throws IOException {
73          BufferedReader expected= new BufferedReader (new FileReader("./Graph/
    expectedbfsout.txt"));
74          BufferedReader actual = new BufferedReader (new FileReader("D:/Graph/
    bfsout.txt"));
75          String line;
76          while ((line = expected.readLine()) != null) {
77              assertEquals(line, actual.readLine());
78          }
79          assertNull("Actual had more lines than the expected.", actual.
    readLine());
80          assertNull ("Expected had more lines than the actual.", expected.
    readLine());
81          }
82  }
```

- Lines from 1 to 7 import some library functions. Firstly, the JUnit library is imported to use the JUnit4 in Eclipse for writing tests. Then, the BufferReader calss is imported to read text from a character input stream.The java.io.File class is an abstract representation of file and directory path names. java.io.FileReader is to read the contents of a file as a stream of characters. To throw the IOException whenever an input or output operation is failed or interpreted, java.io.IOException is imported. The Arrays class in java.util package provides static methods to dynamically create and access Java arrays. The Test annotation tells JUnit that the public void method to which it is attached can be run as a test case.

- Lines from 10 to 19 describe the test method for two important variables, *labels* and *edges*, in *SimpleGraph* class. *labels* has the *String* data type and one dimensional array. *edges* has the *Boolean* data type and two dimensional array.

- Lines from 21 to 25 describe the test method for *setLabel* method in *SimpleGraph*, which accepts two arguments with integer and string data types, namely *index* and *label*, and inserts the information to *labels*.

- Lines from 27 to 32 describe the test method for *getLabel* method, which accepts one argument with integer data type and returns the corresponding label from *labels*.

- Lines from 34 to 38 describe the test method for *addEdge* method, which accepts two arguments with integer data types, namely *source* and *target*, and inserts the information to *edges*.

- Lines from 40 to 46 describe the test method for *neighbours* method, which accepts one argument with integer data type, namely *index*, and returns the integer array which includes the indexes are neighboring to the input *index*.

- Lines from 48 to 61 describe the first test method for *findBFS* method in *BFS* class, which accepts two arguments with the Graph object and the integer data type and returns a string array which includes the labels from *labels* for the selected indexes and the previous index from them by *BFS*. Here, *setLabel* and *addEdge* methods in *SimpleGraph* class are also described here.

- Lines from 63 to 70 describe the second test method for *findBFS* method, which accepts one argument of the string data type and returns the string array which includes the labels from *labels* for the selected indexes and the previous index from them by *BFS*. Here, *readFile* and *writeFile* methods in *BFS* class are also described. The *readFile* method accepts one argument of *File* object and returns the string that includes the *index* and *labels* for the graph to be applied to *findBFS* method. The *writeFile* method accepts two arguments of the *File* object and the string data type array, and writes the input string array, which includes the labels from *labels* for the selected indexes and the previous index from them by *BFS*, to the output file and generate it. This test method throws *IOException* whenever an input or output operation is failed or interrupted when the program is executed.

- Lines from 72 to 81 describe the test method that is used to compare the expected output data file with the output data file from the source code of the student.

3.1.3.6 Simple Test Code Example

Then, to evaluate the performances of students using the informative test code, we also prepare the *simple test code* for them. The following *simple test code* for *BFS* contains only the test methods for *readFile* method, *writeFile* method, and *findBFS* method in *BFS* class.

The *simple test code* only tests the input data file reading and output data file writing functions in the source code. It does not test the internal functions of the code.

Listing 3.4: Simple test code for *BFS*

```
import static org.junit.Assert.*;
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.util.Arrays;
import org.junit.Test;
public class BFSTest {
    @Test
    public void testFindBFS() throws IOException {
        BFS bfs= new BFS();
        File testFileName=new File ("./Graph/graphBFS.txt");
        File OutFileName=new File ("D:/Graph/bfsout.txt");
        String graph=bfs.readFile(testFileName);
        String [ ] path= bfs.findBFS(graph);
        bfs.writeFile(OutFileName, path);
    }
    @Test
    public void assertReaders() throws IOException {
        BufferedReader expected= new BufferedReader (new FileReader("./Graph/expectedbfsout.txt"));
        BufferedReader actual = new BufferedReader (new FileReader("D:/Graph/bfsout.txt"));
        String line;
        while ((line = expected.readLine()) != null) {
            assertEquals(line, actual.readLine());
        }
        assertNull("Actual had more lines than the expected.", actual.
    readLine());
        assertNull ("Expected had more lines than the actual.", expected.
    readLine());
    }
}
```

3.2 Eclipse Metrics Plugin

In this section, we introduce *Eclipse Metrics Plugin* to measure the code quality metrics.

3.2.1 Software Metrics

Software metrics are used for a variety of purposes including the evaluation of the software quality and the prediction of the development/maintenance cost. Software metrics can be measured from software products such as source codes and documents. Most of software metrics are defined on the conceptual modules of software systems, including files, classes, methods, functions, and data flows. This means that software metrics can be measured in any programming language.

At present, a variety of software metrics exist. They can be classified into *basic metrics*, *complexity metrics*, *CK metrics*, and *coupling metrics*. *CK metrics* indicate features of object-oriented software, and has been widely used [14][15].

Basic metrics include the following metrics:

- number of classes (*NOC*)
  Total number of classes in the selected scope
- number of methods (*NOM*)
  Total number of methods defined in the selected scope
- number of fields (*NOF*)
  Total number of fields defined in the selected scope
- number of overridden methods (*NORM*)
  Total number of methods in the selected scope that are overridden from ancestor classes
- number of parameters (*PAR*)
  Total number of parameters in the selected scope What is parameter?
- number of static methods (*NSM*)
  Total number of static methods in the selected scope
- number of static fields (*NSF*).
  Total number of static fields/attributes in the selected scope

Complexity metrics include the following metrics:

- method lines of code (*MLOC*)
- specialization index (*SIX*),
- McCabe cyclomatic complexity (*VG*)
- nested block depth (*NBD*).

CK metrics include the following metrics:

- weighted methods per class (*WMC*)
- depth of inheritance tree (*DIT*),
- number of children (*NSC*)
- lack of cohesion in methods (*LCOM*).

Coupling metrics include the following metrics:

- afferent/efferent coupling (*CA/CE*).

3.2.2 Eclipse Metrics Plugin

Until now, a lot of software metric measurement tools have been developed. Among them, *Eclipse Metrics Plugin* by *Frank Sauer* is the commonly used open source software plugin for *Eclipse IDE* for the metrics calculation and the dependency analyzer. It can measure various metrics and display the results in the integrated view. Actually, 23 metrics can be measured by this tool, which can be used for the quality assurance testing, the software performance optimization, the software debugging, the process management of software developments such as time or methodology, and the cost/size estimations of a project [16].

3.2.3 Adopted Seven Metrics

In this thesis, we use this tool to measure the necessary metrics to evaluate the quality of source codes from the students that pass the test code on *JUnit*. The following seven metrics are actually adopted in this thesis:

1. Number of Classes (*NOC*)
   This metric represents the number of classes in the source code.
2. Number of Methods (*NOM*)
   This metric represents the total number of methods in all the classes.
3. Cyclomatic Complexity (*VG*)
   This metric represents the number of decisions caused by the conditional statements in the source code. The larger value for *VG* indicates that the source code is more complex and becomes harder to be modified.
4. Lack of Cohesion in Methods (*LCOM*)
   This metric represents how much the class lacks cohesion. A low value for *LCOM* indicates that it is a cohesive class. On the other hand, the value close to 1 for *LCOM* indicates the lack of cohesion and suggests that the class might better be split into several (sub)classes. *LCOM* can be calculated as follows:

   1) Each pair of the methods in the class are selected.
   2) If they access to the disjoint set of instance variables, P is increased by one. If they share at least one variable, Q is increased by one. It is noted that P and Q are initialized by 0.

3) *LCOM* is calculated by:

$$LCOM = \begin{cases} P - Q & (\text{if } P > Q) \\ 0 & (\text{otherwise}) \end{cases} \tag{3.1}$$

5 Nested Block Depth (*NBD*)
This metric represents the maximum number of nests in the method. It indicates the depth of the nested blocks in the code.

6. Total Lines of Code (*TLC*)
This metric represents the total number of lines in the source code, where the comment and empty lines are not included.

7. Method Lines of Code (*MLC*)
This metric represents the total number of lines inside the methods in the source code, where the comment and empty lines are not included.

3.3 Summary

In this chapter, we reviewed the *informative test code approach* for the code writing problem, including its concepts and example, and introduced the software metrics to evaluate the quality of generated source codes by students.

Chapter 4

Application to Standard Input/output with Exception Handling

4.1 Introduction

The *code writing problem* in *JPLAS* asks a student to write a source code to satisfy the specifications of a given assignment that are described in the *test code* [11]. The *code writing problem* is implemented based on the *test-driven development (TDD) method* [5], using an open source framework *JUnit* [4]. *JUnit* automatically tests the codes on the server to verify their correctness using the *test code* when they are submitted by students. Thus, students can repeat the cycle of writing, testing, modifying, and resubmitting codes by themselves, until they can complete the correct codes for the assignments.

To register a new assignment for the code writing problem in *JPLAS*, a teacher has to prepare a *problem statement*, a *reference source code*, and a *test code* using a Web browser. The *problem statement* should describe the assignment overview and the important specifications of the code. It is noted that the reference source code is essential to verify the correctness of the problem statement and the test code. Then, a student should write a source code for the assignment while referring the statement and the test code, so that the source code can be tested by using the given test code on *JUnit*.

However, teachers at schools are usually not accustomed to writing a test code that can run on *JUnit*. Some teachers may spend much time in struggling to write a test code, and may register an incomplete test code that does not verify some requirements described in the problem statement correctly. This incomplete test code must be avoided because it may produce inappropriate feedback to a student and undermine confidence to JPLAS.

On the other hand, a commercial tool for generating a test code is usually expensive, and may not cover a test code that verifies the *standard input/output with exception handling* in a source code. The code by implementing the standard input/output with exception handling should be mastered by novice students at the early stage of Java programming educations as the first step programming for human interfaces.

In this chapter, we propose an *informative test code generation* for the code writing problem in *JPLAS*. It generates a test code using a reference source code to test the standard input/output with exception handling through the following steps: 1) a *test code template* is provided by our proposal, 2) a set of standard inputs to be tested are made by a teacher, 3) by running the reference code with each standard input, the corresponding expected standard output is extracted correctly, and 4) this pair of the standard input and the standard output are embedded into the test code template. By repeating steps 3) and 4) for every standard input, the test code can be completed. To run the source code using the test code on *JUnit*, it introduces the classes to handle the standard input/output functions as the memory access functions in [6].

To evaluate the proposed method, first, we applied it to 97 source codes in Java programming textbooks or Web sites that contain the standard input/output. It has been proved that the generated test codes could correctly verify the source codes except for one code using a random generator. Then, we generated the test codes for three problems and asked five students who are currently studying Java programming to write the source codes using them. It was found that they completed the codes that can pass the test codes, whereas the use of exception handling functions was sometimes insufficient or incorrect.

4.2 Standard Stream

In computer programming, the *standard stream* [56] represents a pre-connected input and output (I/O) communication channel between a computer program (process) and its environment (usually, host computer) when it begins execution [18]. The three typical I/O environments are *standard input stream (stdin)*, *standard output stream (stdout)*, and *standard error stream (stderr)*. Originally, the communication channel happened via a physically connected system console (input via keyboard, output via monitor), but standard streams abstract this.

4.2.1 Standard Input Stream (stdin)

Standard input stream is a stream from which a program reads its input data. The program requests data transfer by use of the *read* operation. *System.in* implements the *standard input stream*.

4.2.2 Standard Output Stream (stdout)

Standard output stream is a stream to which a program writes its output data. The program requests data transfer with the *write* operation. *System.out* implements the standard output stream.

4.2.3 Standard Error Stream (stderr)

Standard error stream is a stream to which a program writes its error messages. *System.err* implements the standard error stream.

4.3 Exception Handling

Exception handling is one of the powerful mechanism in Java [19] to handle the runtime errors so that normal flow of the application can be maintained even if errors occur there.

4.3.1 What is Exception Handling

In Java, the *exception* is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime. *Exception handling* is the mechanism to handle runtime errors. Java offers a variety of classes for *Exception handling* to handle each specific runtime error properly, such as *ClassNotFoundException*, *IOException*, *SQLException*, and *RemoteException*. The core advantage of exception handling is to maintain the normal flow of the application. An exception normally disrupts the normal flow of the application that is why we use exception handling.

4.3.2 Hierarchy of Java Exception Classes

java.lang.Throwable class is the root class of the exception hierarchy in Java which is inherited by two subclasses: *Exception* and *Error*. The hierarchy of exception classes in Java are given below:

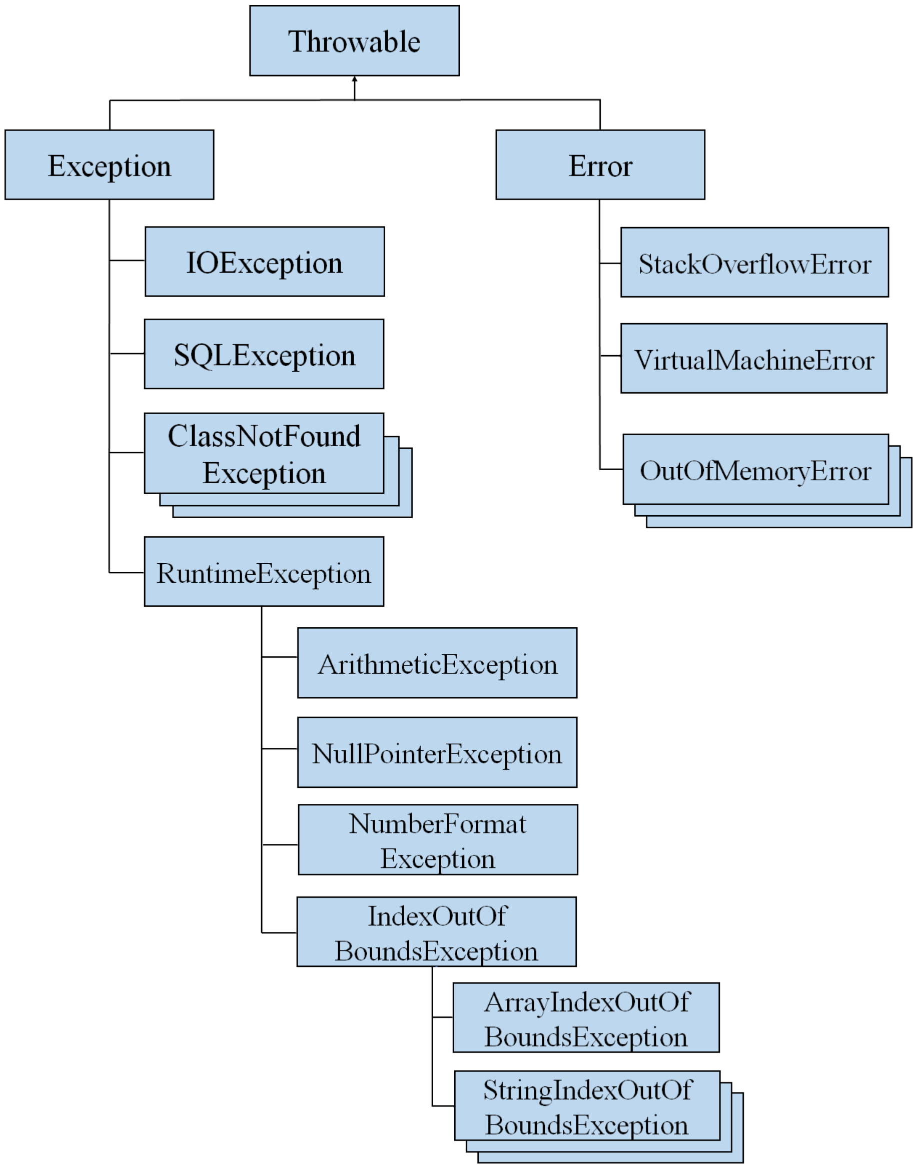


Figure 4.1: Hierarchy of exception classes in Java.

4.3.3 Types of Exceptions

There are mainly two types of exceptions: *checked* and *unchecked* as an *error* is considered as the unchecked exception. However, according to *Oracle*, there are three types of exceptions:

1. checked exception
2. unchecked exception
3. error

4.4 Informative Test Code Generation

In this section, we propose the informative test code generation for standard input/output with exception handling for the code writing problem.

4.4.1 Scope of Source Code under Test

At the early stage of the Java programming education, the responsibility of a student is to master how to write a source code that contains the standard input/output with exception handling. Thus, a teacher in a Java programming course should prepare a considerable number of assignments for writing source codes containing them, where many Java programming textbooks offer such assignments for novice students.

The source code for this study must contain the functions for the standard input/output and the exception handling. Then, if the proper data is given to the code from the standard input, it must handle it correctly and outputs the message specified in the assignment to the standard output. On the other hand, if the improper data is given, it must handle it using the exception handling command without abortion and outputs the corresponding message.

4.4.2 Requirements in Test Code

Subsequently, the test code must satisfy the following requirements:

1. The input data from the standard input (keyboard) must be described in the test code to test the standard input in the source code.
2. The output data to the standard output (console) must be received by the test code to test the standard output in the source code.
3. The input data must be elaborated in the test code for the standard input.
4. The input data in the test code should cover any possible one for the standard input, including the proper and improper ones.
5. The expected output data for each input data must be narrated in the test code correctly.

4.4.3 Solutions for Requirements

The test code generation method adopts the following functions and commands to solve the above-mentioned requirements by referring the test code implementation in [6]:

- To describe the standard input data to the source code, the *Inputln* method in *StandardInput-Snatcher* class is adopted in the test code. It is noted that *StandardInputSnatcher* class is extended from *InputStream* class.

- To receive the standard output data from the source code, the *readLine* method in *Standard-OutputSnatcher* class is adopted in the test code. It is noted that *StandardOutputSnatcher* class is extended from *PrintStream* class.
- Any possible standard input data is prepared by a teacher beforehand. It is used in the argument of *Inputln*.
- To obtain the expected standard output data from the code for each input data, the reference source code is executed with this input data.
- Each pair of the standard input and output data is embedded into the test code.

4.4.4 Conditions of Source Code

Currently, to avoid the complexity, the proposed method confines the applicable source code that satisfies the following conditions:

1. it has the *main* method only.
2. it contains the standard input function.
3. it contains the standard output function for handling the proper input.
4. it contains the standard output function for handling the exception.

It is noted that a source code containing multiple standard input/output functions can be handled by increasing the number of *Inputln* or *assertThat* in the test code accordingly. Besides, if a code does not have the *main* method, it can be handled by describing the proper statements to execute the method for the standard input/output in the test code.

An example source code in this scope is as follows:

source code 2

```
1  import java.util.Scanner;
2  public class Sample {
3      public static void main(String args[]){
4          int number;
5          Scanner scan = new Scanner(System.in);
6          try{
7              System.out.print("Enter an integer");
8              String actual = scan.nextLine();
9              number = Integer.parseInt(actual);
10             System.out.println(number +": is input number");
11         } catch(NumberFormatException e) {
12             System.out.print("NumberFormatException occurs!");
13         }
14     }
15 }
```

source 2 accepts an integer data from a console and outputs a message with this data on a display. In this source code, 1) it has only the *main* method at line 3, 2) *scan* object of *Scanner* class is defined at line 5 as the standard input function, 3) *System.out.println* is called at line 10 as the standard output function for handling the proper input, and 4) *System.out.println* is called at line 12 as the standard output function for handling the exception.

4.4.5 Test Code Template

The proposed method provides the *test code template* containing the required functions for the above mentioned source code. The following code describes the core part of the test code template starting from @*Test*. In advance, several *import* statements to use related libraries, and the instance generations for the *StandardInputSnatcher* and *StandardOutputSnatcher* classes are necessary. Besides, the definitions of these classes are also required to complete the test code template.

In this template, *in.InputIn* at line 29 gives the standard input data to the source code, where *in* is an instance of *StandardInputSnatcher* class. The statements at lines 30-37 run the source code and read the standard output data for this input data, where *out* is an instance of *StandardOutput-Snatcher* class. *expected* at line 38 represents the expected output data of the source code. The blanks *""* at lines 29 and 38 should be filled by the standard input and output data. *assertThat* at line 39 compares the expected data with the output data of the code. The whole statements at lines 25-40 should be prepared for each input data.

test code template

```
 1 import static org.hamcrest.CoreMatchers.is;
 2 import static org.junit.Assert.assertThat;
 3 import static org.junit.Assert.*;
 4 import java.io.InputStream;
 5 import org.junit.Before;
 6 import org.junit.Test;
 7 import Snatcher.StandardOutputSnatcher;
 8 import java.io.BufferedReader;
 9 import java.io.ByteArrayOutputStream;
10 import java.io.IOException;
11 import java.io.InputStream;
12 import java.io.PrintStream;
13 import java.io.StringReader;
14
15 public class TemplateTest {
16     private StandardInputSnatcher in = new StandardInputSnatcher();
17     private StandardOutputSnatcher out = new StandardOutputSnatcher();
18
19     @Before
20     public void setUp() {
21         System.setIn(in);
22         System.setOut(out);
23     }
24
25     @Test
26     public void test1() throws Exception {
27         StringBuffer bf = new StringBuffer();
28         String actual,line,expected;
29         in.Inputln(""); // standard input
30         Sample.main(new String[0]);
31         System.out.flush();
32         while ((line = out.readLine()) != null) {
33             if (bf.length() > 0)
34                 bf.append("\n");
35             bf.append(line);
36         }
37         actual = bf.toString();
38         expected = ""; // expected standard output
39         assertThat(actual,is(expected));
40     }
41 }
```

4.4.6 Test Code Generation Procedure

The test code generation procedure using the *test code template* in the proposed method is as follows:

1) A teacher prepares the reference source code for the assignment.
2) He/she prepares a set of possible standard input data to the source code.
3) He/she runs the source code by using each standard input data and observes the corresponding standard output data.
4) He/she embeds the standard input data into “” at line 5 and the observed standard output data into “” at line 13 in the test code template.

As the possible standard input data in step 2), the following five data types should be considered. Then, the teacher needs to select one value for each data type, which is used in step 3).

- positive integer: 5
- negative integer: -14
- zero integer: 0
- floating-point number: 0.5
- one-byte character: “a b c”
- two-byte character: “A B C”

4.4.7 Generated Test Code Example

This subsection introduces an example of the test code generated by applying the proposed method to **source code 2**. The file name for the generated test code is given as *SampleTest.java*. The following **test code 2** shows a part of the test code.

test code2

```
........

@Test
public void test1() throws Exception {
      StringBuffer bf = new StringBuffer();
      String actual,line,expected;
      in.Inputln("5"); // proper standard input data
      Sample.main(new String[0]);
      System.out.flush();
      while ((line = out.readLine()) != null) {
            if (bf.length() > 0)
            bf.append("\n");
            bf.append(line);
      }
      actual = bf.toString();
      expected = "Enter an integer" +
      "5: is input number";
      assertThat(actual, is(expected));
}

@Test
public void test2() throws Exception {
    StringBuffer bf = new StringBuffer();
    String actual,line,expected;
    in.Inputln("abc"); // improper standard input data
    Sample.main(new String[0]);
    System.out.flush();
    while((line = out.readLine()) != null) {
        if (bf.length() > 0)
        bf.append("\n");
        bf.append(line);
    }
    actual = bf.toString();
    expected = "Enter an integer" + "NumberFormatException occurs!";
    assertThat(actual,is(expected));
}
.......
```

4.5 Evaluation

In this section, we evaluate the effectiveness of the proposed *test code generation method* in terms of generating test codes from existing source codes and writing source codes using the test codes by students.

4.5.1 Test Code Generation Results

First, we evaluate the proposed method in generating test codes from source codes. For this purpose, 97 source codes were collected from Java programming textbooks or Web sites [20]-[24], and the test codes were generated by applying the proposed method. It is noted that some codes in [23] were modified to use the standard input/output through the console instead of using the dialog box. Then, the correctness of each test code was examined by testing the original source code. It was found that our method generated the test codes that can pass original codes correctly except for one source code, which outputs a random number generated in the code. Thus, the effectiveness of the proposed method was confirmed.

The following **source code 3** shows an example source code in [20] where the method successfully generates the test code shown in **test code 3**. It is noted that *try* - *catch* is used here instead of *throws* in the original source code.

source code 3

```
import java.io.*;
class Sample3 {
    public static void main(String[] args) throws IOException {
        try {
            System.out.println("Enter two integers");
            BufferedReader br =
            new BufferedReader(new InputStreamReader(System.in));
            String str1 = br.readLine();
            String str2 = br.readLine();
            int num1 = Integer.parseInt(str1);
            int num2 = Integer.parseInt(str2);
            System.out.println("The sum is " + (num1+num2) + ".");
        } catch(NumberFormatException e) {
            System.out.print("NumberFormatException occurs!");
        }
    }
}
```

test code3

```
.......

@Test
public void test1() throws Exception {
    StringBuffer bf = new StringBuffer();
    String actual,line,expected;
    in.Inputln("2"); in.Inputln("7");// proper standard input data
    Sample.main(new String[0]);
    System.out.flush();
    while((line = out.readLine()) != null) {
        if (bf.length() > 0)
        bf.append("\n");
        bf.append(line);
    }
    actual = bf.toString();
    expected = "Enter two integers" +"The sum is 9.";
    assertThat(actual,is(expected));
}

@Test
public void test2() throws Exception {
    StringBuffer bf = new StringBuffer();
    String actual,line,expected;
    in.Inputln("0.5"); in.Inputln("-3");// improper standard input data
    Sample.main(new String[0]);
    System.out.flush();
    while((line = out.readLine()) != null) {
        if (bf.length() > 0)
        bf.append("\n");
        bf.append(line);
    }
    actual = bf.toString();
    expected = "Enter two integers" + "NumberFormatException occurs!";
    assertThat(actual,is(expected));
}
........
```

4.5.2 Source Code Writing Results

Next, we evaluate the proposed method in writing source codes with generated test codes by five students who are currently studying Java programming and have same technical levels. For this purpose, we prepared the following three problems, where all the students completed the source codes that pass the test codes for any problem.

4.5.2.1 Problem #1

In problem #1, the code accepts an integer data from a console, and outputs a message with this data to a console, where **source 2** is the reference source code and **test 2** is the test code. The source code from a student is expected to use *NumberFormatException* to check the input data format. Then, three students use this class for the exception handling, and one uses *Exception*. However, one student does not use it where he implements the data format checking function.

4.5.2.2 Problem #2

In problem #2, the code accepts an integer index from a console, and outputs the indexed data from the data array. The student code is expected to use *ArrayIndexOutofBoundException* to check the range of the index. Then, only one student uses this class. The other students implement the index checking function in the codes. Two students use *IOException*, and two students do not use any class for the exception handling. No student use *NumberFormatException* to check the input data format, although the class was requested in problem #1. Unfortunately, many students cannot integrate the knowledge that has been studied sequentially.

4.5.2.3 Problem #3

In problem #3, the code accepts a file path from a console, and outputs the string at the first line in the file. The student code is expected to use *FileNotFoundException* or *IOException* to check the file path. Then, three students use *FileNotFoundException*, one uses *Exception*, and one uses *IOException*.

4.5.2.4 Summary of Student Applications

This simple experiment of our proposal shows that the students can generally complete source codes using standard input/output with exception handling that can pass the generated test codes. However, their use of the class for the exception handling is sometimes insufficient or incorrect. It has been observed that these students are not experts, which causes the difference in their source codes, although they have enough programming skills. To let them understand the correct use, it is necessary to improve the proposed method.

4.6 Summary

In this chapter, we proposed the *test code generation method* for the code writing problem in *JPLAS* that requires implementing a Java source code containing the *standard input/output* with *exception handling*. To access the standard input/output from the test code on *JUnit*, the *test code template* is first prepared with the *input/output snatcher classes*. Then, the test code is completed by embedding the input and output extracted by running the *reference source code* into the template. This proposal is helpful in reducing the teacher load in writing the test code for the programming assignment that requires the standard input/output with exception handling, which is common for novice students. The effectiveness is evaluated through applying the method to 97 source codes in Java programming text books or Web sites, and asking five students to write source codes using the generated test codes for three problems.

In future studies, we will extend the proposed method to handle other input/output functions, and other methods than the *main* method, and improve the readability of the generated test code to make it easier for novice students.

Chapter 5

Application to Java Collection Framework

5.1 Introduction

Nowadays, Java has been broadly used in various practical applications as a highly reliable, portable, and scalable objected oriented programming language. Java is a versatile general-purpose programming language that can be used to create cross-platform applications for desktop, mobile, or Web systems [25]. Java was the most popular programming language in 2015 [26] and is in the third in 2018 [27]. Therefore, there have been strong demands of industries for Java programming educations. Correspondingly, a plenty of universities and professional schools are currently offering Java programming courses to meet this challenge. A typical Java programming course consists of grammar instructions and programming exercises.

To improve programming exercise environments, we have developed *Java Programming Learning Assistant System (JPLAS)* [28]. *JPLAS* performs excellently not only in reducing teacher loads by marking answers automatically but also in advancing student motivations with immediate responses. JPLAS offers the code writing problem [29], which asks a student to write a source code that passes the given test code on JUnit [4]. To help a student write a complex code, the detailed information for the source code implementation is described in the test code. Previously, we confirmed the effectiveness of this *informative test code* approach in studying the *three object-oriented programming concepts* for Java [30]. It is expected that a student will learn how to use the concepts by writing a source code which can pass the test code.

Java collections framework (JCF) [31] provides a strong and useful architecture to store or control a group of objects by offering the appropriate library methods. *JCF* and *arrays* are similar in that they both hold references to objects, and can be managed as a group. However, unlike arrays, *JCF* does not need to be assigned a certain capacity when instantiated. It can grow and shrink in size automatically when objects are added or removed. *Array* can hold basic data type elements (primitive types) such as *int*, *long*, or *double*. On the other hand, *JCF* holds *Wrapper Classes* such as *Integer*, *Long*, or *Double*. Thus, the students should master the proper use of *JCF* at Java programming study. In this thesis, *List*, *Set*, and *Map* are selected as the most useful interfaces in *JCF*.

In this chapter, we present the application of the informative test code approach for studying the three interfaces in *JCF*. This test code intends to test whether a method in a source code uses the proper method in the *JCF* library. For evaluations, we generated five informative test codes for *JCF*, and asked 19 students from Japan, Myanmar, China, and Indonesia to implement the source codes. Then, all of them successfully completed the source codes whose software metrics confirm the sufficient quality. However, certain students did not properly use methods in the *JCF* library.

5.2 Informative Test Code Approach for Java Collections Framework

In this section, we present the informative test code approach for the code writing problem using *Java Collections Framework (JCF)* in *JPLAS*. By solving code writing problems with informative tests codes for various *JCF* libraries, it is expected that students can master the proper use of *JCF*.

5.2.1 Review of Java Collections Framework

Java Collections Framework (JCF) is a hierarchy of interfaces and classes that are used for storing and manipulating a group of objects. The most useful interfaces in *JCF* are *List, Set*, and *Map*. An *iterator* method is prepared to go through all the elements in each collection.

List is a more flexible version of an array to be used to insert and retrieve the objects. It allows the duplicated objects. *List* includes the methods for *get, set, add, indexOf, and lastIndexOf*. *List* can be implemented as *ArrayList* or *LinkedList*. *Set* is an unordered collection of objects and does not allow duplicate objects. Any object in *Set* cannot be accessed by using the index. Set can be implemented as *TreeSet, HashSet,* or *LinkedHashSet*. *Map* is an object that maps keys to values and cannot contain duplicate keys. *Map* includes the methods for *put, get, remove, containsKey*, and *containsValue*. The three basic implementations of *Map* are *HashMap, TreeMap*, and *LinkedHashMap*. Iterator is used in *JCF* to retrieve elements one by one [32]. There are three *iterator* interfaces: *Enumeration, Iterator*, and *ListIterator*.

5.2.2 Overview of Informative Test Code for JCF

The *informative test code* describes the necessary information for implementing the source code using the key point under study, such as the grammar, the framework, or the concept. This test code includes the class names, the method names, the access modifiers, and the data types for the important member variables, the argument types, and the returning data types for the methods, and the exception handling in the expected source code. By writing a source code to pass this test code, a student is expected to learn how to use the point under study in the source code. In general, an *informative test code* consists of a test method for testing names and data types of member variables in the source code, named "*variableNameTest*", a test method for testing the number of methods, their names, and returning data types, named "*methodNameTest*", and one or more test methods for testing actions of methods in the code, named "*(action)Test*".

5.2.3 Informative Test Code for List

The **test code 1** demonstrates an informative test code for List. *variableNameTest* tests that in the source code, "*ArrayListImp class*" is implemented, one variable "*list*" is defined, and its data type is *List* object. *methodNameTest* tests that in the code, three methods exist, their names are "*addImp*", "*swapImp*", and "*removeImp*", they have no arguments, and the returning data type is *List* object. *addTest* tests that *addImp* returns the *List* object of *String*, and adds "*Red*", "*Green*", "*Blue*", "*Black*" and "*Orange*" to list in this order. *swapTest* tests that *swapImp* exchanges the first and third data in list. *removeTest* tests that removeImp removes *Red* and *Blue* from list. **source code 1** shows an example source code for the **test code 1**.

test code 1

```
public class ArrayListImpTest {
    @Test
    public void variableNameTest() throws NoSuchFieldException, SecurityException {
        ArrayListImp obj = new ArrayListImp();
        Field f1 = obj.getClass().getDeclaredField("list");
        Field[] f = obj.getClass().getDeclaredFields();
        assertEquals(1, f.length);
        assertEquals(f1.getType(), List.class);
    }
    @Test
    public void methodNameTest() throws NoSuchMethodException, SecurityException
    {
        ArrayListImp obj = new ArrayListImp();
        Method[] m = obj.getClass().getDeclaredMethods();
        assertEquals(3, m.length);
        Method m1 = obj.getClass().getDeclaredMethod("addImp", null);
        Method m2 = obj.getClass().getDeclaredMethod("swapImp", null);
        Method m3 = obj.getClass().getDeclaredMethod("removeImp", null);
        assertEquals(List.class, m1.getReturnType());
        assertEquals(List.class, m2.getReturnType());
        assertEquals(List.class, m3.getReturnType());
    }
    @Test
    public void addTest() {
        ArrayListImp obj = new ArrayListImp();
        List<String> list = obj.addImp();
        assertEquals(5, list.size());
        assertEquals("Red", list.get(0));
        assertEquals("Green", list.get(1));
        assertEquals("Blue", list.get(2));
        assertEquals("Black", list.get(3));
        assertEquals("Orange", list.get(4));
    }
    @Test
    public void swapTest() {
        ArrayListImp obj = new ArrayListImp();
        List<String> list = obj.addImp();
        obj.swapImp();
        assertEquals(5, list.size());
        assertEquals("Blue", list.get(0));
        assertEquals("Red", list.get(2));
    }
    @Test
    public void removeTest() {
        ArrayListImp obj = new ArrayListImp();
        List<String> list = obj.addImp();
        obj.removeImp();
        assertEquals(3, list.size());
        assertEquals("Green", list.get(0));
        assertEquals("Black", list.get(1));
        assertEquals("Orange", list.get(2));
    }
}
```

source code 1

```
public class ArrayListImp {
    List<String> list = new ArrayList<String>();
    public List<String> addImp() {
        list.add("Red");
        list.add("Green");
        list.add("Blue");
        list.add("Black");
        list.add("Orange");
        return list;
    }
    public List<String> swapImp() {
        Collections.swap(list, 0, 2);
        return list;
    }
    public List<String> removeImp() {
        list.remove(0);
        list.remove(1);
        return list;
    }
}
```

5.2.4 Informative Test Code for Set

The **test code 2** reveals an *informative test code for Set*. variableTest tests that in the source code, *TreeSetImp* class is implemented, two variables, "*tset*" and "*iterator*", are defined, and their data types are *TreeSet* of *String* objects and *Iterator* respectively. The *methodTest* tests that the number of the methods is three, their names are "*addImp*", "*removeImp*", and "*displayImp*", they have no argument, and their returning data types are *TreeSet*. The *addTest* tests that every element in tset is *String* object, the number of elements is six, the first element is "*ABC*", and the last one is "*Test*". The *removeTest* tests that two elements are removed from *tset*, the first element becomes "*ABC*", and the last one becomes "*String*". The *displayTest* tests that the output data from tset at the console is "*ABC Ink Jack Pen String Test*" after applying *addImp* to *tset*.

Test code 2

```
public class TreeSetImpTest {
    ...............
    @Test
    public void addTest() {
        TreeSetImp obj = new TreeSetImp();
        TreeSet<String> tset = obj.addImp();
        for (Object x : tset) {
            assertEquals("String".getClass().getName(), x.getClass().getName());
        }
        assertEquals(6, tset.size());
        assertEquals("ABC", tset.first());
        assertEquals("Test", tset.last());
    }
    @Test
    public void removeTest() {
        TreeSetImp obj = new TreeSetImp();
        TreeSet<String> tset = obj.addImp();
        obj.removeImp();
        assertEquals(4, tset.size());
        assertEquals("ABC", tset.first());
        assertEquals("String", tset.last());
    }
    @Test
    public void displayTest() {
        TreeSetImp obj = new TreeSetImp();
        TreeSet<String> tset = obj.addImp();
        ByteArrayOutputStream baos=new ByteArrayOutputStream(1024);
        PrintStream s = System.out;
        PrintStream st = new PrintStream(baos);
        System.setOut(st);
        obj.displayImp();
        System.setOut(s);
        assertEquals("ABC Ink Jack Pen String Test", baos.toString().trim());
    }
}
```

The **source code 2** shows an example source code for **test code 2**. In *addImp*, the six strings are added to *tset* with the consideration of *displayTest*. In *removeImp*, one element among "*Ink*", "*Jack*", "*Pen*", and "*Test*" is removed. Here, "*Pen*" and "*Test*" are actually removed. In *displayImp*, each element in tset is output at the console sequentially using *Iterator*.

Source Code 2

```
public class TreeSetImp {
    TreeSet<String> tset = new TreeSet<String>();
    Iterator iterator;
    public TreeSet<String> addImp() {
    tset.add("ABC");
    ...............
    tset.add("Test");
    return tset;
}
public TreeSet<String> removeImp() {
    tset.remove("Pen");
    tset.remove("Test");
    return tset;
}
public void displayImp() {
    Iterator<String> itr = tset.iterator();
    while (itr.hasNext()) {
        String item = itr.next();
        System.out.print(item + " ");
    }
}
```

5.2.5 Informative Test Code for Map

Test code 3 exhibits an *informative test code* for *Map*. *variableTest* tests that in the source code, "*HashMapImp class*" is implemented, one variable, "*hmap*", is defined, and the data type is "*HashMap*". *methodTest* tests that the number of the methods is three, their names are "*putImp*", "*removeImp*", and "*displayImp*", they have no argument, and their returning data types for the first two methods are "*HashMap*" and no returning data for the last one. *putTest* tests that *hmap* accepts the returning data with a pair of *Integer* and *String*, *putImp* adds "*Coconut*", "*strawberry*", "*Apple*", "*Mango*", and "*Orange*" to *hmap* using 1 to 5 for the key. *displayTest* tests that one element is removed from hmap, and the output data from *hmap* at the console is "*1=Coconut, 2=Strawberry, 4=Mango, 5=Orange*" after applying *putImp* and *removeImp* to *hmap*.

source code 3 shows an example source code for **test code 3**. In *removeImp*, "*Apple*" is removed. In *displayImp*, each key and value in *hmap* are output at the console sequentially using *Iterator*.

Test Code 3

```
public class HashMapImpTest {
    ...............
    @Test
```

```
        public void putTest() {
            HashMapImp obj = new HashMapImp();
            HashMap<Integer,String> hmap = obj.putImp();
            assertEquals(5, hmap.size());
            assertEquals("Coconut", hmap.get(1));
            ...............
            assertEquals("Orange", hmap.get(5));
        }
        @Test
        public void displayTest() {
            HashMapImp obj = new HashMapImp();
            HashMap<Integer,String> hmap = obj.putImp();
            obj.removeImp();
            ...............
            assertEquals("1=Coconut 2=Strawberry 4=Mango 5=Orange", baos.toString
        ().trim());
        }
}
```

Source Code 3

```
public class HashMapImp {
    HashMap<Integer, String> hmap = new HashMap<Integer, String>();
    public HashMap<Integer, String> putImp() {
        hmap.put(1, "Coconut");
        ................
        return hmap;
    }
    HashMap<Integer, String> removeImp() {
        hmap.remove(3);
        return hmap;
    }
    public void displayImp() {
        Iterator<Entry<Integer, String>> itr = hmap.entrySet().iterator();
        String str = "";
        while (itr.hasNext()) {
            Map.Entry<Integer, String> pair = (Map.Entry<Integer, String>) itr.next();
            str += pair.getKey() + "=" + pair.getValue() + "  ";
        }
        System.out.println(str);
    }
}
```

5.2.6 Evaluation

In this section, we evaluate the *informative test code approach for JCF*. We generated five informative test codes for *JCF*, three for *List*, one for *Set*, and one for *Map*. Then, 19 students with different programming skills from Japan, Myanmar, China, and Indonesia were asked to complete the source codes passing the test codes. In the end, we measured the software metrics using *metrics plugin for Eclipse* [33].

Table 5.1 shows the summary of the measured metrics of the source codes from the students. It is noted that all the students completed the source codes. This table indicates that generally, they are high quality since the metrics values exist in superior ranges.

First, NOM (number of methods) shows that one student always uses one more method than the others, because he creates the constructor to initialize the variables. The test code on *JUnit* can test the number of constructors and the number of methods separately.

However, *metrics plugin for Eclipse* counts them together in total. Thus, it appears to be difficult to control the implementation of the constructor by the test code. Next, for *swapImp* in *ArrayListImp*, nine students used *Collections.swap* as the proper *JCF* library function to exchange two objects in the list as in **source code 1**, which is expected to study *JCF*.

By contrast, four students did not adopt the library function, and two students directly set the data in the list, instead of swapping them, which are not expected. For *displayImp* in *TreeSetImp*, 12 students used *iterator* to display the data in *tset* as expected. However, one student used the *enhanced for-loop* statement instead of *iterator*, and two students directly displayed the expected result in the test code by copying it instead of displaying the data in *tset*. The strategy to avoid them will be explored in future works.

Table 5.1: Comparison of metric values for JCF

Problem Name	*NOC*	*NOM*	*VG*	*NBD*	*LCOM*	*TLC*	*MLC*
BookData	2	3-4	2	2	0	31-40	11-19
ArrayListImp	1	4-5	1-1.25	1-1.25	0.5-0.75	29-36	12-19
LinkedListImp	1	4-5	1-1.50	1-1.50	0	23-29	9-15
TreeSetImp	1	3-4	1-1.33	1-1.33	0.5-0.75	25-32	13-18
HashMapImp	1	3-4	1-1.33	1-1.33	0	20-28	9-14

5.3 Summary

In this chapter, we presented the *informative test code approach* to the code writing problem for studying the *Java Collections Framework (JCF)* in JPLAS. The informative test code describes the detailed information for the code implementation including methods and variables. We evaluated the effectiveness of the approach by asking 19 students to write source codes using *JCF* libraries for five informative test codes, where all of them completed high quality source codes that pass the test codes. However, some students did not use *JCF* library functions.

In future works, we will improve the informative test code to confirm the proper use of library functions for *JCF*.

Chapter 6

Student Answer Analyzing Function for Desktop-version JPLAS

6.1 Introduction

To enhance Java Programming educations, we have developed the *Java Programming Learning Assistant System (JPLAS)* as an online Web application system. This online system can be used only in Internet available environments. Since many students who live in developing counties may need to access JPLAS at no Internet-access places, we have implemented offline *Desktop-version JPLAS (D-JPLAS)*, which can run without the Internet [3]. Currently,*D-JPLAS* supports five types of programming problems that cover various stages of Java programming study. *D-JPLAS* offers teacher service function and the student service functions.

In this chapter, we implement the *student answer analyzing function* for the five problem types in *D-JPLAS* so that a teacher can mark the student answers on his/her PC as easily as possible. This function helps a teacher in grading the students and giving feedbacks to them, by summarizing the answer results that are described in a lot of text files, and/or the answer source code files submitted from the students. We evaluate this function for the element fill-in-blank problem solutions by students in Japan, Myanmar, China, and Indonesia.

6.2 Desktop-version Java Programming Learning Assistant System (D-JPLAS)

In this section, we introduce the overview of the *Desktop-version JPLAS (D-JPLAS)*. *D-JPLAS* does not use the online database and the web server. Currently, *D-JPLAS* offers *teacher service functions* for generating programming assignments, managing answer files from students, and analyzing their answers, and *student service functions* for solving and answering assigned problems.

Figure 6.1 illustrates the usage flow of *D-JPLAS*. In *D-JPLAS*, the teacher, who has the authority of managing the assignments, needs to create and assign the programming assignment first. Then, he/she distributes them to the students to solve them for improving programming skills. A student can repeat solving the assignments on his/her own PC on offline. After that, the student submits the answer files to the teacher who stores the files in the respective folder for each problem type and the student. Finally, the teacher analyzes the files submitted by the students using the answer analysis function, and gives feedbacks to the students. In *D-JPLAS*, the teacher and the students exchange the files of assignments and answers by using USB memories, file servers, or emails if the Internet is accessible.

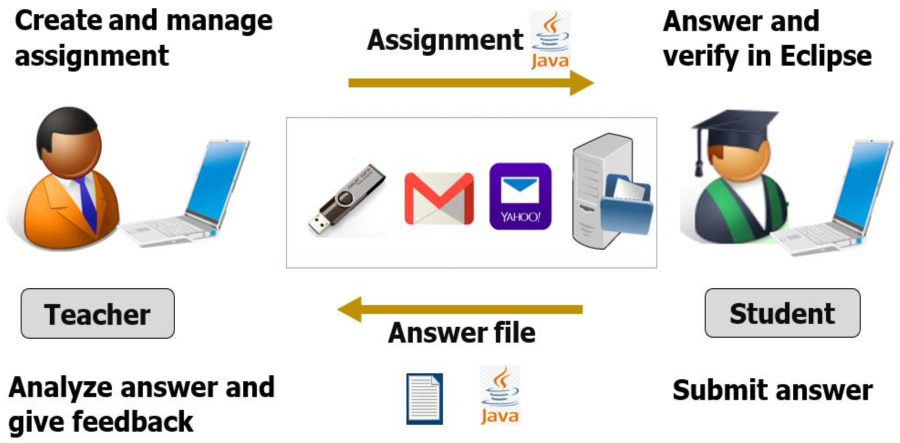

Create and manage assignment
Assignment
Java
Answer and verify in Eclipse
YAHOO!
Teacher
Student
Answer file
Analyze answer and give feedback
Java
Submit answer

Figure 6.1: Usage flow of Desktop-version JPLAS.

6.2.1 Problem Types in D-JPLAS

Currently, *D-JPLAS* supports the following five types of programming problems in *JPLAS*:

- Element fill-in-blank problem (EFP):
  In the element fill-in-blank problem (EFP) [8], a Java code with several blank elements is given to a student who is requested to fill in the proper word for each blank in the problem source code. To solve this problem, a student needs to carefully read the problem code to understand the structure, the algorithm/logic, and the semantics. Subsequently, the student can fill in the blanks correctly by applying grammar and syntax rules.

- Value trace problem (VTP):
  In the value trace problem (VTP), a student is requested to answer the actual values of important variables in the given Java source code that implements a fundamental data structure or algorithm [35, 34]. To trace the values of the important variables, the blank line selection algorithm is developed to automatically make blanks for the output data line of the code where at least one data is changed from the previous one.

- Code correction problem (CRP):
  In the code correction problem (CRP), a student is requested to correct the incorrect elements in the given problem code. It aims the programming practice in reading and debugging the code so that it can pass the given test code on *JUnit*. The problem code has several errors that cannot be passed by the test code. This code is generated from the sample source code by using the error generation algorithm.

- Statement element fill-in-blank problem (SFP):
  In the statement element fill-in-blank problem (SFP), a student is requested to fill in the blank statements in the given problem code. The correctness of the answer is verified by the test code on *JUnit*. The blank statements are selected by generating the Program Dependence

Graph (PDG) of the code and finding the statements that have the largest dependences in PDG.

- Code writing problem (CWP):
  In the code writing problem (CWP), a student is requested to complete the source code that can pass the given test code. The test code describes the detailed specifications of the source code. This problem is based on the test-driven development (TDD) method, using an open source framework *JUnit*. A student can repeat writing, testing and modifying the source code.

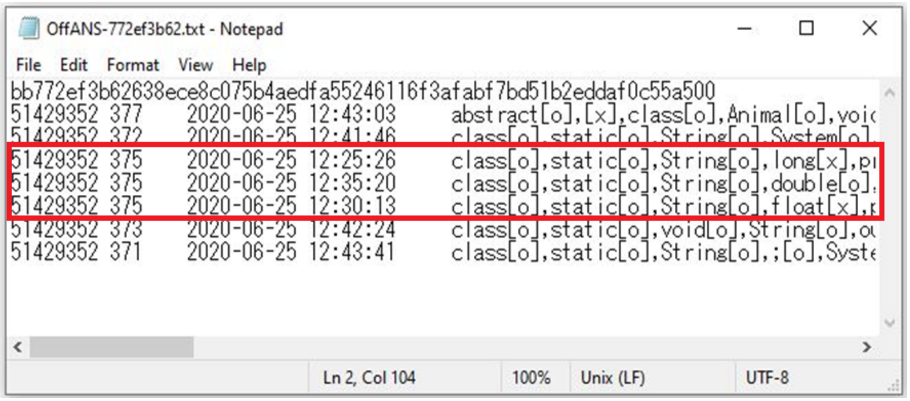
6.3 Student Answer Analyzing Function

In this section, we present the implementation of the function for analyzing student answers for the five problem types in *D-JPLAS*. In EEP and VTP, the student answer is marked through *string matching* with the correct answer. In CRP, SFP, and CWP, the student answer is marked through *software test* running the test code on *JUnit* [4]. These marking functions run on the PCs of students. The students need to submit the answer files of the results to the teacher. If an answer file contains duplicated submissions for a problem, this function selects the one with the last submission time.

6.3.1 Input Data

As the input data to the student answer analyzing function, each row of the answer file contains the student ID, the problem ID, the answer date and time, and the answers and their marking results at each submission.

Figure 6.2 shows the example answer file for EEP, where the answer to problem #375 was submitted three times.

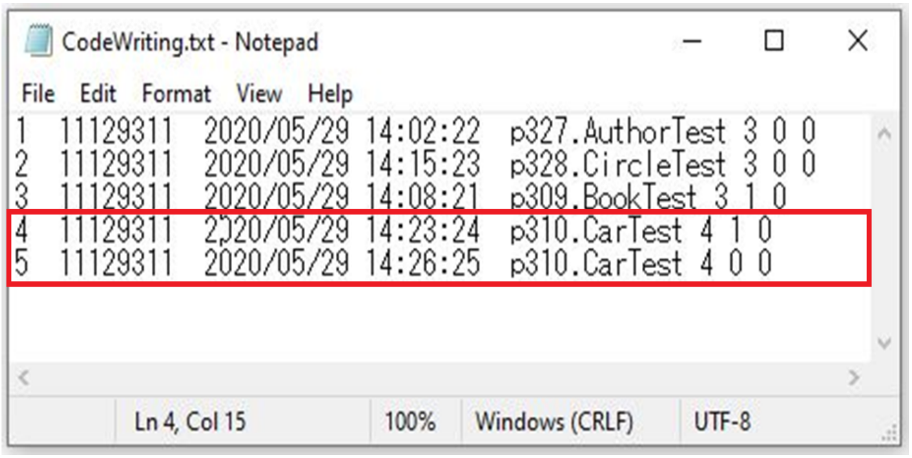


```
bb772ef3b62638ece8c075b4aedfa55246116f3afabf7bd51b2eddaf0c55a500
51429352 377    2020-06-25 12:43:03     abstract[o],[x],class[o],Animal[o],voi
51429352 372    2020-06-25 12:41:46     class[o],static[o],String[o],System[o],
51429352 375    2020-06-25 12:25:26     class[o],static[o],String[o],long[x],p
51429352 375    2020-06-25 12:35:20     class[o],static[o],String[o],double[o],
51429352 375    2020-06-25 12:30:13     class[o],static[o],String[o],float[x],
51429352 373    2020-06-25 12:42:24     class[o],static[o],void[o],String[o],o
51429352 371    2020-06-25 12:43:41     class[o],static[o],String[o],;[o],Syst
```

Figure 6.2: Example answer file for EFP.

- Each row of the answer file describes the following data:
  51429352 2020-06-25 12:41:46 class[o],static[o],String[o],for[o],int[o],void[o],String [o],)[x],
  Here, 51429352 indicates the student ID, 2020-06-025 12:41:46 does answer date and time, class does the answer by the student for the first question, and [o] indicates the correct answer where [x] does the incorrect answer.

Figure 6.3 shows the example answer file for CWP, where the answer to problem #310 was submitted two times.


CodeWriting.txt - Notepad

File Edit Format View Help

```
1  11129311  2020/05/29 14:02:22  p327.AuthorTest 3 0 0
2  11129311  2020/05/29 14:15:23  p328.CircleTest 3 0 0
3  11129311  2020/05/29 14:08:21  p309.BookTest 3 1 0
4  11129311  2020/05/29 14:23:24  p310.CarTest 4 1 0
5  11129311  2020/05/29 14:26:25  p310.CarTest 4 0 0
```

Ln 4, Col 15 | 100% | Windows (CRLF) | UTF-8

Figure 6.3: Example answer file for CWP.

- Each row of the answer file in describes the following data:
  11129311 2020/05/29 14:23:24 p310.CarTest 4 1 0
  Here, 11129311 indicates the student ID, 2020/05/29 14:23:24 does answer date and time, p310.CarTest does the package and class name, and 4 1 0 indicates one test was failed among four tests.

6.3.2 Output Data

As the output data, the function outputs the correct answer rate of each student for each problem.

- EFP, VTP:
  1. Count the number of blank questions and the number of correct answers of each student for each problem.
  2. Calculate the correct answer rate by the following equation Eq. (6.1)

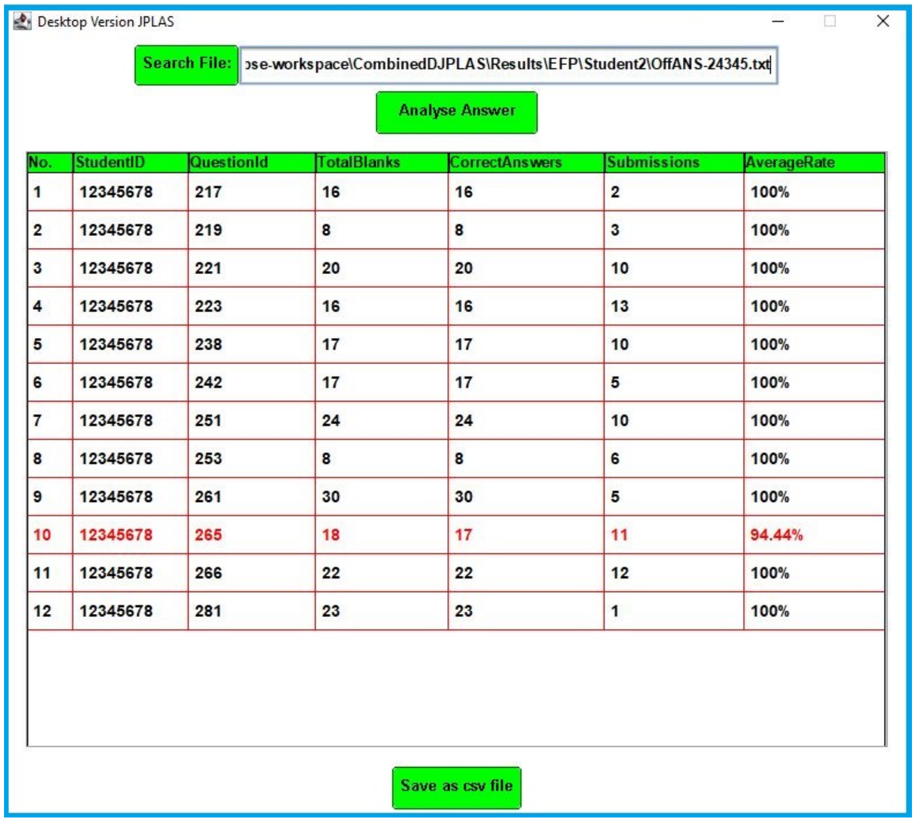
$$Correct\quad Rate = \frac{number\quad of\quad correct\quad answers}{number\quad of\quad blanks} \times 100 \tag{6.1}$$

- CRP, SFP, CWP:
  1. Count the number of tests in the test code and the number of successful tests of each student for each problem.
  2. Calculate the correct rate by the follwing equation Eq. (6.2)

$$Correct\quad Rate = \frac{number\quad of\quad success\quad tests}{number\quad of\quad tests} \times 100 \tag{6.2}$$

6.3.3 Student Result Interfaces

Figure 6.4 shows the answer result interface for a teacher, to helps finding a student who cannot solve a problem well. The corresponding raw is highlighted when the correct answer rate is smaller than the threshold that is given by the teacher. Each row of answer analysis interface also contains student id, question id, number of blanks, number of correct answers, and number of submissions. From the correct answer rate of each student, the teacher can know the programming skill and give proper feedbacks.



ɔse-workspace\CombinedDJPLAS\Results\EFP\Student2\OffANS-24345.txt

No.	StudentID	QuestionId	TotalBlanks	CorrectAnswers	Submissions	AverageRate
1	12345678	217	16	16	2	100%
2	12345678	219	8	8	3	100%
3	12345678	221	20	20	10	100%
4	12345678	223	16	16	13	100%
5	12345678	238	17	17	10	100%
6	12345678	242	17	17	5	100%
7	12345678	251	24	24	10	100%
8	12345678	253	8	8	6	100%
9	12345678	261	30	30	5	100%
10	12345678	265	18	17	11	94.44%
11	12345678	266	22	22	12	100%
12	12345678	281	23	23	1	100%

Figure 6.4: Answer result of each student for each problem.

Figure 6.5 shows the correct answer rate for EFP instances. It helps the teacher to evaluate the difficulty of the assigned problems. If there is a problem with many submissions, it can be too difficult for students.

Figure 6.6 shows the result summary of each student for each problem type. It shows the student-id, the problem type and the average correct answer rate of all the problems for each student. If the average correct rate of a student is lower than the given threshold by the teacher, he/she should give some feedbacks to improve the programming skill.

6.4 Summary

This chapter presented the implementation of the student answer analyzing function to the five problem types in *D-JPLAS*. The proposed function was evaluated by EFP solutions by the students from Japan, Myanmar, China, and Indonesia. Our future works include the use of *D-JPLAS* in Java programing courses.

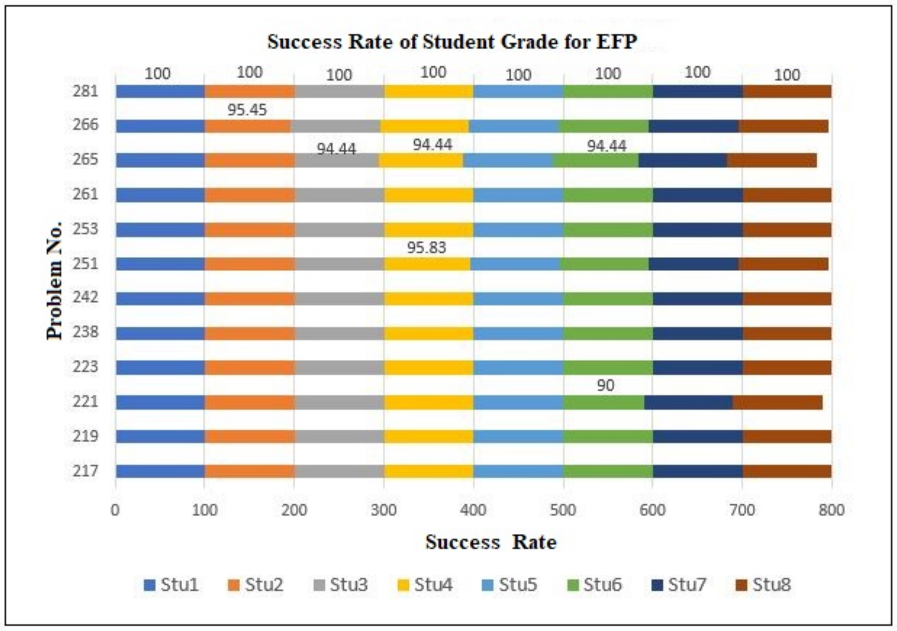
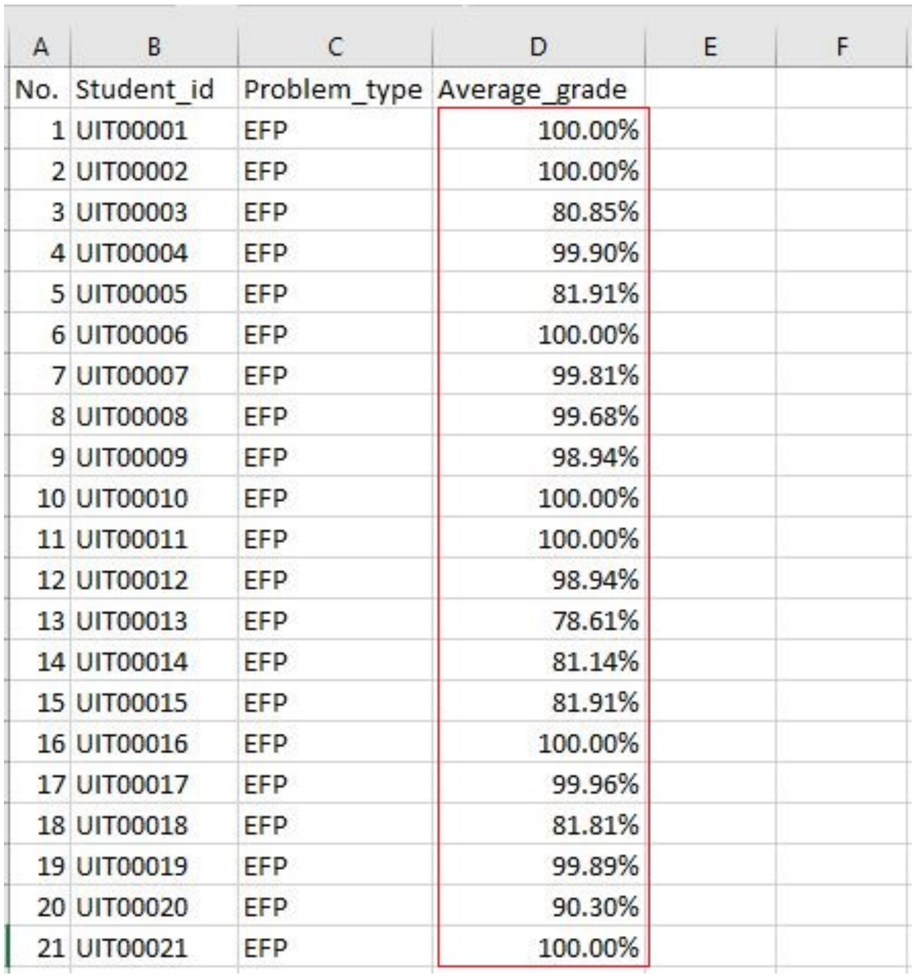


Figure 6.5: Success rate for each EFP.



A	B	C	D	E	F
No.	Student_id	Problem_type	Average_grade		
1	UIT00001	EFP	100.00%		
2	UIT00002	EFP	100.00%		
3	UIT00003	EFP	80.85%		
4	UIT00004	EFP	99.90%		
5	UIT00005	EFP	81.91%		
6	UIT00006	EFP	100.00%		
7	UIT00007	EFP	99.81%		
8	UIT00008	EFP	99.68%		
9	UIT00009	EFP	98.94%		
10	UIT00010	EFP	100.00%		
11	UIT00011	EFP	100.00%		
12	UIT00012	EFP	98.94%		
13	UIT00013	EFP	78.61%		
14	UIT00014	EFP	81.14%		
15	UIT00015	EFP	81.91%		
16	UIT00016	EFP	100.00%		
17	UIT00017	EFP	99.96%		
18	UIT00018	EFP	81.81%		
19	UIT00019	EFP	99.89%		
20	UIT00020	EFP	90.30%		
21	UIT00021	EFP	100.00%		

Figure 6.6: Student result summary.

Chapter 7

Related Works

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

In [36], Brunet et al. presented the concept of *design test* that automatically checks whether the code conforms to the specific *design rule* by using a test-like program. To support it, *Design-Wizard (DW)* had been developed with a fully-fledged API that allows writing design tests for Java codes using *JUnit*. The proposal was applied to three software products in their group and student projects in the undergraduate course. The results showed that this approach was suitable to check conformance between the design rules and the code implementation. Moreover, it has been observed that both designers and programmers appreciated the design tests as executable documents that can be easily kept up to date.

In [37], Akahane et al. presented a Web-based *automatic scoring system* for Java programming assignments to reduce loads of teachers in verifying a huge number of codes and in giving feedbacks to students. The system receives Java application programs submitted by students, and immediately returns the results of *JUnit* tests where the *Java Reflect API* is adopted for testing private classes and methods that have been commonly found in introductory courses. The regular expression is used to compare the output texts of each student program and those of the reference program. Through use in an actual course in their university, it was confirmed that this system was very helpful for students to improve programming skills by correcting mistakes in their programs and repeating their submissions.

In [38], Kitaya et al. presented a Web-based scoring system of programming assignments to students, which is similar to JPLAS. Their test consists of compiler check, *JUnit* test, and result test. The result test verifies the correctness of a student code composed of only the *main* method that reads/writes data from/to the standard input/output devices, by comparing the results of this code and of the reference code. However, the method has several disadvantages from our proposal: 1) it is only applicable to a code composed of the *main* method with the standard input/output, 2) it uses other programs to use the redirection for handling the standard input/output, and 3) it needs several input files to check the correctness for different input data. On the other hand, our method is applicable to a code containing other than the *main* method, it needs only *JUnit* with a test code, and all the input data can be described in a single test code.

In [39], Fu et al. presented a static exception-flow analysis that computes chains of semantically-related exception-flow links and reports entire exception propagation paths. These chains can be used, 1) to show the error handling architecture of a system, 2) to assess the vulnerability of a single component and the whole system, 3) to support the better testing of an error recovery code, and 4) to facilitate the tracing of the root cause of a logged problem.

In [40], Zhu et al. presented a system for mining API usage examples from the test code.

They found that the test code can be a good source for API usage examples that programmers need to know, like our approach. The test code can provide the information on small units of a code like functions, classes, procedures, and interfaces. The information in the test code is helpful in developing and maintaining a source code, including the knowledge sharing and transfer among programmers. However, the repetitive *API* use in a test code makes it complicated for programmers to read it. To address this issue, they studied the *JUnit* test code and summarized a set of test code patterns. They employed a code pattern based heuristic slicing approach to separate test scenarios in code examples. Then, they cluster similar API usages to remove redundancy and provide recommendations for API usage examples for programmers.

In [41], Kolassa et al. presented a system based on *JUnit* to test the partial code in a template of a template-based code generator where it is generated by the template engine. It facilitates the partial testing of a code by supporting the code execution in a mocked environment. They adopted *TUnit*, an extension of *JUnit* based on the *MontiCore* language workbench [42], [43], [44], to support the unit test of an incomplete code in the mocked environment. By using *TUnit*, a code generator template can be tested with mocked contexts such as mocked variables, mocked templates, and mocked help functions that are the inputs to the template. This testing intends to answer the questions: *Is the set of the specified inputs accepted by the code generator template, e.g., the code can be generated?*, *Does the code generator template produce syntactically valid source code?*, and *Are the target language context conditions valid for the generated source code?*

In [45], Rashkovits et al. showed that most of college students understand the concept of Java exception handling at the basic level, and the majority of them have difficulty in understanding advanced properties such as use of multiple exceptions, flow of control in the context of exceptions, handling exceptions further up the calling chain, catching and handling hierarchically related exceptions, and overriding methods that throw exceptions. They also provided a tutorial of exception handling, and quoted that exception handling is perceived as a relatively difficult task by novice programmers. In future works, we will consider to adopt their contributions.

In [46], Nakshatri et al. presented an empirical study of exception handling patterns in Java projects. It forces developers to think in sophisticated ways to handle the exceptions. In this study, empirical data was extracted from projects by analyzing data in *GitHub* and *SourceForge* repositories. The results were compared with recommendations for best practices in exception handling presented by Bloch [47]. It has been observed that most programmers ignore checking exceptions, and higher classes in the exception class hierarchy are more frequently used.

In [48], Xue et al. presented an integrity verification method for exception handling in service-oriented software. In this method, they construct state spaces associated with exception handling, convert the issue of integrity verification into a model of boundedness analysis based on CPN, and reduce the size of state spaces by extending Stubborn Set and Transition Dependency Graph. The experimental results confirmed that the method has good generalization abilities.

In [49], Júnior et al. presented a practical approach to preserve the *exception policy* in a system by automatically checking *exception handling design rules*. They are checked through executions of *JUnit* test cases with *dynamic mock objects* that are generated by the supporting tool. Four versions of *Mobile Media in SPL* were used to evaluate whether the policy was preserved or not. The results show that the approach can effectively detect violations on the policy of software product lines.

In [50], Fairbanks et al. use the term design fragment to refer to patterns that describe how a program interacts with frameworks. Design fragments describe what programmers should build to accomplish some goals and which are the relevant parts of the framework that the programmer's code will interact. The authors have created a catalog of design fragments and also mechanisms to implement (using XML) design fragments to check conformance between application and those design fragments.

In [51], Koile et al. presented a study for a*Tablet-PC-based system* called the classroom Learning Partner (CLP) developed to help in-class formative assessment in huge size of classes. CLP intended to increase instructor-student interaction by increasing student learning. It is evaluated using Tablet-PCs and a Tablet-PC-based classroom presentation system in an introductory computer science class. In this study, the classroom presenter supports student wireless submission of digital ink answers to in-class exercises. In their study, they evaluate the hypothesis that the use of such a system increases student learning by: (1) increasing student focus and attentiveness in class, (2) providing immediate feedback to both students and instructor about student misunderstandings, (3) enabling the instructor to adjust course material in real-time based upon student answers to in-class exercises, (4) increasing student satisfaction. This pilot study evaluates each of the above four parameters by means of classroom observation, surveys, and interviews. In our study, we consider the classroom out of internet access range to provide immediate feedback to students and to adjust course material as quick as possible by developing Desktop-version offline JPLAS.

In [52], Zhou et al. presented an Android application system using a tablet called *Isaly* to provide visual programming environments for educations. In this proposal, the concept of the state-transition diagram is used to make a program by a student. *Isaly* contains several features and user interfaces suitable for the use in a tablet.

In [53], Yamamoto et al. presented an improved group discussion system for the active learning system (ALS) using mobile devices to increase the examination pass rate. In their previous study, it was found that the proposed ALS could not increase the examination pass rate of the students although the self-learning time was increased. The experimental evaluation of the improved group discussion system showed that it can increase the examination pass rate. In future works, we will consider implementing the group discussion function with interfaces for mobile devices in JPLAS, so that students can continue studying Java programming with proper advises or hints from other students.

This paper presented an automatic technique for generating maintainable regression unit tests for programs. They found previous test generation techniques inadequate for two main reasons. First. they were designed for and evaluated upon libraries rather than applications. Second, they were designed to find bugs rather than to create maintainable regression test suites: the test suites that they generated were brittle and hard to understand. This paper presents a suite of techniques that address these problems by enhancing an existing unit test generation system. In experiments using an industrial system, the generated tests achieved good coverage and mutation kill score, were readable by the product's developers, and required few edits as the system under test evolved. While our evaluation is in the context of one test generator, we are aware of many research systems that suffer similar limitations, so our approach and observations are more generally relevant.

In [54], the author presented an automatic technique for generating maintainable regression unit tests for programs according to two main reasons compared to previous studies. First. they designed test generation techniques for and evaluated upon libraries rather than applications. Second, they designed them to find bugs rather than to create maintainable regression test suites: the test suites that they generated were brittle and hard to understand. Thus, they present a suite of techniques that address these problems by enhancing an existing unit test generation system. In experiments using an industrial system, the generated tests achieved good coverage and mutation kill score, were readable by the product's developers, and required few edits as the system under test evolved. While their evaluation is in the context of one test generator, they are aware of many research systems that suffer similar limitations, so our approach and observations are more generally relevant.

In [55], the author applied a programming education tool called pgtracer, which they had developed as a moodle plug-in, at an actual programming course to provide homework assignments to the students. They developed fill-in-the-blank questions based on the course syllabus at each week and evaluated the activities of students by using various functions provided by pgtracer. They also provided the analysis results to the teacher about the activities and achievement of the students for better collaboration between lecture and homework. They achieved the positive feedbacks by interviewing the teacher and surveying student's activities about the usefulness of pgtracer.

In our system, we consider the exception handling to reduce possible combinations in case of checking the standard input and output data with the students source codes. Moreover, we apply the informative test code approach on the three most popular Java Collection Framework, namely, List, Set and Map. Additionally, we implement the *Desktop-version offline JPLAS system(D-JPLAS)* to be helpful for the students who live in low speed internet communication regions. Student Analysis function is integrated to the *D-JPLAS*, therefore, the student answers could be evaluated without accessing to the internet.

Chapter 8

Conclusion

In this thesis, we studied the *informative test code approach* for the code writing problem in two important Java programming topics for novice students in *Java Programming Learning Assistant System (JPLAS)* and the implementation of the new teacher service function in *Desktop-version Java Programming Learning Assistant System (D-JPLAS)*.

Firstly, we presented the *informative test code generation* for *standard input/output with exception handling*, to support a teacher preparing complex test codes for the code writing problem. The proposed method generates an informative test code from the test code template and the model source code. For evaluations, we evaluated this approach by applying the proposal to 97 source codes in Java programming textbooks or Web sites that contain the standard input/output. The results showed that all the students completed the codes that can pass the test code, although the use of exception handling functions was sometimes insufficient or incorrect.

Secondly, we presented the informative test code approach to *Java Collections Framework (JCF)* that is a library of providing a strong and useful architecture for storing and manipulating a group of objects. To evaluate this approach, we generated five informative test codes for *JCF*, and asked 19 students from Japan, Myanmar, China, and Indonesia to solve the code writing problems using these test codes. The software metrics are calculated to confirm the quality of their codes. The results show that all of them successfully completed the source codes.

Thirdly, we presented the implementation of the new function in *Desktop-version Java Programming Learning Assistant System (D-JPLAS)*, to support a teacher summarizing the student answers submitted in text files and analyzing them. *D-JPLAS* provides both teacher service functions and student service functions to offer Java programming study environments using *JPLAS* even without the Internet. The analyzing results can be used to assess the performances of the students and return proper feedbacks to them.

In future works, we will continue studying informative test code approaches for other Java programming topics, implementing other teacher service functions such as detecting illegal copies in student answers for *D-JPLAS*, and applying them in Java programming courses.

Bibliography

[1] "Java," Internet: https://java.com/en/download/faq/whatis_java.xml, Access July 7, 2020.

[2] S. l. Ao et al. ed., IAENG Transactions on Engineering Sciences - Special Issue for the International Association of Engineers Conferences 2016 (Volume II), World Sci. Pub., pp. 517-530, 2018. http://www.worldscientific.com/worldscibooks/10.1142/10727

[3] S. S. Wint, N. Funabiki, and M. Kuribayashi, "Design and implementation of desktop-version Java programming learning assistant system," Proc. HISS, pp. 254-257, Nov. 2018.

[4] "JUnit," Internet: http://www.junit.org/, Access July 7, 2020.

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

[6] "Diary of kencoba," Internet: http://d.hatena.ne.jp/kencoba/20120831/1346398388, Access July 7, 2020.

[7] N. Ishihara, N. Funabiki, M. Kuribayashi, and W.-C. Kao, "A software architecture for Java programming learning assistant system," J. Comp. Soft. Eng., vol. 2, no. 1, Sept. 2017.

[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 J. CS, vol. 44, no. 2, pp. 247-260, May 2017.

[9] 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," Info. Eng. Exp., vol. 1, no. 3, pp. 9-18, Sept. 2015.

[10] N. Ishihara, N. Funabiki, and W.-C. Kao, "A proposal of statement fill-in-blank problem using program dependence graph in Java programming learning assistant system," Info. Engr. Exp., vol. 1, no. 3, pp. 19-28, Sept. 2015.

[11] N. Funabiki, Y. Matsushima, T. Nakanishi, K. Watanabe, and N. Amano, "A Java programming learning assistant system using test-driven development method," IAENG J. CS, vol.40, no.1, pp. 38-46, Feb. 2013.

[12] "BFS," Internet: http://www.geeksforgeeks.org/breadth-first-traversal-for-a-graph, Access July 7, 2020.

[13] "JUnit-Tools," Internet: http://www.junit-tools.org/index.php/getting-started, Access July 7, 2020.

[14] Y. Higo, A. Saitoh, G. Yamada, T. Miyake, S. Kusumoto, and K. Inoue, “A pluggable tool for measuring software metrics from source code,” Proc. IWSM-MENSURA, pp. 2-12, 2011.

[15] T. G. S. Filó and M. A. S. Bigonha, “A catalogue of thresholds for object-oriented software metrics,” Proc. SOFTENG, pp. 48-55, 2015.

[16] “Metric Plugin,” Internet: `http://metrics.sourceforge.net`, Access July 7, 2020.

[17] “Standard Streams,” Internet: `https://en.wikipedia.org/wiki/Standard_streams`, Access July 7, 2020.

[18] D. M. Ritchie, “A Stream Input-Output System”, AT&T Bell Lab. Tech. J., 68(8), Oct. 1984.

[19] “Exception Handling in Java,” Internet: `https://www.javatpoint.com/exception-handling-in-java`, Access July 7, 2020.

[20] M. Takahashi, Easy Java, 5th Ed., Soft Bank Creative, 2013.

[21] H. Yuuki, Java programming lessen, 3rd Ed., Soft Bank Creative, 2012.

[22] Y. D. Liang, Introduction to Java programming, 9th Ed., Pearson Education, 2014.

[23] “Java programming seminar,” Internet: `http://java.it-manual.com/start/about.html`, Access July 7, 2020.

[24] “Kita Soft Koubo,” Internet: `http://kitako.tokyo/lib/JavaExercise.aspx`, Access July 7, 2020. .

[25] M. J. Garbade, “Top 3 most popular programming languages in 2018,” `https://hackernoon.com/top-3-most-popular-programming-languages-in-2018-and-their-annual-salaries-51b4a7354e06`, Access July 7, 2020.

[26] S. Cass, “The 2015 Top Ten Programming Languages - IEEE Spectrum,” Internet: `https://www.linkedin.com/pulse/2015-top-ten-programming-languages-ieee-spectrum-farzin-pashaee`, Access July 7, 2020.

[27] S. Cass and P. Bulusu, “Interactive: The Top Programming Languages 2018,” Internet: `https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2018`, Access July 7, 2020.

[28] N. Funabiki, K. K. Zaw, N. Ishihara, and W.-C. Kao, “Java programming learning assistant system: JPLAS,” IAENG Trans. Eng. Sci., Spec. Issue. Int. Assoc. Eng. Conf. 2016 vol. II, pp. 517-530, 2016

[29] N. Funabiki, Y. Matsushim, T. Nakanishi, K. Watanabe, and N. Amano, “A Java programming learning assistant system using test-driven development method,” IAENG Int. J. Comput. Sci., vol. 40, no. 1, pp. 38-46, Feb. 2013.

[30] K. K. Zaw, N. Funabiki, E. E. Mon, and W.-C. Kao, “An informative test code approach for studying three object-oriented programming concepts by code writing problem in Java programming learning assistant system,” Proc. Global Conf. Consum. Elect. (GCCE2018), pp. 592-596, Oct. 2018.

[31] "JCF," Internet: `https://docs.oracle.com/javase/tutorial/collections/`, Access July 7, 2020.

[32] "Iterators in Java," Internet: `https://www.geeksforgeeks.org/iterators-in-java/`, Access July 7, 2020.

[33] "Metrics 1.3.6," Internet: `http://metrics.sourceforge.net`, Access July 7, 2020.

[34] K. K. Zaw and N. Funabiki, "A concept of value trace problem for Java code reading education," Proc. Int. Cong. Adv. Appl. Info., July 2015, pp. 253-258.

[35] 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," to appear in Inf. Eng. Exp., 2015.

[36] J. Brunet, D. Guerrero, J. Figueiredo, "Design Tests: An Approach to Programmatically Check your Code Against Design Rules," Proc. Int. Conf. on SW Eng. Comp. vol. pp. 225-258, January 2009.

[37] Y. Akahane, H. Kitaya, and U. Inoue, "Design and evaluation of automated Scoring Java programming assignments," Proc. Int. Conf. Soft. Eng., Art. Intel., Net. Para./Dist. Comput., pp.1-6, 2015.

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

[39] C. Fu and B. G. Ryder, "Exception-chain analysis: revealing exception handling architecture in Java server applications," Proc. Int. Conf. Soft. Eng., pp. 230-239, May 2007.

[40] Z. Zhu, Y. Zou, B. Xie, Y. Jin, Z. Lin, and L. Zhang, "Mining API usage examples from test code," Proc. IEEE Int. Conf. Soft. Mainte. Evo., pp. 301-310, 2014.

[41] C. Kolassa, M. Look, K. Müller, A. Roth, D. Rei, and B. Rumpe, "TUnit –unit testing for template-based code generators," Proc. Modellierung Conf., pp. 221-236, 2016.

[42] H. Grönniger, H. Krahn, B. Rumpe, M. Schindler, S. Völkel, "MontiCore: A framework for the development of textual domain specific languages," Proc. Int. Conf. Soft. Eng. (ICSE), 2008.

[43] H. Krahn, B. Rumpe, S. Völkel, "MontiCore: Modular development of textual domain specific languages," Proc. Int. Conf. Model. Tech. Tool. Comp. Perform. Evaluation, pp. 297-315, 2008.

[44] H. Krahn, B. Rumpe, S. Völkel, "MontiCore: "A framework for compositional development of domain specific languages," Int. J. SW Tool. Tech. Transfer, vol. 12, no. 5, pp. 353-372, Sept. 2010.

[45] R. Rashkovits and I. Lavy, "Students' understanding of advanced properties of Java exceptions," J. Info. Tech. Edu., vol. 11, pp. 327-352, 2012.

[46] S. Nakshatri, M. Hegde, and S. Thandra, "Analysis of exception handling patterns in Java projects: an empirical study," Proc. IEEE/ACM Work. Conf. Mining Soft. Rep., pp. 500-503, May 2016.

[47] J. Bloch, Effective Java, 2nd Ed., Prentice Hall PTR, Upper Saddle River, NJ, USA, 2008.

[48] T. Xue, S. Ying, Q. Wu, X. Jia, X. Hu, X. Zhai, and T. Zhang, “Verifying integrity of exception handling in service-oriented software,” Int. J. Grid. Utility Comput., vol. 8, pp. 17-21, 2017.

[49] R. J. S. Júnior and R. Coelho,“Preserving the exception handling design rules in software product line context: a practical approach,” Proc. Latin-American Symp. Depend. Comp. Work., pp. 9-16, 2011.

[50] G. Fairbanks, D. Garlan, and W. Scherlis, “Design fragments make using frameworks easier,” Proc. OOPSLA Conf. vol. 41, pp. 75–88, Oct. 2006.

[51] K. Koile and D. Singer “Development of a Tablet-PC-based System to Increase Instructor-Student Classroom Interactions and Student Learning,” Proc. WIPTE (Workshop on the Impact of Pen-based Tech. on Edu., Purdue University) Apr. 2006.

[52] E. Zhou, Z. Niibori, S. Okamoto, M. Kamada, and T. Yonekura, “IslayTouch: an educational visual programming environment for tablet devices”, J. Space-Based and Situated Comput., vol. 6, no. 3, pp. 183-197, 2016.

[53] N. Yamamoto, “An improved group discussion system for active learning using smart-phone and its experimental evaluation,” J. Space-Base. Situated Comput., vol. 6, no. 4, pp. 221-227, 2016.

[54] B. Robinson, M. D. Ernst, J. H. Perkins, V. Augustine and N. Li, ”Scaling up automated test generation: Automatically generating maintainable regression unit tests for programs,” 26th IEEE/ACM Int. Conf. on Auto. SW Eng. (ASE 2011), pp. 23-32, 2011.

[55] T. Kakeshita and M. Murata, ”Application of Programming Education Support Tool pg-tracer for Homework Assignment,” J. of Learn. Tech. and Learn. Envr, vol1. no.1, pp. 41-60, Mar.,2018.

[56] “Standard Streams,” Internet: `https://en.wikipedia.org/wiki/Standard_streams`, Access July 7, 2020.

[57] S. Heckman, “An Empirical Study of In-Class Laboratories on Student Learning of Linear Data Structures,” Proc. Int. Conf. Int Comp. Edu. Res. pp. 217-225, Aug. 2015.