Towards the role of test design in programming assignments

Lilian P. Scatalon*, Jorge M. Prates*‡, Draylson M. de Souza*, Ellen F. Barbosa* and Rogério E. Garcia†
* University of São Paulo (ICMC-USP), São Carlos, SP – Brazil
† São Paulo State University (FCT-UNESP), Presidente Prudente, SP – Brazil
‡ Mato Grosso do Sul State University (UEMS), Nova Andradina, MS – Brazil
Email: {lilian.scatalon,jorgemprates}@usp.br, {draylson,francine}@usp.br, rogerio@fct.unesp.br

***Abstract*—Software testing can be very helpful to students if adopted in programming assignments throughout the Computer Science curriculum. Many testing practices involve students writing their own test cases. This approach implies that students are responsible for the test design task while performing the test activity. On the other hand, some testing practices follow the opposite approach of providing ready-made test cases, so students only need to execute and evaluate test results for their solution code. In this paper, we investigated the effect of test design in student programming performance. We conducted an experiment comparing two different testing approaches during programming assignments: student-written and instructor-provided test cases. We also assessed students' perceptions of this subject by means of a survey. Results suggest that when students are responsible for test design, i.e. when they write their own test cases, they perform better in programming assignments.**

***Index Terms*—Software Testing, Test Design, Student-written Tests, Instructor-provided Tests, Programming Assignments**

I. Introduction

Traditionally, software testing has been addressed as an isolate topic in upper division computing courses [1]. However, this approach for software testing education does not ensure that testing skills are being reinforced during different computing courses. Indeed, it is possible to observe Computer Science seniors not being able to fully test even simple programs [2].

Students learn how to program during different computing courses, with many opportunities to practice their programming skills. However, they do not learn in the same depth how to validate their own programs. This issue has raised several proposals to spread the teaching of software testing and to integrate testing practices into programming assignments throughout the Computer Science (CS) curriculum [3], [4], [5].

Ideally, the teaching of software testing should begin as early as possible in the CS curriculum, integrated into introductory programming courses [6], [7], [8]. This integration has potential to improve how students learn both subjects, since it provides opportunities to reinforce testing skills, which in turn can result in higher quality code, leading to improved programming skills as well.

A relevant aspect of software testing in this context is to assure that students are adequately testing their programs. In order to get useful feedback about their code, students should rely on a set of test cases that represent the expected behavior of the program. Since it is not feasible to perform exhaustive testing, i.e. use all possible input values to exercise the code, there is the need to select a subset of the input domain, aiming to test the program thoroughly and effectively.

During the testing activity, test design is the task in which the appropriate input values are selected to compose test cases. If instructors are responsible for the test design task, then a test suite would be available for students, which indicates what is expected from the assignment solution. However, this approach of instructor-provided test cases raises an important question. If students do not perform the test design task, will their programming performance be affected?

On the other hand, there are also important issues to consider about the approach of student-written test cases. If students are supposed to write the test cases themselves, then they should be able to know how to select appropriate values to compose test cases. Only then will students appropriately check the validity of their program. They should be equipped with techniques on how to select a set of input values and also to improve their test suite [2].

Therefore, when testing practices are integrated into programming assignments, students need to have some kind of background on basic testing concepts, such as testing criteria. A testing criterion is used to decide which test inputs should be selected [9]. Using testing criteria, students would have the rationale to write test cases, which could make the testing practice more meaningful to them.

In this scenario, we intend to investigate in this paper the role of the test design task in student programming activity. We offered an extracurricular short course on the use of software testing in programming assignments. Students were instructed on basic testing concepts and testing criteria. They also learned how to perform automated unit testing in C using with the assert macro.

We used the final project of the short course to conduct an experiment about the effect of test design on the programming performance of students who attended. We also applied a survey to assess their perceptions of this subject. Results suggest that performing the test design task benefits student programming performance. Also, it was possible to observe that even freshman-level students were able to learn basic testing concepts and apply testing criteria to design their own test cases.

2377-570X/17 $31.00 © 2017 IEEE
DOI 10.1109/CSEET.2017.34

The remainder of this paper is organized as follows. Section II points out how different testing practices adopted in programming assignments can be characterized according to tasks that compose the testing activity, including the test design task. We discuss how our investigation relates to other studies from the literature in Section III. The short course on software testing, which is the context where our investigation took place, is described in Section IV. Details on the experiment and survey are given, respectively, in sections V and VI. We discuss the obtained results in Section VII. Finally, we present conclusions and possible future directions for research in Section VIII.

II. Background

When software testing is integrated into introductory courses, test cases are a valuable resource of programming assignments, in addition to the solution code. Students can use test results as feedback about their code during the process of solving the assignment, before final submission. In this way, they have a mechanism to validate and improve their own code. The goal is that testing practices help students to write high quality code.

Test cases indicate what is expected from a program that correctly solves the problem stated in the assignment. Hence, binding programming and testing activities is very helpful for students. Specially during programming assignments, since students' testing skills are related to understanding what should be done in the assignment [10].

The integration of testing practices in this context usually involves students writing and submitting their own test cases for the programming assignment. This is the case, for example, when instructors recommend students to apply TDD (test-driven development), which is often combined with the use of an automated assessment tool [11], [12], [13].

In TDD, there is a well-defined order to conduct programming and testing activities. At first, the programmer develops test cases. This aspect of TDD is known as test-first programming. Next, he/she executes the test cases and finally writes/refactors the code to the ones that failed. This sequence of steps is repeated iteratively until the unit is complete. Hence, if students are equipped with an automated assessment tool, they will have feedback available for each iteration of the TDD process.

Although TDD is largely used in educational settings [13], it may not be appropriate for every specific context. When comparing students from different levels, novice programmers can be more reluctant to adopt a test-first approach [8], [14]. This kind of evidence helps instructors configure the appropriate testing practice for a given context. For example, choosing to apply TDD or not involves the decision between a test-first or a test-last approach.

There are many ways to configure how software testing should be integrated into programming assignments. Figure 1 shows an initial outline of how to design this integration.

Regarding aspects of the *programming assignment*, it is important to contextualize in which *course* it takes place and to help determine if a given software testing approach is appropriate for students of that course.

Introductory courses are already packed with the programming content, so the addition of software testing should avoid students' cognitive overload. Each kind of testing practice requires different background knowledge and skills. The *programming language* used in the course is also a relevant factor. Each language offers different supporting mechanisms to write and execute test cases.

For the *software testing* approach being integrated into the assignments, instructors should consider which are the *testing concepts* that students should learn in order to be able to perform testing. They should also consider how the *testing practice* should be configured.

One way to define the testing practice is based on the sequence of *testing tasks* that compose the testing activity [9]:

- **test design**: input values are chosen to compose test cases, aiming to effectively test the program;
- **test automation**: the values are inserted into executable code;
- **test execution**: running the code along with the test cases and recording the results; and
- **test evaluation**: evaluating results and taking appropriate actions accordingly, such as reporting to the person responsible for implementing the program under test.

For *test design*, the instructor may require that students write test cases themselves or provide ready-made test cases. The *automation task* is related to the format that test cases will assume.

The simpler format of test cases can be a manual testing approach, with no automation involved, and inputs/expected outputs organized in a table. Despite its simplicity, this can be considered a way to increase students' confidence in the correctness of their code. Students manually enter the inputs, observe the actual outputs (by means of prints, for instance) and compare them to the expected ones listed on the table.

Towards automating execution, a student could use conditional statements (in the format `if(expected!=actual)`) or use assert statements, as we did in this study. Depending on the language used and students' programming experience, testing frameworks (such as JUnit) can be used to automate execution and ease considerably the test evaluation task.

Test execution and *evaluation* might be the more straightforward tasks to integrate into programming assignments. In these tasks students run test cases against the program being developed. By performing these tasks, they can get feedback to improve their code.

Finally, we point out some of the observed *results* that could be observed from the integration of software testing into programming assignments. Both deliverables, program and test suite, can help to evaluate students' *programming* and *testing performance*. Additionally, two other kinds of results are students' actual *behavior* while completing the assignment (given what was required from them) and their *perceptions* of the software testing integration.

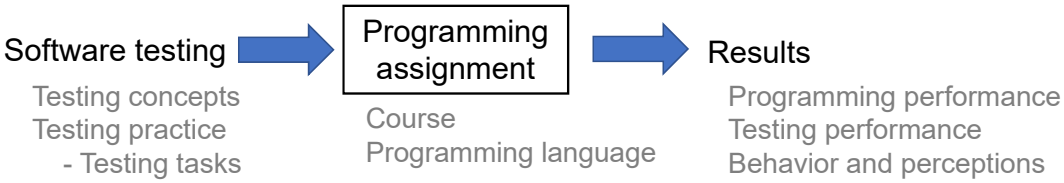

Fig. 1. Characteristics of the integration of software testing into programming assignments

Depending on how the approach is designed, students might face difficulties to cope with programming and testing at the same time, what can influence negatively how they perform and see the testing practice. For example, the more novice the student is, the more reluctant he/she is to adopt a test-first approach (or TDD) [14]. Therefore, maybe test-first is not completely adequate for first-year students, or otherwise some secondary aspect should be reconsidered in this software testing integration approach.

III. Related Work

Several studies have investigated the integration of software testing into programming assignments. Table I shows examples of such studies. More specifically, the testing practice discussed in each one is framed according to the sequence of testing tasks.

The tasks in which students were actively involved are marked with an 'X'. In all studies they performed at least the last two tasks (test execution and evaluation). The first two tasks (test design and automation), which are the ones associated with writing the test cases, present some variations among the studies.

Overall, it is possible to note that students do not need to be actively involved in all testing tasks. The instructor can provide scaffolding for the testing activity, such as ready-made tests [17]. The scaffolding seems reasonable if removes unnecessary difficulties for students at the level in question and adds benefits for their programming performance (such as what happened in S6).

For studies S4 and S6, students received ready-made test cases. Therefore, they were only responsible for execution and evaluation. Researchers that advocate for the use of instructor-provided test cases draw attention to the cognitive overload that might happen, specially for freshman-level students [15], [17]. Students can feel overwhelmed for dealing with both programming and writing test cases.

On the other hand, requiring that students write their own test cases helps improving their testing skills. By doing so, they might be thinking more critically about the problem stated in the assignment [11], [10]. For studies S1, S2 and S3 student-written tests are required, and some of them also involve TDD adoption (S1 and S2).

In particular, the study S5, performed by Isomöttönen and Lappalainen [16], had a slightly different approach. They provided ready-made test cases, but also allowed students to write their own test cases for extra points. That is why this study is coded as 'partial' for the two first tasks. In this case, it can be said that they applied a hybrid approach.

It is interesting to note that both approaches (student-written and instructor-provided test cases) can offer benefits in terms of learning for student. The decision on whether using student or instructor test cases in programming assignments is directly related to the test design task.

In order to gather evidence about this issue, we evaluated testing practices with these two approaches for test design (student-written tests and instructor-provided tests) and compared them in terms of programming performance. We chose to evaluate student programming performance, since, ultimately, the testing activity in this context should be a supporting practice to the programming activity.

Studies in this context often use a single approach for the test design task, either when students (student-written tests) or instructors (instructor-provided tests) are responsible for it. The approach of using student-written test is the most prevalent one [11], [12], [8], [10] but, to the best of our knowledge, there are no studies comparing both approaches or investigating the benefits of assigning to students the responsibility of the test design task.

Lemos et al. investigated the effect of testing knowledge in students' programming performance [18]. However, their study was conducted during an advanced course of software testing. The results suggest that testing knowledge indeed benefit students' programming performance.

Designing test cases involves choosing input values that test the program effectively, since exhaustive testing is not feasible. So, test design is a task that requires testing knowledge to choose appropriate values. Nevertheless, it is interesting to note that, although testing knowledge would help students in performing test design, they are not often instructed in testing concepts during introductory programming courses.

Only few studies report about instructing students in testing concepts in this context [19], [6], [20], [21], [22], [23], [24], [25]. The instruction ranges from some kind of input values selection guideline to teaching testing criteria. In our study, we taught students two testing criteria, *equivalence partitioning* and *boundary value analysis* [26], similarly to the study performed by Barbosa et al. [6].

TABLE I
STUDENT INVOLVEMENT IN TESTING TASKS DURING PROGRAMMING ASSIGNMENTS

study	authors	year	testing tasks			
			design	automation	execution	evaluation
S1	Edwards [11]	2004	X	X	X	X
S2	Spacco and Pugh [12]	2006	X	X	X	X
S3	Janzen and Saiedian [8]	2008	X	X	X	X
S4	Whalley and Philpott [15]	2011			X	X
S5	Isomöttönen and Lappalainen. [16]	2012	partial	partial	X	X
S6	Utting et al. [17]	2013			X	X

IV. A SHORT COURSE ON SOFTWARE TESTING

We conducted our study during an extracurricular short course offered in the São Paulo State University (Unesp). The goal was to introduce testing concepts and explore the use of unit testing during programming assignments.

A course on such subject was relevant in this setting because software testing would only be addressed in a junior-level Software Engineering course. So our idea was to teach students about software testing earlier, emphasizing its relation with the programming activity.

The course duration was eight hours and Computer Science majors from all levels were able to enroll. A total of 21 students enrolled in the course and 11 of them consented to release their data. An overview of the conducted activities during the short course is given in Figure 2.

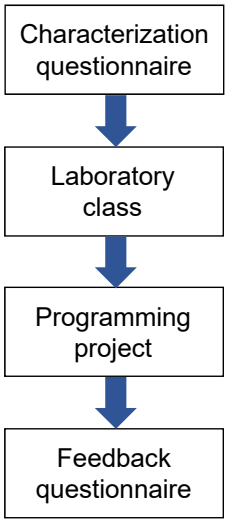

Fig. 2. Overview of the short course on software testing

In the first activity students completed a *characterization questionnaire*. The idea was to identify which was their prior knowledge about programming and find out about their testing habits during programming assignments.

The next activity was a *laboratory class* in which we taught basic testing concepts. We covered concepts such as basic terminology (fault, error, failure), test suite and test cases definition (input/expected output) and evaluation of results after the test execution (program passed/failed). There were several examples and exercises to help students grasp each of these testing concepts.

After introducing testing concepts, the next activity was a *programming project*, composed by three assignments. Students were asked to apply software testing as a supporting practice to complete the proposed assignments. We collected student data from this activity as part of our experiment described in Section V.

Students were trained on the testing practice during the first assignment. During the other two assignments, we divided students in two groups and asked they to use different approaches to the testing practice. We used the data collected from consenting students (n=11). Finally, as the last activity, we applied a *feedback questionnaire* about their experience while using testing practices to solve programming assignments (see Section VI).

Throughout the class, we motivated students to think about their perceptions of software testing. At first, we showed its destructive aspect, since the purpose of executing tests is to reveal the presence of faults in the program. This might discourage students to test their assignments, since, in contrast, programming has a constructive aspect.

On the other hand, we also pointed out the positive aspect of applying software testing in programming assignments. We argued that testing practices could help them find faults on their code before the instructor does so. In this way, they would be learning a skill that helps improving their grades.

We also taught students on how to select values for test cases with *equivalence partitioning* and *boundary value analysis*. We provided a step-by-step that students could follow to select test values, considering the two criteria together:

1) Identify input conditions.
2) For each one, perform the equivalence class partitioning for the input domain and determine valid and invalid input classes.
3) Select:
   a) one input value for each class and
   b) input values from the boundaries of each class.
4) Construct test cases for the selected input values (determine the expected output values).

All these testing concepts were instantiated to a particular testing practice of automated unit testing. Students learned how to write, execute and evaluate test cases using the assert macro from the standard C library.

C is the language used during the first introductory programming courses at Unesp. Following a similar approach of Janzen and Saiedian [8], we chose the assert macro because it would be simple enough to be understood by freshmen and sufficient to automate execution of test cases.

An example that illustrates this testing practice is given next. In this case, the unit under test is the function that computes the factorial of a given integer (and contains a seeded fault).

```
#include <assert.h>

int factorial(int n) {
 if (n < 0)
  return -1;
 else if (n < 1)
  return 1;
 else
  return n * factorial(n - 1);
}


void factorial_tests(){
 assert(factorial(-1) == -1);
 assert(factorial(0) == 1 );
 assert(factorial(1) == 1);
 assert(factorial(2) == 2);
 assert(factorial(3) == 6);
 printf("factorial() passed all test
     cases!\n");
}

int main(){
 factorial_tests();
 return 0;
}
```

V. Experiment

The following subsections provide details about the experiment we conducted and the obtained results. We followed guidelines from Juristo and Moreno [27] and Wohlin et al. [28] to plan and execute the study.

A. Goal

The integration of software testing in computing courses should be done in a way that adds value to the programming assignments. Students should be able to benefit from the testing practice, what can lead them to perform it more willingly.

In this scenario, we intend to investigate the integration of testing practices into programming assignments. We focused on one of the aspects of the testing activity, the test design task, and its effect on student programming performance.

If students are being able to improve their code from the feedback provided by testing results, they probably will feel the need to do so and the integration of software testing will not disrupt the normal course flow to teach the main subject, i.e. the programming concepts.

B. Subjects

We applied a subject characterization questionnaire in the beginning of the short course. In total, 21 students attended, all Computer Science majors, distributed as indicated by Table II. 11 students provided consent on the collected data.

TABLE II
Distribution for student level (n=11)

Student level	#
Freshman	45.45% (5)
Sophomore	9.09% (1)
Junior	9.09% (1)
Senior	36.36% (4)

Since we focused on the application of software testing for programming assignments, we also characterized the students regarding the introductory courses they had already completed or were still attending (see Table III).

TABLE III
Distribution for introductory programming courses (n=11)

Course	#
Introductory Programming I	100% (11)
Introductory Programming II	100.00% (11)
Data Structures I	54.55% (6)
Data Structures II	54.55% (6)
Object Oriented Programming	45.45% (5)

Introductory Programming I and II are first-year courses that involve the teaching of fundamental programming constructs, with an imperative-first approach using the C language. Data Structures I and II are second-year courses about data structures (also in C) and the corresponding algorithms to operate them. Finally, Object Oriented Programming is a second-year course in which students learn about object-oriented concepts using the Java language.

Regarding their prior testing habits, according to Table IV, they usually perform testing while working on their programming assignments. However, as Table IV shows, most students do not use or do not know what a testing criterion is. This is an interesting outcome, considering that testing practices are not addressed during regular programming courses in this setting. Also, it shows that even though students test their code, they are doing it without proper instruction in this subject.

C. Experimental Objects

The experimental objects were the programming assignments from the final project proposed to students at the end of the short course. The project was composed by three assignments, which consisted in implementing alarm clock features (based on the assignments from [17]).

In order to solve them, students had to represent time, as composed of hours and minutes (in a simplified way) and perform basic operations with time calculation.

TABLE IV
STUDENT TESTING HABITS IN PROGRAMMING ASSIGNMENTS

Question: *Do you test the programs you write?*	
Answer	#
I do not know what it means to test a program	0% (0)
There is no need to	0% (0)
Yes, only if there is enough time	0% (0)
Yes, I always test at least a little	90.9% (10)
Yes, I always try to test a lot before delivering the program	9.1% (1)

TABLE V
USE/KNOWLEDGE OF TESTING CRITERIA

Question: *Do you use any testing criteria to test your programs?*	
Answer	#
Yes	45.45% (5)
No	27.27% (3)
I do not know what test criteria is	27.27% (3)

- **Assignment 1.** Tick operation: advance the current time in one minute.
- **Assignment 2.** Alarm clock set features:
  - **(A)** Wake-up time: set the alarm with the desired wake-up time and show the user the remaining sleep time. Involves the implementation of the sum of two times.
  - **(B)** Remaining sleep time: set the alarm with the desired remaining sleep time and show user the resulting wake-up time. Involves the implementation of the subtraction of two times.

Assignment 1 was carried out as a training, with the instructor's assistance. In this way, students were able to become familiar with the testing practice and this problem context. After, they completed assignments 2A and 2B by themselves.

D. Hypotheses

We intend to compare different testing approaches during programming assignments, according to the hypotheses listed on Table VI.

In the first approach students are not responsible for the test design task, they receive ready-made instructor tests (IT). In the second one students need to perform the test design, besides the other testing tasks (test execution and evaluation). So, the test cases are written by the own students (student tests – ST).

The effect of these two testing approaches were observed in students' programming performance. In turn, programming performance was considered in terms of correctness of the program delivered by students to solve the assignment.

E. Variables

During the experiment, students were supposed to apply two different testing approaches to complete programming assignments. In order to configure these testing approaches, some aspects were kept constant for both, while the test design task was the aspect that varied.

Starting from the constant aspects, both assignments involved automated unit testing using the assert macro in C. Also, students were always responsible for the *execution* and *evaluation tasks* of the testing activity. Regarding supporting tools, students were free to use the IDE they were familiar with to write the solution code, the test cases (when applicable) and to execute test cases.

The experiment had one independent variable, the *test design task*. This variable had the two following treatments:

- *instructor-provided test cases* (IT), when students receive ready-made test cases and are not responsible for test design, but only for test execution and evaluation; and
- *student-written test cases* (ST), when students are also actively involved with the test design task and have to write their own test cases.

Since we were aiming to evaluate students' programming performance, the dependent variable was the *correctness* of the solution code submitted for the programming assignment. We measured correctness as the *pass rate* of student solution code.

This metric was calculated dividing the number of test cases for which the unit passed by the total number of test cases for that unit. We used a set of reference test cases, which was the same we delivered to students in the assignments that they were not responsible for test design.

F. Experimental Design

Considering the selection of one independent variable with two treatments, we chose the *paired comparison* design [28]. However, to avoid the learning effect over results, we used two different experimental objects for each subject.

Subjects were divided into two groups randomly. Table VII shows the experimental design and how subjects were assigned

TABLE VI
STUDY HYPOTHESES

Hypotheses type	Formalized hypotheses
Null hypothesis	H_0: $correctness_{IT} = correctness_{ST}$
Alternative hypotheses	H_1: $correctness_{IT} > correctness_{ST}$ H_2: $correctness_{IT} < correctness_{ST}$

to experimental objects (assignments 2A and 2B) and treatments (IT and ST).

G. Results

We analyzed students' programs in order to calculate the correctness of each one. We used the CUnit[1] testing framework in order to execute and evaluate test results.

Individual correctness values for both testing approaches are given in figures 3 and 4. It is interesting to note that some students kept their programming performance somewhat at the same level for both approaches. Most of them (S1, S3, S6, S8, S9, S11) achieved good results for both.

Other aspect from the individual results that draws attention is that many subjects (S2, S4, S7, S10) achieved better results with student-written test cases. Only subject S5 had a considerably lower result with student-written test cases.

These results are summarized in Table VIII, using the means and standard deviations for each approach. Table IX shows results separately for each group. Comparing the means, students that wrote their own test cases (ST approach) achieved better correctness scores.

However, these results cannot support that there is indeed a difference between the testing approaches. We applied the Wilcoxon signed-rank test, but there was not enough evidence to reject the null hypothesis at $\alpha = 0.05$ significance level. This outcome is very likely related to the small size sample.

VI. SURVEY

As the last activity of the short course we applied a feedback questionnaire about students' experience in completing the assignments with the aid of software testing. Students' responses (Table X) allowed us to gather some insights about their perceptions and behavior for the proposed activities.

All students agree that testing practices indeed help to work on programming assignments (Q1). Most of them (81.8%) think it helped for both testing approaches and a small portion of them (18.2%) think it helped only when they wrote the test cases themselves.

The responses also provided an overview of the actual student behavior while working on the programming assignments. $90,9\%$ stated that applied the testing criteria while writing test cases (Q2). Additionally, $72,7\%$ stated that did not face difficulties while doing so (Q4). These numbers suggest that most students were able to understand and to apply testing criteria, including freshmen.

[1]cunit.sourceforge.net

One of the questions (Q3) assessed students' tendency to adopt test-first or test-last approaches. Students were quite divided in this matter, with almost half of them for each approach.

We did not impose a specific order to perform programming and testing activities, but we did demonstrated examples and exercises from the course material in a test-first manner and this might have influenced students' behavior.

The last question (Q5) assessed whether students intend to incorporate testing practices as part of their strategy to complete programming assignments in the future. All of them answered positively.

In the beginning of the short course they had indicated that already performed testing in their programming assignments, but most of them were not used to apply or did not know what a testing criterion was (see Section IV).

VII. DISCUSSION

In this investigation we were able to analyze both students' performance and perceptions of testing practices in programming assignments. Experiment results suggest that making students responsible for writing their own test cases benefit their programming performance (Section V).

Although results were not statistically significant, it is interesting to note that students' perceptions matched with experiment results. All students agree that elaborating test cases indeed helped the programming activity (Q1 in Table X). Most of them think that working with instructor test cases also helps solving programming assignments (81.8%).

This positive effect on programming performance can be due to the fact that test design helps students better understand the problem to be solved in the assignment [10]. In order to select input values for test cases, they need to carefully analyze the problem domain, what forces them to think more critically about the assignment.

Regarding students' actual behavior to complete the assignments (in contrast with what they were asked to do), all students executed test cases while solving the assignment (Q3 in Table X). Besides, almost all of them applied the testing criteria when they were supposed to write their own test cases (Q2 in Table X).

Other interesting finding is that most of the subjects performed well with both test design approaches and some of them were able to improve their performance significantly by designing testing cases (figures 4 and 3).

These two kinds of effect could simply be related to the characteristics of the own subjects, which can have different

TABLE VII
EXPERIMENTAL DESIGN

	Instructor-provided Test Cases (IT)	**Student-written Test Cases** (ST)
Assignment 2A	Group 1	Group 2
Assignment 2B	Group 2	Group 1

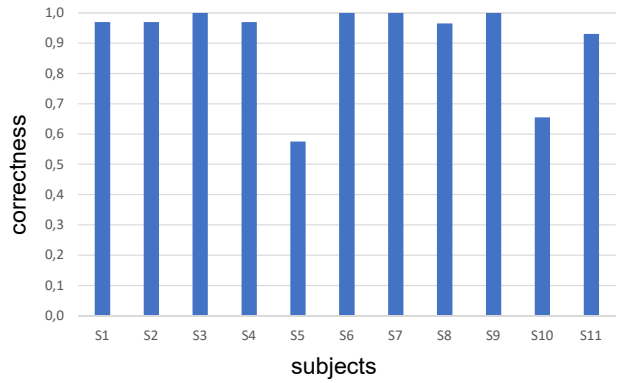

Fig. 3. Individual results for student-written test cases (ST)

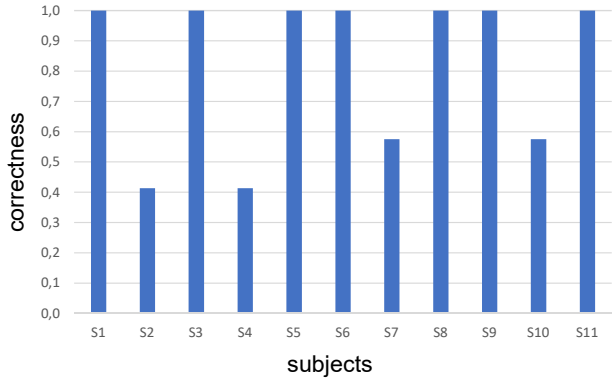

Fig. 4. Individual results for instructor-provided test cases (IT)

TABLE VIII
MEAN AND STANDARD DEVIATION FOR CORRECTNESS

	Instructor-provided Test Cases	**Student-written Test Cases**
Mean	81.63%	91.24%
Std Dev.	25.99%	14.94%

TABLE IX
MEAN FOR CORRECTNESS SEPARATED BY GROUPS

	Instructor-provided Test Cases	**Student-written Test Cases**
Group 1	76.55%	89.70%
Group 2	85.86%	92.53%

aptitudes. However, the test design task could be considered as a possibility to help turn ineffective novice programmers into effective ones [29], [30].

Most of the subjects were freshman or sophomore ($54,54\%$ – Table II), and most of them were able to apply the testing criteria without major difficulties (72.7% – Table X). This result suggests that the material about testing criteria had a difficulty level that most of them were able to follow. However, there were attending students from all levels, and there is the need to evaluate this issue with a more homogeneous sample.

In general, results show that students recognized the importance of designing test cases and that they can do it without major difficulties, even in the first-year level. This was an interesting result considering that software testing is not an

TABLE X
SURVEY RESPONSES (N=11)

Q1. *Did the test cases help you implement the solution code?*	
Yes, for both assignments	81.8% (9)
Yes, just for the assignment with instructor test cases	0% (0)
Yes, just for the assignment that I wrote the test cases	18.2% (2)
No, for both assignments	0% (0)
Q2. *While you wrote the test cases, did you apply the testing criteria?*	
Yes	90.9% (10)
No	9.1% (1)
Q3. *When did you execute the test cases?*	
Only after I have completed the solution code	45.5% (5)
During implementation, even with incomplete versions of the solution code	54.5% (6)
I did not execute the test cases	0% (0)
Q4. *Did you face difficulties while writing test cases by yourself?*	
Yes	27.2% (3)
No	72.7% (8)
Q5. *Do you intend to apply software testing in programming assignments from future computing courses?*	
Yes	100% (11)
No	0% (0)

easy subject, especially for freshman students. However, since we used a simple approach to represent test cases and to select input values, students were able to learn and apply it.

VIII. CONCLUSIONS AND FUTURE WORK

In this paper we investigated the test design task during the process of working on programming assignments. In studies similar to our investigation, the testing activity is usually seen from a holistic point of view. Conversely, in our study we decomposed the testing activity into individual tasks and investigated specifically the effect of the test design task.

We conducted an experiment and a survey with students that participated in a short course we offered about this subject. Results suggest that students' programming performance can be enhanced when they write their own test cases.

Moreover, students' perceptions matched experiment results. Students recognized the relevance of software testing as a supporting practice when completing the proposed assignments.

However, in order to enable students to write and improve their test suite, they need to be instructed on testing concepts. They need to learn how to select input values that will fully and effectively test their program.

This issue motivated us to instruct students on two testing criteria, *equivalence partitioning* and *boundary value analysis*. In this way, they were able to choose input values systematically, instead of adopting a trial-and-error approach [2]. If students are able to write an effective test suite, they will likely see software testing as a helpful practice for programming assignments.

Additionally, it was interesting to notice that the difficulty level of the testing criteria guidelines was adequate even for freshmen. Students were interested, paid attention and were engaged in using the testing practice as a support to the programming activity.

This experience showed the importance of instructing students on how to select appropriate input values and then elaborate test cases. They need background knowledge to perform test design if they are going to be responsible for it.

In our future work, we intend to investigate this effect in more homogeneous and larger samples of students. Isolating results for each level can help to design appropriate testing practices and instructional materials. Moreover, we would like to isolate the effect on programming performance before and after learning testing criteria, similar to the study conducted by Lemos et al. [18].

It is important to remember that, regarding testing practices in programming assignments, one size does not fit all. Students should have multiple and incremental testing experiences throughout computing courses [31], [20]. This means that several different testing approaches should be used throughout the CS curriculum, starting simpler and increasing the level of difficulty gradually as the students are able to cope with the associated learning load.

Therefore, this kind of investigation is relevant to help instructors design test practices to be integrated into programming assignments. In particular, the test design task is an important variation point for the instructor to consider while deciding how the adopted testing practice should be configured.

ACKNOWLEDGMENT

We would like to thank students that participated in the study. The authors also acknowledge Brazilian funding agencies FAPESP (grants 2014/06656-8 and 2016/17575-4), CAPES and CNPq for the financial support provided to this research.

REFERENCES

[1] H. B. Christensen, "Systematic testing should not be a topic in the computer science curriculum!" in *Proceedings of the 8th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '03)*. New York, NY, USA: ACM, 2003, pp. 7–10.

[2] J. C. Carver and N. A. Kraft, "Evaluating the testing ability of senior-level computer science students," in *24th IEEE-CS Conference on Software Engineering Education and Training (CSEE&T)*, May 2011, pp. 169–178.

[3] E. L. Jones, "Software testing in the computer science curriculum – a holistic approach," in *Proceedings of the Australasian Conference on Computing Education (ACSE '00)*. New York, NY, USA: ACM, 2000, pp. 153–157.

[4] S. H. Edwards, "Rethinking computer science education from a test-first perspective," in *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '03)*. New York, NY, USA: ACM, 2003, pp. 148–155.

[5] D. S. Janzen and H. Saiedian, "Test-driven learning: Intrinsic integration of testing into the CS/SE curriculum," in *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '06)*. New York, NY, USA: ACM, 2006, pp. 254–258.

[6] E. Barbosa, J. Maldonado, R. LeBlanc, and M. Guzdial, "Introducing testing practices into objects and design course," in *16th Conference on Software Engineering Education and Training (CSEE&T)*, March 2003, pp. 279–286.

[7] E. Barbosa, M. Silva, C. Corte, and J. Maldonado, "Integrated teaching of programming foundations and software testing," in *Annual Frontiers in Education Conference (FIE)*, Oct 2008, pp. S1H–5–S1H–10.

[8] D. Janzen and H. Saiedian, "Test-driven learning in early programming courses," in *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '08)*. New York, NY, USA: ACM, 2008, pp. 532–536.

[9] P. Ammann and J. Offutt, *Introduction to Software Testing*, 2nd ed. New York, NY, USA: Cambridge University Press, 2017.

[10] C. Fidge, J. Hogan, and R. Lister, "What vs. how: Comparing students' testing and coding skills," in *Proceedings of the Fifteenth Australasian Computing Education Conference (ACE '13)*. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2013, pp. 97–106.

[11] S. H. Edwards, "Using software testing to move students from trial-and-error to reflection-in-action," in *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '04)*. New York, NY, USA: ACM, 2004, pp. 26–30.

[12] J. Spacco and W. Pugh, "Helping students appreciate test-driven development (TDD)," in *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '06)*. New York, NY, USA: ACM, 2006, pp. 907–913.

[13] C. Desai, D. Janzen, and K. Savage, "A survey of evidence for test-driven development in academia," *SIGCSE Bulletin*, vol. 40, no. 2, pp. 97–101, Jun. 2008.

[14] D. S. Janzen and H. Saiedian, "A leveled examination of test-driven development acceptance," in *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 719–722.

[15] 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 (ACE '11)*. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2011, pp. 113–118.

[16] V. Isomöttönen and V. Lappalainen, "CSI with games and an emphasis on tdd and unit testing: Piling a trend upon a trend," *ACM Inroads*, vol. 3, no. 3, pp. 62–68, Sep. 2012.

[17] I. Utting, A. E. Tew, M. McCracken, L. Thomas, D. Bouvier, R. Frye, J. Paterson, M. Caspersen, Y. B.-D. Kolikant, J. Sorva, and T. Wilusz, "A fresh look at novice programmers' performance and their teachers' expectations," in *Proceedings of the ITiCSE Working Group Reports Conference on Innovation and Technology in Computer Science Education-working Group Reports (ITiCSE -WGR '13)*. New York, NY, USA: ACM, 2013, pp. 15–32.

[18] 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*, pp. –, 2017.

[19] S. Frezza, "Integrating testing and design methods for undergraduates: teaching software testing in the context of software design," in *32nd Annual Frontiers in Education (FIE)*, vol. 3, Nov 2002, pp. S1G–1–S1G–4 vol.3.

[20] M. Wick, D. Stevenson, and P. Wagner, "Using testing and JUnit across the curriculum," in *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '05)*. New York, NY, USA: ACM, 2005, pp. 236–240.

[21] J. Collofello and K. Vehathiri, "An environment for training computer science students on software testing," in *Frontiers in Education, 2005. FIE '05. Proceedings 35th Annual Conference*, Oct 2005, pp. T3E–6.

[22] R. Agarwal, S. H. Edwards, and M. A. Pérez-Quiñones, "Designing an adaptive learning module to teach software testing," in *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '06)*. New York, NY, USA: ACM, 2006, pp. 259–263.

[23] S. Elbaum, S. Person, J. Dokulil, and M. Jorde, "Bug hunt: Making early software testing lessons engaging and affordable," in *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 688–697.

[24] H. B. Christensen, "Experiences with a focus on testing in teaching," in *Reflections on the Teaching of Programming*, ser. Lecture Notes in Computer Science, J. Bennedsen, M. E. Caspersen, and M. Kölling, Eds. Springer Berlin Heidelberg, 2008, vol. 4821, pp. 147–165.

[25] V. Thurner and A. Bottcher, "An "objects first, tests second" approach for software engineering education," in *IEEE Frontiers in Education Conference (FIE)*, Oct 2015, pp. 1–5.

[26] M. E. Delamaro, J. C. Maldonado, and M. Jino, *Introdução ao teste de software (in portuguese)*. Elsevier, 2007.

[27] N. Juristo and A. M. Moreno, *Basics of Software Engineering Experimentation*, 1st ed. Springer Publishing Company, Incorporated, 2001.

[28] C. Wohlin, P. Runeson, M. Hst, M. C. Ohlsson, B. Regnell, and A. Wessln, *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated, 2012.

[29] S. H. Edwards, J. Snyder, M. A. Pérez-Quiñones, A. Allevato, D. Kim, and B. Tretola, "Comparing effective and ineffective behaviors of student programmers," in *Proceedings of the Fifth International Workshop on Computing Education Research Workshop*, ser. ICER '09. New York, NY, USA: ACM, 2009, pp. 3–14.

[30] J. Carter, S. White, K. Fraser, S. Kurkovsky, C. McCreesh, and M. Wieck, "ITiCSE 2010 working group report motivating our top students," in *Proceedings of the 2010 ITiCSE Working Group Reports*, ser. ITiCSE-WGR '10. New York, NY, USA: ACM, 2010, pp. 29–47.

[31] E. L. Jones, "Integrating testing into the curriculum – arsenic in small doses," in *Proceedings of the Thirty-second SIGCSE Technical Symposium on Computer Science Education (SIGCSE '01)*. New York, NY, USA: ACM, 2001, pp. 337–341.