Jiang, B. et al. (Eds.) (2025). Proceedings of the 33rd International Conference on Computers in Education. Asia-Pacific Society for Computers in Education

Proposal of Visualization for Test Execution Results of Programming with Automated Testing

Raiya YAMAMOTO[a*], Keita MIYAGI[a], Yasuhiro NOGUCHI[a], Satoru KOGURE[a], Koichi YAMASHITA[b] & Tatsuhiro KONISHI[a]
[a]*Faculty of Informatics, Shizuoka University, Japan*
[b]*Faculty of Business Administration, Tokoha University, Japan*
*ryamamoto@inf.shizuoka.ac.jp

Abstract: In recent years, automated testing has been used in many software developments. Automated testing has advantages such as the early detection of errors through continuous test execution. However, research has reported that software testing is difficult for novices who have not yet acquired basic programming skills. Previous research has observed that even novice learners can perform coding activities and identify bugs in a function in an environment where automated testing is already prepared, but they could not identify the line that has defects in a function and could not eliminate the defect. This study proposes a learning support tool to help novice learners learn how to identify defects in a function using automated testing in programming exercises. Result of evaluation indicated that the proposed tools can help learn a method for identifying the location of defects.

Keywords: Programming education, learning support tool, automated testing

1. Introduction

In recent years, automated testing has become increasingly common in software development because of its ability to detect errors early through continuous execution and function-specific test designs (PractiTest, 2022). This approach helps narrow the scope of the bugs based on failed test cases. However, Edwards (2004) stated that software testing remains difficult for novice learners who have not yet acquired basic programming skills.

Yamamoto et al. (2024) explored the use of an automated testing environment for novices and found that, while they identified buggy functions, they struggled to locate the exact lines causing defects and were unable to fix them. For teaching debugging techniques, Yamamoto et al. (2018) proposed a systematic debugging process based on principles from Why Programs Fail? (Zeller, 2009) and the art of software testing (Myers et al., 2012), including scientific debugging, reproduction problems, deduction, and backtracking. This process involves executing functions with parameters that trigger defects, tracing execution paths, identifying faulty behaviors, and modifying the code while observing the effects. It is suggested that this method of tracing the execution path and identifying the scope of defects was effective even for novice learners. Based on the above, this study proposes a learning support tool to help novice programming learners learn how to identify defects in a function using automated testing in programming exercises.

2. Learning Support Tool with Visualization for Results of Automated Testing

For learners to acquire the ability to locate defects in an automated testing environment, they need to experience a series of processes of selecting a single test case, reviewing its execution path, and limiting the extent of the defect by overlapping the execution path reviewed in multiple tests. However, it is expected that novice learners may have difficulty

performing the following tasks: “confirming whether the assumed execution path is correct or not (Task 1),” “checking the state of superimposing the execution paths (Task 2),” “observing how the candidate ranges changed by superimposing a single test execution path (Task 3),” and “interpreting and making good use of the test result for debugging after a sufficient number of test execution paths have been overlaid (Task 4).” To support novice learners in executing the above tasks, we designed a learning support tool with the following functions: For Task 1: The execution path selected by the learner for each test is shown (Function 1). Supporting Task 2: The overlaid execution paths of the tests selected by the learner are displayed (Function 2). Assisting Task 3: The superimposed execution paths of the previous task were displayed (Function 3). Encouraging Task 4, hints were displayed according to the overlapping status of the execution paths (Function 4).

Figure 1 presents an overview of the developed learning support tool. The learning support tool was developed as an extension of VSCode, providing support for Java.

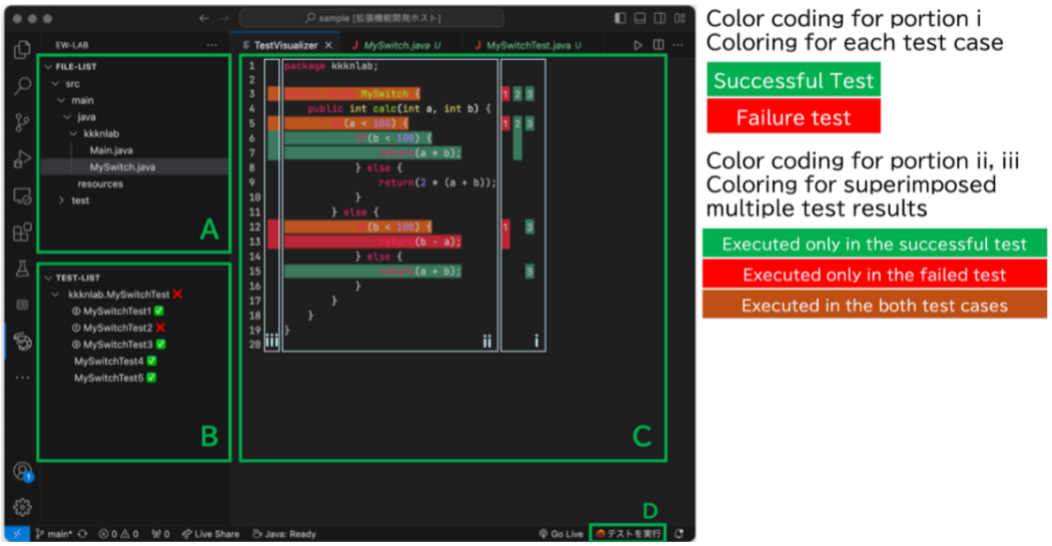

Figure 1. Overview of the Learning Support Tool

The support tool comprises three areas. The file selection area (A in Figure 1) functions as a file browser. The test selection area (B in Figure 1) was used to select the test case. Information regarding success or failure was displayed behind the name of each test. The green check mark indicates that the test was successful, and the red cross mark indicates that the test failed. Circled numbers indicate the order of test execution. The visualization area (C in Figure 1) shows the results of the executed tests and consists of three portions.

In portion i, independent bars describing the execution path that learners select are displayed. The numbers in each bar correspond to the numbers of test selection orders displayed in the test selection area. This figure shows the results of each test (achieving Function 1). The background colors of each line in the source code in portion ii were changed according to the results of the executed tests (achieving Function 2). Portion iii shows the previous color coding of the source code in the previous selection state (achieving Function 3). In addition to these visualizations, hints for interpreting the current state are displayed (achieving Function 4). D in Figure 1 represents the button for test execution.

3. Evaluation and Conclusion

Experiments are conducted to evaluate the effectiveness of the proposed system developed in this study. Nine undergraduate informatics students with basic Java knowledge, but limited experience in automated testing, were recruited for this study. They were divided into two groups based on their pretest scores: one group used the support tool (five students), while the other did not (four students). The evaluation included a pre-test, exercise, post-test, and questionnaire. The pre-test assessed the understanding of four features: selecting effective

test cases, identifying defect locations in the source code, understanding single test execution paths, and interpreting multiple test execution paths. The participants were provided with specifications for the source code containing defects, test codes, and test results. The post-test followed the same format as the pre-test, but used different content. As in the pre-test, the same materials were provided in the exercise, and the participants were asked to debug the code to aim for a state where all tests succeeded. Before the exercise, the support tool group received a tutorial on the use of the tool, while the control group received a tutorial for generic test runner.

Table 1 shows the increase in scores from pre-test to post-test for both groups. In parentheses, the pretest score is written on the left side of the slash, and the post-test score is written on the right side. The scores on both tests were rounded to the second decimal place.

Table 1. *Average score increase pre-test vs. post-test*

Feature Assessed	Allocation of Marks	Group with the support tool	Group without the support tool	Point Difference
Selecting effective test cases	40	4.00 (33.2 / 37.2)	0.75 (35.0 / 35.8)	3.25
Identifying defect locations in source code	40	8.80 (29.6 / 38.4)	-1.50 (33.0 / 31.5)	10.3
Understanding single test execution	50	7.40 (39.6 / 47.0)	-0.75 (49.0 / 48.3)	8.15
Interpreting multiple test execution paths	10	1.60 (7.40 / 9.00)	0.50 (8.75 / 9.25)	1.10

The results showed that the group with the support tool had higher scores than the group without the tool. This suggests that the group that used the tool had a better understanding of the method of locating defects using automated testing. However, the group that did not use the tool tended to have higher pretest scores. Therefore, there is room for further discussion regarding the results for the group without a learning support tool. In addition, this support tool was designed specifically for debugging; therefore, it was not designed to allow learners to design and describe their tests. In the future, it will be necessary to support an environment in which learners can design and describe tests to support debugging in a more practical environment.

Acknowledgements

This work was supported by JSPS KAKENHI, Grant Numbers JP22K12311 and JP23K17014.

References

Edwards, S. H. (2004). Using software testing to move students from trial-and-error to reflection-in-action. *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*. https://doi.org/10.1145/971300.971312

Myers, G. J., Badgett, T., & Sandler, C. (2012). *The art of software testing* (3rd ed.). John Wiley & Sons.

PractiTest. (2022). *The state of testing report 2022*. https://www.practitest.com/assets/pdf/state-of-testing-report-2022v10.pdf

Yamamoto, R., Noguchi, Y., Kogure, S., Yamashita, K., Konishi, T., & Itoh, Y. (2018). Implementation of a lecture package and a learning support system to teach systematic debugging to programming learners who do hit-or-miss debugging, *Transaction of Japanese Society for Information and Systems in Education*, *35*(1), 21-37. (in Japanese)

Yamamoto, R., Iwashimizu, R., Noguchi, Y., Kogure, S., Yamashita, K., & Konishi, T. (2024). Consideration of programming learning activity under automated test environment. *JSAI-Technical Report SIG-ALST, 100*, 83-88. (in Japanese)

Zeller, A. (2009). *Why Programs Fail: A Guide to Systematic Debugging, (2nd ed.).* Morgan Kaufmann Publishers.