Chang, M. et al. (Eds.) (2019). Proceedings of the 27th International Conference on Computers in Education. Taiwan: Asia-Pacific Society for Computers in Education

Learning Support System for Software Component Design based on Testability

Yasuhiro NOGUCHI[a*], Daiki IHARA[b], Satoru KOGURE[a], Koichi YAMASHITA[b], Tatsuhiro KONISHI[a] & Yukihiro ITOH[c]
[a]*Faculty of Informatics, Shizuoka University, Japan*
[a]*Graduate School of Integrated Science and Technology, Shizuoka University, Japan*
[b]*Faculty of Business Administration, Tokoha University, Japan*
[c]*Shizuoka University, Japan*
*noguchi@inf.shizuoka.ac.jp

Abstract: In this paper, we describe a learning support system for software component design for learners at the early programming learning stage. In general, the source code used in instructional programming exercises does not have a testing code and is not designed to rapidly verify behaviors. We think one of the difficulties for students in designing their source code to be testable is that most cannot analyze their component design themselves. Therefore, we summarized the testability features applicable to such learners and designed our learning support system to improve instructional exercises in analyzing source codes and creating improved component design through feedback. The learning outcomes using the system were evaluated based on the subjects' improved scores between pre- and post-tests on analyzing source codes and creating improved component design. The evaluation results indicated the effectiveness of the program with some testability features. We implemented the system in three different programming classes to evaluate the applicability of the system.

Keywords: Programming education, learning support environment, software component design, unit test, testability

1. Introduction

In recent years, in addition to increasing the size and complexity of developed software, the importance of software testing has accordingly increased. Feathers (2004) indicated that a source code without a testing code is a legacy code. However, few studies on software testing and software component design for testing at the early programming learning stage have been conducted. In reality, source codes developed by learners in their exercises do not typically include testing codes for their components. However, there are two problems with learners not writing testing codes and/or not paying attention to the component design for testing. First, because they cannot edit the source code with rapid verification, they tend to take more time for debugging in their exercises. Second, while they continue their programming exercises without testing codes and/or testing designs, their programming style (debug later programming) will be fixed and their programming experience will not be sufficiently varied.

One of the software development processes that enforces programmers to write testing code is test-driven development (TDD). The process makes programmers follow the test-first approach, the programmer cannot write any component before preparing their test codes. Riley and Goucher (2009) reported that TDD can improve the testability of the software components that are being developed. However, the process requires the programmers to have enough knowledge and skills to write test codes with adequate component design for the testing. It is difficult for learners in the early programming learning stage to focus on learning software testing and component design for the testing in the process.

Funabiki et al. (2013) suggested a learning assistant system based on the TDD method. The learning assistant system lets learners follow the TDD process by supporting their testing code creation using a testing framework. Matsuoka (2013) suggested a support system to visualize the learners' testing progress and their testing coverage. However, these studies were focused on the support the learners need to write their testing code. They did not assume that the learners need to improve the design for the tested components before they write their testing codes.

Ohashi (2015) proposed a support system for automated testing by suggesting its weakness based on the following aspects related with tested components:

- Reproducibility: the component always returns to the same value for the same parameters.
- Simplicity: one method in a component should be simplified, as only one test can verify it.

However, the system focused on improving the tested components for stably maintaining their testing codes; the tested components were prepared their testing code templates. As for learners in the early programming learning stage, we think that the system does not sufficiently improve the learners' component design whether the component can be tested or not.

In this paper, we focus on a learning component design in which the learner can write testing code based on testability. For learners at the early programming learning stage, we discussed effective testability features for software component design and related learning strategies. We implemented a web-based learning support system based on the discussion. Our evaluation supported some learning effects of the proposed system for some testability features.

2. Learning Design for Testability-Based Software Component Design

2.1 Testability Related to Early Programming Learning Stage

Boehm (1976) introduced the concept of testability. Treon et al. (1999) suggested self-testable components. Bach (1999) categorized intrinsic testability related to the software component itself according the following features: observability, controllability, algorithmic simplicity, unbugginess, smallness, decomposability, and similarity. Binder (1999) claimed that controllability and observability are important features according to the object-oriented design perspective. We considered that decomposability is also an important feature for learners to understand at the early programming learning stage. A component whose decomposability is high indicates that the component encloses multiple functions. Thus, before testing individual function, the learners should divorce these individual functions from the component. Based on the above discussion, our learning support system focused on observability, controllability, and decomposability, as shown in Figure 24.

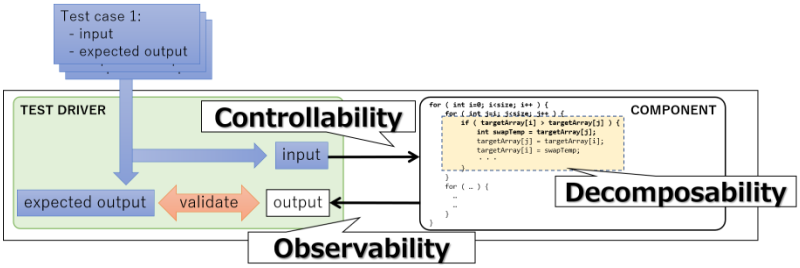

Figure 24. Controllability, observability, and decomposability in the testing model.

2.2 Learning Process for the Testability-Based Software Component Design

We assumed that the learners had already learned basic test driver implementation using a testing framework. We organized the learning items for component design into the following two points.

(I) Understanding the Concept of Testability

(II) Analysis and Improvement for Concrete Code Based on Testability

As for (I), it is necessary for learners to understand the concept of testability. Especially for controllability, observability, and decomposability, the learners should learn what kind of component design reduces testability. We expected that learners would acquire their knowledge from the lecture.

As for (II), the learners should be able to analyze the testability of concrete components based on the concept of the testability. In looking for the reason why we can write test code for the components, they should acquire the skills to analyze the components and identify their lack of testability. Furthermore, to improve the testable components, they should acquire the skills to create their component design by fixing the lack of testability and modifying the code based on the plan. We think

that exercises where learners analyze/modify various patterns of concrete components are required for learners to develop the needed skills. In such exercises, we know that many learners cannot evaluate their analysis and/or their component design themselves, and they do not have confidence in their answers even when they can complete the modification of the components following their design. Thus, to support their analysis, we should provide the learners with feedback on whether their analysis and component design are adequate based on the testability features.

Based on the above, we developed a learning process for the testability-based software component design, which is displayed in Figure 25. Our proposed system supports steps (2) – (5).

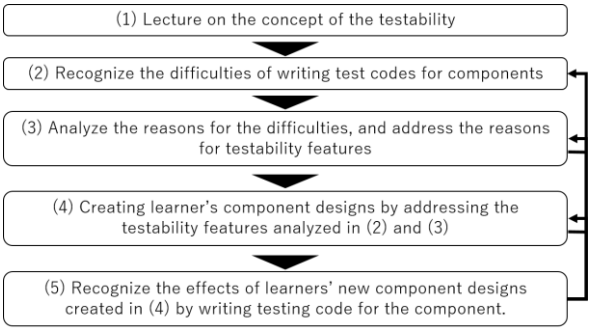

Figure 25. Learning process for the testability-based software component design.

3. Development of the Learning Support System

We implemented our learning support system as a web application. In this section, we explain the learners' usage of the system and the system's feedback (A) – (D) for their inputs in each step.

3.1 System Usage in steps (2) and (3)

Figure 26 shows the interface of the system in step (2). The source code area shows sample codes of the exercise chosen by the learner. The testing code area shows the testing code template for the function the learner should verify. The instruction area shows the system's message to inform the learner how to proceed in the next step. The bottom of the instruction area shows the learner's input field.

First, the exercise explains the learner about the function they should verify via the instruction area. The learner should choose the adequate code area for the function by selecting a rectangle. The system provides feedback (A) on whether the code area chosen by the learner is adequate or inadequate for the function set by the exercise.

In the testing code area, the learner should try to write their testing code for their chosen codes. They should also recognize the difficulties of writing their testing code in the current component design. Finally, they should externalize the reason why the component is difficult to test into the text input field on the bottom of the instruction area.

After submitting, the system gives feedback (B) that highlights words related to the testability features in the explanation. These words the system should highlight are registered by lack testability features of each prepared problem. Then, the system asks the learner to choose an appropriate reason by comparing their own externalized reason. The system suggests that the learner selects the reason related to each testability feature as follows. When the learners choose an answer, the system gives feedback (C) on whether the answer is correct or incorrect in the sample code with the explanation.

- In the testing code, I cannot change the input for testing.
- There is a useless input parameter in the component.
- The component is too restricted to be called from the testing code.
- In the testing code, I cannot observe the output from the component.
- Any components that are mismatched to the code area I should write testing code for.

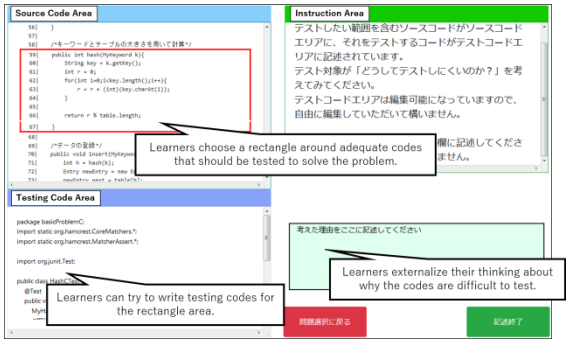

Figure 26. Externalizing their analysis in step (2).

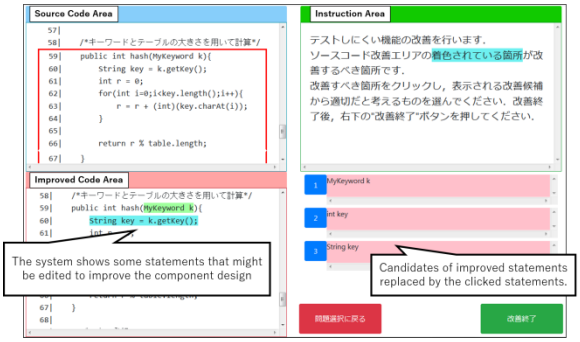

Figure 27. Creating their design in step (4).

3.2 System Usage in step (4)

Figure 27 shows the interface of the proposed system in step (4). The source code area shows the original sample codes. The improved code area shows the sample codes with some highlighted statements. When the learner clicks the highlighted statement, the system suggests some candidates at the bottom of the instruction area. When the learner selects a candidate, the testing code area is updated by the learner's selection. By changing the combinations of statements, the learner changes the sample code into a new component design the learner can write testing code for. When the learner submits the improved code, the system provides feedback (D) based on the learner's improved code. If the learner's improved code has syntax problems, the system sends the learner syntax error messages generated by an appropriate compiler of the programming language. If there are no syntax problems in the learner's improved code, the system decides whether the improved code enables the learner to write their testing code for the function indicated by the exercise. The system provides feedback messages on why their improved code has not fixed the problem the learner analyzed in steps (2) and (3). We defined these messages for every possible combination of candidates when we created the exercise.

3.3 System Usage in step (5)

The system provides the learners with their improved source codes as a project archive for Eclipse IDE. The learners can evaluate their new component design by executing their testing codes in the project.

4. Evaluation

4.1 Hypotheses

We compiled the following three hypotheses on learning the testability-based software module design with the proposed system.

Hypothesis 1. Learners who completed the exercises with the system can better analyze the lack of testability in the source code than those with only IDE support.
Hypothesis 2. Learners who completed exercises with the system can create their component design to improve testability better than those with only IDE support.
Hypothesis 3. The learners who finished exercises with the system will not rely on the system when they apply their acquired skills to other problems.

4.2 Evaluation Process

Figure 28 shows the evaluation process. 15 subjects were college students with basic syntax knowledge in the Java programming language. The pre-test confirmed their basic knowledge of algorithms, basic syntax of the Java programming language, and that they did not have skills for analyzing/improving

codes that lack testability. We divided the subjects into the experimental group with 8 subjects and the control group with 7 subjects to equalize the results of the pre-test.

As for the exercises, we gave both groups same four problems analyzing/improving codes that lack testability. In the post-test, there were three problems where the subjects analyzed the sample codes and improved them. The subjects were asked about (a) the area of source code supporting the function to be tested, (b) why the function is difficult to test, (c) where the area of code to be edited is, and (d) how to improve the code to solve the problem. As for question (c), we evaluated the answer using a point-deduction scoring system.

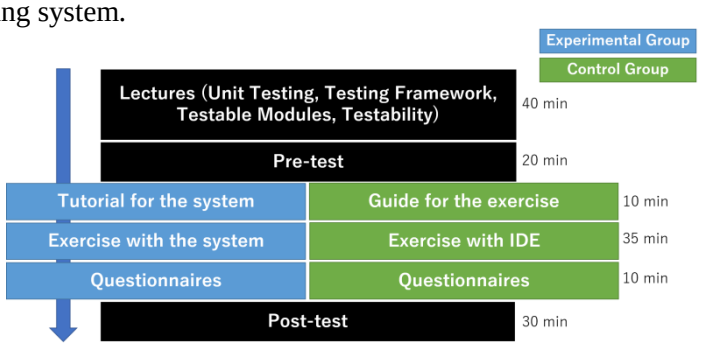

Figure 28. Evaluation process.

4.3 Results

Table 9 shows the average score for answers (a) and (b) in the post-test. The difference was calculated by subtracting the score of the control group from the score of the experimental group. The t-value was calculated using Welch's one-sided t-test. Table 10 shows the average scores for answers (c) and (d) in the post-test.

Table 9
Average Scores for Analyzing Lack of Testability in the Post-Test

Lack of Testability	Controllability		Observability		Decomposability	
Question	(a)	(b)	(a)	(b)	(a)	(b)
Avg. of Experimental Group	0.750	0.625	1.000	0.813	0.250	0.250
Avg. of Control Group	0.571	0.429	1.000	0.071	0.357	0.286
Difference	0.179	0.196	0.000	0.741	-0.107	-0.036
T-value	0.252	0.242	N/A	0.000	0.320	0.443

Table 10
Average Scores for Creating Improved Component Design in the Post-Test

Lack of Testability	Controllability		Observability		Decomposability	
Question	(c)	(d)	(c)	(d)	(c)	(d)
Avg. of Experimental Group	-0.167	0.833	-0.125	0.375	-2.667	1.333
Avg. of Control Group	-1.000	0.500	-2.000	0.000	-3.000	0.667
Difference	0.833	0.333	1.875	0.375	0.333	0.667
T-value	0.024	0.107	0.000	0.010	0.607	0.464

As for Hypothesis 1, Table 9 shows the experimental group's scores of controllability and observability are the same or better than the control group's scores. In Figure 29, the subjects' feelings of understanding in the exercise of the experimental group are better than those of the control group. Although the t-value only supports the score of question (b) for observability, the system does not seem to have bad effects based on the subjects' exercise and questionnaire results. Regarding decomposability, the average scores of the control group were better than those of the experimental group. There were many subjects in the experimental group who felt less understanding compared with the feelings of controllability and observability. In both groups, the subjects who could answer correctly were divided. In the observations of the subjects' post-test results, we found that some subjects in both groups were unable to answer the question. The system may have some educational effect for learners

to analyze the lack of controllability and observability in source code. However, the system is not well designed for decomposability.

As for Hypothesis 2, Table 10 shows that all scores of the experimental group were better than those of the control group. The t-value supports the scores in question (c) on controllability and observability, and the score of question (d) on observability. We think that the system has some educational effect for learners' component design against the lack of controllability or observability.

As for Hypothesis 3, even if the subjects of the experimental group answered the post-test without the system's support, more subjects answered correctly compared with the control group. Therefore, we think that the learning effect with the system is negligible.

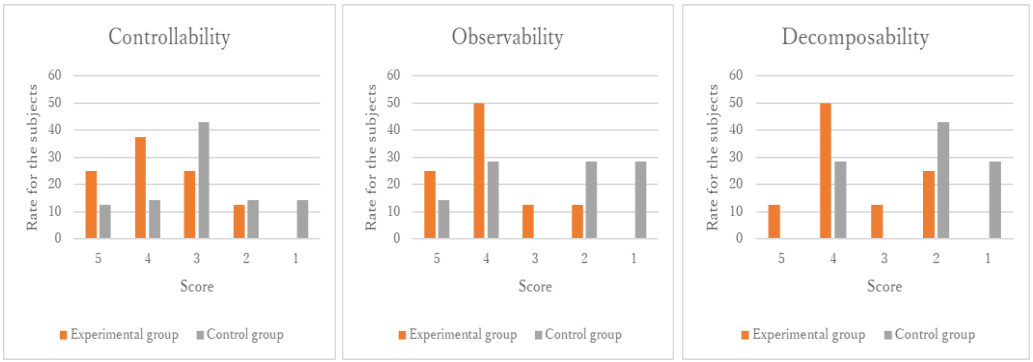

Figure 29. Subjects' feelings of understanding in their exercises.

5. Conclusion

In this study, we built a learning support system for software component design based on testability. The system can support learners' analysis of the lack of testability in the source codes and create their improvement plan for the component design against the lack of testability. The system covered three types of testability: controllability, observability, and decomposability. Our evaluation indicated some learning effects of the proposed system in relation to controllability and observability. We succeeded in introducing the system to three programming classes to evaluate the applicability of the system. After the exercises with the system, the participants tried to analyze larger sample source codes without the system, most participants analyzed the sample codes based on the testability features and suggested their refactoring plans, although all prepared functions with the lack of testability were not analyzed due to the time limitations of the classes.

Acknowledgements

This work was supported by JSPS KAKENHI Grant Number JP18K11566.

References

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

Binder, R. V. (1994). Design for testability with object-oriented systems. *Communications ACM, 37*, 87–101.

Boehm, B. W. (1976). Quantitative evaluation of software quality. *Proceedings of the 2nd of ICSE* (pp. 592–605).

Feathers, M. C. (2004). *Working effectively with legacy code.* Upper Saddle River, NJ: Prentice Hall.

Funabiki, N., Matsushima, Y., Nakanishi, K., Amano, N. (2013). A Java programming learning assistant system using test-driven development method. *IAENG International Journal of Computer Science, 40*(1).

Le Treon, Y., Deveaux, D., & Jézéquel, J. M. (1999). Self-testable components: From pragmatic tests to design-for-testability methodology. In editor name(s) (Ed(s).), *Proceedings of Technology of Object-Oriented Languages and Systems* (pp. 96–107). Nancy, France.

Matsuoka, S., & Katayama, T. (2013). Development of unit testing visualization tool to implement real-time visualization of testing progress. *IEICE Technical Report, 112*(373), 37–42. (in Japanese)

Ohashi, A., Kaminaga, H., Nakamura, S., Morimoto, Y., & Miyadera, Y. (2015). A support system of automated tests in software development exercises. *IEICE Technical Report, 114*(513), 13–18. (in Japanese)

Riley, T., & Goucher, A. (2009). *Beautiful testing: Leading professionals reveal how they improve software.* Sebastopol, CA: O'Reilly Media.