Mutation testing and self/peer assessment: analyzing their effect on students in a software testing course

Pedro Delgado-Pérez*, Inmaculada Medina-Bulo†, Miguel Ángel Álvarez-García‡, Kevin J. Valle-Gómez§
Department of Computer Science and Engineering, Universidad de Cádiz, Cádiz, Spain
Email: {pedro.delgado*, inmaculada.medina†, miguelangel.alvarez‡, kevin.valle§}@uca.es

***Abstract*—Testing is a crucial activity in the development of software systems. With the increasing complexity of software projects, the industry requires incorporating graduates with adequate testing skills and preparation in this field. A challenge in software testing education is to make students perceive the benefits of writing tests and assess their quality with advanced testing techniques. In this paper, we present an experience integrating both *mutation testing* and *self/peer assessment* —two of the most used techniques to that end in the past— into a software testing course during three years. This experience allowed us to analyze the effect of applying these strategies on the students' perception of their manually-written test suites. Noticeably, the computation of the mutation score significantly undermined the initial expectations they had on the developed test suites. Also, the application of peer testing helped them estimate the relative quality of two comparable test suites, as we found a notable correspondence with their respective mutation coverage. Besides, a more in-depth analysis revealed that the students' test suites with more test cases did not always achieve the highest scores, that they found more readable their own tests, and that they tended to cover the basic operations while forgetting about more advanced features. An opinion survey confirmed the impact that the use of mutants had on their perception about testing, and they mostly supported paying a higher level of attention to testing concepts in software engineering degree plans.**

***Index Terms*—Software testing education, mutation testing, peer testing, self/peer assessment.**

I. Introduction

Software testing is a fundamental activity in the life cycle of a software project. Testers aim to apply testing methods to find as many software defects as possible, spending the least possible effort and time. Despite the advances in this discipline, testing takes up most of the resources of a project and, therefore, the industry requires qualified professionals in this domain to achieve a more cost-effective process. Several studies show that the exposition to software testing knowledge also brings benefits to the code developed by students in terms of reliability [1] and programming skills [2]. However, current academic curricula in software engineering still place much more emphasis on development and design than on testing [3]. As a consequence, university degrees are not able to fulfill the demand of graduates with a solid background in testing.

As noted by Lauvås and Arcuri [4], software testing teachers not only face the challenge of instructing students in the use of testing techniques but also of making them understand how fundamental software testing is within the software development, especially when it is often seen as a tedious task. In fact, according to the study by Pham et al. [5], students prioritize obtaining a functional product over testing it, and they may find designing test cases fruitless —they cannot see an apparent reward derived from their test generation efforts. As a result, many of the teams participating in that study resorted to manual testing instead of testing methodically. As such, effective and innovative teaching methods are required to change students' attitude toward software testing.

Both mutation testing and peer testing are two of the strategies highlighted in the software testing teaching literature to enhance students' motivation [4], [6]. *Mutation testing* poses more demanding testing criteria than traditional coverage-based metrics [7], [8] —which usually overestimate the apparent test quality—, and allow students to directly observe examples of bugs that their tests are not able to detect [9]. As for *peer testing*, it exposes students to different solutions and offers the opportunity to share their knowledge with their classmates, engaging them in the learning process and increasing the quality of their developments [10], [11], [12]. Both techniques have been applied in several recent studies, mainly to analyze their impact on the learning of programming skills [13], to motivate students in the use of testing techniques [14] and to allow them to learn from each other [11]. However, it has been less explored what is the effect of applying these approaches on the students' perception, especially regarding the use of advanced testing techniques over manual and unguided testing, and the chance to evaluate their test suites with a comparable solution.

In this paper, we present the results of an experience in which we analyzed the subjective self and peer assessments made by our students of their manually-written test suites in combination with the more impartial measure of the mutation score. The results show that mutation testing had a great impact on their perception of the test suites; their evaluations, made before and after knowing the mutation score, differ even though the tests did not change in between both assessments. We also found that the comparison between their own test suites with that of a peer helped them estimate their relative detection capability and detect possible flaws regarding their completeness. However, that did not happen to the same extent with regard to the design and readability of the tests —two important factors in the software industry [15]—, which may suggest that a guideline on these aspects is required. At the end of the experience, students mostly found mutation testing useful and supported the incorporation of testing concepts into introductory programming courses. Finally, our analysis also reveals that a high number of test cases does not always imply greater fault detection, and that students tend to focus on the most basic operations and, therefore, forget about other more advanced features. These aspects should be emphasized in the classroom to make them aware of the different factors they should consider to improve the test quality and adequacy.

The structure of the paper is as follows. Section II defines the main concepts related to this work and reviews the studies addressing them. Section III presents our research questions and Section IV describes all the aspects related to the design of the experience. Then, Section V shows and discusses the results of the designed study. Section VI collects the lessons learned from the experience. Finally, Section VII summarizes the main findings of this study and provides some research lines that should be explored in the future.

II. Background

A. Mutation testing in software testing education

Mutation testing is a technique inspired by the actual objective of software testing: the detection of real faults. It is based on the insertion of syntactic changes, called *mutations*, that mainly try to resemble realistic bugs. The more mutants the test suite *kills* (i.e., the more of these mutations a test suite is able to reveal), the more likely is that it also identifies real faults. Such mutant detection happens when the output of executing it against the test suite differs from the output of the original version. The number of killed mutants over the whole set of killable mutants is called *mutation score*, which provides an estimation of the fault detection ability of the tests.

Mutation testing has gained attention in the last years not only in research studies and the industry but also in software testing education. In general, students perceive testing as an unproductive and secondary task [5] because they usually have to generate a test suite without tangible evidence of its benefit —they do not face the defects it could seemingly help avoid. However, mutation testing brings them in contact with simulated but plausible buggy versions of a program; they have the potential to engage students in analyzing why a test suite is not able to catch those faults, or even in refining it.

The use of mutation testing tools with educational purposes was proposed many years ago [16], but this technique has been put into practice more extensively in recent years. Oliveira et al. [13], [17] assessed the use of mutations in programming courses as a means to improve the learning process of novice students. Also in introductory courses, Smith et al. [9] used an automated system fed with a pool of buggy versions of the subject under test (SUT) to evaluate the test cases submitted by their students. Some studies have also investigated the accuracy of mutation and code coverage as metrics of the adequacy of students' test suites [7], [18]. A relevant group of papers following the same research line [14], [19] proposed the introduction of mutation testing as a game (the game is called Code Defenders) to make the learning of software testing concepts more enjoyable for the student. Zhang et al. [20] also tried to motivate students to focus on testing by injecting logical errors into their projects. In our study, the aim of using mutants is that our students realize that their manually-designed tests are not as effective as they could initially expect.

B. Self/Peer assessment

As described by Dochy et al. [21], while *self-assessment* refers to the involvement of students in judging their own developments or learning process, *peer assessment* is the process in which students rate their peers, either individually or in groups. Both self and peer assessment foster the active participation of students in their own learning and aim to give them a space to reflect on their results, especially when compared to those of their peers. Several authors have reported on successful experiences related to peer assessment in software testing sessions, generally known as *peer testing* [10], [11], [12], [22], [23], [24]. Peer testing encompasses the activities in which students assess the developments made by their peers, either reviewing each others' code or designing test cases to find defects in them. Peer assessment in software testing courses not only helps students to identify issues in their code —based on the reciprocal critique made by peers— but it can also allow them to learn from other solutions to the same problem [11], [25]. Clark et al. [10] observed several benefits of applying peer testing in her study, including the increase in the quality of the submissions, the awareness of the necessity of testing, the collaborative spirit among classmates, and the learning of technical skills, among others.

Peer testing has been applied following a diversity of approaches. Smith et al. [11] and Clark et al. [10] focused on both peer code review and testing to detect possible errors in the programs implemented by other classmates. In the experience by Smith et al, all the activities from programming to testing were additionally performed in pairs. The studies by Barbosa et al. [24] and Alazzam and Akour [23] went a step beyond by evaluating the use of pair programming in the design of test cases instead of in the implementation of the program (the traditional use of pair programming). The first study found that working with a partner was more efficient than working individually, whereas the second study showed that this collaboration also led to the design of test suites with a higher mutation score. The use of Code Defenders [14], [19] is another form of peer testing, where two students compete in a mutation testing game where they attack each other by creating mutants and defend themselves by adding mutant-killing test cases. Also, Gaspar and Langevin [12] combined pair programming and test-driven development to engage students in competitive learning. In our study, we follow a peer testing approach in which students inspect and grade the test suite designed by another peer, instead of reviewing the code of the tested subject as in previous works. Our approach is similar to the one followed by Gaspar et al. [22], in which students perceived benefits of exchanging their tests with other partners to improve both the test suite itself and the SUT.

III. Research questions

To achieve the goals of this study, we designed an experiment combining two different aspects. As shown in Figure 1,

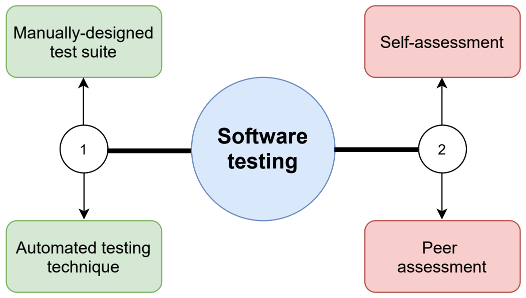

Fig. 1. Twofold approach in the designed experiment.

we confront both two **testing approaches** (*manual* and *automated*) and the **opinion on test quality of two different solutions** (the test suite created by the own student, *self-assessment*, and the one of a peer, *peer assessment*). Therefore, our aim is twofold:

1) By contrasting their subjective assessment with the impartial score offered by an automated technique like mutation testing, we expect students to become more aware of the benefits of applying advanced testing strategies.
2) By comparing their test suites with those of their peers, we want to analyze how the self and peer assessments differ and the extent to which students are able to discern between the quality of two different test suites. Additionally, we expect them to realize that each one may have a different perspective on how complete or well designed a test suite is and that relying only on manual methods may compromise its quality.

These hypotheses are reflected in the following questions:

RQ1: *What is the effect of knowing the mutation score of the manually-written test suite on the student's perception?*

RQ2: *What is the effect of introducing self and peer review of the student-written test suites in this experience?*

Additionally, we wanted to analyze in more detail the relation between the student-written tests and mutation testing to draw further conclusions on the designed test cases. That leads to the following two research questions:

RQ3: *Is the mutation score correlated with the number of designed test cases?*

RQ4: *Are there any programming features that are more likely to be poorly covered by the students than others?*

IV. Study Design

In the following subsections, we detail how we planned our experiments to answer our RQs, including details of the course and set of students, design of the experience, distribution in sessions, material, testing techniques and assessment criteria.

A. Course and participants

We carried out this experiment in the course *Software Verification and Validation* of the Degree of Computer Science and Engineering at our University. This is a 3rd-year course for students in the *Software Engineering* branch (one of the five different branches in this Degree). Students are taught the fundamental concepts and the importance of software verification and validation, and are instructed in the use of several testing tools and the testing techniques behind them, among others. We think that integrating this experience into a 3rd-year course is preferable to doing so into an introductory course. First, students at this point have a better understanding and background of the entire software development lifecycle; second, we counter the threat that low-performing test suites are directly related to the inability of novice programmers to understand the code of the SUT, to create the necessary test cases or to reason about the possible faults that could appear.

The experience was developed in three different years with the same configuration. In total, we have at our disposal the results and opinions about the experience of 20 students. This practice is part of a series of practices and exams that they have to undertake throughout the course, from which their final grade is derived. Therefore, while this experience has no definitive influence on their final grade, we believe it is sufficient to encourage them to take the proposed tasks seriously.

B. Design of the experience

For the sake of the experience's success, we leveraged different findings in the literature to prepare our experiment. Thus, we made some decisions about the following findings:

Finding #1: *According to different reports, students are not fond of testing tasks [5], [26], and they may find writing tests unproductive as there is no immediate benefit:* To address this issue, we designed a concise and manageable experience that was effective without being overly burdensome. Namely, we distributed the work into two sessions of 2,5 hours (5 hours in total), which were focused on a single test subject.

Finding #2: *Students do not like testing their own programs because testing judges their programming skills by revealing the faults in them [26], [27] :* To avoid this problem, we opted for providing them with a test subject. As such, the goal was to create a good test suite for that code instead of revealing existing defects in it. For once, we wanted students to be focused on testing more than on the programming aspects.

Finding #3: *According to the following report [5], the test subject size was a relevant factor for students, as they may feel that there is no need to test small projects:* In our experience, we took into account this consideration but to produce the opposite effect. We selected a test subject of low to moderate complexity, which consists of two C++ classes (Parent and Child adapted from a listing in [28]) representing the inheritance relationship between them. By using a small SUT, the effect of observing plausible faults not covered by their test suites could be more shocking for students than observing them in a complex or unmanageable program.

Finding #4: *Different studies [7], [8] have reported great differences between the use of statement and mutation coverage, being the latter more effective at detecting weaknesses in test suites:* As a result of this, we focus on mutation testing as the metric of test effectiveness. Mutation testing, in addition to providing an impartial measure of test quality, offers the

Task	Session 1	Session 2
Reading of material and preparation	20 min.	35 min.
Manual design of test suite	90 min.	–
Application of mutation testing	–	75 min.
Self assessment	15 min.	15 min.
Peer assessment	25 min.	25 min.

TABLE I
DISTRIBUTION OF THE TASKS IN SESSIONS

opportunity to gain first-hand knowledge of the types of faults that their tests would not reveal. We also believe that the effect of seeing uncovered faults is more tangible and convincing than the effect of a simple grade assigned by the lecturer.

Finding #5: *Different studies applying an approach related to peer testing have shown to provide benefits to students [10], [22], [24]:* In the designed experience, students play the role of both developers and reviewers at different times. In this study, peer assessments allow us to analyze the differences in the relative test quality of the student-written test cases when compared to the self-assessments, and how these assessments relate to the mutation score. In addition, this approach exposes students to a different test suite for the same program, and we can evaluate how that affects their perception.

C. Sessions

We divided the experience into two well-defined sessions of work (Table I represents the structure of the experience):

- **Session 1**: Students are asked to *manually design a test suite* as adequate as possible for the SUT. This process is done without the guidance of any measurable testing criteria, such as the statement or mutation coverage. They are only informed about the five assessment criteria that will be used to evaluate their test suites (these criteria are detailed in Section IV-E). After that, they are asked to use the rubric with the assessment criteria to evaluate their own test suite and the test suite designed by a peer.
- **Session 2**: Students are asked to *evaluate the fault detection capability of their test suites* (i.e., those designed in the first session) by applying mutation testing. After that, students are again asked to perform the same self and peer assessments of those test suites (without any modifications). The decision of evaluating again the initial version is intentional: we wanted to analyze whether knowing the mutation score changes their perception of the test suites they had already assessed.

The process can be summarized in terms of the documents that each student should submit:

- *Session 1*: Manually-written test suite, self-assessment survey and peer-assessment survey.
- *Session 2*: Execution matrix (it contains which test cases kill which mutants) and mutation score, self-assessment survey and peer-assessment survey.
- *After the sessions*: Anonymous questionnaire, inspired by the one designed by Oliveira et al. [13] and extended with new specific questions related to the experience. With these questions, we mainly want to know whether students can actually draw a lesson from this experience beyond the results of our analysis of their test suites. We also aim to understand how they perceive this experience in order to learn some lessons ourselves.

To differentiate the assessments made in the first and second sessions, we will refer to them as pre-mutation score (*pre-MS*) and post-mutation score (*post-MS*) assessments, respectively.

D. Course material and application of mutation testing

For the first session, we prepared a manual explaining how to create and execute new test cases for the SUT. We provided the students with a library for the design and execution of tests for object-oriented (OO) software. By using this library, adding a test case is as simple as expressing in a new function the desired sequence of calls to the public methods of the classes, and call *check_test_case* as many times as required to include the planned assertions. The following is a valid test case with two asserts, shown in the manual as an example.

```
void testChildConstructor(){
  Child c("Pritchett", "Simon");
  check_test_case(strcmp(c.getFirstName(), "Simon") == 0);
  check_test_case(strcmp(c.getLastName(), "Pritchett") == 0);
}
```

For the generation and execution of mutants, we applied *MuCPP* [29], a mutation tool for C/C++ code. The use of this tool is easy enough to be understood in one-session work; two command lines suffice to generate all the mutants and then execute them on the previously-designed test suite. We prepared a step-by-step manual on how to install the tool and configure the project to incorporate their own test suite, and how to use the tool and interpret its output. *MuCPP* counts with a set of traditional mutation operators (i.e., those commonly applied in any general-purpose language) and class operators (i.e., those affecting OO features).

Shams and Edwards [30] mentioned some issues related to the use of mutation testing in the classroom. In our study, they do not represent an obstacle because of the following reasons:

Issue #1: *Incomplete or buggy solutions*: In those cases, not all the expected mutants would be generated. *In our study:* Mutation testing is directly applied to the instructor-provided code, so the same mutants are available to all of the students.

Issue #2: *Compile-time dependencies*: The student-written tests may not compile when bound together with the instructor-provided reference solution. *In our study:* Students develop the test suite directly for the reference solution. Therefore, their tests should only refer to the elements of this original version, avoiding this type of compilation errors.

Issue #3: *Equivalent mutants*: These mutants, which actually do not represent a buggy version of the program despite the inserted change, may distort the mutation score calculation. *In our study:* Thanks to a previous inspection of all the mutants, we can inform the students which of the mutants are equivalent and should be subtracted for the mutation score computation. In total, without counting equivalent mutants, our students had to execute their test suites against a set of 39 killable mutants generated by 15 different operators.

E. Assessment Criteria

These experiments required the definition of a set of assessment criteria so that students could judge and value the developed test suite. To this end, we used five criteria that cover different aspects related to the quality of test suites:

1) **C1 - Correctness**: The test cases pass successfully when executed against the original program, which is considered to be fault-free.
2) **C2 - Completeness**: The test suite presents an appropriate coverage of functionalities by exercising the diverse scenarios that may apply when using the program.
3) **C3 - Assertions**: The test cases include a sufficient number of assertions to cover the possible changes in the internal state of the program as a result of the actions.
4) **C4 - Design**: Each test case is designed to test a particular functionality (i.e., a test case does not combine multiple and unrelated scenarios), and there are no overlapping or redundant test cases.
5) **C5 - Legibility**: The programming style is clear and allows understanding the purpose of the test cases and their assertions.

Students had to value each criterion in the range 1 (minimum) to 4 (maximum). With C1, we just wanted to make students aware that each input should produce the expected output according to the code of the program. C2 and C3 represent the aspects that could be measured with a testing criterion, such as statement or mutation coverage. Finally, C4 and C5 seek to preserve and encourage the clarity and readability of tests, an aspect that has been recently shown to be of utmost importance in the industry [15]. Also, as reported by Edwards and Shams [8], students tend to merge different scenarios in a single test case, leading to poor test case design.

Notice that C2 and C3 could be the more affected criteria in the student's assessments after knowing the mutation score (in principle, correctness, design and legibility are not directly related to the fault detection capability). As such, to better appreciate differences between the pre-MS and post-MS assessments, we weighted both C2 and C3 with 30% of the total assessment score. The rest of the criteria have the following weights: C1 (10%), C4 (15%) and C5 (15%).

V. Results and discussion

A. Effect of the mutation score on the student's perception

Table II furnishes the data collected from each student during this experience, which includes the number of designed test cases, the mutation score (as a percentage of all the mutants detected) and the marks assigned in the self and peer assessments in the range 1-4 points. Focusing on the mutation score, it stands out the low number of mutants killed overall. The scores range from 8% to 61%, although the mean is 31% and only 3 of the students were able to detect more than half of all the mutations. These results favor the expected effect from this practice, as a low mutation score reveals significant deficiencies in the developed test suites.

Student	Test cases	MS	Session			
			1		2	
			Self	Peer	Self	Peer
1	12	36%	3.25	3.25	2.8	2.65
2	8	28%	3.4	3.25	3.1	3.1
3	6	26%	3.2	2.8	3.1	3.1
4	8	33%	3.1	2.2	2.8	1.9
5	5	8%	2.4	3.6	-	-
6	9	26%	3.0	3.25	3.3	3.4
7	12	28%	2.8	3.2	2.35	1.85
8	9	10%	3.0	3.55	2.9	2.95
9	17	49%	4.0	2.8	3.4	2.65
10	6	18%	2.8	2.6	2.35	2.0
11	9	13%	2.8	2.4	1.75	2.45
12	4	23%	2.65	2.8	2.05	2.5
13	16	41%	2.65	2.65	2.5	2.65
14	13	59%	3.25	2.35	2.85	2.05
15	10	51%	3.4	2.95	2.75	2
16	11	22%	2.45	3.7	1.85	2.2
17	8	18%	1.9	2.65	1.85	2.25
18	14	31%	2.8	3.0	2.6	2.8
19	11	38%	3.25	3.55	2.35	2.65
20	14	61%	3.15	3.1	2.35	2.05
Mean	10	31%	3.0	3.0	2.6	2.5

TABLE II
List of the results by student, including number of test cases, mutation score (MS), and weighted mean of the marks assigned to the criteria (C1-C5). These marks are divided by session (1 and 2) and type of assessment, including the self review (*Self*) and the review made by the student of a peer's test suite (*Peer*).

To analyze the effect of the mutation score on the students, we can focus on the pre-MS and post-MS assessments. The mean value of the self and peer assessment fell 0.4 (from 3 to 2.6) and 0.5 points (from 3 to 2.5) from the first to the second session, respectively. Notably, 95% of the students depreciated the value of their test suites in their post-MS assessments. The greatest difference appears in the student 11 (-1.05 points), whose mutation score was notably low (13%). Also, 79% of the students reduced the marks assigned to their peers in the second session. In this case, the largest drop is 1.5 points, from 3.7 to 2.2 (student 16).

It is interesting to know which of the assessment criteria gave rise to these decreases. Table III shows the mean marks per criterion (C1-C5). As expected, the greatest differences appear in C2 (completeness) and C3 (assertions), both directly related to the mutation coverage. Regarding the self-assessment, we can observe a drop of 0.7 and 0.4 points in C2 and C3, respectively; these drops are even more notable in the reviews of the peers: 0.65 and 1 points, respectively. On the contrary, the design-related criteria (C4 and C5) were barely affected by the knowledge of the score. Interestingly, some of the students seemed to consider their tests not as "correct" as they initially expected, judging by the decrease in C1 (-0.3), although this was not the original purpose of this criterion.

B. Self and peer assessments

We can evaluate the student's perception of test quality by comparing the values assigned in the self and peer assessments, which can be seen again in Table II. In the first session, we can observe that nine of the students attached lower scores to their peers than to themselves, and that nine of

Assessment	Session	C1	C2	C3	C4	C5
Self	1	3.6	2.4	2.7	3.35	3.8
	2	3.3	1.7	2.3	3.35	3.7
Review by peer	1	3.1	2.5	3.15	3.25	3.35
	2	3.2	1.85	2.15	3.05	3.4

TABLE III
MEAN OF THE VALUES ASSIGNED IN THE SELF AND PEER ASSESSMENT DIVIDED BY SESSION (1 OR 2) AND CRITERION (C1-C5).

them did the contrary, being 1.2 points the greatest difference in both cases (student 9 and 16, respectively). Therefore, half of them felt that their tests were better than the ones of the reviewed peer, and the other half that they were worse. To better understand whether these differences are justified, we performed a correlation test analyzing the following variables:

1) X: The difference in mutation score of the own test suite and the reviewed test suite.
2) Y: The difference between the mean mark in the self and peer assessments (when acting as a reviewer).

With this statistical test, we wanted to observe whether an increase/decrease of the marks in the self and peer assessments corresponds with a similar increase/decrease in the mutation score of their test suites. As an example, let's consider student 12, who reviewed student 13. For this student, X = 23% - 41% = -18%, and Y is 2.65 - 2.8 = -0.15. In this case, the increase in the value attached to the peer's test suite corresponds with an increase in its mutation score; this means that this student seemingly realized that the peer's test suite was of a higher quality than his/her own test suite.

a) Pre-MS assessment: The data for the first session corresponding to X and Y are displayed in the scatter plot in Figure 2, showing the tendency line. We run the Pearson's correlation test between these two variables. The result (*r*: 0.587; *p-value*: 0.006) suggests that indeed there is a direct correlation between the differences in the assessments and the mutation scores, but this correlation is medium. We can interpret from this result that most of the students were able to understand which of the two test suites was the most adequate, but they could not accurately measure that difference.

b) Post-MS assessment: We can run the same correlation test but with the post-MS assessments, when they know the mutation score (Figure 3 represents these data). The result of the Pearson's correlation test (*r*: 0.830; *p-value*: 1.099e-05) shows that the correlation is now much stronger. This means that an impartial measure as the mutation score influenced their subjective assessment, this time assigning marks that were more in line with the fault detection ability of those tests.

We can analyze each criterion individually again in Table III, but focusing on the differences between the self and peer review this time. In the first session, the students gave higher marks to their peers than to themselves in C2 (from 2.4 to 2.5) and C3 (from 2.7 to 3.15). This leads us to think that, in general, they realized that some test cases or assertions were missing in their own test suites on reviewing the tests of their peers. Contrarily, they gave lower marks in C1 (from 3.6 to 3.1), C4 (from 3.35 to 3.25) and C5 (from 3.8 to 3.35).

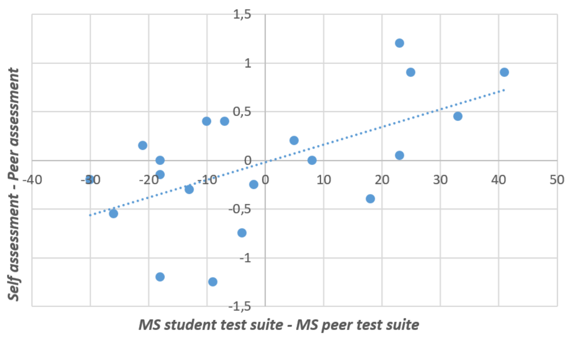

Fig. 2. Scatter plot - Session 1: Axis x: difference in mutation score (MS) of the student and the peer test suite (variable X); Axis y: difference between mean marks in the self assessment and peer assessment (variable Y).

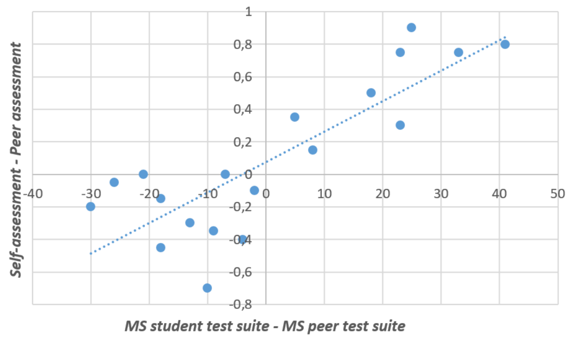

Fig. 3. Scatter plot - Session 2: Axis x: difference in mutation score (MS) of the student and the peer test suite (variable X); Axis y: difference between mean marks in the self assessment and peer assessment (variable Y).

Regarding C1, we found severe decreases (from 4 to 1) in two of the students, and we conjecture that they simply failed to execute the peer's test suite because they did not follow the steps correctly. Regarding C4 and especially C5, the decrease corresponds with our expectations that a test suite is not as straightforward to interpret for others as it is for its designer.

C. Student's opinions after the experience

Table IV compares the results of the questionnaire filled out by the students in our study and the study by Oliveira et al. [13]. We can observe a notable degree of similarity in the responses to SQ2, SQ6 and SQ8. SQ1 and SQ3 also present similar results but there was a transference to the new options —added to make those questions more complete. Interestingly, none of our students thought that their knowledge of testing reached an intermediate level before the course (SQ1), and 50% of them considered that practicing mutation testing could result in both better programs and programmers (SQ3).

Analyzing these questions altogether, we can infer that our students were aware that testing had not been properly addressed in previous courses and, however, that integrating testing techniques and tools into introductory programming courses could have had a positive effect on them. Remarkably, all of our students thought that using educational testing tools

#	Question	Answers	Result	Oliv.
SQ1	What were your previous knowledge of software testing before the course?	- None	33.3%	42.9%
		- Basic level	66.7%	-
		- Intermediate level	0%	57.1%
SQ2	Do you encourage mutation testing concepts to be presented in conjunction to programming foundations?	- Yes	25%	38.1%
		- Yes, but superficially	50%	52.4%
		- No	25%	9.5%
SQ3	What might be the effects of practicing mutation analysis by novice programmers?	- Better programs	25%	61.9%
		- Better skilled programmers	16.7%	28.6%
		- Better programs and skilled programmers	50%	-
		- None	8.3%	9.5%
SQ4	Do you consider regular testing tools useful to teach programming foundations?	- Yes	8.3%	36.4 %
		- Yes, using basic functionalities	91.7%	54.5%
		- No	0%	9.1%
SQ5	Do you consider the mutation testing tools useful to teach programming foundations?	- Yes	16.7%	19.4%
		- Yes, using basic functionalities	66.7%	19.5%
		- No	16.7%	61.1%
SQ6	Disregarding the specific course on software testing, considering your course's curriculum, software testing concepts were:	- Fairly presented	16.7%	13.6%
		- Poorly presented	58.3%	59.1%
		- Not presented	25%	27.3%
SQ7	Do you think that the use of educational testing tools might be useful to create good programming habits?	- Yes	100%	59.1%
		- No	0%	41.1%
SQ8	Do you think that designing test cases through mutation testing might be useful to improve the learning capacity of novice programmers?	- Yes	75%	72.7%
		- No	25%	27.3%

TABLE IV
FIRST PART OF THE QUESTIONNAIRE, WITH GENERAL QUESTIONS PROPOSED BY OLIVEIRA ET AL. [13]. COLUMN *Result* AND *Oliv.* SHOW THE RESULTS COLLECTED IN OUR EXPERIMENTS AND BY OLIVEIRA ET AL., RESPECTIVELY. NOTE THAT WE ADDED A NEW OPTION TO THE QUESTIONS SQ1 AND SQ3 (THOSE ARE MARKED WITH '-' IN THE COLUMN *Oliv.*)

#	Question	Answers	Result
SQ9	How did you find designing test cases manually without the aid of any testing tool?	- Easy, in general	25%
		- Hard, especially ensuring that the code was covered appropriately	50%
		- Hard, especially following a logic order in the design of test cases	25%
SQ10	What is your perception after applying mutation testing to your test suite?	- It changed my perception of test suite design	8.3%
		- It allowed me to discover fault-prone code areas not properly covered	75%
		- Mutants do not seem particularly useful in improving test quality	16.7%
SQ11	Considering your self and peer assessments, do you think that we tend to value our own developments positively and overlook certain shortcomings in them?	- Yes	50%
		- Yes, but only as far as design and legibility criteria are concerned.	8.3%
		- No, the assessments were fair.	41.7%
SQ12	From a practical viewpoint, mutation testing:	- Is very useful	58.3%
		- Is very useful, but is not comfortable to use	33.3%
		- Does not compensate for the effort required.	8.3%

TABLE V
SECOND PART OF THE QUESTIONNAIRE, WITH SPECIFIC QUESTIONS ABOUT THE EXPERIENCE.

might benefit the programming habits (SQ7), while only 59% of the subjects in the study by Oliveira et al. marked that option. Also, 100% and 83% of the students deemed that regular (SQ4) and mutation testing tools (SQ5), respectively, would be useful to teach program foundations (mainly focusing on the basic functionalities). This result contrasts with that of the study by Oliveira et al., where 61% of the students did not find mutation testing as a useful technique to that end. These results make us believe that our experience had a great impact on their perception of the importance of software testing.

We completed the survey with four additional questions, shown in Table V. Most of the students (75%) found it hard to design test cases manually (SQ9) and they realized that some areas were not properly covered by them thanks to the mutants (SQ10). There is not a great difference between the option 'yes' and 'no' regarding the fairness of the assessments (SQ11); this is in line with the analysis of the self and peer assessments in the previous section. In contrast, only 8% recognized a possible bias when assessing the design and the legibility, as Table III revealed. This shows that we often fail to realize that our designs might not be as readable as we tend to think. Finally, almost 92% found mutation testing very useful, but 33% of them also found it uncomfortable to use (SQ12).

D. Relation between the test suites and mutation testing

In this section, we analyze the student-written test suites in relation to the mutation score. First, we run the Pearson's test to evaluate the correlation between the number of test cases in the test suites and their mutation score. The result (*r*: 0.673; *p-value*: 0.001) reveals that, as we could expect, there is a significant positive correlation between these two measures, but the association between both variables is far from being perfect. In fact, if we look at the scatter plot in Figure 4, we can observe that two of the students designed 14 test cases with a very different outcome: 31% and 61% of mutation score. Also, the student's test suite with more test cases (17) achieved a lower score than another one with 10 test cases (49% vs 51%).

Operator	Description	Mutants	Students	Mean
IPC	Explicit call of a parent's constructor deletion	1	19	96.4%
OMR	Overloading method contents replace	3	20	90.4%
IHI	Hiding variable insertion	1	13	79.9%
CTD	*this* keyword deletion	1	11	63.5%
CTI	*this* keyword insertion	1	11	63.5%
COD	Conditional operator deletion (e.g., !)	1	11	63.1%
ARS	Arithmetic assignment operator replacement (e.g., $+=$ by $-=$)	4	14	56.6%
IOD	Overriding method deletion	1	6	41.0%
ROR	Relational operator replacement (e.g., $>$ by $>=$)	5	13	40.2%
AIS	Increment/decrement operator insertion (e.g., $++$)	8	14	33.2%
CCA	Copy constructor and assignment operator overloading deletion	1	2	17.7%
IOP	Overriding method calling position change	3	3	12.6%
ARB	Arithmetic operator replacement (e.g., $+$ by $-$)	7	6	9.4%
CDD	Destructor method deletion	1	1	8.8%
PVI	*virtual* modifier insertion	1	0	0%

TABLE VI

ANALYSIS BY MUTATION OPERATOR. *Students* SHOW THE NUMBER OF STUDENTS THAT DETECTED SOME MUTANTS OF THE OPERATOR AND *Mean* REPRESENTS THE WEIGHTED MEAN MUTATION SCORE (SUM OF THE MUTATION SCORE IN THE OPERATOR OF EACH STUDENT MULTIPLIED BY HER TOTAL MUTATION SCORE, DIVIDED BY THE SUM OF THE TOTAL MUTATION SCORES OF ALL STUDENTS).

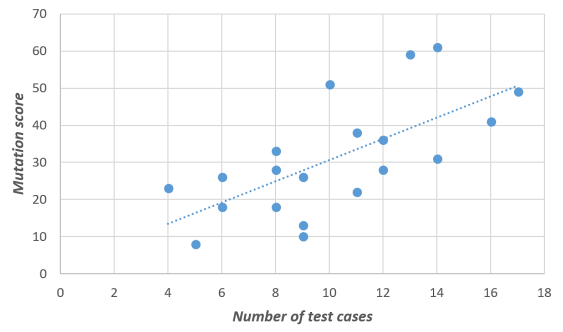

Fig. 4. Scatter plot - Relation between the number of test cases in the test suites and their mutation score.

Second, we sought to evaluate whether the mutation score was evenly distributed among the mutation operators. In other words, we wanted to observe whether the students' test suites had covered the mutants from all of the operators or some mutants were prone to remain undetected. To this end, we computed the mutation score obtained by each student in each operator, and then we calculated the mean mutation score for each operator. Note that this is a weighted mean so that those students that achieved higher mutation scores (and therefore worked harder on the test suite) contribute more to this result.

Table VI presents the list of mutation operators, followed by a brief description (further details can be found in [29]) and ordered by the mean mutation score. From the data in this table, it seems that the features related to the construction of objects were covered in general, such as the initialization (IPC), the presence of several constructors (OMR) and, to a lesser extent, the use of the *this* keyword (CTD and CTI). Also, many of the students (between 11 and 14) were able to detect some of the mutants generated by the traditional operators (COD, ARS, AIS and ROR) except for ARB, whose mutants mainly affected the computation of memory allocation for different arrays. However, their mutation score is not as high as the score of the operators at the top of the table because they generated more mutants and most of those students only detected some of them. On the contrary, some particular OO features addressed by class operators (see Section IV-D) were not exercised by most of the students (between 0 and 6), such as the copy (CCA) and destruction (CDD) of objects, class inheritance (IOD and IOP) and the polymorphism (PVI). This shows that some advanced features may easily go unnoticed, probably because students are less accustomed to working with them and put emphasis on the most common operations.

E. Threats to validity

A common threat of this kind of study stems from the number of subjects participating in the experiment. There is a wide choice of branches and courses in the Degree and, as a result, the number of students enrolled in each course is usually low. To counter this threat, we repeated the same experience in three consecutive years until we had a significant number of students to accomplish this study.

With regard to peer testing, there is the threat that those students who obtained a higher mutation score than the one of the assigned peer feel that they did an acceptable job, even though their mutation score is still very low. We see no easy way to completely avoid this issue; however, they had been taught during the whole course that they should aspire to 100% in the score to increase the confidence that their test suites will be able to detect faults. The positive results in favor of the use of mutation testing in the questionnaire lead us to think that they mostly regarded this experience as useful. It is also possible that mentioning to students that the SUT was supposed to be correct may have affected their motivation to test the system comprehensively. However, this was necessary to avoid that they focused on making changes to the code or even optimizing it instead of on designing the required tests. Nonetheless, this practice takes place at an advanced stage of the course, when they have been repeatedly told that they should always strive to develop a test suite as strong as possible, independently of their belief in the absence of faults.

As for the assessment criteria, we told them to value different aspects related to test suites to guide them as much as possible in their assessments so that the marks among students were comparable; we also assigned a weight to each criterion to calculate a mean mark. Had these criteria or their respective weights been different, the mean scores in the assessments could have been different too. However, we took five desirable properties of a test set and assigned weights to better observe differences between the pre-MS and post-MS assessments.

Finally, the available mutants in the SUT might not be representative of the features they target both because of the few mutants generated and because the difficulty to reveal each mutant depends on the fragment of code. However, the main purpose in RQ4 was to observe whether there were mutants hardly covered by the students more than pointing to specific uncovered features. Being this the situation, this fact reveals that there are cases to which students, in general, do not pay special attention or do not associate with possible faults.

VI. Lessons Learned

a) Effect of knowing the mutation score: The results of this experience reveal that the knowledge of the mutation score significantly decreased the expectations the students had on the test suites, especially concerning their coverage of functionalities and possible changes in the internal state of the program. It is interesting to observe the high correspondence between the mutation score and the values assigned in the post-MS assessments as that shows the extent to which the mutation score had an impact on their subjective perception. The survey reaffirms this conclusion as they mostly found mutation testing useful and recommended the integration of testing concepts and tools into introductory programming courses.

As the main lesson learned, this experience shows that applying mutation testing after the manual design of test cases is an effective method to make students more aware of the need of using advanced testing techniques to increase the quality of the test suites. However, the survey also reveals a need to develop or adapt testing tools for educational purposes, probably engaging them with more attractive GUIs.

b) Effect of introducing self and peer review: The fact that half of the students recognized that their test suites were worse than the peer's test suite is an interesting finding that supports the introduction of peer review in the classroom. The results show that the differences between the self and peer assessments matched well with the differences in the mutation score of both test suites. Thus, the application of self and peer assessments can be a useful method to help students calibrate the relative quality of their manually-designed test suites.

The analysis of the assessment criteria allows us to observe that our students mainly found shortcomings in the completeness and the set of assertions of their tests; in contrast, they did not detect as many deficiencies in their own test cases when evaluating the criteria related to their clarity. Although more than half of the students later acknowledged a possible bias in their assessments, this gives evidence that the importance of the design and legibility aspects should be reinforced in testing courses as they might not be given the necessary attention.

c) Fault detection capability of the designed test cases: The results show a significant correlation between two measures: the mutation score and the number of test cases. Still, we can observe some cases where a high number of test cases did not turn out in a higher mutation score when compared with other more reduced test suites. It can be useful to stress this result in the classroom to show students that it is not only the quantity but also the quality of the test cases that matters.

The analysis of the execution matrices indicates that most of the students paid more attention to some features and areas of code than others, especially forgetting about advanced features like those of the OO paradigm. This outcome is consistent with the findings by Edwards and Shams [8], who observed that several bugs introduced by the students in their solutions went unnoticed by most of their test suites. Thus, a lesson learned is that programming courses should underscore the problems derived from the wrong use of these features and how to check if an OO program presents the expected behavior.

VII. Conclusion and Future Work

In this paper, we have presented the results of an experience based on the concepts of self and peer assessment and the application of mutation testing in contrast to manual review. The results derived from their combination lead us to think that our students have now better understood the importance of the coverage criteria taught throughout the course and of applying advanced software testing techniques and tools in general. As the main finding, this paper evinces the effect that the mutation score had on their perception of the developed test suites when compared to their initial expectations. This impact on their perception can be drawn from the change in the evaluation of the same test suite before and after applying mutation testing, and from their support to an increase of software testing education in the questionnaire. Another remarkable finding is that our students were able to perceive the differences between their test suites and the peer's test suite, as there is a notable correlation between the mutation coverage and the assigned marks. Additionally, we could observe that our students found their own tests more readable, that the test suite size is not always related to the fault detection power, and that some language features were less covered than others. All this information could be wisely used to provide general feedback to the group, discuss with them the reasons behind these results, and make them more aware of the shortcomings.

In the future, we would like to investigate whether this experience has any beneficial effect in subsequent sessions where they have to develop tests for other programs. It would be equally interesting to analyze whether reviewing the test suites designed by their peers leads to a significant change or improvement in their tests afterward, especially with regard to the design and legibility criteria. We should also find ways to make evident to them that an increase in the SUT complexity would require greater testing efforts. Finally, a pending task is to adapt the mutation tool with a focus on education, thus making mutation testing more engaging to encourage students to use it in their future developments as professionals.

Data Availability

The material of the experience and the experimental data can be found in: https://doi.org/10.5281/zenodo.4464300

Acknowledgment

The work was partially funded by the European Commission (FEDER), the Spanish Ministry of Science, Innovation and Universities under the project FAME (RTI2018-093608-B-C33), the European project ASSETs (612678-EPP-1-2019-1-IT-EPPKA2-SSA-B), and the University of Cádiz.

References

[1] O. A. L. Lemos, F. F. Silveira, F. C. Ferrari, and A. Garcia, "The impact of software testing education on code reliability: An empirical assessment," *Journal of Systems and Software*, vol. 137, pp. 497–511, 2018. [Online]. Available: https://dx.doi.org/10.1016/j.jss.2017.02.042

[2] J. L. Whalley and A. Philpott, "A unit testing approach to building novice programmers' skills and confidence," in *Proceedings of the Thirteenth Australasian Computing Education Conference - Volume 114*, ser. ACE '11. AUS: Australian Computer Society, Inc., 2011, p. 113–118.

[3] T. Astigarraga, E. M. Dow, C. Lara, R. Prewitt, and M. R. Ward, "The emerging role of software testing in curricula," in *IEEE Transforming Engineering Education: Creating Interdisciplinary Skills for Complex Global Environments*, 2010, pp. 1–26.

[4] P. L. Jr. and A. Arcuri, "Recent trends in software testing education: A systematic literature review," in *31st Norsk Informatikkonferanse, NIK 2018, Universitetet i Oslo, Oslo, Norway, September 18-20, 2018*. Bibsys Open Journal Systems, Norway, 2018.

[5] R. Pham, S. Kiesling, O. Liskin, L. Singer, and K. Schneider, "Enablers, inhibitors, and perceptions of testing in novice software teams," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 30–40.

[6] V. Garousi, A. Rainer, P. Lauvås, and A. Arcuri, "Software-testing education: A systematic literature mapping," *Journal of Systems and Software*, vol. 165, p. 110570, 2020.

[7] K. Aaltonen, P. Ihantola, and O. Seppälä, "Mutation analysis vs. code coverage in automated assessment of students' testing skills," in *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, ser. OOPSLA '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 153–160. [Online]. Available: https://doi.org/10.1145/1869542.1869567

[8] S. H. Edwards and Z. Shams, "Do student programmers all tend to write the same software tests?" in *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education*, ser. ITiCSE '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 171–176. [Online]. Available: https://doi.org/10.1145/2591708.2591757

[9] R. Smith, T. Tang, J. Warren, and S. Rixner, "An automated system for interactively learning software testing," in *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 98–103. [Online]. Available: https://doi.org/10.1145/3059009.3059022

[10] N. Clark, "Peer testing in software engineering projects," in *Proceedings of the Sixth Australasian Conference on Computing Education - Volume 30*, ser. ACE '04. Australian Computer Society, Inc., 2004, p. 41–48.

[11] J. Smith, J. Tessler, E. Kramer, and C. Lin, "Using peer review to teach software testing," in *Proceedings of the Ninth Annual International Conference on International Computing Education Research*, ser. ICER '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 93–98. [Online]. Available: https://doi.org/10.1145/2361276.2361295

[12] A. Gaspar and S. Langevin, "Active learning in introductory programming courses through student-led "live coding" and test-driven pair programming," in *International Conference on Education and Information Systems, Technologies and Applications, Orlando, FL*, 2007.

[13] R. A. P. Oliveira, L. B. R. Oliveira, B. B. P. Cafeo, and V. H. S. Durelli, "Evaluation and assessment of effects on exploring mutation testing in programming courses," in *2015 IEEE Frontiers in Education Conference (FIE)*, 2015, pp. 1–9.

[14] B. S. Clegg, J. M. Rojas, and G. Fraser, "Teaching software testing concepts using a mutation testing game," in *Proceedings of the 39th International Conference on Software Engineering: Software Engineering and Education Track*, ser. ICSE-SEET '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 33–36. [Online]. Available: https://dx.doi.org/10.1109/ICSE-SEET.2017.1

[15] A. Arcuri, "An experience report on applying software testing academic results in industry: we need usable automated test generation," *Empirical Software Engineering*, vol. 23, no. 4, pp. 1959–1981, 2018.

[16] J. R. Horgan and A. P. Mathur, "Assessing testing tools in research and education," *IEEE Software*, vol. 9, no. 3, pp. 61–69, 1992.

[17] R. A. Oliveira, L. Oliveira, B. B. Cafeo, and V. H. Durelli, "On using mutation testing for teaching programming to novice programmers," in *Proceedings of the 22nd International Conference on Computers in Education (ICCE'14), Nara, Japan*, 2014, pp. 394–396.

[18] S. H. Edwards and Z. Shams, "Comparing test quality measures for assessing student-written tests," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE Companion 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 354–363. [Online]. Available: https://doi.org/10.1145/2591062.2591164

[19] G. Fraser, A. Gambi, M. Kreis, and J. M. Rojas, "Gamifying a software testing course with code defenders," in *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 571–577. [Online]. Available: https://doi.org/10.1145/3287324.3287471

[20] P. Zhang, J. White, and D. C. Schmidt, "Holicow: Automatically breaking team-based software projects to motivate student testing," in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, 2016, pp. 436–439.

[21] F. Dochy, M. Segers, and D. Sluijsmans, "The use of self-, peer and co-assessment in higher education: A review," *Studies in Higher Education*, vol. 24, no. 3, pp. 331–350, 1999. [Online]. Available: https://doi.org/10.1080/03075079912331379935

[22] A. Gaspar, S. Langevin, N. Boyer, and R. Tindell, "A preliminary review of undergraduate programming students' perspectives on writing tests, working with others, & using peer testing," in *Proceedings of the 14th Annual ACM SIGITE Conference on Information Technology Education*, ser. SIGITE '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 109–114. [Online]. Available: https://doi.org/10.1145/2512276.2512301

[23] I. Alazzam and M. Akour, "Improving software testing course experience with pair testing pattern," *International Journal of Teaching and Case Studies*, vol. 6, no. 3, pp. 244–250, 2015.

[24] J. R. Barbosa, P. Valle, J. Maldonado, M. Delamaro, and A. M. R. Vincenzi, "An experimental evaluation of peer testing in the context of the teaching of software testing," in *2017 International Symposium on Computers in Education (SIIE)*, 2017, pp. 1–6.

[25] D. Nicol, A. Thomson, and C. Breslin, "Rethinking feedback practices in higher education: a peer review perspective," *Assessment & Evaluation in Higher Education*, vol. 39, no. 1, pp. 102–122, 2014. [Online]. Available: https://doi.org/10.1080/02602938.2013.795518

[26] T. B. Hilburn and M. Townhidnejad, "Software quality: A curriculum postscript?" *SIGCSE Bull.*, vol. 32, no. 1, p. 167–171, Mar. 2000. [Online]. Available: https://doi.org/10.1145/331795.331848

[27] D. Carrington, "Teaching software testing," in *Proceedings of the 2nd Australasian Conference on Computer Science Education*, ser. ACSE '97. New York, NY, USA: Association for Computing Machinery, 1997, p. 59–64. [Online]. Available: https://doi.org/10.1145/299359.299369

[28] R. S. Wiener and L. J. Pinson, *The C++ Workbook*. USA: Addison-Wesley Longman Publishing Co., Inc., 1990.

[29] P. Delgado-Pérez, I. Medina-Bulo, F. Palomo-Lozano, A. García-Domínguez, and J. J. Domínguez-Jiménez, "Assessment of class mutation operators for C++ with the MuCPP mutation system," *Information and Software Technology*, vol. 81, pp. 169–184, 2017. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2016.07.002

[30] Z. Shams and S. H. Edwards, "Toward practical mutation analysis for evaluating the quality of student-written software tests," in *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research*, ser. ICER '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 53–58. [Online]. Available: https://doi.org/10.1145/2493394.2493402