So, H. J. et al. (Eds.) (2020). Proceedings of the 28th International Conference on Computers in Education.
Asia-Pacific Society for Computers in Education

Introducing a Mock Technique into a Learning Support System for Program Design Based on Testability

Masaya MURAMATSU[a*], Yasuhiro NOGUCHI[a], Satoru KOGURE[a], Koichi YAMASHITA[b], Tatsuhiro KONISHI [a], & Yukihiro ITOH[c]
[a] *Faculty of Informatics, Shizuoka University, Japan*
[b] *Faculty of Business Administration, Tokoha University, Japan*
[c] *Shizuoka University, Japan*
* muramatsu.masaya.16@shizuoka.ac.jp

Abstract: While software development requires automated testing skills, there are few opportunities for test education during a novice programming learning phase. Therefore, we constructed a learning support system that can conduct an exercise to analyze and improve program design from the viewpoint of *Testability* in the design of the component to be tested. However, our previous system did not support design improvement using mock techniques, and the method of improvement that learners could choose was limited. Therefore, in this paper, we extended the system to support learning for analyzing and improving the design of the test target component with mock techniques. We experimented to evaluate the learning effect of the system whether the learners increase their skills of analyzing and improving the design of test target components, even in the situation where applying object-oriented design principles are required for the learners.

Keywords: Programming education, learning support environment, unit test, testability, program design, mock

1. Introduction

In recent years, as the software has become larger and more complex, the importance of program verification has increased. While knowledge and practical skills for testing are required, there are few opportunities for testing education at early programming learning stages, such as the college study of computer science. Therefore, research have been conducted to support the learning and implementation of unit test automation, such as Arcuri, Fraser, and Galeotti (2014) and Proulx and Jossey (2009).

Noguchi, Ihara, Kogure, Yamashita, Konishi, and Itoh (2019) developed the Testability Learning System (TLS), which supports the learning of the designs of test target components, focusing on the index of Testability organized by Bach (1999). By using TLS, learners can acquire how to design testable components in two phases: analysis phase and improvement phase. In the first phase, the learners analyze the causes of difficulties to test the test target component. In the second phase, the learners improve the designs of the component based on the causes analyzed. In both phases, the learners complete the tasks with the guiding of predefined feedbacks from the system. However, TLS does not support mock techniques thereby limiting design improvement methods.

There are two problems with the lack of methods of improvement with mock techniques. First, learners must decide from limited candidates for the methods of improvement for the analyzed cause and it causes that learners narrow the understanding of the relationship between the cause and the appropriate component design that solves the cause. Second, the test target components to be practiced are limited because some components are difficult to be tested without mock techniques. For instance, a component that receives essential data through calling an external API is a lack of *Controllability*, when specific data are required to invoke the component's behavior to be tested. If learners cannot use mock techniques, they can choose only a method of improvement that moves the codes of calling the external API to the outside of the component and prepares some interfaces of the component to be set every essential data (e.g. setters, method parameters, etc.). The limitation of methods of improvement has two problems mentioned above. Learners who learned the method as only a choice to improve the lack of controllability cannot notice that the method makes the component depend on other components by overusing control coupling. Also, a component that uses different data based on the condition of the component cannot be considered as a sample component at the learning. In this case, the component cannot sustain the provided function because the method moves most code fragments to out of the component. Moreover, most components excluding simple codes only for programming exercises categorized as such components.

Redefinition, polymorphism, and dynamic binding are important features of object-oriented design for testable components by using mock techniques (hereinafter, "OOD features"), and many researchers have conducted on the difficulty of the learning and some educational methods (e.g., Or-Bach, & Lavy, 2004; Hadar, & Leron, 2008; Arif, 2000; Bergin, 2003; Schmolitzky, 2006; Benaya, & Andzur, 2008; Alkazemi, & Grami, 2012; Daniel, 2019). Therefore, it is important to provide some supports to understand and utilize the OOD features while they analyze and improve the design of the test target components.

In this paper, we extended previous TLS so that learners can analyze and improve the design of the test target component with mock techniques. The extended TLS prepares the learning flow for supporting such analysis and improvement, and the system can support the learners in completing the flow. We conducted experiments to evaluate the extended system and verify the learning effects of the extended portion from the experimental results.

2. Related Works

For the learning and implementation of unit test automation, many research focused on the testing components to reduce the design/implementation costs (e.g., Schroeder, & Rothe, 2005; Allowatt, & dwards, 2005; Teixeira, & Silva, 2018). As for the practice for early programmers with the design of the test target components, there are some cases that Test Driven Development (TDD) activities used in a programming exercise (e.g., Janzen, & Saiedian, 2008; Funabiki, Matsushima, Nakanishi, Watanabe, & Amano, 2013). In these practices, however, their students are often provided sample projects including full set of automated unit tests. It means that testing/tested component design is provided for students and they focused on implementing the behaviors of the tested components and adding extra test cases. There are some research to support TDD activities for early programmers, such as Johnson and Kou (2007). As mentioned above, these research do not focus on supporting learning of design and implementation for the test target component.

In this paper, we extended TLS (Noguchi, et al., 2019) for supporting learners to improve their design of components with mock techniques. We summarize the purpose and functions of the previous TLS and clarify our contributions in this paper. The previous TLS focused on three indicators of Testability—*Controllability*, *Observability*, and *Decomposability*—that are related to the interface design of the component to be tested. ***Figure 1*** shows the relationship between the three indicators and automated testing. *Controllability* is an indicator of whether the test driver can control the test target component to cause its behavior to be tested. *Observability* is an indicator of whether the test driver can observe the test target component's behavior to be tested. *Decomposability* is an indicator of whether the test target component consists of only the codes to be tested.

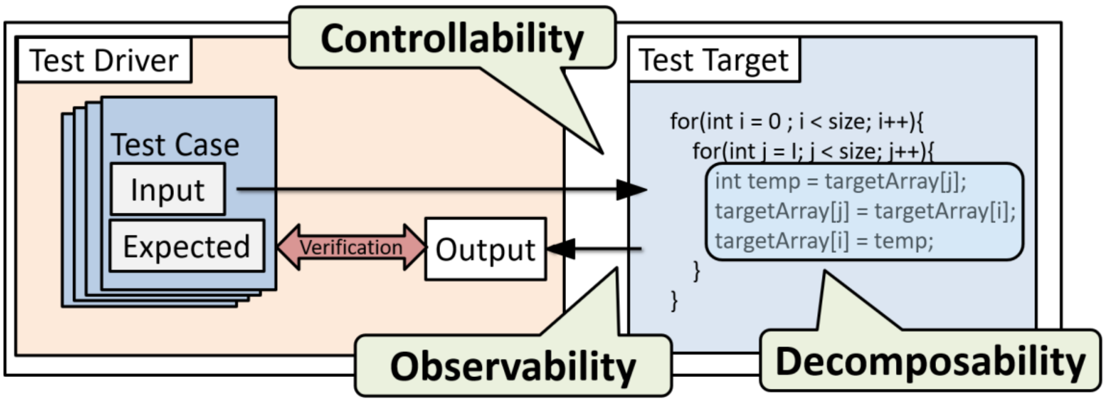


Figure 1. Testability Indicators

In the previous TLS, learners can choose an exercise whose sample program code includes a test target component with some difficulties to be tested, and the learners analyze the causes of the difficulties and improve the design of the test target component to resolve the difficulties to be tested. The learners can improve the design based on their analyzed causes of the difficulties by choosing the methods of improvement provided by the system. At each step of the analysis and design improvement, the system evaluates the learners' choices and provides feedback. When the learners completed the steps, their improved program code can be downloaded as an Eclipse project, and the learners can confirm whether they can write automated tests for the improved test target component in Eclipse by executing the automated tests. However, the previous TLS cannot provide methods of improvement with mock techniques, because the TLS focused on the learners at early programming learning stages who have not learned the OOD features that are essential for using the methods.

Regarding the difficulties in learning the OOD features, Liberman, Beeri, and Kolikant (2011) analyzed an object-oriented programming course where 22 in-service CS teachers whose programming mother tongue was procedural (Pascal, C, or Basic) learned object-oriented programming. Then, they categorized the difficulties in learning inheritance into four clusters: one, by not presenting a clear model, participants develop their model for a new situation from past knowledge, etc.; two, by explaining it with similar concepts that learners are familiar with, they stumble during implementation; three, by confusing inheritance of the is-a relation with the composition of the has-a relation; and four, by assuming that an object of a class exists when the definition of the class is written, even before executing construction code for the object. In our study, we provide learners an obvious design model that satisfies Testability to support automated testing. Moreover, both the correctness of its design and the implementations can be evaluated according to whether the automated testing can be applied or not. Thus, in our study, learners can avoid the first and the second difficulties because they are guided by the obvious design model and teachers are not required to explain design principles by using similar concepts that learners are familiar with. As for the third and fourth difficulties, learners are supported by providing feedback from the system about the design and implementation errors.

Nandigam, Tao, Gudivada, and Hamou-Lhadj (2010) proposed that unit testing using a mock object framework is an effective practical subject for teaching object-oriented design principles. They organized the learning of object-oriented design principles using the mock object framework into four elements of instruction: the design must have an associated with unit test class, one must first prepare interface specification for a unit, the designer of each unit must strive to use interfaces instead of concrete classes, and everything that an object needs are passed into the object either as a parameter to the constructor or using an appropriate setter method. Whereas they aimed at learning object-oriented design principles, our study aimed at learning design concepts and practical techniques for improving component design based on Testability. Also, object-oriented design principles are only one of the necessary elements as the design improvement method. In our study, the mock object framework is a selectable element as one of the methods of design improvement when the learners themselves evaluate the component design based on the criteria of Testability. If the design improvement by the mock object framework is effective, they should choose it; otherwise, they should choose a different design proposal. Moreover, since our proposed system covers the four main learning items Nandigam, et al. proposed, it may be able to be positioned as an educational support system of training developers in the object-oriented design principles the researchers proposed.

3. Learning Support Strategies for Design Improvement with Mock

3.1 Difficulties for Design Improvement with Mock

The OOD features enable developers to change the behavior of the invoked method without modifying the method name in code. It means that a developer can design a component that can be worked on a production purpose and a testing purpose without changing the codes of the component. As for the typical design for such components, ***Figure 2*** shows a common design example used in testing frameworks such as mockito. Suppose that it is designed so that the behavior of the method to be tested on the test target component ("TestTargetMethod" in ***Figure 2***) can be changed from the testing component ("Testing Component") without modifying the code of the test target component. As for exchanging the production purpose and the testing purpose of the test target component, the test target component can use the production component and the mock via the common interface ("FunctiontoReplace" in "Interface"). The method to be tested invokes these components through the common interface, not using the concrete name of these components ("Production Component" and "Mock"). Additionally, the method to be tested can accept a parameter of the common interface ("obj: Interface" in "Test Target Component") which enables the testing component to insert the concrete component for the purpose ("Production Component" or "Mock").

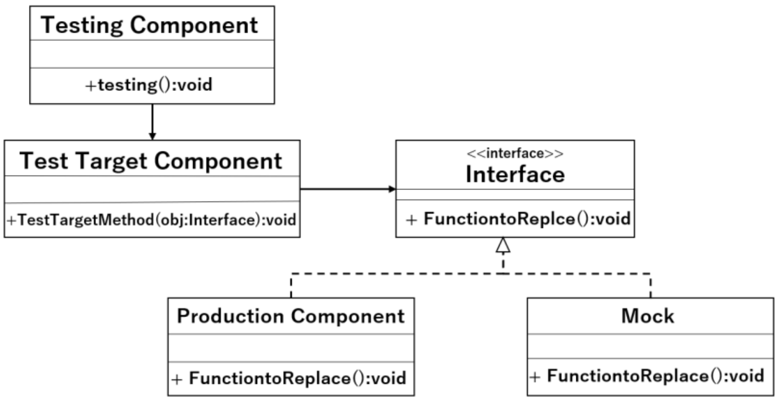

Testing Component
+testing():void
Test Target Component
+TestTargetMethod(obj:Interface):void
<<interface>>
Interface
+ FunctiontoReplce():void
Production Component
+ FunctiontoReplace():void
Mock
+ FunctiontoReplace():void

Figure 2. Design example with mock

Regarding the difficulties for novice programming learners, they must learn the OOD features as their new knowledge, design the components based on these OOD features to realize automated testing, and modify the codes to realize the design of components from the original component which has difficulties of *Controllability* and/or *Observability*. In contrast with the learners' situations of the previous TLS, there are major design changes from the original sample components, and collaborations of some part of design changes realize a part of testable features of the component. Thus, the learners sometimes lost steps in their improvement process; they cannot identify the design of components in their specific step; they cannot understand what part of design changes realizes a part of testable features of the component. As the result, it often happened that learners cannot notice their mistakes in the design changes themselves.

3.2 Design Improvement Process with Mock

We summarized the design improvement process with a mock with the following 6 steps:

Step 1. Choosing code fragments in the test target method should be replaced by a mock object that assigns parameters invoking the behavior to be tested or observes the results of the behavior to be tested.
Step 2. Defining the interface of the method invoked from the test target method that is commonly used in a production component and a mock.
Step 3. Moving the corresponding code fragments chosen at Step 1 into the production component, and change the invoking the production component via the interface defined in Step 2 from the test target component.
Step 4. Adding a parameter of the common interface into the test target method (or into the constructer of the test target component) to enable the testing component to insert a concrete object ("Production Component" or "Mock").
Step 5. Defining a mock object implemented based on the common interface, and, implementing the mock object based on the difficulties in the original test target component as it assigns parameters invoking the behavior to be tested or it observes the results of the behavior to be tested.
Step 6. Implementing the testing component that invokes the behavior of the test target method and verifies the responses of the method via inserted mock object.

At the end of Step 4, the design of components has been improved to enable the testing component to insert a concrete component. It means the improved code is possible to work as a production purpose or a testing purpose. At the end of Step 5, a mock object has been prepared that can assign parameters and observe the results. At the end of Step 6, the testing component has inserted a mock object suitable for the testing situations that enable the testing component to control/observe the behavior of the test target object via the mock object.

There are various conditions for the design of the test target component, it is not necessary to complete the steps from Step 1. Moreover, learners must evaluate the design of the test target component and decide which step is appropriate for the component. Also, sometimes the learners decide to skip the steps that are not necessary to the design conditions.

3.3 Extending the Learning Flow

Figure 3 shows our extended learning flow from the previous TLS that did not support the design with mock techniques. In this study, we extended the step (1), (4)-B1, and (4)-B2 for improving the design with mock techniques. At the step (1), we explained the essential concepts for the design analysis and improvements in lecture-style method: *Testability* concepts, OOD features, design example with mock, and other knowledge for testable component design with mock techniques. In this study, we employ the existing steps (2), (3), (4)-A, and (5) for analysis and improvement of the design without mock techniques. The extended step (4)-B1 contains Step 1 and the step (4)-B2 contains Step 2-6 discussed in section 3.2.

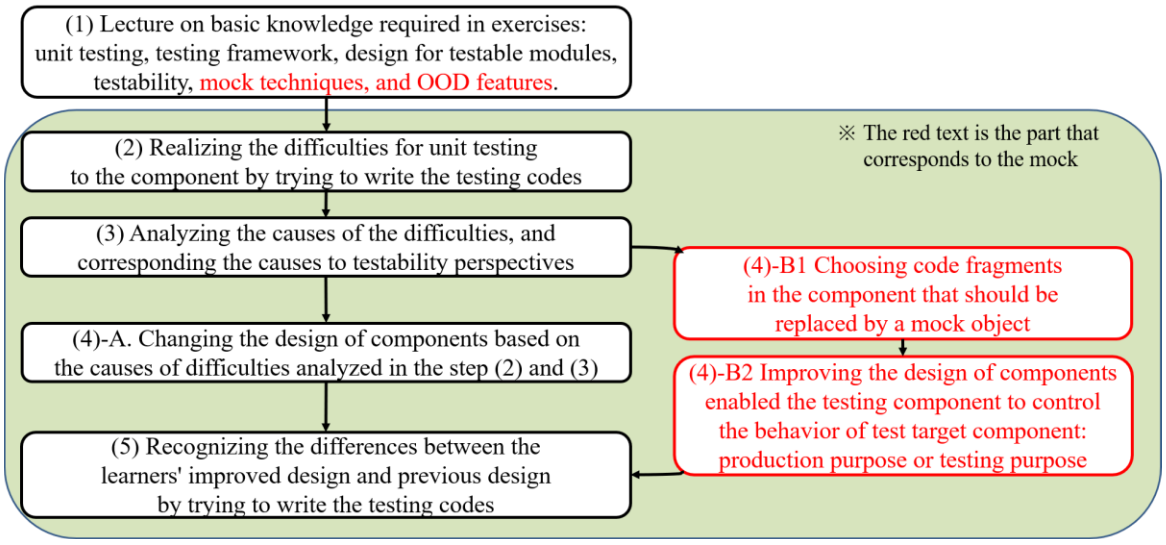

(1) Lecture on basic knowledge required in exercises: unit testing, testing framework, design for testable modules, testability, mock techniques, and OOD features.
※ The red text is the part that corresponds to the mock
(2) Realizing the difficulties for unit testing to the component by trying to write the testing codes
(3) Analyzing the causes of the difficulties, and corresponding the causes to testability perspectives
(4)-B1 Choosing code fragments in the component that should be replaced by a mock object
(4)-A. Changing the design of components based on the causes of difficulties analyzed in the step (2) and (3)
(4)-B2 Improving the design of components enabled the testing component to control the behavior of test target component: production purpose or testing purpose
(5) Recognizing the differences between the learners' improved design and previous design by trying to write the testing codes

Figure 3. Extended Learning Flow

3.4 Extended System's Supporting Functions

3.4.1 Design Comparison View

To let learners identify their design of the component at the current step and the differences between the expected design of components and one at the current step, we extended the previous TLS to provide learners a view showing the both for comparing in ***Figure 4***. This view shows the design of components in UML class diagram like format. In the view of their design at the current step, the mnemonics of the components follow the current real components the learners modified in their practice. Additionally, the system emphasizes the differences between the expected design and the design at the current step. Furthermore, the system also identifies where the learners should improve at the next step and the learners have successfully improved.

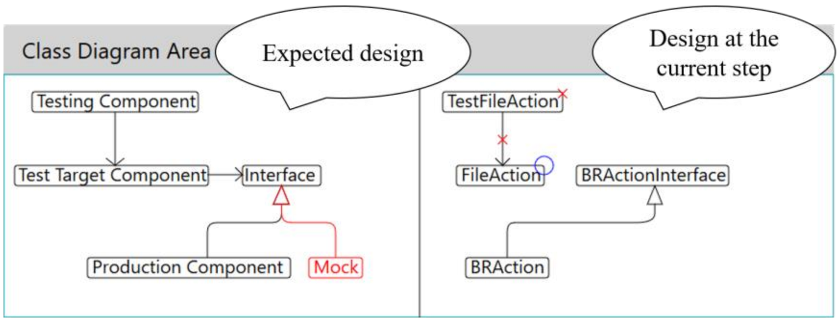
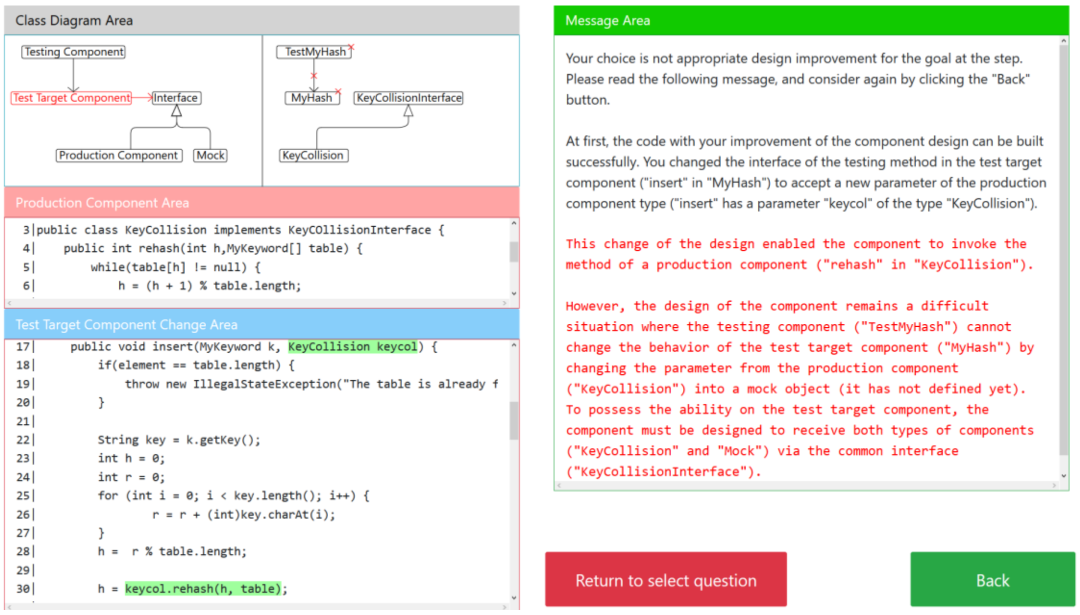


Figure 4. Comparison view between the expected design and the design at the current step

3.4.2 *Message Feedback Strategies in Extended Steps*

At each step, the system should provide feedbacks to learners who decide to choose a method of improvement. The feedbacks should encourage the learners by confirming their decision, and let the learners understand what features of the component are changed by their decided design change. In this study, the expected design is fixed and the system provides a limited number of candidates to the learners. Thus, the system's feedback messages for possible learners' choices can be predefined, and the code mnemonics corresponding to the expected design can be inserted into the feedback messages.

Regarding the steps extended in our study, at the step (4)-B1, the system provides candidates of code fragments in the test target method should be replaced by a mock object. If the learners choose incorrect code fragments (the learners can edit code fragments from the provided candidates in "Test Target Component Change Area" in ***Figure 5***), the system provides learners feedback messages that explain how the difference between the code fragments as a mock object and the expected design. At the step (4)-B2 corresponds to Steps 2-6 in section 3.2, the learners can choose a candidate for the appropriate method of design improvement while checking the current design of components and the difference from the expected design by using the comparison view as ***Figure 5***. If the learners choose the correct method of design improvement, the system provides learners feedback messages that what the features of the components are realized by the learners' chosen design change. As for the incorrect method of design improvement, the system provides feedback messages for the problems let by the decision: syntax errors, exceptions occurred, losing the function of the original component design, losing the features of the component, and remaining difficulties for testing. Like Figure 5, the feedback messages generally consist of the four parts of information: instruction of the next step for the learner, explanations for confirming the learners' decided design changes, explanations for the effects of the learners' design changes, and explanations for the remaining problems in the component design caused by the inappropriate design changes.



```
public class KeyCollision implements KeyCOllisionInterface {
    public int rehash(int h,MyKeyword[] table) {
        while(table[h] != null) {
            h = (h + 1) % table.length;
```

```
     public void insert(MyKeyword k, KeyCollision keycol) {
         if(element == table.length) {
             throw new IllegalStateException("The table is already f
         }

         String key = k.getKey();
         int h = 0;
         int r = 0;
         for (int i = 0; i < key.length(); i++) {
                 r = r + (int)key.charAt(i);
         }
         h =  r % table.length;

         h = keycol.rehash(h, table);
```

Your choice is not appropriate design improvement for the goal at the step. Please read the following message, and consider again by clicking the "Back" button.

At first, the code with your improvement of the component design can be built successfully. You changed the interface of the testing method in the test target component ("insert" in "MyHash") to accept a new parameter of the production component type ("insert" has a parameter "keycol" of the type "KeyCollision").

This change of the design enabled the component to invoke the method of a production component ("rehash" in "KeyCollision").

However, the design of the component remains a difficult situation where the testing component ("TestMyHash") cannot change the behavior of the test target component ("MyHash") by changing the parameter from the production component ("KeyCollision") into a mock object (it has not defined yet). To possess the ability on the test target component, the component must be designed to receive both types of components ("KeyCollision" and "Mock") via the common interface ("KeyCollisionInterface").

Figure 5. System Interface at Step 4 in 4-B(2)

4. Evaluation Experiment

4.1 Hypothesis

There are two hypotheses about the learning effects in exercises with sample components required design improvements by using mock techniques.

H1. Learners using our extended TLS increase their ability to analyze the causes of difficulties on the test target component rather than learners using IDE with a mock framework.
H2. Learners using our extended TLS increase their ability to improve the design of components rather than learners using IDE with a mock framework.

4.2 Evaluation Process

Figure 6 shows an overview of the evaluation process. At first, we collected 17 college students for the evaluation. 6 students are 3rd grade, 9 students are 4th grade, and 2 students are 1st grade in the graduate school. These subjects have basic knowledge of Java programming language confirmed 3 questions about them in the first 5 minutes in the pre-test. However, we eliminated 1 subject in the pre-test after our lecture for the subjects, because he acquired enough abilities only from the lecture.

In the pre-test, we also gave the subjects 2 problems improving the design of the component with mock techniques; one lacks *Controllability*, another lacks *Observability*. We categorized 8 subjects into the experimental group and 8 subjects into the control group based on the pre-test. The experimental group completes a new exercise improving the design of components with mock techniques with our extended TLS. The control group completes the same exercise with Eclipse IDE and the JUnit testing framework. While the experimental group received the tutorial for the TLS, the control group received the guide for analyzing/improving steps based on ***Figure 3***. After these exercises, we gave a post-test with the same design improvement problems in their pre-test to both groups.

Regarding the evaluation of the design improvement problems in the pre-test and the post-test, the subjects answered on the answer sheets in 30 minutes. The answer sheet provided the subjects the codes of the original components. On the answer sheet, they can choose code fragments by enclosing with a square, and modify the codes into their improved codes. We evaluated the subject's answers whether the subject can answer corresponding to the results of the analysis step (2) and (3) in ***Figure 3*** and whether the subject can complete their design improvement corresponding to the step (4)-B1 and (4)-B2 including 6 steps discussed in section 3.2. Finally, we converted the evaluation scores of each portion on 10-scale points. As the result, a subject was evaluated in total of 40 points in both the post-test and the post-test. Additionally, we measured the difference between the scores of the pre-test and the post-test.

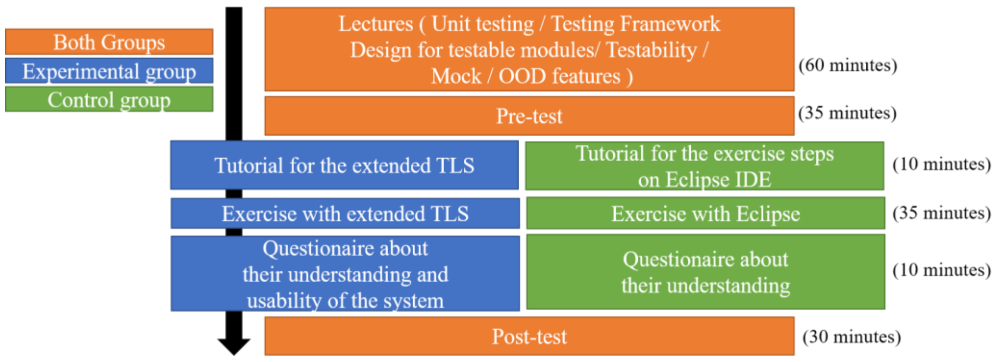

Both Groups
Experimental group
Control group
Lectures (Unit testing / Testing Framework Design for testable modules/ Testability / Mock / OOD features)
(60 minutes)
Pre-test
(35 minutes)
Tutorial for the extended TLS
Tutorial for the exercise steps on Eclipse IDE
(10 minutes)
Exercise with extended TLS
Exercise with Eclipse
(35 minutes)
Questionaire about their understanding and usability of the system
Questionaire about their understanding
(10 minutes)
Post-test
(30 minutes)

Figure 6. Overview of the Evaluation Process

4.3 Results

Table 1 shows the average scores in the pre-test and the post-test, the difference scores between both groups, and the difference scores between the pre-test and the post-test. The asterisks *, ** indicate that the coefficients are statistically different from zero at the 5 and 3 percent level based on Welch's two-sided t-test, respectively. The difference in the average scores in the pre-test was not statistically different between both groups. Also, the differences from the score of the pre-test and the post-test indicate the learning effects of the exercises in each group. Therefore, the difference in the learning effects in the experimental group and the control group is evaluated by comparing with the both difference scores.

For observing the learning effects for individual abilities, **Table 2**, **Table 3** show the differences in the average score of the analysis steps and the design improvement steps. Also, these tables show the scores separately for the type of problems: controllability and observability.

For observing which step the subjects completed or failed, ***Figure 7*** shows the number of subjects who scored 0.5 or more points at each step in the post-test. Each step evaluated from 0 to 1 at an interval 0.25. Besides, it shows the number of subjects who completed the whole steps with their score more than the threshold. The threshold worked for focusing the design improvement portion on whether the learner can proceed with the next step excluding the effects from negligible mistakes like misspelling, slight syntax problems.

Table 1.
Differences in the average scores between the pre-test and the post-test (out of 40 points)

	Pre-test	Post-test	Difference
Experimental	16.56	30.27	13.71
Control	15.62	19.82	4.20
Difference	0.94	10.45	**9.51****

Table 2.
Difference in the average score of the analysis steps (out of 10 points for each problem)

	Observability			Controllability		
	Pre-test	Post-test	Difference	Pre-test	Post-test	Difference
Experimental	8.13	9.38	1.25	5.63	10.00	4.38
Control	5.94	6.56	0.63	6.25	8.13	1.88
Difference	2.19	2.82	**0.63**	-0.62	1.87	**2.50**

Table 3.
Difference in the average score of the design improvement steps (out of 10 points for each problem)

	Observability			Controllability		
	Pre-test	Post-test	Difference	Pre-test	Post-test	Difference
Experimental	1.25	5.33	4.08	1.56	5.57	4.01
Control	1.44	1.46	0.02	1.99	3.67	1.68
Difference	-0.19	3.87	**4.06****	-0.43	1.90	**2.33**

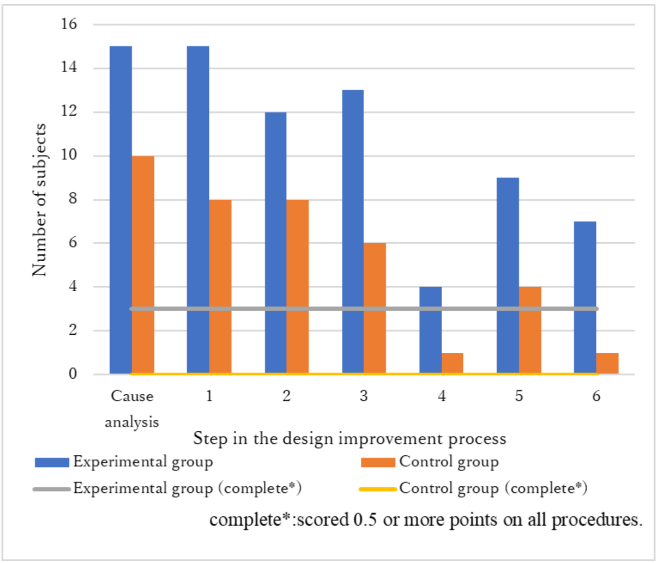


Figure 7. Number of Subjects who completed each step in the post-test

4.4 Discussion

As for the difference of the learning effect between the experimental group and the control group in **Table 1**, the score 9.51 that was statistically different from zero at the 3 percent level confirmed more learning effects of our extended TLS than the learning with only standard IDEs.

Regarding H1, **Table 2** shows that the scores of the post-test for the experimental group were 9.38 and 10.0 for the problem that lacks *Controllability* and the problem that lacks *Observability*. These scores indicated the subjects in the experimental group almost completely resolved the problems in the post-test. Even though the differences in the learning effects: 0.63 and 2.50 in **Table 2** did not have a significant difference by t-test, learning with our extended TLS has enough educational effects for increasing learners' ability to analyze the causes of difficulties of the test target component to be automated tested with mock techniques.

Regarding H2, **Table 3** shows that the scores of the post-test for the experimental group were 5.33 and 5.57 for the problem that lacks *Controllability* and the problem that lacks *Observability*. These scores indicated the subjects in the experimental group still made some mistakes in the post-test. As for the differences of the learning effects, the difference score 4.06 for the problem that lacks *Observability* had a significant difference by t-test. Even though the difference score 2.33 for the problem that lacks *Controllability* did not have a significant difference by t-test, both difference scores are not negative. Thus, H2 is partially supported by the results.

Besides, ***Figure 7*** shows that a greater number of subjects in the experimental group satisfied at each step in the post-test rather than the subjects in the control group. It indicated that learning with our extended TLS has learning effects at every step rather than the learning with only standard IDEs. Also, the learning with our extended TLS may not support all learners but is potentially able to let learners increase their abilities to complete all steps because 2 subjects in the experimental group completed all steps. Regarding OOD features and Testability, learners choose appropriate code fragments based on the *Decomposability* perspective at Step 1; At Step 2, 3 and 5, learners require knowledge of inheritance and polymorphism to identify the common interface for multiple components and modify the code for invoking these components via the interface; At Step 4, learners improve the design of the test target component for realizing the concept of dynamic binding. The relatively small number of the subjects in both groups satisfied Step 5. It indicated that our extended TLS can support learners in Step 1, 2, 3, and 5 related to the concept of decomposability and inheritance. However, it also indicated that most learners lack of understanding of dynamic binding and the method of design improvement, and our extended TLS is also far from enough to support learners for the point.

5. Conclusion

In this paper, we extended previous TLS (Testability Learning System) for supporting learners to improve the design of the test target component with mock techniques. We summarized the design improvement process into 6 steps and implemented supporting features in the extended TLS. For modifying the component into the expected design, it is required that the learners can utilize some principles of the object-oriented design: inheritance, polymorphism, and dynamic bindings. Also, to choose the appropriate method of improvement, they must analyze the component design by the Testability perspectives: *Controllability*, *Observability*, and *Decomposability*. Although it is difficult for novice programming learners to apply these knowledges into the concrete design of components, our preliminary evaluation indicated some learning effects of the system to increase the learners' practical skills of analyzing and improving the design of test target components. On the other hand, the evaluation indicated that most learners made mistakes the design improvement step related with dynamic binding even though the learners can receive some feedbacks by the system.

Acknowledgements

This work was supported by JSPS KAKENHI Grant Number JP18K11566.

References

Alkazemi, B.Y., & Grami, G.M. (2012). Utilizing BlueJ to teach polymorphism in an advanced object-oriented programming course. *Journal of Information Technology Education, 11*, pp. 271-282.

Allowatt, A., & Edwards, S.H. (2005). IDE Support for test-driven development and automated grading in both Java and C++. *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange (eclipse '05)*, pp. 100-104.

Arcuri, A., Fraser, G., & Galeotti, J.P. (2014). Automated unit test generation for classes with environment dependencies. *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering (ASE '14)*, 79-90.

Arif, E.M. (2000). A methodology for teaching object-oriented programming concepts in an advanced programming course. *ACM SIGCSE Bulletin*, *32*(2), pp. 30-34.

Bach, J. (1999). Heuristics of software testability, Retrieved from http://www.satisfice.com/tools/testable.pdf

Benaya, T., & Andzur, E. (2008). Understanding object-oriented programming concepts in an advanced programming course. *Proceedings of the 3rd International Conference on Informatics in SecondarySchools—Evolution and Perspectives: Informatics Education (ISSEP '08)*. Lecture Notes in ComputerScience, vol. 5090, pp. 161-170.

Bergin, J. (2003). Teaching polymorphism with elementary design patterns: *Companion of the 18th annual ACM SIGPLAN conference on object-oriented programming*, systems, languages, and applications (OOPSLA '03). *ACM*, 167-169.

Daniel, D. (2019). Teach all OOP principles in a single solution and expanding to solve similar problems. *Proceedings of the 10th Annual CITRENZ Conference*, pp. 60-65.

Funabiki, N., Matsushima, Y., Nakanishi, T., Watanabe, K., & Amano, N. (2013). A Java Programming Learning Assistant System using test-driven development method. *IAENG International Journal of Computer Science*, *40*(1), pp. 38-46.

Hadar, I., & Leron, U. (2008). How intuitive is object-oriented design?. *Communications of the ACM, 51*(5), pp. 41-46.

Janzen, D., & Saiedian, H. (2008). Test-driven learning in early programming courses. *Proceedings of the 39th SIGCSE technical symposium on Computer science education(SIGCSE '08)*, pp. 532-536.

Johnson, P.M., & Kou, H. (2007). Automated recognition of test-driven development with Zorro. *Proceedings of the AGILE (AGILE '07)*, pp. 15-25.

Liberman, N., Beeri, C., & Kolikant, Y.B.-D. (2011). Difficulties in learning inheritance and polymorphism. *ACM Transactions on Computing Education Transactions on Computing Education (TOCE), 11*(1), 4.

mockito, Retrieved from https://site.mockito.org

Nandigam, J., Tao, Y., Gudivada, V.N., & Hamou-Lhadj, A. (2010). Using mock object frameworks to teach object-oriented design principles. *The Journal of Computing Sciences in Colleges, 26*(1), pp. 40-48.

Noguchi, Y., Ihara, D., Kogure, S., Yamashita, K., Konishi, T., & Itoh, Y. (2019). Learning support system for software component design based on testability. *Proceedings of ICCE2019*, pp. 306-311.

Or-Bach, R., & Lavy, I. (2004). Cognitive activities of abstraction in object orientation: an empirical study. *ACM SIGCSE Bulletin*, *36*(2), pp. 82-86.

Proulx, V.K., & Jossey, W. (2009). Unit test support for Java via reflection and annotations. *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java (PPPJ '09). ACM*, 49-56.

Schmolitzky, A. (2006). Teaching inheritance concepts with Java. *Proceedings of the 4th international symposium on Principles and practice of programming in Java PPPJ '06*. *ACM*, pp. 203-207.

Schroeder, P.J., & Rothe, D. (2005). Teaching unit testing using test-driven development. *Workshop on Teaching Software Testing 2005*, Retrieved from http://www.testingeducation.org/conference/wtst4/pjs_wtst4.pdf

Teixeira, F.A., & Silva, G.B. (2018). EasyTest: an approach for automatic test cases generation from UML activity diagrams. *14th International Conference on Information Technology-New Generations*, pp. 411-417.