Received July 30, 2020, accepted August 19, 2020, date of publication August 24, 2020, date of current version September 16, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3018718

An Empirical Study on the Ability Relationships Between Programming and Testing

PENG YANG[1], ZIXI LIU[2,3], JIN XU[2], YONG HUANG[2], AND YA PAN[3], (Member, IEEE)

[1]School of Information Engineering, Guangzhou Panyu Polytechnic, Guangzhou 511483, China
[2]Mooctest Inc., Nanjing 210000, China
[3]School of Computer Science and Technology, Southwest University of Science and Technology, Mianyang 621002, China

Corresponding author: Ya Pan (panya@swust.edu.cn)

This work was supported in part by the National Key Research and Development Program of China under Grant 2018YFB1403400, in part by the National Natural Science Foundation of China under Grant 61690201, and Grant 61772014, in part by the Science and Technology Project of Guangzhou under Grant 201904010220, and in part by the Project of Technology Development Foundation of Guangdong under Grant 706049150203.

ABSTRACT Under the software quality management mechanism, developers are generally required to review and test their own code firstly to ensure that the submitted code meets specific quality standards. At the same time, with the popularity of test-driven development (TDD) and extreme programming (XP), programming and testing are complementary in the process of software development, i.e., software testing has become as important as programming. Despite its importance, there is no empirical study that investigates the ability relationships between programming and testing. This article presents such a study, where we designed software tasks to investigate the ability of programming and testing. We distributed the program tasks to software vocational students and analyzed the results from multiple dimensions. Our main findings show that (i) almost half of the developers with strong programming ability do not have a good testing ability; (ii) some developers with weak programming ability can do well in testing; (iii) compared with programming ability, testing fundamentals have a greater impact on the testing ability; and (iv) most developers can do well at finding bugs but lack experience in writing test scripts.

INDEX TERMS Software testing, programming ability, testing ability.

I. INTRODUCTION

Software testing has become a significant process to ensure the quality of software systems [1]. In fact, there are corresponding testing tasks at each coding stage [2]. For example, unit testing is a correctness testing for program modules (i.e., the smallest unit of software design) [3]. Testing can help developers check for defects in the production code and ensure the software system is as robust as possible under different using conditions [2], [4], [5]. However, developers seem to be unable to complete unit-testing well [6], and many developers do not realize the importance of testing [7]–[9], resulting in the difficulty of software maintenance.

A high-quality software testing process plays a vital role in software quality assurance. With the development of test-driven development (i.e., TDD) [10], unit testing can help developers to understand the specific needs of software development and increase developer confidence in code changes [10]. This extends the benefits of testing to include faster refactoring of production code or description of production requirements. In addition, in almost all software development models [11], developers are required to write unit test scripts of their own production code and achieve a specific coverage. This is because programmers are familiar with the structure and functions of their production code, while other testers who are not familiar with the code cannot write test scripts efficiently. Therefore, it can be reasonably expected that there is a correlation between the programming ability of the developer and the testing ability.

Various studies have investigated the quality of the production code or test code. Many focus on discussing different software indicators and how these indicators affect software quality and reliability. Examples include building code quality models, building metrics suits for projects, assessment of test cases, etc. [12]–[14]. The topic most relevant to our research is the quality of production and test code. Some

The associate editor coordinating the review of this manuscript and approving it for publication was Zhaojun Li.

works measure the quality of test code based on the readability, mutation score, code coverage, and code/test smell. These studies calculate the indicators to mine the factors that influence code quality. Although there are currently some methods for evaluating the quality of code, no research has been conducted to link the quality of production code and test code and study the potential relationship between them. Since developers need to complete some testing tasks, such as unit testing [2], and testers often need to have a particular programming foundation, it is meaningful to study the programming and testing abilities of developers.

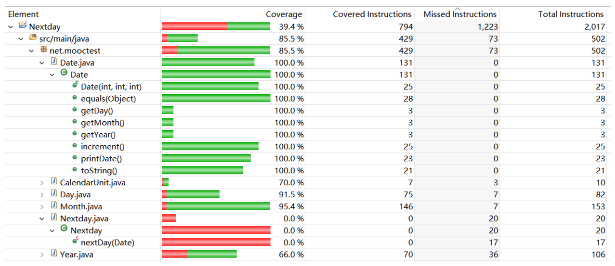
In this article, we design two programming tasks, which are used to evaluate the programming and testing abilities separately. We choose a class of vocational students in the software testing direction as the testers. They all have a particular programming foundation and master the object-oriented programming method. Besides, they have learned the basics of software testing and automated testing. We analyze the programming and testing abilities of students by collecting their coding information and calculating a set of indicators. We evaluate students' programming level, code maintainability, and coding efficiency from multiple dimensions, and analyze the correlation between programming and testing capabilities based on these indicators.

In summary, we have found the following conclusions in our empirical study. First, the Pearson correlation coefficient of the programming ability and the testing ability is only about 0.3, which indicates that the two abilities are not strongly correlated. Additionally, more than half of developers with good programming ability have a good testing ability, while a few developers with good testing ability have good programming ability. Second, the developers with intermediate programming ability do best at writing readable test code and have a particular ability to detect bugs. However, most of them lack experience in designing test cases, which leads to low running efficiency. Third, the impact of test foundation on testing ability is more significant than the effect of programming ability. Even if developers have strong programming ability, their testing ability is generally weak due to the lack of a testing foundation.

The remainder of this article is structured as follows. In Section II, we use two examples to illustrate our research motivation. In Section III the foundations for this study are presented. Section IV illustrates our study and methodology in detail. The results of our study are reported in Section V, and Section VI discusses the related work. Finally, Section VII concludes the paper and elaborates on our future work.

II. MOTIVATING EXAMPLE

Both programming ability and testing ability are closely related to coding. In some cases, the obstacles encountered when writing test scripts are also inevitable when programming. Meanwhile, in some cases, developers who do not have strong programming ability can also write test scripts well.



Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
Nextday	39.4 %	794	1,223	2,017
src/main/java	85.5 %	429	73	502
net.mooctest	85.5 %	429	73	502
Date.java	100.0 %	131	0	131
Date	100.0 %	131	0	131
Date(int, int, int)	100.0 %	25	0	25
equals(Object)	100.0 %	28	0	28
getDay()	100.0 %	3	0	3
getMonth()	100.0 %	3	0	3
getYear()	100.0 %	3	0	3
increment()	100.0 %	25	0	25
printDate()	100.0 %	23	0	23
toString()	100.0 %	21	0	21
CalendarUnit.java	70.0 %	7	3	10
Day.java	91.5 %	75	7	82
Month.java	95.4 %	146	7	153
Nextday.java	0.0 %	0	20	20
Nextday	0.0 %	0	20	20
nextDay(Date)	0.0 %	0	17	17
Year.java	66.0 %	70	36	106

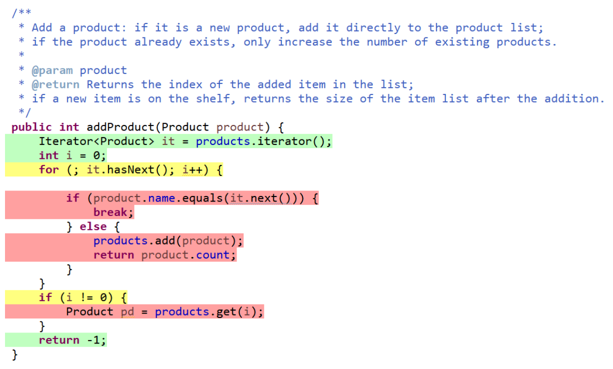
FIGURE 1. The code coverage rate of a testing task.

```
public  static  Date nextDay(Date d) {
    Date dd = new Date(d.getMonth().getCurrentPos(),
         d.getDay().getCurrentPos(), d.getYear().
         getCurrentPos());
    dd.increment();
    return dd;
}
```

LISTING 1. The nextDay function.

Figure 1 is the detailed code coverage rate of a testing task submitted by a vocational student. Since the testing task is required to write the unit test code while the production code has already been written, we focus on the code coverage rate of the production code. The code coverage rate of the entire production code is 85%. It is noticeable that most of the classes in production code are covered by 100%, but the class "Nextday" has a coverage of 0. In fact, this class contains only one function, and this function involves a calling relationship of multiple classes, which is shown in List 1. However, the called classes and functions are 100% covered, which indicates that this student knows how to write test cases for each function, but is not familiar with the calling relationship between functions. This student's performance in the programming task confirmed this conclusion. This student completed all the functions without any calling relationship but did not implement a function with a calling relationship. This real example shows that if a developer does not understand the logic of object-oriented programming well, there are obvious flaws in both the production and test code.

In real cases, we found that some developers can complete the testing tasks well, but their programming skills are not outstanding. In programming tasks, a student completed most of the development tasks, but he did not complete the functions related to the loop structure, and the only loop structure he wrote did not meet the requirements of the task, which is shown in Figure 2. In Figure 2, the green part indicates the code is covered, the yellow part suggests that the branch is not fully covered, and the red part shows that the code is not covered, which means that this part of the production code does not meet the requirements. Judging from the completion of the programming task of this student, he is not familiar with writing the loop structure but masters the basic programming language. Although this student is not familiar with the loop structure, it is basically not necessary to use a loop statement in the test script. Thus the test code of this student has



```
/**
 * Add a product: if it is a new product, add it directly to the product list;
 * if the product already exists, only increase the number of existing products.
 *
 * @param product
 * @return Returns the index of the added item in the list;
 * if a new item is on the shelf, returns the size of the item list after the addition.
 */
public int addProduct(Product product) {
    Iterator<Product> it = products.iterator();
    int i = 0;
    for (; it.hasNext(); i++) {

        if (product.name.equals(it.next())) {
            break;
        } else {
            products.add(product);
            return product.count;
        }
    }
    if (i != 0) {
        Product pd = products.get(i);
    }
    return -1;
}
```

FIGURE 2. **The code coverage of a loop structure.**

achieved a high coverage rate. This case shows that even if some developers are not good at programming, as long as they understand the basic syntax, they may have good testing capabilities.

These examples show that developers' programming ability and testing ability have a special relationship, but not a simple linear relationship. Therefore, in this study, we focus on evaluating the relationship between programming and testing ability from multiple perspectives such as coding quality, efficiency, code coverage rate, and bug detection rate.

III. BACKGROUND

A. TEST-DRIVEN DEVELOPMENT

Test-driven development (i.e., TDD) [10] is a new development method different from the traditional software development process. It requires developers to write the test code before writing the production code for a certain function. Then, only the function code that passes the test is written [15]. Therefore, the test cases are used to drive the entire development. This helps developers to write concise and high-quality code and speeds up the development process [16].

This can avoid and find errors as early as possible by bringing test works before coding and running all tests frequently. TDD greatly reduces the cost of subsequent tests and repairs and improves the quality of the source code. Under the protection of testing, the source code is continuously refactored to eliminate duplicate designs, optimize the design structure, improve code reuse, and thus improve the quality of software products [10]. In this article, we apply TDD to evaluate the programming ability.

B. UNIT TESTING

The purpose of unit testing is to discover various errors that may exist within each module, mainly based on white-box testing. It refers to the inspection and verification of the smallest testable unit in software [3]. In general, the unit is the smallest functional module that is artificially specified. The meaning of a unit in the unit testing should be determined according to the actual situation [17] (e.g., a function in C or a class in Java).

Unit testing is different from other tests. Unit testing can be considered as part of the coding work and should be completed by the developers. Meanwhile, unit testing should start as early as possible. In TDD, developers must write all the unit test cases before writing the source code. In the traditional software development process, unit test cases should be written after building the framework of functions. In this article, testers need to finish the unit testing tasks, and we investigate the testing ability by evaluating the unit test cases.

C. MUTATION TESTING

Mutation testing (i.e., mutation analysis) is a form of white-box testing [18]. It is an effective software testing method that improves the source code of a program [19]. These so-called mutations are based on well-defined mutation operations, which either simulate typical application errors (e.g., using the wrong operator or variable name), or force effective testing (e.g., making each expression equal to 0) [20]. Compared to logic testing and path-based testing, mutation testing can reflect the defect detection capabilities of test cases more intuitively. Driven by previous research in the field of software testing, mutation scores are the most critical code coverage criterion and one of the most relevant metrics for developers [19], [21]. In this article, the mutation score is one of the metrics for evaluating the testing ability.

IV. DESIGN OF STUDY

Figure 3 depicts an overview of our approach. First, we design some project tasks for software engineering vocational students. All these tasks are Java projects, and the details are described in Section IV-B. Students must finish the source code or unit test cases in the allotted time. Then, within the limited time, students can submit code multiple times. After each submission, the server will run unit test scripts to update scores based on the coverage rate and other factors in Section IV-C. Finally, we collect and calculate the scores of each indicator to form a radar chart of capabilities.

A. RESEARCH QUESTIONS

The quality of production and test code is related to the influence of multiple factors. The question of these factors arises here: is there a potential correlation between programming and testing abilities if these factors are used to evaluate the ability of a developer? With the increasing pace of software product development, many unit test tasks need to be completed by developers. Prior researchers argue that the quality of unit test scripts written by developers is not high [6]. Therefore, it is necessary to study the relationship between developers' programming ability and testing ability so as to promote the development of software development better.

Due to the normal distribution of students' intelligence skills, including their learning abilities, actual hands-on abilities, etc. [22], we focus on students belonging to intermediate programming levels. By analyzing the distribution of test abilities of this group of students, we can determine their

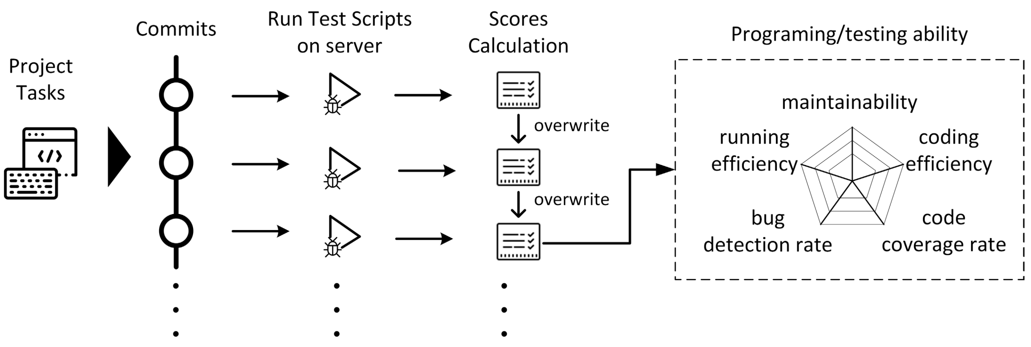


FIGURE 3. **Overview of our approach.**

overall testing ability and determine which skills they are best at or lack.

Further, since there are many differences in the design of the test code and production code, we consider it necessary to refer to the testing fundamentals of developers. Analyzing the impact of programming ability and test fundamentals on testing ability can help testers improve their test level.

To conduct our research, we refine our goal into three different research questions (RQs):

- **RQ1**: Are programming ability and testing ability strongly related?
- **RQ2**: What does a developer with intermediate programming skills do best or worst when writing test scripts?
- **RQ3**: Which ability has a greater impact on testing ability?

We try to answer these questions through a set of experiments. The answers to these questions provide us with an opportunity to better understand the relationship between the programming ability and testing ability of developers in the actual software development field. Additionally, the results can provide us with ideas on how to improve production and testing relationship. Meanwhile, by studying the experimental results, it could help us adjust teaching methods so that students who are good at coding can write better test scripts, and students with weak coding ability can improve their programming thinking during testing.

B. DESIGN OF PROJECT TASKS

1) PROGRAMMING ABILITY TASK

We design the TDD task to evaluate the programming ability of students. We build a Java code framework containing at least five classes, and all the unit test cases are already written. We also list the required functions and write some basic annotations in the source code to help students understand the task precisely. Students must finish the source code, and the more test cases that pass, the higher the score is. When students submit the source code, the server will automatically run unit test scripts and calculate the branch coverage.

In our study, we have designed three programming ability tasks, two of which are object-oriented programming, and one is a pure algorithm implementation. Students need to complete these three tasks as much as possible within three hours. In the algorithm task, there is only one entity class and one implement class containing two functions. The goal of the algorithm task is to implement the first-come-first-served scheduling algorithm and short job priority scheduling algorithm.

Different from the algorithm task, the other two object-oriented tasks have multiple classes and calling relationships. They implement a store management system and a bank simulation system, respectively. In the store management system, requirements include depositing and withdrawing funds, calculating interest for different accounts respectively, outputting a list of customers, and the number of accounts they have and outputting the total interest paid by the bank on all accounts. Similarly, the store management system also has several calling relationships between classes and functions. Overall, the code structure of the store management system is more complicated than the bank simulation system.

Whenever students submit the source code, we automatically run unit test scripts on the server and calculate the pass rate of test cases and code maintainability. For the specific calculation method, refer to the subsections IV-C1 and IV-C3.

2) TESTING ABILITY TASK

Different from the programming ability task, the source code in testing ability task is provided, and students are required to write the unit test scripts. The difficulty with this type of task is that the source code is complicated (i.e., the number of classes is more than five). If students cannot understand the function of the source code, the unit test coverage will not be high.

In our study, we have designed three testing ability tasks according to the difficulty level. Students need to complete these three tasks as much as possible within three hours. The first task only contains one class and four functions in it, which implements checking if the object is a triangle,

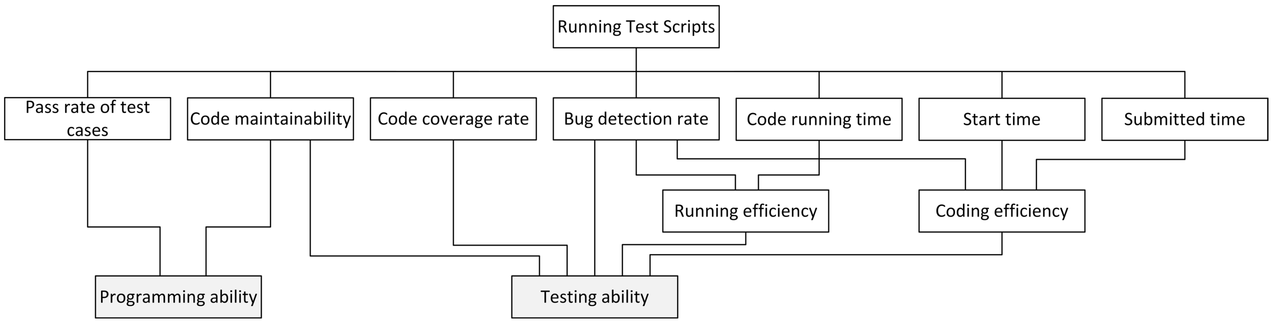


FIGURE 4. **The indicators for programming and testing ability evaluation.**

checking the type of triangle, calculating the diff between borders, and obtaining the length of borders, respectively. Compared with the first task, the other two tasks are more complicated. The second task implements the calculation of dates containing five classes, one of which is an interface class. The structure of the third task is the most completed containing nine classes and a great number of calling relationships between functions.

Whenever students submit the source code, we automatically run unit test scripts on the server and calculate the code coverage rate, code maintainability, bug detection rate, running efficiency, and coding efficiency. For the specific calculation method, refer to subsections IV-C2, IV-C3, IV-C4, IV-C5 and IV-C6.

C. PROGRAMMING AND TESTING ABILITY FACTORS

In order to evaluate the capabilities of developers, we mainly evaluate six code-related indicators, *i.e., Pass rate of test cases, Code coverage rate, Code maintainability, Bug detection rate, Running efficiency, and Coding efficiency.* On the one hand, we focus on evaluating developers' programming capabilities by calculating the pass rate of test cases. Since in the programming ability task, the unit test code has already been written, and these test codes reflect all the coding requirements. Therefore, the more test cases passed, the more production code is completed, which indicates that the programming ability is stronger. On the other hand, we evaluate the testing ability by collecting the other five factors. We assess the quality of the test code itself through code coverage rate, code maintainability, and bug detection rate. Meanwhile, we evaluate testers' testing ability and proficiency through running efficiency and coding efficiency. The overall indicators for programming and testing ability evaluation are depicted in Figure 4.

1) PASS RATE OF TEST CASES

In general, each test case of a unit test corresponds to three results, i.e., pass, error, and failure. The passed test cases indicate that the real result of the program is running as expected, the error results indicate that the test code cannot run normally, and the failure results indicate that the running result is inconsistent with the expected. If the unit test case design is accurate, but the test case fails, it means that the production code does not meet the software design requirements. In the programming tasks, we provide all accurate test cases that reflect development requirements. Thus, failed test cases indicate that the developer does not complete the corresponding coding requirements.

We use the Junit framework[1] to write and run test code. Then, the pass rate of test cases can be expressed as the ratio of the number of passed test cases $Num(passedCases)$ to the total number of test cases $Num(testCases)$.

$$PassRate = \frac{Num(passedCases)}{Num(testCases)} * 100\% \tag{1}$$

2) CODE COVERAGE RATE

Code coverage is the most commonly used indicator for testing code quality assessment [4]. The code coverage rate describes the amount of production code covered during test case execution. In particular, statement coverage refers to which statements in the source code are executed, branch coverage refers to which branches are executed, and path coverage describes the paths traversed in the control flow of the program. In our study, we use the branch coverage rate to calculate the code coverage rate.

We use the JCov plugin[2] to run students' code, which can convert the tested code and the test code into meta nodes and catch nodes, respectively. Then, the code coverage rate can be represented as the ratio of the number of catch nodes $Num(catchNode)$ to the number of meta nodes $Num(metaNode)$.

$$CoverageRate = \frac{Num(catchNode)}{Num(metaNode)} * 100\% \tag{2}$$

3) CODE MAINTAINABILITY

The test code with good quality should be easy for testers to understand, and when source code functionality changes, test code should be easy to refactor [23]. Both the bad code smells, and the bad test smells have a negative impact on the maintainability of the code [24]. Thus, we consider using the

[1]https://junit.org/junit5/
[2]https://hg.openjdk.java.net/code-tools/jcov/

standardization of code as the basis for calculating maintainability.

To check the standardization of the code, we use the Checkstyle plugin[3] and the Google's coding standards for source code in the Java[4] for code inspection. Suppose the vector V represents the feature of code specification, each dimension of which is expressed as a check item of the code specification. V_{size} represents the total number of code specific check items that are violated. The value of each dimension in the vector indicates the number of times the user violated this code specification, or 0 if not. V_{max} represents the maximum value of each code check violation for all user datasets. Therefore, the loss score for the code under testing can be expressed as:

$$Loss_i = V_i^T * V_{max} * \frac{100}{V_{size}}$$

Since the scale of code in each task is different, it is not appropriate to calculate the code maintainability directly. Thus, we divide the highest bug detection rate $BugDetectionRate_i$ (refer to subsection IV-C4) by the loss score $Loss_i$, and the player with the highest score C_{max} is counted as 100 points. Finally, the code maintainability scores of other students are calculated by linearization and normalization. The calculation formula of code maintainability is as follows:

$$C_i = \frac{BugDetectionRate_i}{Loss_i}$$
$$CodeMaintainability = \frac{C_i}{C_{max}} * 100\% \tag{3}$$

4) BUG DETECTION RATE

In terms of the ability to detect actual bugs, mutation testing is one of the best predictors of test case quality. Many studies have shown that mutation test scores reflect the quality of test code well [21].

Since the mutation testing can reflect the defect detection capabilities of test cases intuitively, we use the mutation factor $mutation_i$ to calculate the bug detection rate. As mentioned in subsection IV-B2, the difficulty of each task is quite different. It is not accurate to calculate the scores without considering the difficulty of each task. Thus, we regard the maximum mutation kill rate $mutation_{max}$ among all testers as 100 points in each task. The bug detection rate can be calculated as follows:

$$BugDetectionRate = \frac{mutation_i}{mutation_{max}} * 100\% \tag{4}$$

5) RUNNING EFFICIENCY

For test scripts, the efficiency of the code can indicate how efficient the test case is. Additionally, the running efficiency can reflect the reliability of test code [25]. In existing research [12], [25], the reliability of software products is measured by the meantime to failure, and the meantime to repair. In our study, the server runs the code submitted by testers and will stop immediately if it encounters a failed test case, so we cannot calculate the time interval between successive failures. Thus, we use the time it takes to run all test cases to measure the efficiency of the test code.

To calculate the running efficiency, we first use the ratio of the bug detection rate $BugDetectionRate_i$ to the running time of the code $RunDuration_i$. Since the running efficiency of code is related to the structural complexity of source code, we regard the maximum running efficiency score F_{max} as 100 points. The running efficiency can be calculated as follows:

$$F_i = \frac{BugDetectionRate_i}{RunDuration_i}$$
$$RunningEfficiency = \frac{F_i}{F_{max}} * 100\% \tag{5}$$

6) CODING EFFICIENCY

For both developers and testers, coding efficiency is an essential factor to reflect their ability. Efficient coding can adapt to rapid iterative updates of today's software products. Process metrics, including the time to produce the product, describe the effectiveness and quality of the software products' process [25]. Thus, we consider the time it takes a tester to write test cases as an indicator of his coding efficiency.

To assess the coding efficiency, we calculate the ratio of the bug detection rate $BugDetectionRate$ to the fastest writing time that reaches this rate. The writing time can be calculated as the time lag between the start time of the task $StartTime$ and the time for tester i to reach his highest bug detection rate $RunTime_i$. We regard the maximum coding efficiency score G_{max} as 100 points to eliminate the difference among tasks. The coding efficiency can be calculated as follows:

$$G_i = \frac{BugDetectionRate_i}{RunTime_i - StartTime}$$
$$CodingEfficiency = \frac{G_i}{G_{max}} * 100\% \tag{6}$$

V. STUDY RESULTS

In this section, we present the results of our study and answer each RQ separately.

A. RQ1: ARE PROGRAMMING AND TESTING ABILITY STRONGLY RELATED?

To reflect the overall level of students' programming and testing ability intuitively, the programming ability score is expressed as the average of code coverage rate for three programming tasks(i.e., subsection IV-B1). The testing ability score is expressed as the average of code coverage rate, code maintainability, bug detection rate, running efficiency, and coding efficiency for three testing tasks (i.e., subsection IV-B2).

In our study, 26 students have valid scores in both programming and testing tasks. The Pearson correlation coefficient [26] of students' programming ability and testing ability

[3]https://checkstyle.sourceforge.io/

[4]http://google.github.io/styleguide/javaguide.html

score is 0.3752, which indicates that there is a slight positive correlation between them, but the correlation is not obvious. In Figure 5, each column represents a student's performance, with the student's name label on the X-axis hidden. The green bar represents the student's programming score, and the blue bar represents the student's testing score. From left to right, the student's programming scores are arranged in descending order.

As mentioned in Section IV-B1, two of the three programming tasks are object-oriented projects, and one is an algorithm task. Since all the testers in our research are vocational students in software testing and have not studied algorithms, all of them have 0 points in the algorithm task. Thus, the highest mark on the programming ability task is only 67 points, which is equivalent to two object-oriented questions with full marks.

As can be seen from the Figure 5, of the four students who ranked first in programming ability, only two of them have high testing scores, while the other two have only intermediate testing scores. Judging from the testing scores, two of the three highest-ranked students did not have outstanding programming scores. More than half of the students with high testing scores do not have high programming scores and even have poor programming scores. In general, students with good programming skills have intermediate or high testing ability, while there is a big difference in programming ability among students with good testing ability.

Answer to RQ1: Our study results show that programming and testing ability is not strongly related. Most developers with strong programming ability have good or medium testing ability. However, among the developers with good testability, the programming ability is from strong to weak.

B. RQ2: WHAT DOES A DEVELOPER WITH INTERMEDIATE PROGRAMMING SKILLS DO BEST OR WORST WHEN WRITING TEST SCRIPTS?

In our study, students' programming ability scores are normally distributed, and most of them received 34 points. Thus, we mainly study this group of students, which accounted for 60% of the total students. Since testing scores can only represent comprehensive testing abilities, we analyze five indicators in detail, as shown in Table 1. In order to better visualize the data, we make a radar chart of the five testing ability indicators of each student and superimpose them, as shown in Figure 6.

As can be concluded from Table 1 and Figure 6, among the five indicators, students' code maintainability scores are generally high, while the coding efficiency is generally low. Furthermore, compared to the code coverage rate and running efficiency, the bug detection rate is relatively high. Since these students have programming experience and the code specifications are relatively easy to control, the code

TABLE 1. The details of five testing ability indicators.

	max	min	median	average
Code coverage rate	99.66	0	23.81	34.2
Coding efficiency	53.47	0.08	1.84	16.52
Bug detection rate	99.31	17.3	21.71	39.66
Running efficiency	95.2	16.88	20.53	38.74
Code maintainability	100	18.98	89.8	71.73

maintainability score is the highest among the five factors. These students' bug detection rate is relatively high, mainly because they can use test assertions correctly and master test methods such as boundary testing. It can be seen from the lowest coding efficiency that these students are not proficient at writing test code, resulting in the bug detection rate not being directly proportional to the code runtime. Among these five factors, the score of running efficiency and the code coverage rate is at a medium level, suggesting that students are not familiar with writing test scripts. The lowest coding efficiency score also confirms this conclusion.

Answer to RQ2: Our study results show that the students with intermediate programming skills do best at writing standard code, and they have a particular ability to detect bugs. Specifically, they understand how to use test assertions and master some test techniques such as boundary value testing. However, these students are not familiar with designing test cases, resulting in low test coverage rate and extremely low code running efficiency.

C. RQ3: WHICH ABILITY HAS A GREATER IMPACT ON TESTING ABILITY?

Although the process of writing test scripts is similar to programming, the difference between programming and testing is that testers need to understand the basics related to testing methods, e.g., designing test cases and guidelines for testing coverage. To evaluate the testing fundamentals of students, we obtain their scores of the software testing foundation course. In this course, students are taught software testing methods, classification of testing methods, and how to design test cases. The exam content of this course includes multiple-choice questions and true-false judgments about testing fundamentals(e.g., unit testing, dynamic and static testing, black-box testing, white-box testing, and test case design methods). There are also two design questions in the exam, which are designing test cases for black-box testing and white-box testing, respectively.

To study the impact of testing fundamentals and programming ability on testability, we present the code coverage rate of the programming task as programming ability, and the testing ability is represented as the average of code coverage rate and bug detection rate. We rank the testing ability score from the highest to the lowest and determine that the testability of the first half is high, and the testability of the latter half is low. Then, for the score of testing fundamentals and programming ability, we divide them into three levels of A,

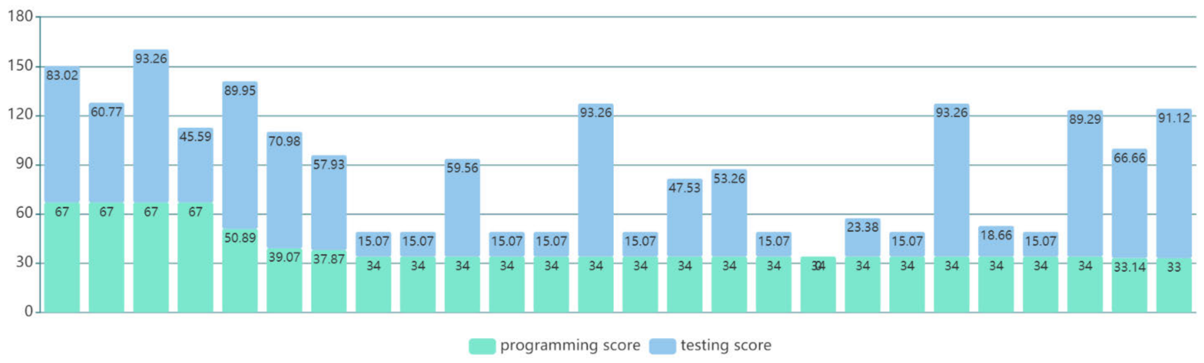


FIGURE 5. The programming and testing scores of all the students in our study.

TABLE 2. The details of software testing foundation exam.

Basics	Description
Logical coverage methods	Judgment method of six logical coverage types including statement coverage, decision coverage (i.e., branch coverage), condition coverage, decision-condition coverage, condition combination coverage, and path coverage.
Designing test cases	Specific analysis methods for equivalence class division, boundary value analysis, causality diagram analysis decision table analysis, and scene analysis.
Static testing and dynamic testing	The difference between dynamic and static testing, and the criteria of these testing method.
White-box testing and black-box testing	The use cases and classification basis of white-box and black-box testing. In addition, the test case design for these two test methods.

B, and C according to the performance ranking, accounting for 30%, 60%, and 10%, respectively. The results are shown in Table 3.

TABLE 3. The impact of testing fundamentals and programming ability on test ability.

Testing ability: high		**Testing ability: low**	
Testing fundamentals	Programming ability	Testing fundamentals	Programming ability
A	C	B	B
A	B	B	B
A	B	A	**B**
A	B	B	B
A	C	B	B
B	**A**	B	B
B	**A**	**C**	B
B	C	**C**	A
A	B	B	B
B	**A**	**B**	A
B	B	A	**B**
A	C	**C**	B
B	B	B	**C**
B	B	B	B
B	**A**	B	B
B	**A**	**C**	B

In our study, we found that among students with good testing ability, proficiency of testing fundamentals is generally more potent than the programming ability. Even some students with poor programming ability achieved high testing scores due to a good grasp of the testing fundamentals. As for students with bad testing ability, many of them have testing fundamentals that are worse than their programming ability.

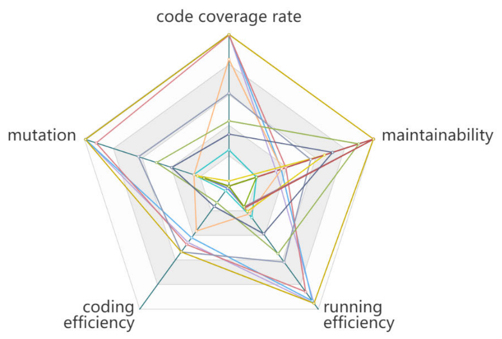


FIGURE 6. The radar chart of five testing ability indicators.

Even if the programming ability is good, if the developer does not understand test methods and fundamentals, the testability will be reduced.

Answer to RQ3: Our study results show that compared to the programming ability, testing fundamentals have a greater impact on the testing ability. Even if developers have a poor programming ability, as long as they master the testing method, the testing ability can be outstanding. Conversely, if a developer does not have a testing foundation, even if the programming ability is strong, the testing ability will be weak.

D. THREATS TO VALIDITY

There is one main threat to validity. The experimental evaluation is based on a small number of participants. However, the participant group in this article is a class of students, which is inherently representative, since there are differences in levels between each student in the class. We also try to recruit more participants to participate in the experiment, but it is difficult to ensure that the participants' programming and testing experience are equivalent. In addition, our current experiment is on Java programs; thus, another condition for us to recruit participants is to be familiar with Java development language. We plan to involve other languages and recruit more participants for evaluation in the future.

VI. RELATED WORK

Software testing is now considered to be an important process for improving the quality of software systems. In the past, research focused on studying the effect of different testing methods, such as unit testing [6], mutation testing [27], and exploratory testing [28]. Our study tries to analyze the relationship between the developer's programming and testing ability based on white-box testing, instead of focusing on the factors that affect the quality of a specific test method. In this section, we compare our study to prior work in the field of test code quality assessment and programming ability evaluation.

A. TEST CODE QUALITY ASSESSMENT

Athanasiou *et al.* [29] aimed at constructing a test code quality model with a set of source code metrics, including completeness, effectiveness, and maintainability. Completeness mainly concerns the complete coverage of the source code. Effectiveness shows the ability of the test code to detect bugs and locate the cause of bugs. Maintainability reflects the ability of test code to adapt to changes in source code, and the extent to which test code can be used as documentation. The experimental results show that the developer's code quality is positively related to some aspects of the issue handling ability (throughput and productivity).

As mentioned previously, our study also aims to measure a set of indicators to evaluate the test code quality. Different from Athanasiou *et al.* [29], our goal is to study the comprehensive programming capabilities of developers, not just the developer's programming efficiency or the ability to fix bugs. Since we design the same programming tasks for testers instead of collecting unrelated open-source projects and calculating metrics, our ability assessment is more fair and reasonable.

Another study in the field of assessing test code is the paper of Grano *et al.* [14]. To evaluate the quality of test cases, they mainly considered dependent metrics, such as mutation score and independent metrics, including code coverage, test smells, code smells, and readability. In addition to the indicators listed above, we consider coding efficiency and running efficiency additionally. The goal of Grano *et al.* [14] was to classify effective and non-effective test cases without running the code. However, we evaluate both the source code and the test code and then analyze the potential relationships between the indicators of the two codes.

B. PROGRAMMING ABILITY EVALUATION

As for assessing the programming ability, there are many studies focusing on how to evaluate and improve students' programming skills in the teaching field, e.g., [30], [31] and [32]. These studies tried to evaluate programming ability by designing exams or program tasks. Different from their concerns, we design test-driven development programming tasks to evaluate the students' programming ability. We also evaluate the level of students' testing fundamentals by considering their scores of software testing foundation course.

Rashid *et al.* [12] presented a comprehensive study on using indicators to assess code quality through indicators with multiple classifications. They compared and analyzed different metrics types, including product quality metrics, in-process quality metrics, and maintenance quality metrics. When evaluating these three types of indicators, multiple technical methods were used to evaluate the relevant indicators and the correlation between the indicators. The experimental results showed that these indicators are effective at detecting the status and quality of the project. In our study, we selected relevant indicators as metrics for the evaluation of programming and testing ability. Different from Rashid *et al.* [12], we mainly explore the relationship between capabilities instead of between indicators.

In addition to the related work mentioned above, there are a few other studies on factors affecting testing or programming code, e.g., [7], [13], [33], and [34]. Nevertheless, these papers only studied the indicators that can reflect the source code or testing code and did not compare and analyze the relationship between the two codes at the same time. To the best of our knowledge, we are the first to conduct empirical research on the relationship between developers' programming ability and testing ability.

VII. CONCLUSION AND FUTURE WORK

In this article, we conduct an empirical study on the ability relationships between programming and testing. We selected a class of software testing vocational students as the research testers. To evaluate the programming ability of students, we designed TDD programming tasks and used the code coverage rate to represent the ability. To assess the testing ability of students, we measured the code coverage rate, code maintainability, bug detection rate, running efficiency, and coding efficiency. In addition, the students' scores of software testing foundation course were represented as the level of testing fundamentals.

We analyzed different factors of the source code and test code to discover the potential connection between programming and testing. Our studies show that: (i) the programming ability and testing ability are not strongly related; (ii) developers with intermediate programming ability do best at writing standard code but are not familiar with designing test cases and writing test scripts; (iii) compared to the programming ability, testing fundamentals have a more significant impact on the testing ability.

In the future, we plan to further explore the correlation between programming and testing by comparing programming and testing capabilities. Furthermore, we will investigate other testing methods such as integration testing and automated testing.

ACKNOWLEDGMENT

The authors would like to thank the students of Guangzhou Panyu Polytechnic, who provided assistance during the experimentation process.

REFERENCES

[1] A. Bertolino, "Software testing research: Achievements, challenges, dreams," in *Proc. Future Softw. Eng. (FOSE)*, May 2007, pp. 85–103.

[2] G. J. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing*. Hoboken, NJ, USA: Wiley 2011.

[3] Wikipedia. (Feb. 23, 2020). *Unit Testing*. [Online]. Available: https://en.wikipedia.org/wiki/Unit_testing

[4] P. Ammann and J. Offutt, *Introduction to Software Testing*. Cambridge, U.K.: Cambridge Univ. Press, 2016.

[5] P. Kong, L. Li, J. Gao, K. Liu, T. F. Bissyande, and J. Klein, "Automated testing of Android apps: A systematic literature review," *IEEE Trans. Rel.*, vol. 68, no. 1, pp. 45–66, Mar. 2019.

[6] F. Trautsch and J. Grabowski, "Are there any unit tests? An empirical study on unit testing in open source Python projects," in *Proc. IEEE Int. Conf. Softw. Test., Verification Validation (ICST)*, Mar. 2017, pp. 207–218.

[7] D. Spadini, F. Palomba, A. Zaidman, M. Bruntink, and A. Bacchelli, "On the relation of test smells to software code quality," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Sep. 2018, pp. 1–12.

[8] X. Wang, W. Sun, L. Hu, Y. Zhao, W. E. Wong, and Z. Chen, "Software-testing contests: Observations and lessons learned," *Computer*, vol. 52, no. 10, pp. 61–69, Oct. 2019.

[9] Y. Peng, X. Jin, S. Hongyu, H. Yong, and X. Jianfeng, "Crowdsourced testing ability for mobile apps: A study on mooctest," *Int. J. Performability Eng.*, vol. 15, no. 11, p. 2944, 2019.

[10] K. Beck, *Test-Driven Development: By Example*. Reading, MA, USA: Addison-Wesley, 2003.

[11] N. M. A. Munassar and A. Govardhan, "A comparison between five models of software engineering," *Int. J. Comput. Sci. Issues (IJCSI)*, vol. 7, no. 5, p. 94, 2010.

[12] J. Rashid, T. Mahmood, and M. W. Nisar, "A study on software metrics and its impact on software quality," 2019, *arXiv:1905.12922*. [Online]. Available: http://arxiv.org/abs/1905.12922

[13] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, Jun. 1994.

[14] G. Grano, F. Palomba, and H. C. Gall, "Lightweight assessment of test-case effectiveness using source-code-quality indicators," *IEEE Trans. Softw. Eng.*, early access, Mar. 4, 2019, doi: 10.1109/TSE.2019.2903057.

[15] D. Astels, *Test Driven Development: A Practical Guide*. Upper Saddle River, NJ, USA: Prentice-Hall, 2003.

[16] T. Bhat and N. Nagappan, "Evaluating the efficacy of test-driven development: Industrial case studies," in *Proc. ACM/IEEE Int. Symp. Int. Symp. Empirical Softw. Eng. ISESE*, 2006, pp. 356–363.

[17] P. Runeson, "A survey of unit testing practices," *IEEE Softw.*, vol. 23, no. 4, pp. 22–29, Jul. 2006.

[18] Wikipedia. (Mar. 7, 2020). *Mutation Testing*. [Online]. Available: https://en.wikipedia.org/wiki/Mutation_testing

[19] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Trans. Softw. Eng.*, vol. 37, no. 5, pp. 649–678, Sep. 2011.

[20] D. Schuler and A. Zeller, "Javalanche: Efficient mutation testing for java," in *Proc. 7th joint meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng. Eur. Softw. Eng. Conf. Found. Softw. Eng. Symp. ESEC/FSE*, 2009, pp. 297–298.

[21] J. H. Andrews, L. C. Brand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Proc. 27th Int. Conf. Softw. Eng. ICSE*, 2005, pp. 402–411.

[22] R. N. Bracewell and R. N. Bracewell, *The Fourier Transform and its Applications*, vol. 31999. New York, NY, USA: McGraw-Hill, 1986.

[23] G. Meszaros, *xUnit Test Patterns: Refactoring Test Code*. London, U.K.: Pearson, 2007.

[24] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley, "Are test smells really harmful? An empirical study," *Empirical Softw. Eng.*, vol. 20, no. 4, pp. 1052–1094, Aug. 2015.

[25] G. Kaur and K. Bahl, "Software reliability, metrics, reliability improvement using agile process," *IJISET-Int. J. Innov. Sci., Eng. Technol.*, vol. 1, no. 3, pp. 143–147, 2014.

[26] J. Benesty, J. Chen, Y. Huang, and I. Cohen, "Pearson correlation coefficient," in *Noise Reduction in Speech Processing*. Berlin, Germany: Springer, 2009, pp. 1–4.

[27] F. Hariri, A. Shi, V. Fernando, S. Mahmood, and D. Marinov, "Comparing mutation testing at the levels of source code and compiler intermediate representation," in *Proc. 12th IEEE Conf. Softw. Test., Validation Verification (ICST)*, Apr. 2019, pp. 114–124.

[28] A. N. Ghazi, K. Petersen, E. Bjarnason, and P. Runeson, "Levels of exploration in exploratory testing: From freestyle to fully scripted," *IEEE Access*, vol. 6, pp. 26416–26423, 2018.

[29] D. Athanasiou, A. Nugroho, J. Visser, and A. Zaidman, "Test code quality and its relation to issue handling performance," *IEEE Trans. Softw. Eng.*, vol. 40, no. 11, pp. 1100–1125, Nov. 2014.

[30] D. Parsons, K. Wood, and P. Haden, "What are we doing when we assess programming," in *Proc. 17th Australas. Comput. Edu. Conf. (ACE)*, vol. 27, 2015, p. 30.

[31] C. Daly and J. Waldron, "Assessing the assessment of programming ability," *ACM SIGCSE Bull.*, vol. 36, no. 1, pp. 210–213, 2004.

[32] A. T. Chamillard and K. A. Braun, "Evaluating programming ability in an introductory computer science course," in *Proc. 31st SIGCSE Tech. Symp. Comput. Sci. Edu. SIGCSE*, 2000, pp. 212–216.

[33] L. Montecchi, P. Lollini, and A. Bondavalli, "A template-based methodology for the specification and automated composition of performability models," *IEEE Trans. Rel.*, vol. 69, no. 1, pp. 293–309, Mar. 2020.

[34] L. Chen, S. Casper, and Q. Yuqing, "A performance analysis platform for performance evaluation of smart production lines," *Int. J. Performability Eng.*, vol. 16, no. 6, p. 834, 2020.

PENG YANG received the B.Sc. degree in computer science and technology and the M.Sc. degree in computational mathematics from Hunan Normal University, China, in 2000 and 2003, respectively. She is currently an Associate Professor with the School of Information Engineering, Guangzhou Panyu Polytechnic, China. Her research interests include software testing, data mining, and machine learning.

ZIXI LIU received the B.Sc. degree in software engineering from the Southwest University of Science and Technology, in 2020. She is currently pursuing the master's degree with Nanjing University. Her research interests include software testing, program analysis, and deep learning.

JIN XU received the master's degree in software engineering from Nanjing University, in 2020, where she is currently pursuing the Ph.D. degree with the School of Engineering Management. Her research interests include software testing and operation management.

YONG HUANG received the master's degree in software engineering from the Information Engineering University of the People's Liberation Army. He is currently the Deputy General Manager of research and development with Nanjing Mooctest Information Technology Company Ltd. His research interests include informatization project management, software architecture design and optimization, software testing theory and technology.

YA PAN (Member, IEEE) received the master's degree from Chongqing University. From 2015 to 2016, she continued to study at Northeastern State University as a Visiting Scholar. She is currently an Associate Professor with the Department of Computer Science and Technology, Southwest University of Science and Technology. Her research interests include software testing and quality assurance technology.