Integration of Software Testing to Programming Assignments: An Experimental Study

Gustavo M. N. Avellar, Rogério F. da Silva, Lilian P. Scatalon,
Stevão A. Andrade, Márcio E. Delamaro, and Ellen F. Barbosa
Institute of Mathematics and Computer Science (ICMC)
University of São Paulo (USP), São Carlos, SP – Brazil
Email: {gustavo.avellar, rogeriofrr, lilian.scatalon}@usp.br, {stevao, delamaro, francine}@icmc.usp.br

***Abstract*—This Research Full Paper reinforces that Software Testing can be a helpful practice to students while working on programming assignments. Considering Software Testing as a process, the testing activity is composed by a sequence of steps. When students write and submit their own test cases, they are responsible for the test design, automation, execution, and evaluation. Otherwise, instructors can provide ready-made test suites, needing only to execute it and evaluate it. In this scenario, we conducted an experimental study to investigate how the programming performance of students from the Computer Science area is affected when Software Testing is integrated with programming. We proposed three different approaches: (i) ad hoc programming; (ii) programming and testing by writing the test suite; and (iii) programming and testing with a ready-made test suite. We assessed students' programs in terms of correctness, measured by the pass rate of the reference test suite. Results indicate that students had a lower performance with ad hoc programming in comparison with both approaches involving Software Testing. On the other hand, using ready-made test cases raised better results than when students had to write their own test cases. We also assessed students' attitudes towards testing by means of a survey.**

***Keywords*—Software testing, Programming assignments, Ad hoc testing, Student-written tests, Instructor-provided tests.**

I. Introduction

Over the years, Software Testing has been usually addressed as a separate topic in Computer Science programs [1]. However, even when this topic is addressed, students' testing skills might not be properly developed, due the fact that students usually understand the theory behind Software Testing, but do not practice it properly during programming assignments. Therefore, it is still possible to identify Computer Science students who are not able to completely test even simple programs, as well as students that do not learned the value of testing in introductory programming courses, or at any point of their undergraduate or graduate programs [2], [3].

However, the same Computer Science programs that do not focus to develop students' testing skills, provides different courses centered on the teaching and learning process of programming concepts, with several opportunities available to students to practice and enhance their programming skills. This plurality of teaching and learning approaches and emphasis does not occur when the topic in question is Software Testing, in contrast with industry, that typically spends half or more of software project resources on testing [2]. Hence, this slightest emphasis on the Software Testing activity has motivated the development of several studies that integrate testing and programming throughout the Computer Science curriculum without reducing any topic coverage [4], [5], [6], [7].

A common approach for this integration is providing ready-made test suites, or test cases, to students, so they can be responsible only for its execution, enabling them to correct their codes according to the feedback provided by the results of the tests execution [8]–[10]. Another possible scenario is when students create their own test suites. In this case, students must be able to select the appropriate values to construct the test cases. Thus, it is necessary to have at least the basic knowledge regarding testing criteria, which are used to decide which entries should be selected to test the operation of the program [11], [12].

In a related perspective, different studies also investigated the more often method used by students to test their programs, which is mostly based on intuition, i.e., ad hoc or exploratory testing [13]–[15]. One approach to ad hoc testing, or ad hoc programming, is to consider it as improvisation, since it clearly distinguishes from the formal testing paradigms that emphasize systematic approaches based on well-defined and standardized processes [13]. Even though there is evidence of similar efficiency of failure identifying between ad hoc testing and formal testing [14], [16], with regard to reliability, ad hoc testing can be considered relatively weak and is sometimes viewed as an unsatisfactory approach to Software Testing when compared with formal Software Testing methods, which usually lead to more confidence throughout the development process [13]

In this context, our goal with this paper is to describe our quantitatively data-driven investigation on the relationship between the integration of Software Testing to the programming activity and the students performance in programming assignments. To this end, we conducted an experimental study inspired in Scatalon et al. [17] during a class of Software Testing and Inspection course for Computer Science area students (undergraduate and graduate level) of the Institute of Mathematics and Computer Science (ICMC) of the University of São Paulo (USP). The experimental study occurred at the end of the semester and it was not part of the students' course evaluation. In our study, we proposed to the students to solve three different programming tasks using three different approaches. We evaluated the correctness of the program developed by students using Software Testing methods in the following circumstances: (i) ad hoc programming; (ii) programming and testing with students writing their own test suite; and (iii) programming and testing with a ready-made test suite provided by the instructor. Therefore, the research question we set out to investigate is related to the influence that the integration of Software Testing to the programming activity promotes in the students' performance while developing programming assignments.

Additionally, we assessed students' attitudes and perceptions towards the integration of Software Testing with programming by means of a survey. The results presented indications that both approaches that integrate formal Software Testing to the programming activity provide programs with higher quality compared to the approach that does not include formal testing during programming. The results also presented that the approach which the instructor provided a ready-made test suite achieved the best results of code correctness.

The remainder of the paper is organized as follows. In Section II, the theoretical basis of this study is presented. Section III describes how our investigation relates to other studies from the literature. Details on the experimental design are given in Section IV. The obtained results are presented in Section V and discussed in Section VI. Section VII explains the threats to the validity of this experimental study. Finally, our conclusions and perspectives for future work are presented in Section VIII.

II. Background

A standard definition for ad hoc[1] is "for the particular end or case at hand without consideration of wider application". Thus, ad hoc testing, or exploratory testing, can be described as an exploratory test that is expected to run only once, or as an activity that simultaneously includes the processes of learning, test design, and test execution of a program. In others terms, ad hoc testing is any testing activity in which the tester actively controls the design of the tests, how the tests are performed, and uses information gained while testing to design new and improved test cases [13], [14].

Although ad hoc is usually adopted as a synonymous to careless work, all testing performed by programmers is exploratory in some degree. However, when applied by a expert programmer, ad hoc testing can be highly effective, since one of the more adequate applications of ad hoc testing is for the discovery of defects, which are commonly due to the missing of entire classes of test cases. In fact, the issues that need to be addressed in order to choose the appropriate Software Testing approach are highly contextual, but Software Testing learning means more than just ensuring the proper functioning of a program [14].

Studies demonstrate that the learning of Software Testing should start as early as possible in Computer Science programs, and preferably together with introductory programming courses [7], [11], [18], [19]. Therefore, as this integration reinforces the reflection of the problem to be solved, it might lead to better results in students' programming assignments, increasing quality of code and students' understanding and interpreting skills [6], [20].

[1]https://www.merriam-webster.com/dictionary/ad%20hoc

The integration of Software Testing into programming courses of Computer Science programs can be an efficient way to prevent students from engaging in practices that are negative to programming learning. Particularly for students who are still in introductory disciplines, it is common to think that if the program produces the expected output or if the compiler did not point out problems, the code is fully correct. Hence, Computer Science students will be more successful in learning programming if there is a shift from a trial-and-error approach to a reflection and action approach [21].

Based on mechanisms to validate the program, students can use both test design process, their execution, and evaluation as support to the program development. The purpose of this approach is to help students writing better programs and making the Software Testing activity a standard practice when developing programs, since Software Testing can not be effectively taught and evaluated only through lectures, without any practice. Thus, it is necessary to find ways to encourage or require students to get in the habit of properly test their programming projects [12].

In the Test-Driven-Development (TDD), the test cases are usually designed by students themselves [12], [22], [23]. Its principles have been discussed over the years and have a strong connection with Extreme Programming (XP) software development methodology, which considers Software Engineering practices in its procedures, such as Software Testing, that is a fundamental practice for the development of quality software [24], [25]. The use of TDD imposes a certain order to execute the tasks that must be iteratively repeated, requiring first the creation of test cases, followed by the tests execution, and the program correction. Even though TDD can be used in educational environments, it is not possible to generalize the methodology in the same way that it is defined for every target audience. Therefore, for both freshmen and seniors students, and in different programs of Computer Science, it is necessary to study the context of application of the TDD methodology to make it a positive integration between Software Testing and programming [23].

To define the integration approach between Software Testing and programming it is important to consider the tasks that compose the Software Testing activity [26]:

- **test design**: input values are selected to create test cases, aiming to successfully test the program;
- **test automation**: the values are placed into executable code;
- **test execution**: running the code together with the test cases and obtaining results; and
- **test evaluation**: evaluating results and conducting proper actions.

In test design, students may be required to create their own test cases. However, it is also possible to provide ready-made test suites in such a way that it is only necessary to the students to execute them and evaluate the obtained feedback in order to correct the program. Test automation is related to the format that test cases will take. The simplest format for test suites is the manual test approach, with no automated tools, whereas the involved inputs and the expected results are arranged in the form of a table. Students manually enter the inputs, observe the program outputs and compare them with the expected results listed in the table. Despite simplicity, this can increase students' confidence in correcting their code.

To execute the tests, students could use instructions in the format `if (expected != actual)`. Depending on the programming language used and on the programming experience of students, it is possible to use Software Testing automation tools, such as JUnit[2], which considerably facilitates the testing activity and its evaluation, as we adopted in this experimental study (detailed in Section IV). Both test execution and evaluation can be directly integrated into programming activities, enabling students to execute test cases while programming to improve their code based on the obtained results.

III. Related Work

Several studies investigated the integration of Software Testing to the programming activity, as well as assessed and compared the efficiency of ad hoc programming and formal Software Testing methods based on test cases.

In order to compare ad hoc programming and formal test case based methods, the study conducted by Itkonen et al. [16] evaluated a sample of advanced software engineering students when performing manual testing on an application with seeded defects. The authors' intent was to investigate the difference in the number of detected defects between testing with and without ready-made test suites by the same sample of students. Comparing both approaches, they found no significant differences in defect detection efficiency between ad hoc programming and test case based methods. Although, the authors state that the design and execution of test cases can provide to the students many other benefits besides program defect detection efficiency.

With similar results, Prakash and Gopalakrishnan [27] also performed a comparison between test case based methods and ad hoc programming using two groups of students with similar background. In this case, the whole student sample had no background on Software Testing, hence the authors decided to give checklists to the ad hoc programming group, while the other group received ready-made test cases. Their results revealed that, even though the test cases were as detailed as the checklist, test cases did not help too much, since both groups found similar amount of defects. Additionally, the students that followed ad hoc programming were able to find defects related to diverse quality criteria, while defects found by students that used test cases were clustered in limited quality criteria.

[2]https://junit.org

In the studies that integrate Software Testing with the programming activity, the students usually perform at least the tasks of tests execution and evaluation. The tasks of test design and test automation are associated with actively writing test cases and present variations among the covered studies. Whalley and Philpott [8] and Utting et al. [10] provided ready-made test cases to the students, leaving them responsible only for the testing tasks of execution and evaluation. In the studies conducted by Edwards [21], Spacco and Pugh [12], and Janzen and Saiedian [11], all the testing activities are at some point held by the students. In particular, Isomöttönen and Lappalainen [9] used an approach in which the authors provided ready-made test cases, but also allowed students to write their own test cases for extra points. All of these studies showed programming learning enhancement at some point when adopting Software Testing integrated with the programming activity.

Considering that the instructor can and should provide some guidelines for the testing activity, students do not need to be actively involved in all testing tasks. Since students designing test cases or receiving ready-made test cases can both offer learning benefits. Thus, the decision about which approach use is directly related to the context of students and the intent of the course [10].

The study conducted by Scatalon et al. [17], described an experimental study that investigate and compare the two testing-related approaches used in programming activities: (i) test cases designed by students; and (ii) ready-made test cases provided by the instructor. In a short course, the authors presented to the students the necessary concepts of Software Testing and evaluated both Software Testing integration approaches. The results showed that students' performance was better in the approach in which they were committed to the creation of the test cases.

In order to gather more evidence on this topic, in this experimental study we evaluated the two approaches of Software Testing integration with the programming activity and ad hoc programming, in which we considered the exploratory testing approach, i.e., students evaluated the program based only on their intuition. To our knowledge, there are no studies that compare the integration efficiency or investigate students' performance in programming assignments when they design test cases, receive ready-made test cases, and test the program using an ad hoc approach.

IV. Experimental Study

Next, we describe the details related to the planning of our experimental study and its execution. Concerning to the experimental study planning, as presented in Figure 1, we followed the guidelines proposed by Wohlin et al. [28]. The last phase of planning, Validity Evaluation, is discussed in Section VII.

A. Goal Definition

The integration of Software Testing with the programming activity can be done in order to add value to the development

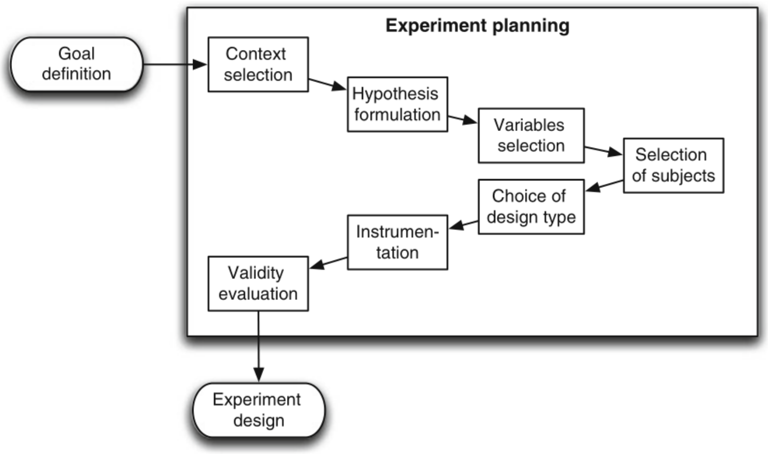

Fig. 1. Experimental study planning phases [28]

process of a program, enabling students to benefit from the practice of testing. Therefore, it is appropriate to investigate the benefits of this integration to the performance of students during programming assignments.

In this context, the objective of this experimental study is to analyze the performance of students in programming activities, considering different approaches of integration with Software Testing, for evaluation purposes, regarding the correctness of the source code developed, from the point of view of the researcher, in the context of undergraduate and graduate students from the area of Computer Science.

B. Research Question and Hypothesis Formulation

The question we set out to explore in the experimental study is: *how the Computer Science students' performance in programming is affected when programming is integrated with Software Testing?* In this context, we considered three different approaches to the students conduct the programming activity, that are described as follows:

- **A1 – ad hoc programming:** in this activity, students perform a programming task according to a certain specification, and do not perform formal Software Testing;
- **A2 – programming and testing by writing their own test suite:** in this activity students develop a different programming task, design, execute, and evaluate their own test cases;
- **A3 – programming and testing with a ready-made test suite provided by the instructor:** in this activity, students perform another programming task, execute, and evaluate test cases provided by the instructor.

In order to verify the students' programming efficiency in the referred approaches, the presented research question was converted into hypotheses that were verified through the conducting of the experimental study. The hypotheses to be evaluated are presented in Table I and described as follows:

- **Null hypothesis (H_0):** determines that there is no difference in students' programming performance, considering all three approaches.
- **Alternative hypothesis 1 (H_1):** determines that students' performance is best when they design, execute, and evaluate test cases to develop the program.
- **Alternative hypothesis 2 (H_2):** determines that students' performance is best when they receive test cases provided by the instructors, being responsible only for their execution and evaluation.

C. Variables Selection

During the experimental study, students were asked to use three different integration strategies between testing and programming activities. Thus, the integration approach between activities was varied, while the following variables remained constant throughout the experimental study: (i) the students received specifications of three different programming activities to be developed, once at time, using Java programming language; (ii) all students used Eclipse[3] as the integrated development environment (IDE) for development of the activities; (iii) the tests, when performed, were automated and performed using JUnit tool; and (iv) students were also responsible for performing and evaluating test cases, in the approaches that Software Testing was performed.

Thus, the experimental study has an independent variable: the integration approach between programming and testing activities. With three different treatments: the approaches A1, A2, and A3. Since the purpose of the study is to evaluate students' performance in programming activities, the dependent variable is the correctness of the source code for each one of the three activities submitted by the students.

D. Selection of Subjects

The subjects selected to participate in the experimental study were students of the Software Testing and Inspection course for undergraduate and graduate students at ICMC–USP. The goal of the course is to provide an overview of the area of software verification, validation, and testing (VV & T) for

[3]http://www.eclipse.org

TABLE I
HYPOTHESIS FORMULATION

	Formal hypotheses
H_0	$\text{Correctness}_{A1} = \text{Correctness}_{A2} = \text{Correctness}_{A3}$
H_1	$\text{Correctness}_{A1} < \text{Correctness}_{A3} < \text{Correctness}_{A2}$
H_2	$\text{Correctness}_{A1} < \text{Correctness}_{A2} < \text{Correctness}_{A3}$

Computer Science students of different levels, emphasizing the teaching and learning of strategies, methods, criteria and tools associated with Software Testing and Inspection that can be applied during software development.

The experimental study was carried out at the end of the first semester of 2018 and it was not part of the students' course evaluation. In total, 20 students participated, which 19 authorized the use of their data through our Term of Data Use Consent. Through the application of a characterization form, it was possible to determine some characteristics of the subjects.

Most of the students were enrolled in Computer Engineering undergraduate program (42.10%), as presented in Table II. Regarding previous habits related to testing activities, according to Table III, most students stated that they test their programs at least a little (57.90%).

TABLE II
DISTRIBUTION OF SUBJECTS PER PROGRAM

Program	%
Computer Engineering	42.10% (8)
Computer Science	31.58% (6)
Information Systems	10.53% (2)
Master's in Computer Science	10.53% (2)
PhD in Computer Science	5.26% (1)
TOTAL	100% (19)

TABLE III
HABITS OF SOFTWARE TESTING PRIOR TO EXPERIMENT

Q: Do you test the programs you write?	
Answer	**%**
Yes, I always try at least a little	57.90% (11)
Yes, if I have time	21.05% (4)
Yes, I always try to test it as much as possible	21.05% (4)
I do not know what it means to test a program	0% (0)
It is not necessary	0% (0)
TOTAL	100% (19)

E. Choice of Design Type

The type of experimental study we defined is based on a factor with more than two treatments, since the independent variable is the integration approach between programming and Software Testing, and its three treatments are the approaches A1, A2, and A3.

Figure 2 shows an overview of the activities performed in the experimental study. Initially, in conjunction with a characterization questionnaire, students were required to complete a consent form accepting to participate in the experimental study and authorizing the analysis and publication of the data collected. Then, we provided instructions for the tasks, they developed them, and filled in a survey with their opinions regarding the context of the experimental study.

F. Instrumentation

We provided to the students only the basic structure of the class `Time`, developed in Java, which code can be seen next:

```
public class Time {

        private int hours;
        private int minutes;
        private int seconds;

        public Time() {
                hours = 0;
                minutes = 0;
                seconds = 0;
        }
        public void tick() {
        }
        public Time addTime (Time time) {
                return null;
        }
        public Time subTime (Time time) {
                return null;
        }

}
```

The purpose of this class is to perform operations over time. Since it is a common context for everyone, such operations were suggested by Utting et al. [10], and also used in another experimental study in the context of Software Testing and programming assignments [17]. Thus, the following instructions were provided:

- `Time` class objects encapsulate a time value in European style (24 hours). The class has three attributes to represent time, which are hours, minutes, and seconds. The possible range of values is from 00:00:00 (midnight) to 23:59:59 (a second to midnight). The class also has the statement of three methods, which should be developed by students at three different times using the three approaches of integration between programming and testing.
- **Task 1**: implement the method `tick()` – add a second to the class time and make the necessary adjustments to the attributes that represent hours, minutes, and seconds. This activity uses the A1 approach, i.e., the method must be implemented according to the specification and should not be formally tested;
- **Task 2**: implement the method `addTime(Time time)` – add the value passed as a parameter to the class time, make the necessary adjustments, and return its value. This activity uses the A2 approach, students were responsible for the whole testing process.
- **Task 3**: implement the method `subTime(Time time)` – subtract from the passed value as parameter the

time represented in the current class, make the necessary adjustments and returns its value. This activity uses the A3 approach, students use the test suite provided by the instructor.

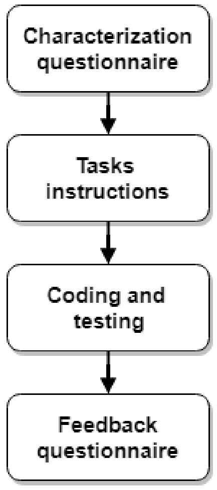

Fig. 2. Experimental study activities overview

As previously discussed, Software Testing has been usually addressed as an isolate topic in Computer Science programs. Therefore, students usually test and evaluate their programs based only on their intuition. In this case, it is to be expected that ad hoc programming (A1) presents lower results when compared to the other approaches (A2 and A3). Thus, we decided to use ad hoc programming as the first task of the experimental study with the objective of firstly obtain data from the students common habit when programming, to later compare it with the approaches that involve Software Testing.

In order to evaluate the students' performance in each one of the three tasks, the correctness of the source code was measured by means of the pass rate of our reference test suite. This metric was calculated by dividing the number of test cases in which the activity was successfully performed, by the total number of test cases designed by instructors for that activity.

V. Results

The experimental quantitatively data from the execution phase is the input to the analysis and interpretation of results. After collecting the experimental data, our intent is to be able to draw conclusions based on this data. According to the guidelines proposed by Wohlin et al. [28], we divided the quantitative analysis of results between descriptive statistics and hypothesis testing. The objective of descriptive statistics is to provide different visual interpretative forms of the obtained results, while hypothesis testing aims to guarantee that the obtained results can be generalized and were not obtained by chance. Due to the data format and to the variables and treatments we established to analyze in this experimental study, we applied the Kruskal-Wallis non-parametric hypothesis test [29] with one factor and more than two treatments.

Although the initial sample was composed of 19 students, 3 of them were characterized as outliers, being removed for the statistical analyzes. The removal of these students from the sample is due to the condition of their submitted projects, that were unfinished or not submitted. Hence, the final sample of this experiment was composed of 16 students.

The individual correctness of each student related to each approach is illustrated in Figure 3. The experimental study took 2 hours to be conducted. We determined 30 minutes to instructions and forms filling and 30 minutes to complete each programming activity.

Only 4 students had 100% success in all activities (S1, S2, S4, and S6). It is interesting to note that in addition to the students who achieved 100% accuracy in the code in all three activities, more than half of the sample maintained similar performance in all activities (S3, S8, S10, S11, S12, S13, S14, S15, and S16). It is also important to mention that the highest correctness percentages were found when there were integrated programming tests (S4, S8, S10, S11, S14, and S15). A greater variability of performance could be observed in only 2 students (S5 and S9). Finally, only one student obtained a better score when there was no integration of tests during programming (S7).

These results are summarized in Table IV, which presents the average and standard deviation for each approach. Comparing averages, the best results achieved belong to the A3 approach, in which we provided ready-made test suite and students only performed the tests and evaluated the results.

TABLE IV
Average and standard deviation of code correctness for each approach

	Average	**Standard deviation**
A1	71.13%	25.52%
A2	89.36%	15.71%
A3	92.57%	19.65%

With regard to the validation of conclusions of this experimental study, we used the R[4] environment to perform the Kruskal-Wallis non-parametric hypothesis test, with significance level equal to 0.05. The test result was $p = 0.01937$, presenting sufficient evidence to reject the null hypothesis. Based on the result of Kruskal-Wallis, that showed that there is significant difference between the samples, and on the analysis of descriptive statistics, we concluded that there is enough evidence to affirm that the approaches that integrate programming and Software Testing (A2 and A3), generate higher quality code in comparison to the approach that does not include formal Software Testing (A1).

This perspective can also be seen in Figure 4, where it is possible to see a large dispersion in the data for approach A1, which does not include testing during programming, with its lower limit close to 50% and the median value of 57.14%. Regarding approach A2, in which students designed

[4]http://www.r-project.org

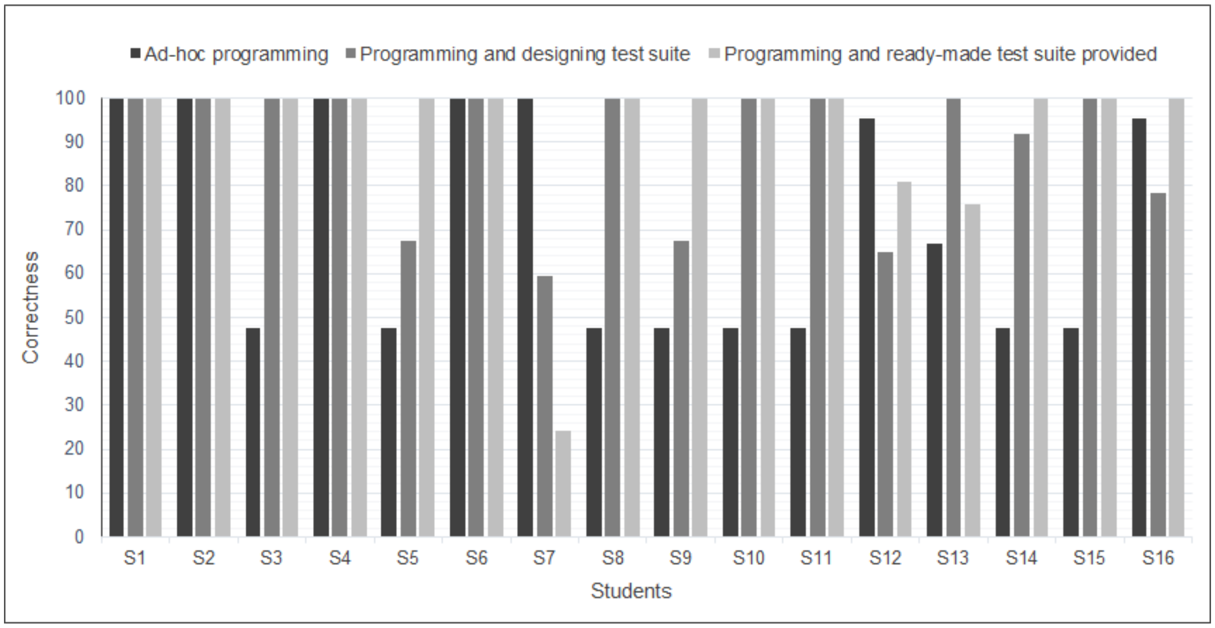

Fig. 3. Individual performance per task

test cases, it presented a median of 100% and a lower limit close to 60%, with the mostly concentration of data above it. Finally, approach A3, in which test cases were provided by the instructor, also obtained a median of 100% and presented only 3 sample subjects that did not reach 100% of correctness.

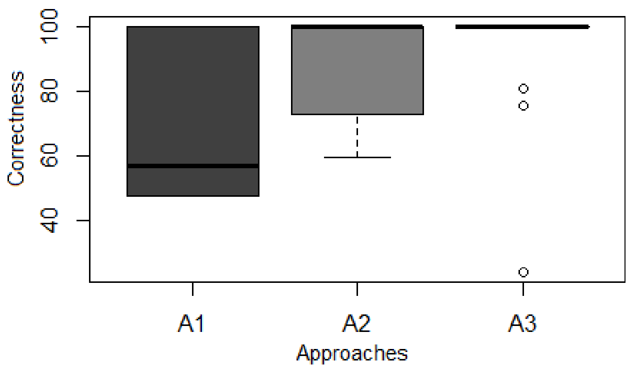

Fig. 4. Box plot

VI. Discussion

With this experimental study, it was possible to analyze both students' performance and perception with respect to ad hoc programming and two approaches of Software Testing and programming integration.

The result of the hypothesis testing in conjunction with the analysis of the descriptive statistics produced evidences that the integration of tests into the programming activity is beneficial, more specifically, in relation to the two approaches we adopted (A2 and A3), in which the students designed test cases and received ready-made test cases from the instructor.

Although there is little difference between the correctness results for the A2 and A3 approaches, the statistical analysis led us to the acceptance of the H_2 alternative hypothesis, which states that the correctness of the code implemented in the scenario that the instructor provides ready-made test suite is greater than when the students have to design the test suite (A2), and even higher than when there is no formal Software Testing methods integrated with the programming activity, i.e., ad hoc programming (A1).

Therefore, regardless of the testing approaches, and considering that the task contained in A1 was the less complex, ad hoc programming still presented lower correctness results, which may be due to there is no proper support to validate the program's execution. However, because of the establishment of a closer basis to what is common to students daily activities, ad hoc programming is an essential approach when comparing their performances.

Additionally, the students' opinion is in the same direction of the statistical results, since most part of the students agreed that Software Testing together with the programming activity helped in the development of the programming tasks, even without knowing the results (Q1 at Table V). Although it was not answered by all of the students (87.5%) that they would use Software Testing in programming activities in the future (Q3), no student had difficulty in designing test cases (Q2).

VII. Threats to Validity

The threats to validity can be separated into four different types, being prioritized according to the study objectives [28], [30]. In this experimental study, we focused on the achievement, in decreasing order, of the following validities: Internal, External, Construct, and Conclusion Validity.

TABLE V
SURVEY RESULTS

Q1: In the last two tasks did the test cases help implement them?	
Yes, for both tasks	8 (50.00%)
Yes, but just for task 2	4 (25.00%)
Yes, but just for task 3	3 (18.75%)
No, for both tasks	1 (6.25%)
Q2: Did you have any difficulties designing your test cases?	
Yes	0 (00.00%)
No	16 (100.00%)
Q3: Would you use Software Testing on future programming activities?	
Yes	14 (87.50%)
No	2 (12.50%)

Internal Validity refers to what can affect the independent variables of the study regarding causality, without the researcher's due knowledge. In the context of our study, threats to Internal Validity are related to the instrumentation of independent variables. In order to mitigate these validity threats, the study was conducted at once with the whole sample of participants in a single day, during a class at the end of the Software Testing and Inspection course for students (undergraduate and graduate level) of the Computer Science area. This course aims to approach the introductory concepts of Software Testing, which ensures that participants have at least minimal knowledge of the content required for the development of the experimental tasks, as well as the students' knowledge level on Software Testing are somehow similar. In addition, we ensure that each task was performed by the students only for the amount of time we stipulate, there were no interruptions during the conduction of the experiment, and all students present in the laboratory were performing only the tasks of the experiment. Further, we used an automated testing tool to acquire the correctness of the programs developed by the students. Although eliminating all learning effects is not completely possible, we strive to minimize it by providing three different programming tasks to the students. The same three tasks were developed by all students at the same time and in the same order, increasing the problems from the first activity to the third.

External Validity is related to whether the obtained results can be generalized to a different context, as well as to what extent the findings are of interest to other people outside the experimental settings. In order to avoid threats to External Validity, we selected undergraduate and graduate students from the Computer Science area with experience in programming and enrolled in a Software Testing course. Additionally, the programming language we chose for the development of the programming tasks and tests were already known and commonly used by the students. Further, we strive to make the experimental environment as usual as possible in relation to a class of the Software Testing and Inspection course. Besides, the settings used to conduct this study were extracted from a similar experimental study [17], due to the fact that it proposes a model that can be extended to different contexts, as it actually was performed in this study.

Construct Validity is related to the experimental design and to what extent the results correspond to the theory behind it, as well as to the social factors involved with the experiment. In order to mitigate the design threats and make the experimental study sufficiently clear, the guidelines for experimentation in Software Engineering proposed by Wohlin [28] were adopted, both to plan the experimental study as to conduct its operation. In addition, students did not know that their performance would be compared for each established treatment, which is important to avoid hypothesis guessing and attempts to be more aware of errors. Moreover, we defined correctness as the main comparison measure for the students' performance and also assessed students attitudes towards Software Testing during the programming activity, to enable us to have different types of measures that can be compared against each other.

Finally, regarding Conclusion Validity, which aims to guarantee the reliability of the conclusions drawn through the conducting of the study, the data collected were characterized using descriptive statistics and analyzed by hypothesis testing, i.e., the hypothesis are statistically evaluated. With respect to the assumptions of statistical tests, the hypothesis test suitable for this study was the Kruskal-Wallis. As mentioned, we were able to achieve statistical significance and reject the hypothesis that do not represent the obtained results.

VIII. CONCLUSIONS AND FUTURE WORK

In this experimental study, we investigated the relationship between the programming activity and the Software Testing activity. The intent was to analyze the correctness of the source code developed by the students when adopting different approaches of programming and Software Testing.

The first task we proposed to the students develop was concerned to ad hoc programming. Even though it was the simplest one, it obtained the lowest pass rate. The following two tasks were more complex, since they involved conditions that were not met in the first one, but combined different approaches of Software Testing. Thus, we conclude that students' performance in programming can be enhanced when they execute and evaluate test cases provided by instructors, as well as when they design their own test cases.

In addition, the students' view of Software Testing, obtained through the survey, is aligned to the results of the experimental study. Students recognized the relevance of Software Testing activity as a resource that helps in understanding what should be developed in code.

As future work, since the short term effect observed in this study is significant, we also intent to investigate the long term effect of Software Testing integration with the programming activity in the students' academic performance. Additionally, we aim to investigate the benefits that integration of other categories of Software Testing can bring to the quality of source code produced by students, such as structural testing and mutation testing.

Acknowledgments

We acknowledge the financial support of the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001/Procad 071/2013; the Brazilian National Council for Scientific and Technological Development (CNPq – process 134045/2018-1); and the São Paulo State Research Foundation (FAPESP – processes 2017/19492-1 and 2014/06656-8). We also thank the students that participated in this experimental study, the instructors that contributed with the execution of the study, and the reviewers for their comments.

References

[1] H. B. Christensen, "Systematic testing should not be a topic in the computer science curriculum!" in *ACM SIGCSE Bulletin*. ACM, 2003, pp. 7–10.

[2] T. Shepard, M. Lamb, and D. Kelly, "More testing should be taught," *Commun. ACM*, vol. 44, no. 6, pp. 103–108, Jun. 2001. [Online]. Available: http://doi.acm.org/10.1145/376134.376180

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

[4] E. L. Jones, "Software testing in the computer science curriculum–a holistic approach," in *Proceedings of the Australasian conference on Computing education*. ACM, 2000, pp. 153–157.

[5] 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*. ACM, 2003, pp. 148–155.

[6] D. S. Janzen and H. Saiedian, "Test-driven learning: intrinsic integration of testing into the cs/se curriculum," in *ACM SIGCSE Bulletin*. ACM, 2006, pp. 254–258.

[7] C. Desai, D. S. Janzen, and J. Clements, "Implications of integrating test-driven development into cs1/cs2 curricula," in *Proceedings of the 40th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '09. New York, NY, USA: ACM, 2009, pp. 148–152. [Online]. Available: http://doi.acm.org/10.1145/1508865.1508921

[8] 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*. Australian Computer Society, Inc., 2011, pp. 113–118.

[9] 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, 2012.

[10] I. Utting, A. E. Tew, M. McCracken, L. Thomas, D. Bouvier, R. Frye, J. Paterson, M. Caspersen, Y. B.-D. Kolikant, J. Sorva *et al.*, "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*. ACM, 2013, pp. 15–32.

[11] D. Janzen and H. Saiedian, "Test-driven learning in early programming courses," in *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*, ser. SIGCSE '08. New York, NY, USA: ACM, 2008, pp. 532–536. [Online]. Available: http://doi.acm.org/10.1145/1352135.1352315

[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*. ACM, 2006, pp. 907–913.

[13] C. Agruss and B. Johnson, "Ad hoc software testing: A perspective on exploration and improvisation," in *Florida Institute of Technology*, 2000, pp. 68–69.

[14] J. Bach, "Exploratory testing explained," *The testing practitioner*, pp. 253–265, 2004.

[15] J. Itkonen and K. Rautiainen, "Exploratory testing: a multiple case study," in *2005 International Symposium on Empirical Software Engineering, 2005.*, Nov 2005, pp. 10 pp.–.

[16] J. Itkonen, M. V. Mantyla, and C. Lassenius, "Defect detection efficiency: Test case based vs. exploratory testing," in *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, Sep. 2007, pp. 61–70.

[17] L. P. Scatalon, J. M. Prates, D. M. de Souza, E. F. Barbosa, and R. E. Garcia, "Towards the role of test design in programming assignments," in *Software Engineering Education and Training (CSEE&T), 2017 IEEE 30th Conference on*. IEEE, 2017, pp. 170–179.

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

[19] E. F. Barbosa, M. A. Silva, C. K. Corte, and J. C. Maldonado, "Integrated teaching of programming foundations and software testing," in *Frontiers in Education Conference, 2008. FIE 2008. 38th Annual*. IEEE, 2008, pp. S1H–5.

[20] 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-Volume 136*. Australian Computer Society, Inc., 2013, pp. 97–106.

[21] S. H. Edwards, "Using software testing to move students from trial-and-error to reflection-in-action," *ACM SIGCSE Bulletin*, vol. 36, no. 1, pp. 26–30, 2004.

[22] D. Astels, *Test Driven Development: A Practical Guide*. Prentice Hall Professional Technical Reference, 2003.

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

[24] K. Beck, "Embracing change with extreme programming," *Computer*, no. 10, pp. 70–77, 1999.

[25] K. Beck and E. Gamma, *Extreme programming explained: embrace change*. Addison-wesley Professional, 2000.

[26] M. Delamaro, M. Jino, and J. Maldonado, *Introdução Ao Teste De Software*. Elsevier Brasil, 2013.

[27] V. Prakash and S. Gopalakrishnan, "Testing efficiency exploited: Scripted versus exploratory testing," in *2011 3rd International Conference on Electronics Computer Technology*, vol. 3, April 2011, pp. 168–172.

[28] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[29] W. H. Kruskal and W. A. Wallis, "Use of ranks in one-criterion variance analysis," *Journal of the American statistical Association*, vol. 47, no. 260, pp. 583–621, 1952.

[30] T. D. Cook and D. T. Campbell, *Quasi-Experimentation: Design and Analysis Issues for Field Settings*. Houghton Mifflin Company, Boston, 1979.