Article

Design of an Online Programming Platform and a Study on Learners' Testing Ability

Nien-Lin Hsueh *, Lien-Chi Lai and Wei-Hsiang Tseng

Department of Information Engineering and Computer Science, Feng Chia University, Taichung 407, Taiwan
* Correspondence: nlhsueh@fcu.edu.tw

Abstract: Online judge (OJ) systems are essential in programming education as they efficiently evaluate learners' programming skills and reduce instructor workload. However, these systems often overlook the importance of software-testing concepts. To address this gap, we developed a system called Pytutor that integrates software-testing concepts to assess learners' programming abilities and proficiency by exploring students' test cases and learning behaviors. Drawing on software engineering theory and practical techniques, test capabilities are evaluated by analyzing the code coverage and mutation testing of Defining Test Cases. Since our experiment is conducted in an online environment, we can collect students' learning behaviors and further analyze the relationship between software engineering abilities and learning behaviors. We also analyzed the differences in programming and testing abilities between computer science majors and non-computer-science majors. Our findings suggest that better testing abilities may contribute to the improvement in programming abilities, whereas in the current Taiwanese education context, computer science majors do not necessarily have better testing abilities. This result provides suggestions for us to strengthen software-testing education no matter which type of students it is targeted at.

Keywords: software testing; online judge

Citation: Hsueh, N.-L.; Lai, L.-C.; Tseng, W.-H. Design of an Online Programming Platform and a Study on Learners' Testing Ability. *Electronics* **2023**, *12*, 4596. https://doi.org/10.3390/electronics12224596

Academic Editor: Mohamed Wiem Mkaouer

Received: 5 September 2023
Revised: 5 November 2023
Accepted: 6 November 2023
Published: 10 November 2023

1. Introduction

In modern society, programming has become a fundamental skill. With advancements in educational technology, many educational institutions have started adopting online judge (OJ) systems as teaching tools [1–3] to assist learners in acquiring and improving their programming abilities. OJ systems provide automated code evaluation and feedback, offering assistance to learners throughout the learning process. These systems not only accelerate the pace of instruction but also reduce the human and time costs associated with manual grading by instructors. Additionally, they enhance learning outcomes and effectively lower dropout rates [4]. Regarding OJ systems, Wasik mentioned the need for the secure, reliable, and continuous evaluation of code submitted by users worldwide. The systems can generally be categorized into two types based on their objectives. Competition-oriented systems typically support a limited number of programming languages and primarily aim to provide users with a platform for programming competitions, often including elements such as leaderboards. Educational systems, on the other hand, support a wider range of programming languages and incorporate gamification elements to engage users in problem-solving processes [5]. Qian et al. visualized and analyzed learners' performance on OJ systems by using a dashboard that provides real-time insights into the distribution of learners' mistakes and their mastery of knowledge. This fine-grained analysis assists teachers in monitoring and providing feedback on learners' activities [6].

In the field of software engineering, software testing is also a critical component. By writing test cases, software developers can ensure the correctness and reliability of their code. In education, teaching learners about software testing can contribute to improving the quality of their code [7]. Therefore, the ability to write test cases is crucial in the process of learning programming. However, the traditional OJ systems used for education only focus on testing the code written by learners, neglecting the importance of test-case development. Previous studies have taught learners about software-testing concepts in courses and investigated their testing abilities or programming skills. Buffardi and Edwards conducted research on the testing validity of novice programmers and found a strong positive correlation between the quality and quantity of early testing and assignment outcomes [8]. On the other hand, regarding the relationship between testing abilities and programming skills, Spacco and Fossat demonstrated that testing practices can improve learners' programming abilities and habits [9]. Fidge and Hogan observed the relationship between learners' testing and programming abilities through code coverage and program scores but found a suboptimal testing performance due to learners' lack of awareness [10]. Yang and Liu, in addition to coverage, employed mutation testing to assess learners' testing abilities and demonstrated a moderate positive correlation between the test efficiency index and programming abilities [11].

This study aims to explore learners' abilities and learning behavior in software engineering within an online Python programming course. We developed a system called Pytutor, which is a programming-practice platform used to assist in assessing learners' abilities. (our system's name is similar to the well-known code-visualization platform Python Tutor, https://pythontutor.com/, but we are totally different, because ours is an online judge system, accessed on 5 November 2023). The distinguishing feature of Pytutor is its ability to test not only learners' code but also provide practice opportunities for writing test cases. In the learning process of software engineering, learners' ability indicators and learning behavior serve as important measures to evaluate their progress and achievements. The ability indicators may include programming skills and testing abilities, while learning behavior encompasses activities such as watching course videos. Furthermore, there may be variations in programming and testing abilities among students from different departments or with different learning outcomes. For example, learners majoring in computer science may demonstrate better programming and testing abilities compared to non-computer-science students, or high-scoring learners may outperform low-scoring learners in programming and testing abilities. We compare these differences to gain insights into learners' performance in software engineering learning, which may contribute to providing targeted learning support and guidance. Lastly, there may be a correlation between learners' ability indicators and their level of engagement in software engineering learning with their final learning outcomes. We analyze data related to learners' ability indicators, learning participation, and final learning outcomes to determine whether there is a correlation among these variables. This will help us understand the relationship between learning behavior and final learning outcomes and potentially provide guidance on maximizing learners' learning achievements.

We will focus on the following three aspects in our research:

- RQ1: What is the performance of learners in terms of ability indicators in software engineering and learning behavior?
- RQ2: How do learners from different groups perform in terms of programming ability and testing ability?
- RQ3: Is there a relationship between ability indicators and learning behavior?

2. Related Work

The online judge system has evolved from automatic assessment systems. It possesses the capability to evaluate whether the program submitted by the user can pass a set of test cases. Depending on the usage scenarios, different online judge systems exhibit distinct characteristics.

Wasik classified online judge systems into four categories based on their purposes: "online compilers", "data mining, education, and competitive programming", "recruitment platforms", and "development platforms" [5]. The category of data mining, education, and competitive programming has piqued our interest. Therefore, we discuss three research-related areas that provide the foundation for designing courses and the necessary functionalities for platforms. Firstly, we explore the importance of testing capability for programmers and found in previous research that testing capability helps learners understand specifications and write more-robust programs. Next, we discuss which appropriate functionalities should be incorporated into automated assessment systems that emphasize testing capability.

2.1. Importance of Software-Testing Ability

In the current automated assessment systems, the emphasis is on evaluating learners' programming abilities.Testing capability involves understanding and formulating appropriate specifications, which are then used as test cases to examine the programs. Edwards observed that reinforcing test-driven development in the curriculum positively impacts students' ability to test programs [12]. Ala-Mutka pointed out that students must learn to design test cases before submitting their code and then proceed with testing the code [13]. Cerioli and Cinelli mentioned that at the very least, students should be provided with a partial set of tests to evaluate their projects as it helps improve their understanding of the problem [14]. Fidge and Hogan stated that testing capability is the ability to understand problems while programming capability is the ability to solve them, yet they found that learners' testing capability is significantly lower than their programming capability [10]. Fraser introduced gamification elements into the OJS in a software-testing course to enhance students' learning performance [15].

2.2. Programming Education and Assessment

We investigated the beneficial functionalities that automated assessment systems can bring to learners in programming education. If an automated assessment is applied in formative learning, the feedback provided to learners after they modify and resubmit their programs is helpful. Sun designed an online judge system that enables teachers to assign essential programming concepts to beginners for learning programming [16]. Suleman mentioned that multiple submissions aid in iterative learning [17]. Carless et al. described feedback as a key factor in developing high-quality learning [18]. Malmi et al. considered that feedback on assignments allows students to modify their submissions [19]. When students know what issues their submissions have and where their programs went wrong, they can use this information to learn from their mistakes. Providing fully automated feedback can be challenging, and many existing systems have room for improvement in this aspect. Brito developed a system that enhances students' programming competencies through problem solving and the utilization of problem-type repositories. This system also incorporates competitive programming features, allowing students to assess their programming skills in a timed environment [20]. Haynes-Magyar designed a self-directed learning environment named "Codespec". In this learning environment, students have the freedom to choose from various problem-solving approaches, including pseudocode problems, Parsons problems, Faded Parsons problems, fix-code problems, or write-code problems to tackle practice exercises [21]. Xia et al. mentioned that traditional teaching methods encounter challenges in conveying the complex internal processes of algorithms. To enhance the effectiveness of classroom instruction, they introduced an interactive computer-algorithm learning platform named "Progressive Blockly". This platform's main features include visual programming, dynamic visual demonstrations of algorithmic steps, and interactive instruction on the theoretical aspects of computer algorithms [22]. Polito introduced gamification features into their 2TSW system. Gamification is already widely used and involves integrating one or more game design elements and gameplay features into typically non-competitive domains, such as the learning environment [23]. Swacha also performed a comprehensive bibliometric survey on gamification education and found out that computer science is the area presenting the most interest in the gamification area [24].

2.3. *Software-Testing Education and Assessment*

In most automated assessment systems, learners submit their programs, and the system uses a pre-established test dataset to test the functionality of the code. Ala-Mutka pointed out that the scope of evaluation depends on the design of the test cases. The accuracy of the assessment and the value of the formative evaluation heavily rely on the design of the test cases used [13]. Vujosević-Janičić et al. stated that the grading is influenced by the test cases used [25]. Montoya-Dato et al. emphasized the need for the careful construction of the test case set to prevent erroneous programs from passing the tests. When incorrect programs are misjudged as correct, students who create erroneous code may remain unaware of the errors [26]. Jiang developed an online practice system that enables students to practice Selenium test scripts online and provides fair grading and feedback [27]. Kasahara et al. mentioned that low-quality code can lead to issues such as reduced productivity. Therefore, they propose the combination of gamification and code assessment to motivate students to write high-quality code in assignments [28]. García-Magariño et al. introduced UnitJudge to address the limitations of traditional online judges, which primarily focused on evaluating shorter code exercises and lacked feedback on the causes of errors. This system can individually test different code segments to provide valuable information for students to address errors in longer code practices [29].

3. Methods

Within this section, the developed OJ system for the study will be presented. Furthermore, the evaluation of learners' performance indicators will be elucidated, along with an explication of the terminologies and parameters utilized in the computation of these indicators.

3.1. *System Design*

In this section, we introduce our developed OJ system, Pytutor, including the key design aspects, features, operation process, and security.

3.1.1. System Design Principles

As shown in Figure 1, our Pytutor is a web-based system based on the Django framework, which follows the Model–View–Template architecture for easily extension. The template module provides an interface for learners practicing the exercises in a Python-coding editor. When learners submit their code, the view-control module will call the automated assessment tool for automated evaluation. The software-testing module calculates the code coverage, mutation scores, and our defined metrics (see Section 3.3). To make sure our system is easily used and safe, we built the system by referencing the issues mentioned by Ihantola [30].

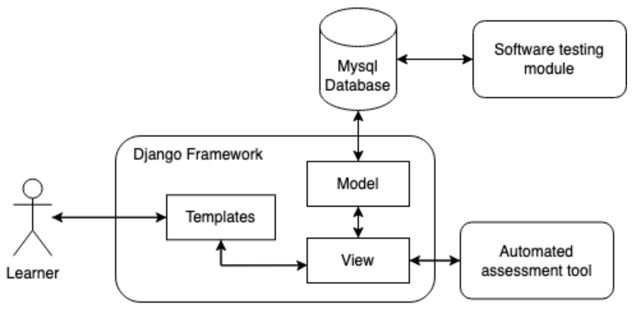


Figure 1. The architecture of Pytutor.

The system interface includes a compile error feature that presents the compilation error messages in their original form. Learners can refer to these messages to effectively debug their code and learn how to correct errors directly.

In terms of problem design, each problem is predesigned with a set of test cases known as Hidden Test Cases (HTCs). Upon completing their program, learners can submit it for evaluation, and the system will assess their programs by using these HTCs. Learners can promptly view the scores obtained from the test results.

In addition to HTCs, the system offers Open Test Cases (OTCs). Learners can utilize these OTCs to assess their programs. They also have the ability to design their own test cases, referred to as Defining Test Cases (DTCs), and execute their programs to verify the accuracy of their solutions under these DTCs. During the verification process using DTCs, learners are required to ensure the correctness of the test outcomes. Once learners believe that their programs are error-free following testing with the provided test cases, they can proceed to submit their programs. At this stage, the system will evaluate the learners' programs by using HTCs and promptly display the obtained scores. In summary, the system offers learners three categories of test cases: predesigned Hidden Test Cases, open-ended Open Test Cases, and the flexibility to create their own custom Defining Test Cases. This comprehensive approach enables learners to thoroughly test and refine their programs prior to final submission.

The types of test cases are as follows:

- Open Test Case (OTC): The system provides the Open Test Cases (OTC). Learners can test their program under these OTCs and see the result to evaluate their code independently.
- Hidden Test Case (HTC): Each problem is predesigned with some test cases called Hidden Test Cases (HTCs). After learners complete their program, they can submit them to the system, which will then test the programs by using these HTCs. The system will provide immediate feedback to learners in the form of scores based on the test results.
- Defining Test Case (DTC): Learners also have the option to design additional test cases, referred to as Defining Test Cases (DTCs). These test cases are the ones that learners believe the code should pass. Once learners feel confident that their programs are error-free under these test cases, they can submit their programs for further evaluation.

Moreover, learners are expected to adhere to two principles while answering questions. Firstly, the "Completion Principle" allows learners to make multiple attempts until they successfully pass all the HTCs. Secondly, the "Cautionary Principle" enforces a deduction of 3 points for each formal submission made by learners, encouraging them to think thoroughly before formally submitting and ensuring error-free programs.

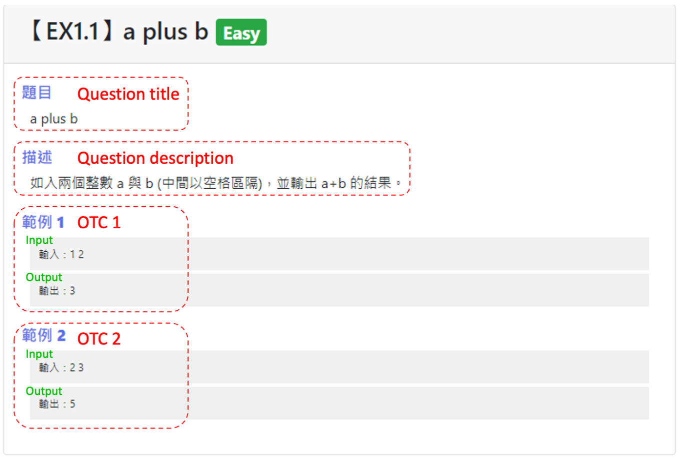
In summary, this system provides features such as displaying compilation errors, HTCs, OTCs, and DTCs in its interface. Learners can utilize these features to debug, test, and evaluate their programs while following the "Completion Principle" and the "Cautionary Principle" to improve their scores and testing proficiency.

3.1.2. System Interface

The answering page of the system comprises three main areas: the question-information area, the answering area, and the result-display area. In the question-information area, learners can find the question name, description, and OTC, all of which are illustrated in Figure 2.

Our Pytutor offers two distinct interfaces, namely the testing mode and the formal submission mode, in the answering area and result-display area. Figure 3 illustrates the testing-mode interface, where the upper area serves as the code-writing section for learners while the lower area allows them to design their DTC and view the output of their test results, including the history of the last three results.

In formal mode, the system tests the code submitted by learners using HTCs. It then displays the obtained score or error message based on the testing result. Additionally, to prevent accidental submissions, we provide a confirmation mechanism where learners need to check the "Confirm Submission" option before proceeding with the formal submission.


【EX1.1】a plus b Easy

題目 Question title

a plus b

描述 Question description

如入兩個整數 a 與 b (中間以空格區隔)，並輸出 a+b 的結果。

範例 1 OTC 1

Input

輸入：1 2

Output

輸出：3

範例 2 OTC 2

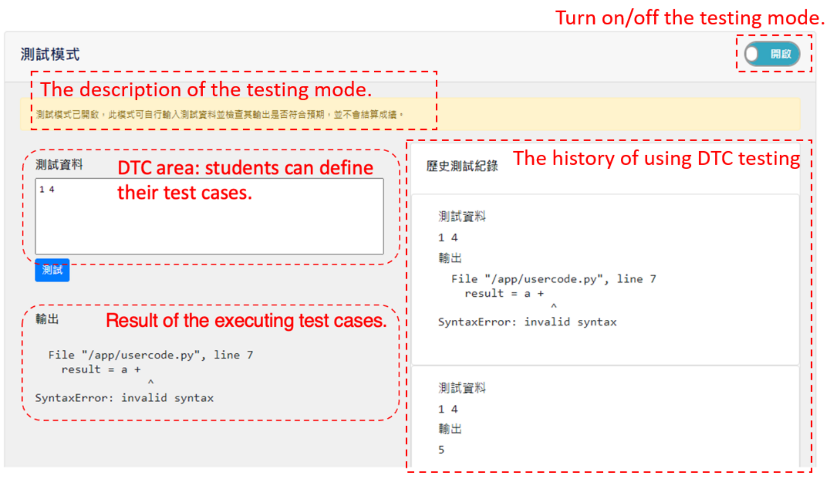
Input

輸入：2 3

Output

輸出：5

Figure 2. The question-information area.


Turn on/off the testing mode.

測試模式 開啟

The description of the testing mode.

測試模式已開啟，此模式可自行輸入測試資料並檢查其輸出是否符合預期，並不會結算成績。

測試資料 DTC area: students can define their test cases.

```
1 4
```

測試

輸出 Result of the executing test cases.

```
  File "/app/usercode.py", line 7
    result = a +
               ^
SyntaxError: invalid syntax
```

歷史測試紀錄 The history of using DTC testing

測試資料

```
1 4
```

輸出

```
  File "/app/usercode.py", line 7
    result = a +
               ^
SyntaxError: invalid syntax
```

測試資料

```
1 4
```

輸出

```
5
```

Figure 3. The result-display area in the testing mode.

3.1.3. Security

Due to security considerations [31,32], learners' programs are submitted to the server for execution and results retrieval. To prevent the submission of malicious programs or those with infinite loops that could potentially harm the server, we implemented Docker container technology [33]. Upon program submission, the system automatically creates a dedicated virtual container for execution. After obtaining the output, the container is promptly deleted. Consequently, any malicious program will only affect the specific container, which the system deletes within seconds. This approach effectively mitigates security risks.

3.2. Process

The workflow diagram of the system is illustrated in Figure 4. After writing their program, learners can either test it by using the testing mode or directly submit it by using the formal submission mode. In the testing mode, learners can design DTCs or use OTCs to validate their code and observe the output results. If the output does not meet their expectations or errors occur, they can revise the program and retest or formally submit it after making corrections. Once the testing results match their expectations, learners can choose to design additional DTCs for further testing or proceed directly to the formal submission mode without additional testing.

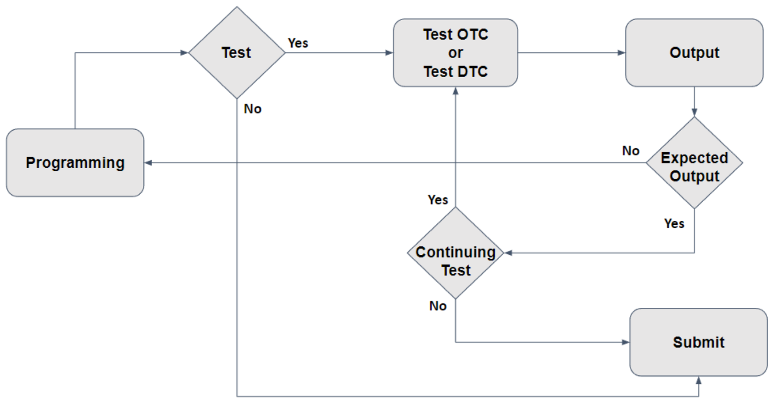


Figure 4. The workflow diagram of the system.

3.3. Indicators

Our research aimed to investigate whether students' sensitivity to test cases during the process of learning programming is related to their programming ability. Regarding the assessment of programming ability, since we are in an online environment, we can evaluate this ability based on system logs, not just the scores in student assignments. Our proposed approach considers the number of submission attempts, the code complexity, and their pass rate in the online system. This consideration is based on some previous work, our observations, and practical knowledge of software engineering. If a learner achieves the same score as others but with relatively fewer submission attempts, their programming ability should be considered superior. Research by Xu Bin and others also shows that students' programming ability is related to the number of submissions in the online review system. They used an exploratory factory analysis model to identify underlying variable structures from log data submitted by students in an online assessment system and assess students' programming proficiency to predict "at-risk" learners [34]. In our approach, we extend this model to consider the factor of program complexity because complex programs need more tries and submissions. We standardized the number of submission attempts by code complexity, which is based on the theory of cyclomatic complexity [35]. In addition to the number of submission attempts, the pass rate is also a factor to judge the programming capability. In Yang et al.'s work [11], the programming ability is evaluated by the pass rate and code maintainability. In our model, we did not consider the code maintainability issue because our model was targeted at new learners and the assignments were not complex enough to consider this issue. We also introduced the concept of "resilience" to explore whether attitude affects programming behavior. When the amount of code submitted was significantly higher than the complexity of the assignment, students showed higher "resilience (r)" in solving the problem. Pereira et al. also proposed the same term "resilience" to refer to students who submitted more code than the median number of attempts and analyzed its relationship with final grades [36]. The detailed of evaluation approach is described in Sections 3.3.1 and 3.3.2.

As far as testing capability is concerned, it is seen as the ability to provide effective test cases to explore code errors. The factors and techniques we take into concern in our online context are program complexity, number of test cases and their code coverage, and mutation scores. The basic principle is the higher the coverage of a test case, the higher the testing efficiency [37], which also means the better the testing ability of the learner who provides this test case. In Yang et al.'s work [11], the test ability is also evaluated by the coverage rate and mutation score. We did not apply the other factors maintainability and code efficiency because we think maintainability is suitable for capstone projects, not for the small programs in our course setting. Regarding the issue of program code efficiency, since our system provides timing constraints, if the program runs for too long, it will be evaluated as a wrong answer. What is more interesting is that in our system, test cases can be divided into Hidden Test Cases (HTCs), Open Test Cases (OTCs), and Defining Test

Cases (DTCs). The first two test cases are provided by teachers, and the last DTC is defined by students, so the test ability is mainly evaluated by DTCs. We will provide detailed definitions in Section 3.3.3.

3.3.1. Resilience of Program Assignment (r)

The concept of resilience, represented by r, indicates that a learner should persist in their efforts despite making mistakes or not obtaining the complete answer. This value is associated with tr, which keeps track of how many times a learner attempts to respond when facing challenges. Specifically, tr includes both syntax errors (in tests and official submissions) and submissions that fail to meet all the requirements of the HTC tests. Meanwhile, ch_p represents the challenge level of a given program, p. A higher ch_p implies that p is more complex, making it more difficult to complete. The value of ch_p is determined by the decision count in the p program, with each case being tested once. We assume that a learner's tr should be proportional to ch_p. So, if a student's tr is higher than ch_p, it indicates that the learner faced significant challenges with this problem but was willing to continue trying. Therefore, we apply an exponential function to weight tr. The reason for using an exponential function is that it increases the weighting when there is a substantial difference between tr and ch_p but does not create a disproportionately large gap. The constant k fine tunes the curve of our equation. The calculation of r can be expressed by using the following equation:

$$r = e^{k\times(tr-ch_p)} \tag{1}$$

Taking Figure 5 as an example, when the learner's challenge count equals the expected challenge count (i.e., $tr = ch_p = 10$ on the x-axis of the graph), it is referred to as the "resilience origin", and $r = 1$ at this point. Beyond the resilience origin, the learner's r is higher while it is lower below the origin. To control the magnitude of the weighting within an appropriate range, we choose the coefficient $k = 0.1$.

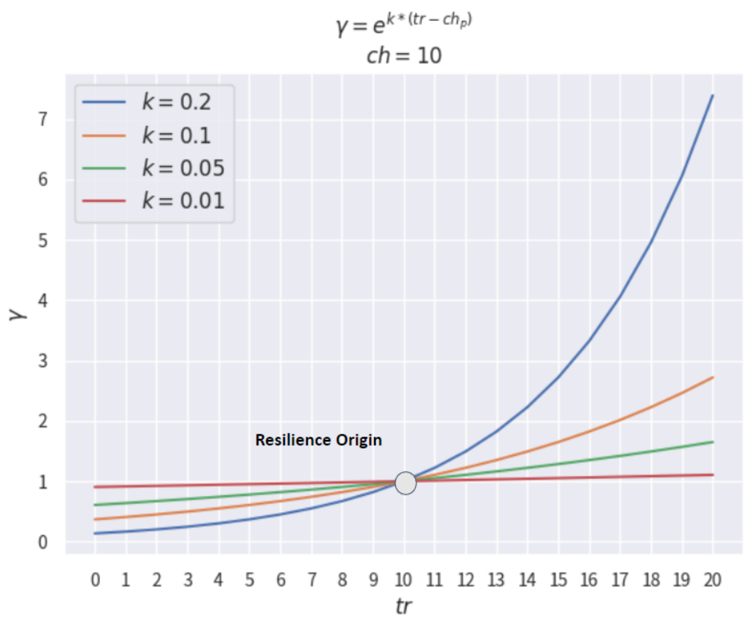


Figure 5. The impact of the value of k on the resilience of answering questions.

3.3.2. Programming Ability Index (PAI)

The Programming Ability Index serves as an indicator for assessing the programming proficiency, *taking into account the differences in programming skills that arise from the frequency of encountering setbacks under the same score.* A higher PAI indicates that a learner encounters fewer setbacks when achieving similar scores, thus reflecting a more-robust programming ability.

In our system evaluation, learners can continually submit their answers until they attain perfection. However, the pure program score, denoted by s and derived from the basic score and the HTC pass rate, does not sufficiently encapsulate the full scope of a student's programming competence. We consider that the learners' r is quantified by considering the number of submission attempts and instances of setbacks that they experience. Therefore, the PAI is defined as the difference in r under similar scores, which signifies the disparity in the programming proficiency. The PAI can be calculated as follows:

$$PAI = \frac{s}{r} \tag{2}$$

Taking Figure 6 as an example, when $r = 1$, PAI = s, referred to as the "programming ability origin". When $r < 1$, it indicates that learners achieve higher scores with greater efficiency than expected, resulting in the PAI being higher than the programming-ability origin. Conversely, if $r > 1$, it implies lower efficiency and leads to a lower PAI.

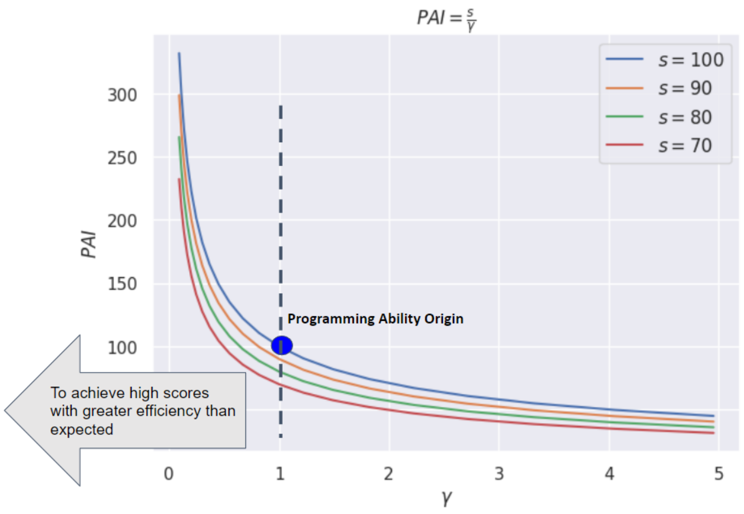


Figure 6. The impact of the value of r on PAI.

The learner's score, denoted as s, is formulated considering both the base score and the proportion of HTCs successfully passed. Specifically, the base score, labeled as *base*, represents the minimum attainable score when the submitted code clears at least one HTC. On the other hand, p signifies the ratio of HTCs passed by the submitted program relative to the total number of HTCs. With these components in mind, the score is determined by the equation

$$s = base + (100 - base) \times p \tag{3}$$

Using the previously described method, when a learner's r in answering is higher, it indicates lower proficiency in solving problems quickly. Thus, the resilience (r) serves as a dual indicator, representing both the learner's programming ability and acting as an inverse indicator of programming proficiency.

By applying this method to calculate the PAI for each learner, the average PAI is determined to be 121.80, which exceeds the maximum score of 100 for the questions. This suggests that learners achieve higher scores with greater efficiency than expected on most questions.

3.3.3. Testing Efficiency Index (TEI)

Before discussing the Testing Efficiency Index (TEI), we define the Testing Quality Index (TQI) as an indicator that measures the quality of the test case. To assess the quality of the test case, we conduct code coverage and mutation testing, yielding two key parameters: the branch coverage b_t and mutation score ms_t. The branch coverage b_t represents the fraction of branches in the code that are executed or covered by the test case t. Meanwhile, the mutation score ms_t quantifies the fraction of mutants neutralized by the test case t relative to the overall mutants produced. Recognizing the integral roles of both metrics, the TQI for each test case t is computed as the geometric mean of b_t and ms_t, with the index ranging from 0 to 100. The formal representation of this relationship is

$$TQI_t = \sqrt{b_t \times ms_t} \tag{4}$$

In our analysis of each student, we integrated both their DTC and OTC. Following this, we conducted code coverage and mutation tests anew to quantify any enhancement in their TQI. Specifically, TQI^+ stands for the outcome when learners incorporate DTCs derived from OTCs. The numerator, TQI_{OD}, represents the TQI calculated by combining both the *OTCs* and the *DTCs* provided by the learners. In contrast, the denominator, TQI_O, strictly considers the TQI originating from the *OTCs*. The mathematical representation of this approach is given by

$$TQI^+ = \frac{TQI_{OD}}{TQI_O} \tag{5}$$

However, TQI^+ might not comprehensively encapsulate a learner's Testing Efficiency Index. For instance, if two learners possess identical TQI^+ values but one creates fewer test cases, it implies superior efficiency in the test-case quality for that learner. Recognizing this, we factor in the number of test cases that each learner designs.

The Testing Efficiency Index (TEI), defined as *the degree of enhancement in the software-testing metrics of DTCs relative to OTCs, is determined for every learner for each question*. In this context, TEI quantifies the extent of improvement; te represents the collective number of test cases during the testing phase, equating to the sum of DTCs and OTCs; and ch_t denotes the anticipated test cases needed for a program—a number that invariably rises with the program's complexity. The corresponding TEI formula is expressed as

$$TEI = \begin{cases} 1 & , te = te_o \\ TQI^+ \times (1 + e^{-k \times (te - ch_t)}) & , te > te_o \end{cases} \tag{6}$$

As the complexity of the program increases, the expected test-challenge index also increases. The constant k is used to adjust the extent of the weighting. We assume that learners who use OTCs for testing possess a basic testing ability. Therefore, we set the TEI value of the learners who meet this assumption to 1 by default. Furthermore, we believe that learners who have a willingness to design their DTCs should have their testing ability built on top of their basic testing ability. However, since the value of the exponential function may be less than 1, it could result in some learners having a lower TEI than those who only possess a basic testing ability due to poor DTC quality. To avoid this situation, we add 1 to the result of this exponential function.

Taking Figure 7 as an example, when the total number of test cases for a learner is equal to the expected challenge index (i.e., $n = ch = 10$ on the x-axis of the graph), it is referred to as the "testing ability origin". When $te < 10$, it indicates that the learner can achieve good testing quality with higher efficiency than expected, resulting in a higher TEI than the testing ability origin. Conversely, if te is greater than or equal to 10, the TEI will be lower. As for the coefficient k, to avoid drastic changes or unclear increments, we set $k = 0.1$, which provides a moderate and discernible increase.

Using this approach, the average TEI for each learner is only 1.89, indicating that many learners exhibit a low testing efficiency or a lack of a willingness to test, resulting in a majority of the TEI values being 1 for most questions.

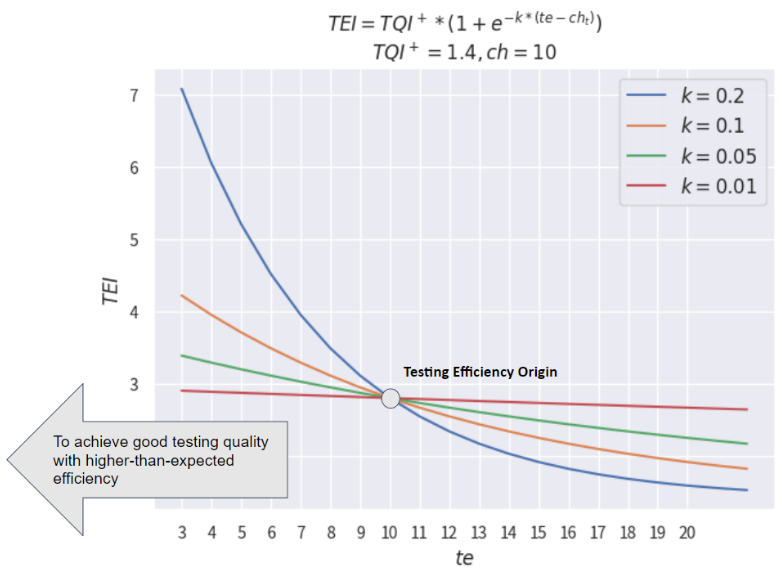


Figure 7. The impact of the value of *k* on TEI.

4. Experiment

We conducted a study on the programming and testing abilities of learners, focusing on whether they could design effective test cases to enhance their programming skills. We designed an online course to evaluate learners' sensitivity to test cases, even without prior knowledge of software-testing concepts, as they started learning programming from scratch. Additionally, we analyzed the relationship between learners' backgrounds and their learning outcomes based on their programming and testing abilities.

4.1. Course Design

This course is an online self-learning course, but we also planned five physical meetings. Montoya-Dato et al. found that questions answered by instructors foster both independent learning and reflective thinking, which are considered crucial for deep learning [26]. The purpose of these meetings is to allow learners to have face-to-face discussions with classmates and teachers regarding course-related questions.

The course is designed to be conducted asynchronously online. Learners can watch prerecorded course videos or practice programming on the Pytutor platform at any time. If learners have any questions about the course content or programming, they can always send inquiries through the Microsoft Teams course group to the teacher or teaching assistant.

The course consists of seven units, comprising 64 videos. The first five units cover the fundamentals of programming, including an introduction to programming, basic structures and operations, logical operations, and object-oriented programming. The last two chapters focus on practical application examples, specifically data processing and object design. There are a total of 50 exercises on the Pytutor platform, with 1 exercise in Unit 1, 9 exercises in Unit 2, 13 exercises in Unit 3, 18 exercises in Unit 4, and 9 exercises in Unit 5. Among the 50 exercises, each exercise includes two Open Test Cases and five Hidden Test Cases. The average cyclomatic complexity of all the exercises is 4.06. The average branch coverage for exercises with Open Test Cases is 0.92, with an average mutation score of 0.75. The distribution of the Test Quality Index (TQI) for each exercise is illustrated in Figure 8.

Our assessment methods consist of a formativeassessment and summative assessment. The formative assessment is based on the scores that the learners achieve during the learning process, calculated as the average score of all the exercises on Pytutor. On the other hand, the summative assessment relies on the grades that the learners achieve in the final programming exam.

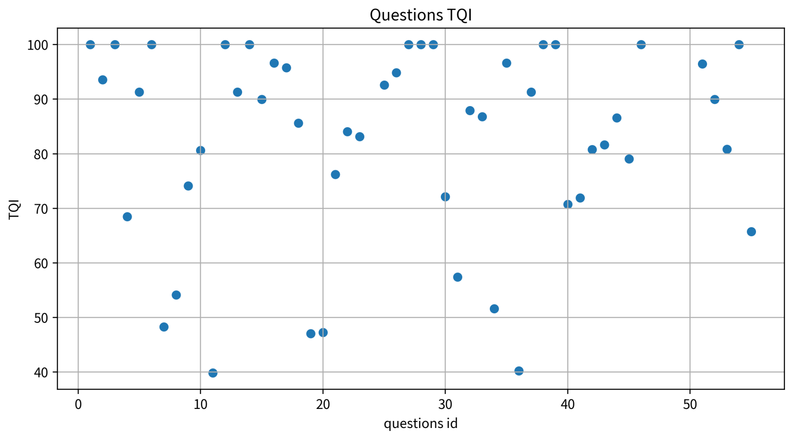


Figure 8. The distribution of the Test Quality Index (TQI) for each exercise. Some exercises have very low TQI and need to be improved by learners.

4.2. Participants

The course had a total of 69 participants. Among them, 27 were computer-science-department learners. In terms of programming-language experience, 54 learners had previous programming experience, with 15 of them having learned Python. Regarding their motivations for taking the course, 44 learners expressed an interest in programming, 30 learners wanted to acquire a second skill, and 21 learners chose the course because of the convenience of online learning. Additionally, 13 learners mentioned that they enrolled in the course together with their classmates and all expressed an interest in programming.

4.3. Procedure

At the beginning of our course, we do not make all the content available. Instead, we progressively release the videos and exercises for each unit over a span of seven weeks. Learners can watch the course videos to acquire knowledge and practice programming on the Pytutor platform. There are no restrictions on accessing unit content, and learners do not have to watch all the videos before engaging in programming exercises. The opened content remains accessible without time limitations, allowing learners to review past videos and resources at any time. Two weeks after the opening of the content for the seventh week, learners undergo the summative assessment. The summative assessment consists of four questions, with a total duration of 40 min for answering.

4.4. Measures

We collected data from three sources in our study:

- Questionnaires: at the beginning of the course, we conducted a survey that included information such as the learners' department, grade level, programming experience, and motivations.
- Event logs: OpenEdu provided various behavior logs for video watching, including *load_video*, *play_video*, *seek_video*, *pause_video*, and *stop_video*. We utilized these logs to calculate the viewing time and completion rate of individual videos for each learner. On the Pytutor platform, we collected the learners' behavioral data while answering the exercises. In the testing mode, we recorded the code, execution results, and test data for each test attempt. In the formal submission mode, we recorded the learners' code and the number of passed Hidden Test Cases. Additionally, we also recorded the learners' answering times for each exercise.
- Scores: We calculated the learners' scores for each exercise and averaged them to obtain the formative-assessment score. We also recorded the summative-assessment score.

4.5. Result

4.5.1. The Engagement of Learners

We collected data on the learners' behavior in terms of watching course videos and answering exercises on OpenEdu and Pytutor. We conducted some descriptive analyses, as shown in Table 1.

Table 1. Statistical data of learners' engagement.

Pytutor	
The average number of submissions per learner per question.	1.43 times
Proportion of learners who perform at least one public test case before submission.	59%
Proportion of learners who perform at least one Defining Test Case before submission.	44%
Proportion of learners who submit without performing any tests.	23%
The average number of attempts (*tr*) of learners.	3.23
The average number of test cases (*te*) of learners.	3.41
The average number of public test cases executed by each learner per question.	3.96 times
The average number of Defining Test Cases executed by each learner per question.	2.94 times
The average time spent by each learner on a question.	14.45 mins
OpenEdu	
Average completion rate of video viewing by learners.	56%
Average playback speed of video viewing by learners.	2.08

In the logs of video-based learning, the average completion rate represents the proportion of the total viewing time for course videos among all learners. In our experiment, each learner watched an average of 56% of all course videos. The average video playback rate refers to the average speed at which the learners watch videos. Our calculation yielded the average playback speed is 2.08.

Regarding the behavior of exercise answering, we found that learners, on average, made 1.43 formal submissions per exercise. This indicates that even with the cautious grading approach we adopted in the course design, most learners made two or more submissions. On average, learners spent 14.45 min on each exercise. The average value of *tr* was 3.43, indicating that most learners encountered three or more frustrations but still continued their attempts. The average value of *te* was 3.41, meaning that, besides OTCs, learners added, on average, more than one DTC. The average number of times learners used OTCs for testing was 3.96, showing that before conducting DTC testing, most learners used public test cases to verify if their code met the requirements of the exercise before proceeding to DTC testing. The proportion of exercises in which learners conducted at least one OTC test before formal submission was 59%, indicating that the majority of learners used the testing mode and OTC to evaluate their code before making a formal submission. On average, learners conducted 2.94 DTC tests per exercise, taking into account the repeated usage of the same DTC. The proportion of exercises with at least one added DTC test before submission was 44%, indicating that more than half of the exercises, on average, did not have an added DTC test before formal submission. The proportion of exercises that were formally submitted without any OTC or DTC testing was 23%, indicating that, on average, 23% of the exercises were submitted without any testing by the learners.

4.5.2. The Indicators of Learners

Before analyzing the learners' TEI and PAI, we excluded learners who had no activity during the later stages of the experiment. In total, we had 61 learners remaining after the exclusion. We calculated various indicators for these learners, including the PAI average and TEI average, which represent the average performance and task engagement for each learner across different exercises. The distribution of these indicators for all learners is depicted in Figure 9. Descriptive statistics for the indicators are presented in Table 2.

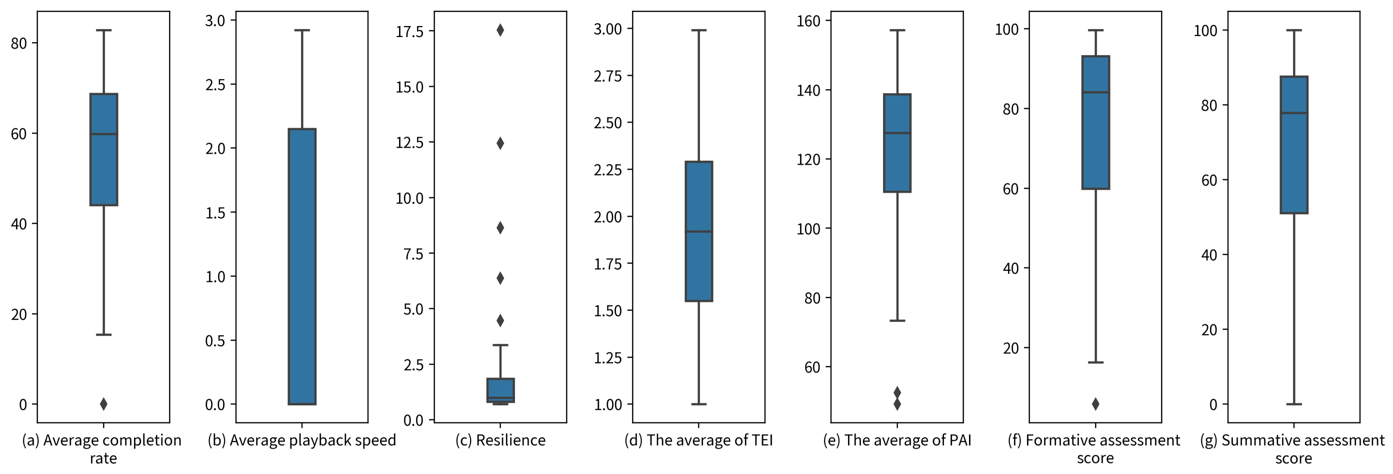


Figure 9. The box-and-whisker figure of indicators. The results showed that some students showed unusually high resilience when resolving the program assignments.

Table 2. The descriptive table of indicators.

	Average Completion Rate	Average Playback Speed	*r*	The Average of TEI	The Average of PAI	Formative Assessment	Summative Assessment
mean	0.56	2.08	1.97	1.89	121.80	72.66	68.04
std	0.17	0.70	2.82	0.49	23.64	26.17	23.69
min	0.00	0.00	0.70	1.00	49.18	5.82	0.00
25%	0.44	1.64	0.81	1.55	110.52	59.86	51.09
50%	0.60	2.16	0.99	1.92	127.55	84.08	77.93
75%	0.69	2.63	1.85	2.29	138.75	93.18	87.61
max	0.83	2.92	17.53	2.99	157.19	99.64	100.00

5. Discussion

Our research began by defining the learners' programming ability and testing ability, and we used an OJ system with testing functionality in conjunction with an online course to examine the correlation between our defined indicators and the learners' engagement. In this study, we collected data on the learners' behaviors during the learning process, including video viewing and completing practice exercises. After calculating the indicators for each learner, we explored their correlation with the learners' backgrounds and learning outcomes. We will answer each RQ in this section.

5.1. RQ1: What Is the Performance of Learners in Terms of Their Ability Indicators in Software Engineering and Learning Behavior?

We categorized the learners' abilities into four types and grouped them based on their backgrounds and learning achievements. We categorized each learner into four types based on their PAI and TEI, using percentile ranks to determine their abilities relative to the entire group. Learners with an ability index ranging from 1 to 50 were classified as low, while those ranging from 51 to 99 were classified as high. The four types are as follows:

- High PAI and high TEI (HPHT)
- High PAI and low TEI (HPLT)
- Low PAI and high TEI (LPHT)
- Low PAI and low TEI (LPLT)

In Figure 10, among the 61 learners, 11 belong to the category of HPHT, 20 belong to HPLT, 19 belong to LPHT, and 11 belong to LPLT.

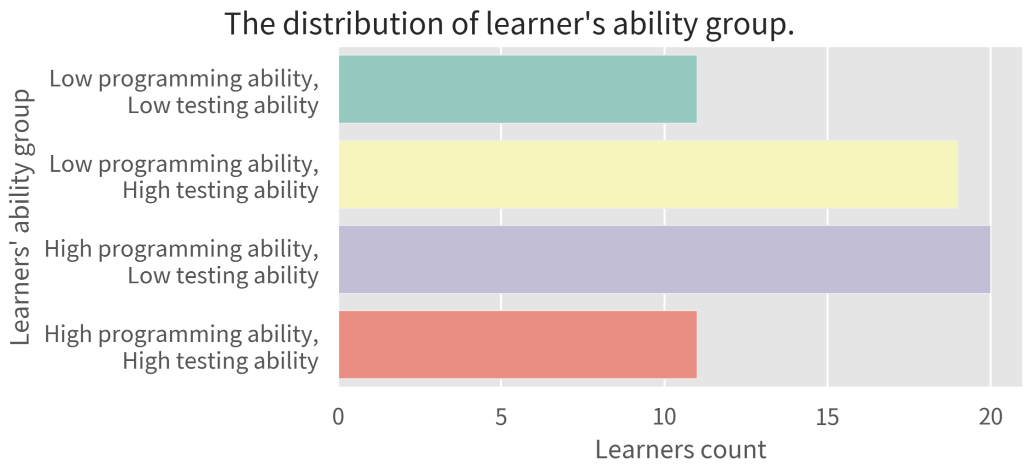


Figure 10. The bar plot of the learners in four categories.

Table 1 presents the learning behaviors of the participants gathered from the Pytutor and OpenEdu platforms. According to the data in the table, we observed that even when we imposed punitive deductions for multiple formal submissions, each learner, on average, made 1.43 attempts per question. Based on the values in the table, including the proportion of learners who perform at least one public test case before submission (59%), proportion of learners who perform at least one Defining Test Case before submission (44%), and proportion of learners who submit without performing any tests (23%), it is evident that a majority of the formal submissions were made without sufficient testing. The lack of testing resulting in the need for multiple formal submissions contributes to the increase in the average number of formal submissions.

The value of the average number of attempts (*tr*) of learners is 3.23. The value of the average number of public test cases executed by each learner per question is 3.96 times. Based on these two values, the learners utilize OTCs to ensure the correct execution of their code. Moreover, from the value of the average number of test cases (*te*) of the learners (3.43), it can be determined that learners incorporate DTCs to test their code. It is evident from the average number of DTCs executed by each learner per question (2.94 times) that DTCs are used for multiple rounds of code testing, indicating that learners identify segments of code requiring modification during DTC testing. On average, learners spend 14.45 min on each question. Our summative assessment entails completing four programming questions within a 40 min timeframe. Calculated proportionally, learners are expected to complete 69% of the questions in the summative assessment. In Table 2, the average for the summative assessment is 68.04, which closely aligns with the anticipated value. However, the median is 77.93, indicating that a majority of students perform even better.

Regarding video-viewing behavior, the average completion rate stands at only 56%, signifying that most learners do not watch all of the videos. Additionally, the average playback speed is 2.08, indicating that learners tend to watch course videos at an accelerated pace, reflecting a lower level of patience when it comes to learning with video content.

Based on the above experimental results, it was found that the majority of learners have relatively low patience with course content. This is evident in the low viewership of course videos, the use of high-speed video playback, and formal submissions on the programming-practice platform without sufficient testing. However, it was also discovered through the analysis of the DTC that Defining Test Cases can effectively identify segments of code that require modification. We recommend that instructors and course designers consider subdividing course materials or videos into smaller partitions based on specific knowledge points. Furthermore, they can encourage learners to create additional Defining Test Cases to improve their programming-practice scores.

5.2. Rq2: How Do Learners from Different Groups Perform in Terms of Programming Ability and Testing Ability?

In order to investigate whether learners' PAI and TEI vary across different groups, we categorized learners based on their backgrounds and learning outcomes. The groups categorized by learning outcomes are divided into two types: scores from formative assessments and scores from summative assessments, and we separated learners into high-score and low-score groups based on the median. Since the distribution of the learners' indicators did not follow a normal distribution, we employed the Mann–Whitney U test to examine the differences.

Background

When considering the grouping based on the learners' backgrounds, as shown in Figure 11, among the 26 learners with a computer science (CS) background, 19 fall into the High Programming Ability (HP) group while only 11 fall into the High Testing Ability (HT) group. Among the 35 non-CS background learners, 12 fall into the HP group and 19 belong to the HT group. Based on our inference, CS learners mostly possess programming backgrounds, leading to greater self-confidence in their code. Consequently, they tend to conduct fewer tests, resulting in a relatively lower testing proficiency for the majority. However, they excel in their programming ability. Conversely, non-CS learners tend to conduct more tests to avoid error messages.

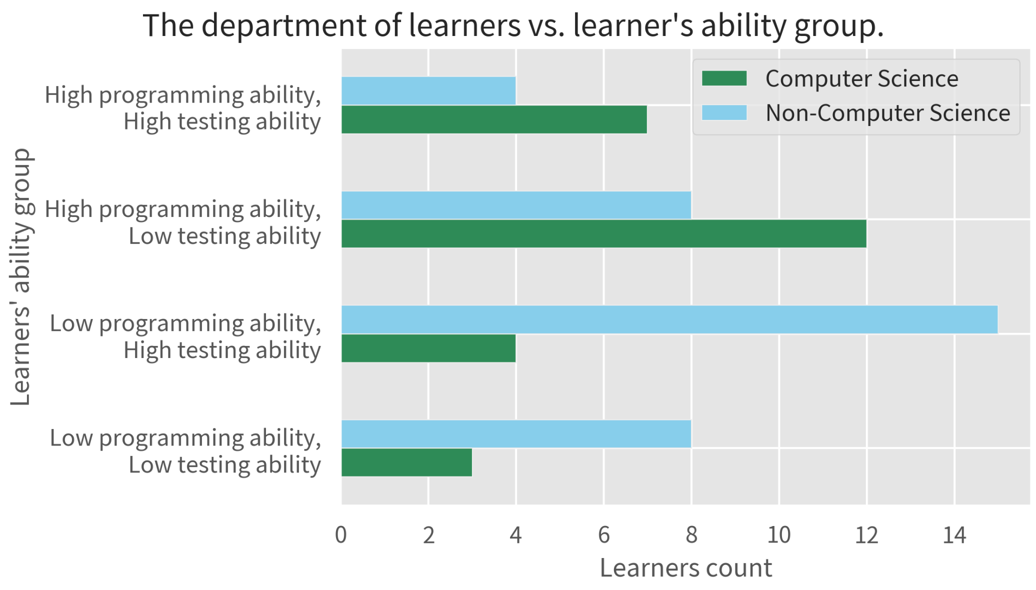


Figure 11. The learners categorized by departmental background across four learning-ability categories.

Regarding resilience, the distributions for both groups are depicted in Figure 12a. The Mann–Whitney U test resulted in a p-value of $0.001 < 0.05$, indicating a significant difference. The distribution of the PAI is presented in Figure 12b, with a p-value from the test of $0.0003 < 0.05$, demonstrating a significant difference. On the other hand, the distribution of the TEI, as shown in Figure 12c, yielded a p-value of $0.277 > 0.05$, suggesting no statistically significant difference. Based on the above results, there is a significant difference in resilience and the PAI, while there is no significant difference in the TEI. We speculate that CS learners can achieve high scores without the need for many test cases.

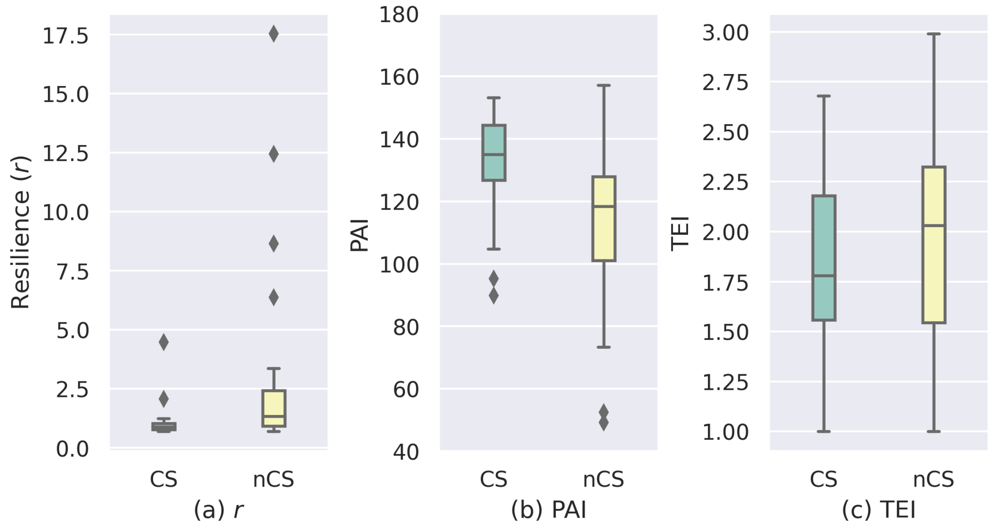


Figure 12. The box-and-whisker plot of indicators for different backgrounds. CS students tend to have more concentrated abilities. Non-computer science majors showed higher resilience, but they also need more testing, resulting in higher TEI.

Formative assessment

Based on the median of 84.08 for the formative assessments, we divided learners into two groups: 31 with a high score and 30 with a low score. The distribution of the ability types categorized by the formative assessment is shown in Figure 13. Among the 31 learners in the high-scoring group, 21 possess higher testing abilities. This indicates that having strong testing abilities is crucial for achieving relatively high scores in the formative-learning process.

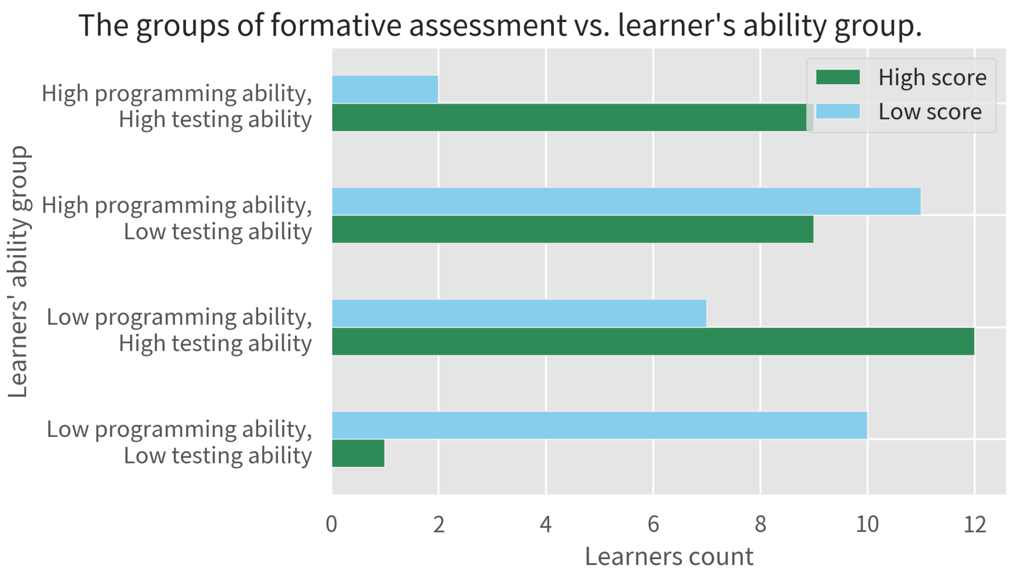


Figure 13. The learners are categorized based on the difference in formative-assessment results across four learning-ability categories.

Regarding resilience, the distributions of the two groups are displayed in Figure 14a. The *p*-value from the Mann–Whitney U test is 0.153, which is greater than 0.05, indicating no significant difference. For the PAI, the distribution is shown in Figure 14b, and the *p*-value from the test is 0.035, which is less than 0.05, indicating a significant difference. Similarly, for the TEI, the distribution is presented in Figure 14c, and the *p*-value from the test is 0.036, which is less than 0.05, also showing a significant difference.

During the formative-learning process, the r does not affect the outcomes of the learning process. However, the differences in the PAI and TEI are more significant in terms of outcomes. Therefore, having a better programming ability or testing ability has a greater impact on achieving excellent learning outcomes.

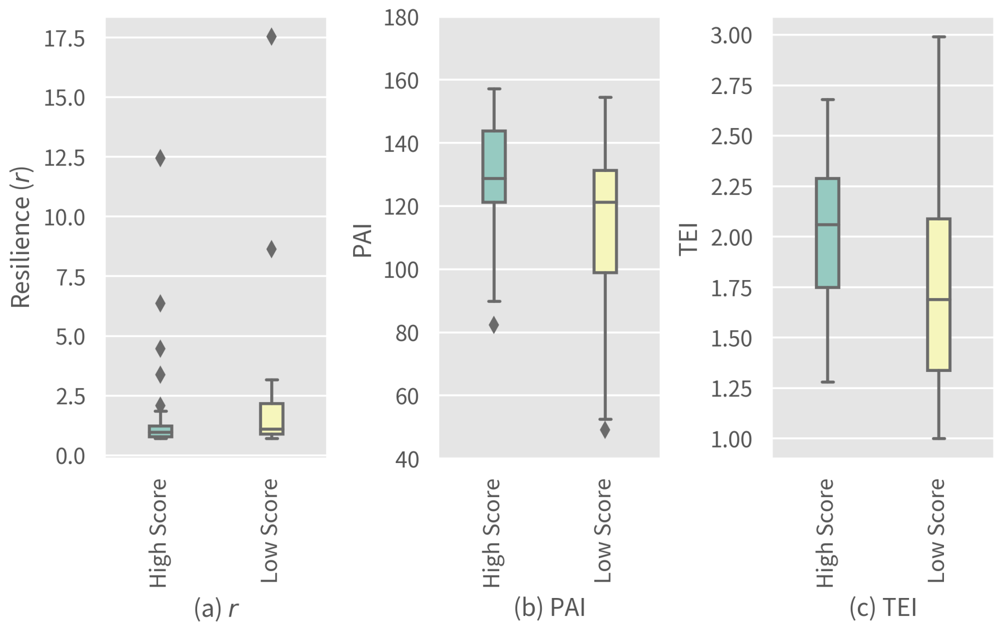


Figure 14. The box-and-whisker plot of indicators for formative assessments. For r, there was no significant difference between the two groups of high and low groups. For the TEI index, the average of the high group is significantly higher than that of the low group.

Summative assessment

Based on the median of 77.93 for the summative assessments, we divided the learners into two groups: 31 high achievers and 30 low achievers. The distribution of the ability types categorized by the summative assessments is shown in Figure 15. Among the 31 learners in the high-score group, there are 18 HP learners and 13 LP learners. Among the 30 learners in the low-score group, there are 13 HP learners and 17 LP learners.

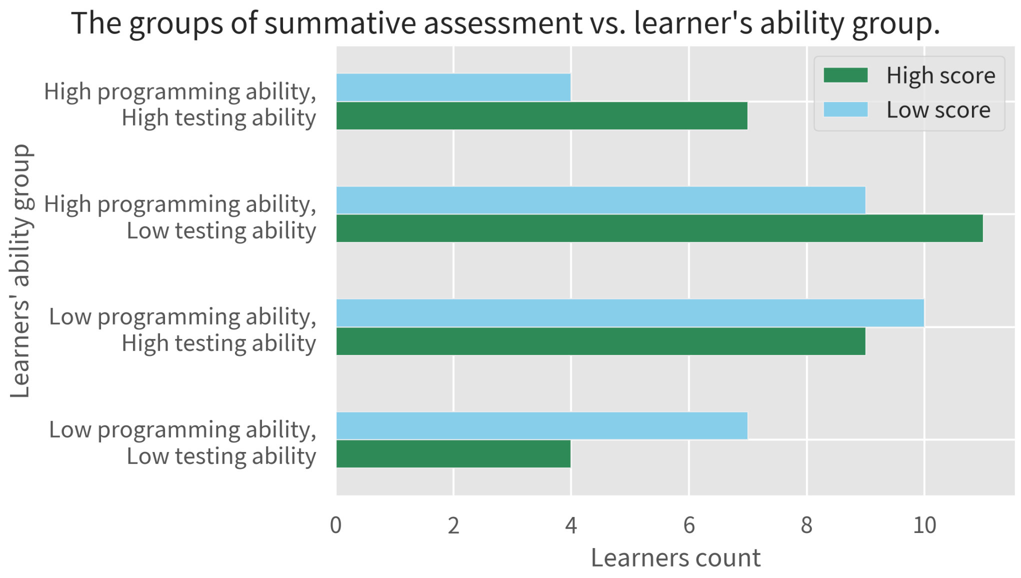


Figure 15. The learners are categorized based on the difference in summative-assessment results across four learning-ability categories.

Regarding resilience, the distributions of the two groups are displayed in Figure 16a. The p-value from the Mann–Whitney U test is 0.105, which is greater than 0.05, indicating no significant difference. For the PAI, the distribution is shown in Figure 16b, and the p-value from the test is 0.016, which is less than 0.05, indicating a significant difference. Similarly, for the TEI, the distribution is presented in Figure 16c, and the p-value from the test is 0.93, which is greater than 0.05, also showing no significant difference. In the analysis of the summative assessment, there is no significant difference between r and the TEI, while the PAI shows a significant difference. We speculate that due to the limited time available, learners may not be able to design higher-quality test cases to thoroughly evaluate their code. Therefore, learners with a superior PAI have a competitive advantage.

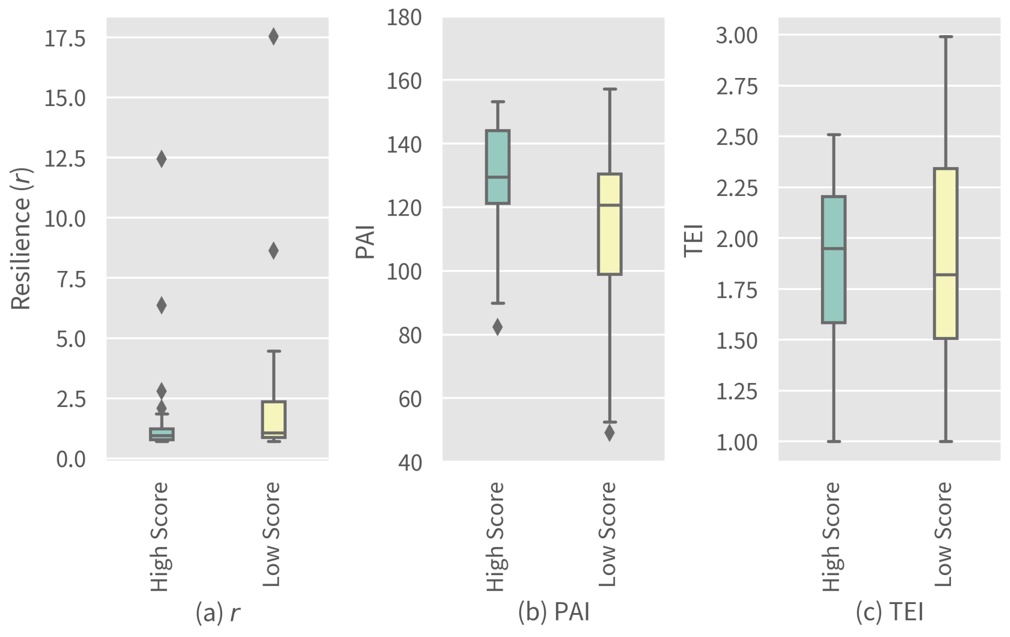


Figure 16. The box-and-whisker plot of indicators for summative assessments. Summative assessment emphasizes conceptual understanding rather than actual coding. This results in students with high scores not necessarily having high TEI.

We analyzed the differences in programming and testing abilities among the learners based on their majors, formative assessment, and summative assessment. The experimental results indicate that both programming and testing abilities influence learning outcomes. The learners' academic backgrounds to some extent affect their programming abilities. However, many learners with lower programming abilities also exhibit high testing abilities and achieve good results. We recommend that instructors of introductory programming courses consider incorporating software-testing concepts into their teaching to help learners write better programs.

5.3. Rq3: Is There a Relationship between Ability Indicators and Learning Behavior?

To analyze the relationships between the performance indicators, we conducted a Spearman's rank correlation coefficient analysis, as depicted in Figure 17. The correlation coefficient between the learners' PAI and r is -0.863, indicating a strong negative correlation. The correlation between the PAI and TEI is -0.385, indicating a moderate negative correlation. Conversely, there is a moderate positive correlation between r and the TEI. In our defined indicators, a higher r corresponds to a lower PAI, as learners with higher programming abilities are expected to encounter fewer setbacks. In the relationship between the PAI and TEI, it is possible that learners have yet to acquire knowledge related to software testing, and for many questions, high scores can be achieved without comprehensive testing. Therefore, the impact of the PAI on scores becomes more significant.

The moderate positive correlation between r and the TEI suggests that as the frequency of program errors increases, learners are more likely to conduct multiple rounds of testing to debug their code.

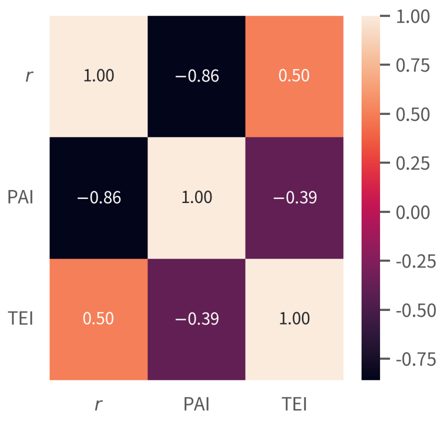


Figure 17. The correlation heatmap of the ability indicators.

In addition, we also investigated the correlations between ability indicators and learning behaviors, as well as between ability indicators and learning outcomes, as shown in Figure 18. The correlation coefficient between the indicator r and video playback speed is 0.203, suggesting that learners might struggle to grasp concepts while watching course videos at high speeds, making them more likely to encounter errors when writing code. The correlation coefficient between r and the video-completion rate is 0.111, indicating a near lack of correlation. The indicator PAI exhibits a negative correlation with the video playback speed, implying that learners may tend to quickly finish watching videos and proceed to the assignment phase, thus increasing the likelihood of errors. The PAI also shows almost no correlation with the video-completion rate. The indicator TEI has a correlation coefficient of 0.237 with the video playback speed, indicating a low positive correlation. Learners may aim to enter the assignment phase quickly, leading them to conduct multiple tests to ensure the correctness of their code. The TEI's correlation coefficient with the video-completion rate is 0.331, displaying a moderate positive correlation. Learners who watch more videos tend to perform more tests to ensure the accuracy of their code.

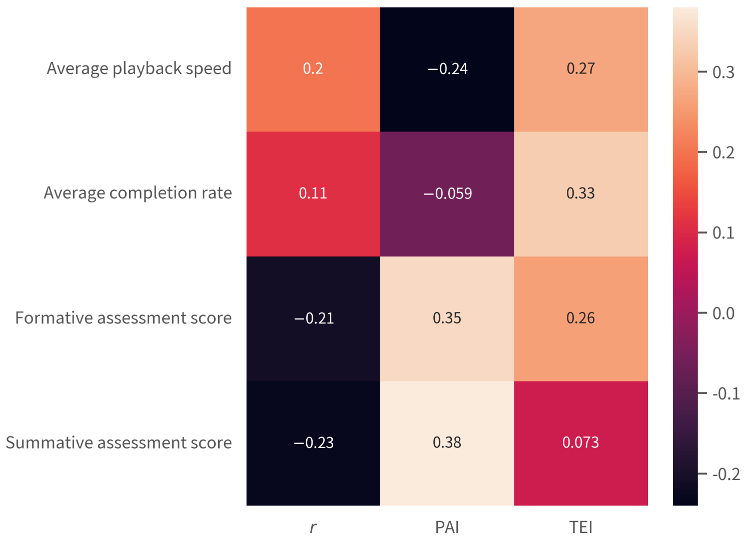


Figure 18. The heatmap that illustrates the correlation between ability indicators and learning behaviors.

In the analysis of the correlation between learning outcomes and ability indicators, *r* exhibits low negative correlations with both formative-assessment scores and summative-assessment scores. The PAI shows a moderate positive correlation with both formative-assessment scores and summative-assessment scores, suggesting a certain degree of a relationship between programming ability and learning outcomes. As for the TEI, it shows a low positive correlation with formative-assessment scores, indicating that testing ability has some impact on achieving higher scores during the learning process. However, the TEI displays no correlation with summative-assessment scores, suggesting that programming ability becomes the dominant factor in achieving high scores under time pressure.

From the above analysis, it is evident that learners' programming abilities are correlated with their programming-exercise scores. However, the playback speed at which learners watch videos shows a negative correlation with programming abilities. This suggests that learners may tend to quickly complete video viewing and move on to the exercise phase. This behavior may lead to the incomplete acquisition of course knowledge and an increased likelihood of errors. On the other hand, programming abilities show a positive correlation with both formative-assessment and summative-assessment scores. Additionally, testing abilities also exhibit a positive correlation with formative-assessment scores. Therefore, testing abilities can help learners achieve a better performance during their learning process. Consequently, emphasizing the importance of course materials and the concept of software testing in the course can influence learning outcomes.

5.4. Comparison

There have been many excellent studies in the field of programming education in the past. Compared with their studies, we focus more on the assessment of students' programming and testing abilities. Suleman introduced an automated assessment system to address the challenges of code compilation, execution, and testing commonly encountered in introductory computer science classes. Despite the need for improvements highlighted in learners' feedback, it underscores the necessity of automating code assessments [17]. Sun developed a system named YOJ to support the teaching of introductory computer science courses. This system enables students to practice programming independently and assists instructors in assigning homework related to key programming concepts. Furthermore, the system can be adapted for use in other courses [16]. Within the Pytutor system, we created an environment that allows instructors to design questions of varying levels of difficulty. To tackle questions of different complexities, learners are required to use a larger number of test cases to evaluate their code and meet the requirements of the questions. Therefore, we developed a feature that enables learners to add Defining Test Cases to assess whether their code meets the specified criteria.

Brito emphasized that hands-on practice is one of the means to solidify programming skills, but issues such as an inadequate number of practice problems and excessive time spent evaluating student-submitted solutions needed to be addressed. Therefore, they implemented an online platform to serve as a repository for problems and to automatically assess student-submitted code [20]. Haynes-Magyar mentioned that learning programming requires the development of at least four skills: code reading and comprehension, code writing, pattern understanding, and pattern application. Developing these skills necessitates specialized task-oriented practice to enhance specific abilities. A web-based interactive programming-practice environment was designed to empower learners to decide what and when to learn. Consequently, they created a self-directed learning environment named Codespec. It provides learners with exercises on the same problems but with different solution approaches to enhance the cultivation of various skills [21]. In the Pytutor, instructors have the capability to add sample code when designing questions. This provides significant flexibility for instructors to design questions according to the specific learning approaches they wish to employ. For example, if an instructor wants to design a coding exercise that focuses on debugging skills, they can insert erroneous code into the default code, allowing learners to identify the faulty sections of the code through error messages or test cases.

Yang et al.'s study also emphasized the assessment of students' programming/testing abilities. Yang et al. conducted a study with the objective of assessing the programming and testing abilities though indicators such as students' programming proficiency, code maintainability, and coding efficiency. Their methodology involved the creation of three programming assignments, the utilization of an automated assessment tool for computing the indicators of abilities, and the application of the Pearson correlation coefficient to examine the relationship between these abilities. The study's participants were students with prior programming experience and knowledge of software-testing concepts. Notably, Yang et al. found that strong programming skills do not guarantee strong testing abilities, and conversely, individuals with weaker programming skills can excel in testing capabilities. Compared with their research, the similarities and differences are summarized as follows:

- Yang et al. used three programming tasks to assess students' abilities. On Pytutor, we provided 50 questions for students to practice various programming concepts.
- In Yang et al.'s research, their participants were undergraduate students with a programming background who had also studied concepts related to software testing. Our course design focuses on fundamental Python programming concepts, so most of the participating learners do not have a background in urban or software testing.
- Pytutor enables students to practice programming questions at any time, offering them more opportunities to demonstrate their testing abilities when assessing code. In Yang et al.'s experiment, students were required to submit their assignments within a limited timeframe, and the code was assessed through unit tests.
- From our research, we found that programming ability and testing ability are not correlated. However, testing ability can assist learners in achieving better scores on assignments. Yang et al.'s study also mentioned that nearly half of the developers with excellent programming skills have relatively lower testing abilities. Conversely, some developers with lower programming skills also perform well in testing.

5.5. Limitation

In our course design, some programming questions have TQI_O scores exceeding 80, so learners' self-Defining Test Cases (DTCs) do not have much room for improvement, resulting in a negligible increase in the TEI calculation. Additionally, the concept of software testing was not integrated into the course when teaching programming concepts, so the testing ability here reflects learners' sensitivity to test cases. Regarding data collection, the system currently lacks the capability to analyze learners' actual behaviors while answering questions, such as whether learners use local editors to write code and perform tests before submitting them to the system.

6. Conclusions and Future Work

We developed an OJ system with testing functionality called Pytutor and applied it in online programming-education courses. Unlike most traditional OJ systems, Pytutor allows learners to add their own test cases to test their own code. We defined programming ability and testing ability to differentiate learners' capabilities. In the past, OJ systems determined passing based on the number of Hidden Test Cases passed. We defined a scoring system based on the proportion of Hidden Test Cases passed and considered the resilience of learners who persisted in attempting even in the face of errors or incomplete correctness as a parameter for programming ability. Additionally, we designed criteria for testing ability based on the number of test cases, code coverage, mutation testing scores, and program cyclomatic complexity.

We analyzed the ability indices of 61 learners along with their collected learning behaviors and data from the platform. Our findings indicate that learners from computer science majors have better programming abilities than non-CS learners, but there was no significant difference in testing abilities. Having good programming and testing abilities can lead to higher scores during the learning phase, but programming ability is the decisive factor for exam performance.

In our research, we observed that learners with higher testing abilities can better solve programming problems during the learning process. Based on the findings, we recommend that programming-education practitioners consider incorporating software-testing concepts into course design—whether or not learners are computer science majors. This approach allows learners to receive education in both programming fundamentals and software testing, ultimately leading to improved learning outcomes. Although this result may not have a huge impact on the education sector, it can be used as a reference for cultivating students' software engineering abilities.

With the rise of artificial intelligence, the behavior of writing programs will also undergo major changes, which will of course also impact software engineering education [38]. In the future, we plan to develop more interesting topics:

- Using AI as a virtual assistant to assist in program education. How to guide—rather than directly provide answers—will be challenging.
- Developing a more-powerful online editor so students enjoy writing code there, and we can record student behavior in detail. By analyzing detailed programming behaviors, we can analyze more and learn how to further improve programming education.
- Conducting more experiments to understand the learning effects when using our tool. In future courses, we plan to divide students into experimental and control groups and conduct more comparative analyses on them.

Author Contributions: Conceptualization, N.-L.H., L.-C.L. and W.-H.T.; methodology, N.-L.H., L.-C.L. and W.-H.T.; software, N.-L.H., L.-C.L. and W.-H.T.; validation, N.-L.H., L.-C.L. and W.-H.T.; writing—original draft preparation, N.-L.H., L.-C.L. and W.-H.T.; writing—review and editing, N.-L.H., L.-C.L. and W.-H.T. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the National Science and Technology Council, Taiwan R.O.C., under grants MOST112-2813-C-035-075-E.

Data Availability Statement: The data presented in this study are available on request from the corresponding author. The data are not publicly available due to the data contains personal information of students.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Wang, J.Y.; Liang, J.C.; Chang, C.C. The CodingHere Platform for Programming Courses. *Inf. Eng. Express* **2022**, *8*, 1–14.
2. Hidalgo-Céspedes, J.; Marín-Raventós, G.; Calderón-Campos, M.E. Online Judge Support for Programming Teaching. In Proceedings of the 2020 XLVI Latin American Computing Conference (CLEI), Loja, Ecuador, 19–23 October 2020; pp. 522–530.
3. Zinovieva, I.; Artemchuk, V.; Iatsyshyn, A.V.; Popov, O.; Kovach, V.; Iatsyshyn, A.V.; Romanenko, Y.; Radchenko, O. The use of online coding platforms as additional distance tools in programming education. *J. Phys. Conf. Ser.* **2021**, *1840*, 012029.[CrossRef]
4. Pereira, F.D.; Oliveira, E.; Cristea, A.; Fernandes, D.; Silva, L.; Aguiar, G.; Alamri, A.; Alshehri, M. Early dropout prediction for programming courses supported by online judges. In Proceedings of the Artificial Intelligence in Education: 20th International Conference, AIED 2019, Chicago, IL, USA, 25–29 June 2019; pp. 67–72 .
5. Wasik, S.; Antczak, M.; Badura, J.; Laskowski, A.; Sternal, T. A survey on online judge systems and their applications. *ACM Comput. Surv.* **2018**, *51*, 1–34.[CrossRef]
6. Fu, Q.; Bai, X.; Zheng, Y.; Du, R.; Wang, D.; Zhang, T. VisOJ: Real-time visual learning analytics dashboard for online programming judge. *Vis. Comput.* **2022**, *39*, 2393–2405. [CrossRef]
7. Lemos, O.A.L.; Silveira, F.F.; Ferrari, F.C.; Garcia, A. The impact of Software Testing education on code reliability: An empirical assessment. *J. Syst. Softw.* **2018**, *137*, 497–511.[CrossRef]
8. Buffardi, K.; Edwards, S.H. Effective and ineffective software testing behaviors by novice programmers. In Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research, San Diego, CA, USA, 12–14 August 2013; pp. 83–90.
9. Spacco, J.; Fossati, D.; Stamper, J.; Rivers, K. Towards improving programming habits to create better computer science course outcomes. In Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education, Canterbury, UK, 1–3 July 2013; pp. 243–248.
10. Fidge, C.; Hogan, J.; Lister, R. What vs. how: Comparing students' testing and coding skills. In Proceedings of the Conferences in Research and Practice in Information Technology Series 2013, Chicago, IL, USA, 13–16 March 2013; pp. 97–106

11. Yang, P.; Liu, Z.; Xu, J.; Huang, Y.; Pan, Y. An empirical study on the ability relationships between programming and testing. *IEEE Access* **2020**, *8*, 161438–161448.[CrossRef]
12. Edwards, S.H. Improving student performance by evaluating how well students test their own programs. *J. Educ. Resour. Comput.* **2003**, *3*, 1–es. [CrossRef]
13. Ala-Mutka, K.M. A survey of automated assessment approaches for programming assignments. *Comput. Sci. Educ.* **2005**, *15*, 83–102.[CrossRef]
14. Cerioli, M.; Cinelli, P. GRASP: Grading and Rating ASsistant Professor. In Proceedings of the ACM-IFIP 2008, Leuven, Belgium, 1–4 December 2008; pp. 37–51.
15. Fraser, G.; Gambi, A.; Kreis, M.; Rojas, J.M. Gamifying a software testing course with code defenders. In Proceedings of the 50th ACM Technical Symposium on Computer Science Education, Minneapolis, MN, USA, 27 February–2 March 2019; pp. 571–577.
16. Sun, H.; Li, B.; Jiao, M. YOJ: An online judge system designed for programming courses. In Proceedings of the 2014 9th International Conference on Computer Science & Education, Vancouver, BC, Canada, 22–24 August 2014; pp. 812–816.
17. Suleman, H. Automatic marking with Sakai. In Proceedings of the 2008 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on IT Research in Developing Countries: Riding the Wave of Technology, Wilderness, South Africa, 6–8 October 2008; pp. 229–236.
18. Carless, D.; Salter, D.; Yang, M.; Lam, J. Developing sustainable feedback practices. *Stud. High. Educ.* **2011**, *36*, 395–407. [CrossRef]
19. Malmi, L.; Korhonen, A.; Saikkonen, R. Experiences in automatic assessment on mass courses and issues for designing virtual courses. *ACM SIGCSE Bull.* **2002**, *34*, 55–59. [CrossRef]
20. Brito, M.; Gonçalves, C. Codeflex: A web-based platform for competitive programming. In Proceedings of the 2019 14th Iberian Conference on Information Systems and Technologies (CISTI), Coimbra, Portugal, 19–22 June 2019; pp. 1–6.
21. Haynes-Magyar, C.C.; Haynes-Magyar, N.J. Codespec: A Computer Programming Practice Environment. In Proceedings of the 2022 ACM Conference on International Computing Education Research, Virtual, 7–11 August 2022; Volume 2, pp. 32–34. [CrossRef]
22. Xia, Z.; Hu, B.; Diao, W.; Huang, Y. Design of Interactive Computer Algorithm Learning Platform: Taking the visual programming tool "Progressive Blockly" as an example. In Proceedings of the 2021 International Conference on Computer Engineering and Application (ICCEA), Kunming, China, 25–27 June 2021; pp. 189–193.
23. Polito, G.; Temperini, M.; Sterbini, A. 2tsw: Automated assessment of computer programming assignments, in a gamified web based system. In Proceedings of the 2019 18th International Conference on Information Technology Based Higher Education and Training (ITHET), Magdeburg, Germany, 26–27 September 2019; pp. 1–9.
24. Swacha, J. State of research on gamification in education: A bibliometric survey. *Educ. Sci.* **2021**, *11*, 69.[CrossRef]
25. Vujošević-Janičić, M.; Nikolić, M.; Tošić, D.; Kuncak, V. Software verification and graph similarity for automated evaluation of students' assignments. *Inf. Softw. Technol.* **2013**, *55*, 1004–1016. [CrossRef]
26. Montoya-Dato, F.J.; Fernández-Alemán, J.L.; García-Mateos, G. An experience on Ada programming using on-line judging. In Proceedings of the Reliable Software Technologies—Ada-Europe 2009: 14th Ada-Europe International Conference, Brest, France, 8–12 June 2009; pp. 75–89.
27. Jiang, W.; Deng, J. Design and Implementation of On-Line Practice System Based on Software Testing. *J. Phys. Conf. Ser.* **2021**, *1738*, 012115. [CrossRef]
28. Kasahara, R.; Sakamoto, K.; Washizaki, H.; Fukazawa, Y. Applying gamification to motivate students to write high-quality code in programming assignments. In Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education, Aberdeen, UK, 15–17 July 2019; pp. 92–98.
29. García-Magariño, I.; Pita, I.; Arroyo, J.; Fernández, M.L.; Bravo-Agapito, J.; Segura, C.; Gilaberte, R.L. UnitJudge: A novel online automatic correction system for long programming practices by means of unit tests. In Proceedings of the 2023 10th International and the 16th National Conference on E-Learning and E-Teaching (ICeLeT), Tehran, Iran, 28 February–2 March 2023; pp. 1–5.
30. Ihantola, P.; Ahoniemi, T.; Karavirta, V.; Seppälä, O. Review of Recent Systems for Automatic Assessment of Programming Assignments. In Proceedings of the 10th Koli Calling International Conference on Computing Education Research, Koli, Finland, 28–31 October 2010; pp. 86–93. [CrossRef]
31. Kuo, J.Y.; Wen, Z.J.; Hsieh, T.F.; Huang, H.X. A Study on the Security of Online Judge System Applied Sandbox Technology. *Electronics* **2023**, *12*, 3018.[CrossRef]
32. Paiva, J.C.; Leal, J.P.; Figueira, Á. Automated assessment in computer science education: A state-of-the-art review. *ACM Trans. Comput. Educ.* **2022**, *22*, 1–40.[CrossRef]
33. Peveler, M.; Maicus, E.; Cutler, B. Automated and manual grading of web-based assignments. In Proceedings of the 51st ACM Technical Symposium on Computer Science Education, Portland, OR, USA, 11–14 March 2020; p. 1373.
34. Xu, B.; Yan, S.; Jiang, X.; Feng, S. SCFH: A student analysis model to identify students' programming levels in online judge systems. *Symmetry* **2020**, *12*, 601.[CrossRef]
35. Shepperd, M. A critique of cyclomatic complexity as a software metric. *Softw. Eng. J.* **1988**, *3*, 30–36.
36. Pereira, F.D.; Oliveira, E.H.; Oliveira, D.B.; Cristea, A.I.; Carvalho, L.S.; Fonseca, S.C.; Toda, A.; Isotani, S. Using learning analytics in the Amazonas: Understanding students' behaviour in introductory programming. *Br. J. Educ. Technol.* **2020**, *51*, 955–972.[CrossRef]

37. Kochhar, P.S.; Thung, F.; Lo, D. Code coverage and test suite effectiveness: Empirical study with real bugs in large systems. In Proceedings of the 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Montreal, QC, Canada, 2–6 March 2015; pp. 560–564.
38. Welsh, M. The end of programming. *Commun. ACM* **2022**, *66*, 34–35.[CrossRef]