Experience Report: Can Software Testing Education Lead to More Reliable Code?

Otávio Augusto Lazzarini Lemos*, Fabiano Cutigi Ferrari†, Fábio Fagundes Silveira*, and Alessandro Garcia‡
*Science and Technology Department – Federal University of São Paulo at S. J. dos Campos – Brazil
{otavio.lemos, fsilveira}@unifesp.br
†Computing Department – Federal University of São Carlos – Brazil
fabiano@dc.ufscar.br
‡Informatics Department – Pontifical Catholic University of Rio de Janeiro – Brazil
afgarcia@inf.puc-rio.br

***Abstract*—Software Testing (ST) is one of the least known aspects of software development. Yet, software engineers often argue that it demands more than half of the costs of a software project. Thus, proper testing education is of paramount importance. In fact, the mere exposition to ST knowledge might have an impact on programming skills. In particular, it can encourage the production of more reliable code. Although this is intuitive, to the best of our knowledge, there are no empirical studies about such effects. Evidence on this matter is important to motivate – or demotivate – classical testing education. Concerned with this, we have conducted a study to investigate the possible impact of ST knowledge on the production of reliable code. Our controlled experiment involved 28 senior-level Computer Science students, 8 auxiliary functions with 92 test cases, and a total of 112 implementations. Results show that code delivered after the exposition to ST knowledge is, on average, 20% more reliable (a significant difference at the 0.01 level). Also, implementations delivered afterwards are not significantly larger in terms of lines of code. This indicates that ST knowledge can make developers produce more reliable software with no additional overhead in terms of program size.**

***Index Terms*—software testing, computer science education, student experiments.**

I. Introduction

Software Testing (ST) is one of the most important and least known aspects of software development. In fact, it is common that Computer Science (CS) students graduate into industry without knowing how to test a program [1]. ST is thus regarded as one of the *dark arts* of software development [2]. Yet, researchers and practitioners argue that testing often demands more than 50% of the costs of a software project [3].

There is another aspect of ST that seems to be overlooked: the mere exposition to its knowledge might help developers produce more reliable programs. In fact, there are several ideas in the ST body of knowledge that can produce positive effects in programmers' skills. For instance, consider the awareness that *virtually all programs contain faults* [2, 5], a principle taught early in ST courses. Such an idea can instil a healthy skepticism in programmers towards their own code, making them more cautious. Moreover, the formal testing techniques themselves encourage designing programs with attention. Take, for example, *boundary value analysis*, a functional testing criterion that requires writing tests for border inputs. Developers exposed to this strategy can be more careful about corner cases in their implementations, hence improving the quality of their code.

Although the effect of testing knowledge on programmers seems to be intuitive, there is little empirical evidence to support it. We can find in the literature substantial work into developing ways to improve the training of ST in CS programs. For instance, Patterson et al. proposed the integration of testing tools into programming environments [6], Jones has explored the integration of testing into introductory CS courses through testing labs and diverse forms of courseware [7], and Elbaum et al. presented a web-based tutorial to engage students in learning software testing strategies [4]. However, to the best of our knowledge, there are no experimental studies into the effects of ST education on the developers' programming skills *per se*, in terms of reliability.

Investigations into this topic are important because recent data shows that computing academic curricula tend to emphasize *development* at the expense of *testing* as a formal engineering discipline [1, 8, 9]. In fact, as reported by Astigarra et al. [8], the bulk of academic CS curricula tend to place a heavy emphasis on design and implementation, rather than on quality assurance topics such as ST. On the other hand, even when ST courses are in fact present in curricula, it is unclear the extent to which the techniques that are taught are in fact adopted by the industry (*e.g.*, mutation [10] and data-flow [3] testing seem to be rarely practiced). Empirical evidence showing that ST education can lead to more reliable programming can motivate the creation or maintenance of these courses.

In this paper, we present a study that investigates the impact of ST education on reliable programming. We have conducted a controlled experiment involving 28 senior CS undergraduate students, 8 auxiliary functions in 4 different domains (basic mathematics; array manipulation; string manipulation; and file input/output) with 92 test cases, and a total of 112 implementations. Subjects implemented two different functions before and after learning basic ST concepts and three techniques (functional – or *black-box* – testing, structural – or *white-box* – testing, and mutation testing), and the quality of the produced code before and after was compared. Our goal was not to ver-

ify how well the techniques were applied afterwards, but how ST knowledge could impact on the subjects' programming skills in terms of producing more reliable implementations (*i.e.*, we did not measure the quality of the testing code itself, neither which specific techniques were being applied). To evaluate the reliability of the implemented functions in terms of correctness, the produced implementations were executed against the systematically developed test sets before and after the training took place. To improve the external validity of our experiment, we included a control group of 8 subjects at the same academic year who were not taking the ST course.

Our study provides evidence that ST knowledge can significantly impact on programming skills in terms of reliability. In fact, subjects were more than twice more likely to deliver correct implementations after learning the ST concepts and techniques (150%). Moreover, the correctness of the subjects' implementations was, on average, 20% higher after the exposition to ST knowledge took place. Interestingly, we noticed that the positive effect is present even when no specific testing technique is explicitly applied, possibly a consequence of the exposure to the testing theory itself. Subjects in the control group did not perform as well as the ones in the treatment group.

As an additional investigation, we looked into the subjects' efforts – in terms of time and lines of code written – and complexity of the produced code – in terms of cyclomatic complexity. Our study indicates that subjects do invest more effort in their implementations after taking the ST lessons, in terms of time. However, such effort did not result in the production of significantly more lines of application code, implying that the implementations produced afterwards were more reliable but not significantly larger than the ones produced in the first session. Cyclomatic complexity was significantly higher in the second session, probably because the second session implementations covered important corner cases through conditionals that were omitted in the first session.

The remainder of this paper is structured as follows. Section II presents background knowledge required to understand our study, and Section III presents how our study was set up in terms of subjects, experimental design, metric and statistical procedures. Section IV presents the results and analysis of our experiment, while Section V discusses such results in more details. In the sequence, Section VI presents our study limitations and Section VII summarizes related research. Finally, Section VIII concludes the paper.

II. Software Testing Knowledge

In this paper, the functional, structural, and fault-based testing techniques together with basic software testing principles and concepts – such as the ones discussed in this section and in Section I – were taken as basic software testing knowledge (*ST knowledge*, from now on). Our goal is to evaluate whether such knowledge could impact on the programming skills of software developers, in terms of producing more reliable implementations.

A *test case* (or simply, a *test*) consists of a set of inputs and expected output for a program [11]. The output is assessed via an *oracle*, which determines what is the correct result of the program under testing given an input [12]. In our case, the oracle is a tester supported by an automated testing tool (in this paper, JUnit[1]) that implements *assertions*. Formally, a test case is an ordered tuple: $< (I_1, ..., I_n), O >$, where O is the expected output of the program when $I_1, ..., I_n$ are used as inputs.

Software testing is defined as the execution of a program against test cases with the intent of revealing faults [2]. The varied testing techniques are defined based on the underlying artifacts used to derive test cases. Three basic testing techniques are *functional*, *structural*, and *fault-based testing*.

Functional – or *black box* – testing derives test cases from the specification of a program. Two of the most well-known functional-based testing selection criteria are *equivalence partitioning* and *boundary-value analysis*. Equivalence partitioning divides the input and output domains of a program into a finite number of valid and invalid *equivalence classes*. It is then assumed that a test case with a representative value within a given class is equivalent to testing any other value in the same class. This criterion requires a minimum number of test cases to cover the valid classes and an individual test case to cover each invalid class. Boundary-value analysis complements equivalence partitioning by requiring test cases to cover values at the boundaries of equivalence classes [2].

Structural – or *white-box* – testing is a technique that complements functional testing. It derives test cases from the internal representation of a program [2]. Some of the well-known structural testing criteria are *statement*, *branch*, or *definition-use* [13] *coverage*. These criteria require that all commands, decisions, or pairs of assignment and use locations of a variable be covered by test cases.

Fault-based testing has *mutation testing* as its most representative criterion. The main idea behind mutation testing is to define a set of mutation operators which, when applied to components of a program, introduce certain types of *faults* into the program. Typical mutations include changing variable names in expressions, or changing arithmetic operators (*e.g.*, a $>$ for a $<$). The goal then is to construct a set of tests cases T which will distinguish between a program P and any nonequivalent program P' which can be generated from the original P by the application of mutations to components of P [14].

III. Study Setup

The goal of our study is to investigate the impact of ST knowledge on programming skills. Such impact is evaluated in terms of reliability. In this endeavor, we are interested in the following research question: $\mathbf{RQ}_1$ – Can ST knowledge help developers improve their programming skills in terms of delivering more reliable implementations?

[1] http://junit.org/ - 07/jun/2015

As an additional investigation, we want to check whether developers tend to invest more (or less) effort in their implementations after learning ST, and whether there is a difference in the complexity of the code produced before and after the exposition to ST knowledge. Such additional investigation raises two other research questions: $\mathbf{RQ}_2$ – Does ST knowledge impact on the effort invested by developers on their implementations?; and $\mathbf{RQ}_3$ – Does ST knowledge impact on the complexity of the produced code?

Our investigation develops in terms of hypotheses H_1, H_2, H_3, and H_4, where the first is related to research question $\mathbf{RQ}_1$, the second and the third are related to research question $\mathbf{RQ}_2$, and the forth is related to research question $\mathbf{RQ}_3$. The null (*0*) and alternative (*A*) definitions of each hypothesis are described in Table I.

TABLE I
HYPOTHESES FORMULATED FOR OUR EXPERIMENT.

	Null hypothesis (0)	Alternative Hypothesis (A)
H_1	$\text{Correctness}_{\text{woTK}} = \text{Correctness}_{\text{wTK}}$	$\text{Correctness}_{\text{woTK}} < \text{Correctness}_{\text{wTK}}$
H_2	$\text{Time}_{\text{woTK}} = \text{Time}_{\text{wTK}}$	$\text{Time}_{\text{woTK}} < \text{Time}_{\text{wTK}}$
H_3	$\text{Size}_{\text{woTK}} = \text{Size}_{\text{wTK}}$	$\text{Size}_{\text{woTK}} < \text{Size}_{\text{wTK}}$
H_4	$\text{Complexity}_{\text{woTK}} = \text{Complexity}_{\text{wTK}}$	$\text{Complexity}_{\text{woTK}} < \text{Complexity}_{\text{wTK}}$

Legend: H = Hypothesis; woTK = without testing knowledge; wTK = with testing knowledge; Size = code size; Complexity = code complexity.

The experimental setup adopted for our study shares several characteristics of a previous study conducted by the same authors [15]. The main difference is that in this paper we evaluate the impact of the testing knowledge on programming skills, while the previous experiment targeted agile practices. Moreover, in the experiment presented in this paper two functions and a control group were added to the experimental design. This was done to add more rigor to the study, since subjects implemented two functions in each session and we compared outcomes with a group of students that was not taking the ST course. The next paragraphs discuss our setup in detail.

A. Subjects, Target Functions, Test Sets, and Tools

Subjects: Our study involved 28 senior CS undergraduate students (20 in the treatment group and 8 in the control group). All students had basic Java programming knowledge learned along a one-semester, 72-hour Object-Oriented Programming course (8 hours per week). The students in the treatment group were asked to perform the tasks of our experiment *before* and *after* learning ST knowledge while taking a 72-hour Software Testing course. The first session took place on the first week of the semester, and the second took place around two and a half months later. The control group did exactly the same thing, except that they were not taking the ST course, but an Object-Oriented (OO) Design in Java course (mostly learning UML and OO patterns and their implementations in Java).

All sessions were performed in a closed-lab environment. Students had two hours to complete their tasks, and were asked to use a Java IDE to implement and test their assigned functions. Since we were more interested in the effects of the testing theory itself on programming skills, not as much on the application of the specific testing techniques themselves, no functional, structural, or mutation testing tools were readily available to them. In this way, our results represent more the impact that *learning* the testing concepts and techniques have on programming skills. In fact, as we discuss in Section V, by analyzing the code produced by the subjects of the treatment group, it appears that some of them have not explicitly applied the learned techniques, and still got better results.

Target Functions. The features involved in our study were *auxiliary functions*, that is, supportive actions of software systems. It is important that these functions be developed with care because the history of software development shows that they can be the source of significant failures [15]. To select a representative and variable set of such features, we looked into the Apache Commons project[2], which provides libraries of reusable Java components. We also selected functionalities that could easily be found through searches issued to code search engines; that is, we tried to identify commonly used auxiliary functions that were not readily available in the Java API. We categorized these functions into four domains: array manipulation (Array), basic mathematics (Math), string manipulation (String), and file input/output (File I/O). To obtain a richer set, we selected two functionalities within each domain. The auxiliary functions used in our study are listed in Table II.

Another characteristic of the selected functions is that they are narrowly scoped. The idea is to perform a conservative evaluation: if particular knowledge can impact on the implementation of smaller features, we can expect them to further impact on larger ones. Another advantage is that this type of function enables the adoption of more systematic test case selection techniques to evaluate them in the experiment, such as functional testing. Such characteristic provides more control to the experiment.

Test Sets. To evaluate the programs implemented by the subjects, we developed full functional test sets for each of the selected functions. The last column of Table II shows the number of test cases developed for each one. To construct the test sets, we applied the equivalence partitioning and boundary-value analysis criteria (see Section II). These criteria were used to select representative test cases for each test set, trying to cover as many functional specificities of the functions as possible. To show an example of how test cases were developed, Table III presents the equivalence classes and boundary values (when applicable) for the $\mathbf{a}_1$ functionality. *ar1* and *ar2* are the $\mathbf{a}_1$ input arrays; $|ar|$ represents the array *size*; and *arX*[i] represents an element of the array. *Int.MIN* and *Int.MAX* correspond to the minimum and maximum integer values. Since the study was conducted using the Java language, the highest and lowest possible integers were used as boundary values for that data type. A similar rule was applied to other types for other functions. Here, we do not use the specific values to represent the test cases independently of language.

[2]http://commons.apache.org/ - 07/jun/2015

TABLE II
FUNCTIONS USED IN THE EXPERIMENT.

Domain	F	Description	Sample Test Case	# TCs
Array	$\mathbf{a}_1$	*Array equality*: given two arrays, the program should return *true* or *false* according to the contents of the arrays being equal or not.	<([1, 2, 3], [1, 2, 3]), *true*>	20
	$\mathbf{a}_2$	*First index with different value*: given an array and a number, the program should return the first index of the array that contains a value different from the number.	<([0, 0, 0, 0, 0, 1], 0), 5>	12
Math	$\mathbf{m}_1$	*Power of two*: given a number, the program should return *true* or *false* according to it being or not a power of two.	<(4), *true*>	6
	$\mathbf{m}_2$	*Factorial*: given a number, the program should return its factorial.	<(5), 120>	7
String	$\mathbf{s}_1$	*Capitalization of phrases*: given a string, the program should return the same string with the first letters of words capitalized.	<("one two"), "One Two">	7
	$\mathbf{s}_2$	*Maximum common prefix*: given two strings, the program should return the maximum common prefix between them.	<("pref suf", "pref fus"), "pref ">	11
File I/O	$\mathbf{i}_1$	*Create text file*: given two strings, the program should create a text file whose content and location/name is indicated in the first and second string.	<("abc", ".\dir\text.txt"), *creates file .\dir\text.txt with "abc" as content*>	13
	$\mathbf{i}_2$	*File copy*: given two strings, the program should copy the file indicated in the first string to the location indicated in the second.	<(".\test.file", ".\tmp\"), *.\test.file is copied to .\tmp* >	16

Legend: F = Function; TCs = Test Cases.

TABLE III
EQUIVALENCE CLASSES AND BOUNDARY VALUES CONSIDERED FOR TESTING *Array Equality* ($\mathbf{a}_1$).

Input Cond.	Valid Classes	Invalid Classes	Boundary Values
|ar1|	|ar1| > -1 (C1)		|ar1| = 0 (B1)
ar1 is *null*	No (C2)	Yes (C3)	
|ar2|	|ar2| > -1 (C4)		|ar2| = 0 (B2)
ar2 is *null*	No (C5)	Yes (C6)	
|ar1|, |ar2|	|ar1| > |ar2| (C7) |ar2| > |ar1| (C8)		|a1| - |a2| = 1 (B3) |a2| - |a1| = 1 (B4)
ar1[i]	*Int.MIN* ≤ ar1[i] ≤ *Int.MAX* (C9)		ar1[i] = *Int.MIN* (B5) ar1[i] = *Int.MAX* (B6)
ar2[i]	*Int.MIN* ≤ ar2[i] ≤ *Int.MAX* (C10)		ar2[i] = *Int.MIN* (B7) ar2[i] = *Int.MAX* (B8)

Tools. Eclipse[3] was the IDE used to develop the functions, and JUnit was the framework used to develop the test cases. Students received instructions in order to make sure they would concentrate their effort only on implementing the intended auxiliary functionalities (and tests for them). For instance, the subjects were instructed to implement functions as static methods in a class with a predefined name. We did this because static methods are easier to implement since they do not require object instantiation. Moreover, auxiliary functions usually rely only on parameter values to fulfill their responsibility. This also enables the execution of our established test sets more easily.

B. Experimental Design and Procedure

For the conducted experiment, we adopted the *repeated measures* with cross-over and control group experimental design (or *pre-post test with control group*), in which each subject implemented functions before and after acquiring the ST basic knowledge – in the case of the treatment group – or OO design basic knowledge – in the case of the control group. Such type of design supports more control to the variability among subjects [16]. To minimize the variability of the difference among functions, we randomized the assignments among students for both groups. Finally, to cancel function asymmetry, each function was assigned to be implemented before and after the treatment by different subjects.

The experiment was conducted in two sessions. In the first session, prior to learning any ST or OO design concepts, students implemented two functions; and in the second session, after learning ST or OO design, they implemented other two functions. Since each subject implemented 4 functions, we collected a total of 112 implementations.

Students had to implement functions from different domains in the first and second sessions. For instance, a student implementing a_1 and i_2 in the first session would implement a Math and a String function in the second session. We did this to cancel the impact of function domains on each other

[3] http://eclipse.org/ - 07/jun/2015

while implementing the functionalities in the first and second sessions.

To help understanding the adopted experimental design, Table IV presents part of the assignments used for the experiment for one of the groups.

TABLE IV
PARTIAL TASK ASSIGNMENTS TO SUBJECTS.

Subject	1^{st} Session	2^{nd} Session
	Functions	Functions
01	a_1 & m_1	s_1 & i_1
02	a_1 & s_1	m_1 & i_1
03	a_1 & i_1	s_1 & m_1
04	s_1 & m_1	a_1 & i_1
05	s_1 & i_1	a_1 & m_1
06	m_1 & i_1	a_1 & s_1
07	a_2 & m_2	s_2 & i_2
08	a_2 & s_2	m_2 & i_2
09	a_2 & i_2	s_2 & m_2
10	s_2 & m_2	a_2 & i_2
⋮	⋮	⋮

C. Metrics

We adopted a straightforward metric to evaluate the *reliability* of the developed functions: their correctness in terms of their *Functional Test Set Success Rate* (FTSSR). For a given implementation, FTSSR is computed by dividing the number of successful test cases by the total number of test cases developed for a given function. The FTSSR is a continuous variable: it grades implementations from 0.0 to 1.0. For instance, an implementation of the $\mathbf{a}_1$ function that passed 10 test cases would receive an FTSSR score of 0.5, since there are 20 test cases developed for it in its test set (see Table II).

For the effort evaluation, we measured the total development time in minutes subjects took to implement the two functions in each session, and the average number of produced lines of code (LoC). Some studies have found a positive correlation between the size of program modules in LoC and fault-proneness [17, 18] (*i.e.*, the larger a module in LoC, the more faults it tends to present). Therefore, by measuring the difference in LoC from the first to the second session we are also secondarily evaluating an additional reliability metric. That is, if code produced in the second session is not larger than code produced in the first session, we have an additional evidence that reliability has not decreased afterwards from such a perspective.

To evaluate complexity, we computed the average McCabe cyclomatic complexity metric [19] (M). We used the Eclipse Metrics to measure both LoC and M[4]. Subjects were responsible for registering the time taken to implement functions.

D. Statistical Analysis

From a statistical standpoint, a simple observation of the means or medians from sample observations is not enough to infer about the actual populations. This happens because the reached differences might be a coincidence caused by random sampling. To check whether the observed differences are in fact significant, statistical hypothesis tests can be applied.

In our study, each subject developed functions before and after learning the ST and OO design concepts. In this case, the *paired* statistical hypothesis tests can compare measures within subjects rather than across them. Paired tests are considered to greatly improve precision when compared to unpaired tests [16]. Since our results seemed to follow a normal distribution, according to a Shapiro-Wilk normality test, we decided to apply the paired Student t-test.

To have a more rigorous evaluation of our results, for the statistical tests ran in our experiment, we adopted a confidence level of 99%. Our analyses thus consider p-values below 0.01 significant. For the statistical tests we adopted the R language and environment[5].

IV. Results and Analysis

Table V presents the results of our experiment for the treatment group. For FTSSR, the table shows the results for each function implemented and the average. For other metrics, it shows only the average (Cyclomatic Complexity - M - and Lines of Code - L) and total (Development Time - T). To allow a visual analysis of two metrics that were significantly affected for the treatment group, Figure 1 shows a boxplot of the FTSSR and M outcomes for that group. Note that some subjects were removed from our analysis either for producing outliers in terms of FTSSR, or for not completing all assigned tasks (for instance, some students did not implement one of the two functions required for each session). The outliers are discussed in Section VI.

We can notice that results related to reliability improved significantly and consistently after the ST course took place: the average FTSSR went from 0.70 to 0.84, a 20% improvement. Also, the number of implementations that passed all tests more than doubled in the second session: they went from 4 to 10. Moreover, the single subject that produced two implementations that passed all tests only did so in the second session (subject #13). Note that 13 out of the 20 subjects (65%) achieved better reliability results after taking the ST course. The boxplot also shows that the subjects' performance was more varied in the first session, while in the second session they were consistently better. The second session box is smaller and higher than the first session one.

Other interesting outcomes are worth noting. For instance, the minimum FTSSR was improved by approximately 65% from the first to the second session (it went from 0.43 to 0.71). This indicates that even less proficient programmers can benefit from the ST knowledge, with respect to producing more reliable software. Also note that the subject that reached such a minimum in the first session, raised her FTSSR score to 0.91, on average, in the second session: a very significant improvement (one of the functions developed by the student in that session passed all tests). Another interesting outcome is that there is only a single function that did not pass any of the tests, and it was produced in the first session only.

[4]http://metrics.sourceforge.net - 07/jun/2015

[5]http://www.r-project.org/ - 07/jun/2015

TABLE V
OUTCOMES FOR THE TREATMENT GROUP. S = SUBJECT; FTSSR = FUNCTIONAL TEST SET SUCCESS RATE; F_n = FUNCTION N; μ = AVERAGE; Σ = TOTAL; T = DEVELOPMENT TIME (IN MINUTES); L = LINES OF CODE; M = CYCLOMATIC COMPLEXITY.

S	1^{st} Session						2^{nd} Session					
	FTSSR			ΣT	μL	μM	FTSSR			ΣT	μL	μM
	F_1	F_2	μ				F_1	F_2	μ			
1	0.95	0.00	0.48	40	11.00	2.00	0.83	1.00	0.92	61	19.50	3.50
2	0.55	0.71	0.63	28	16.50	1.50	0.56	1.00	0.78	80	16.00	5.00
3	0.85	0.42	0.63	16	14.50	2.50	0.83	0.86	0.85	80	18.00	4.00
4	0.83	0.86	0.85	28	16.00	4.50	0.85	0.92	0.88	7	15.50	5.00
5	0.85	0.83	0.84	34	14.50	3.00	0.71	0.82	0.77	82	16.50	7.00
6	0.85	1.00	0.93	45	17.00	5.00	0.83	0.71	0.77	40	17.50	4.50
7	0.83	0.64	0.73	30	16.00	2.00	0.92	1.00	0.96	51	10.00	3.50
8	0.58	0.43	0.51	32	10.00	2.50	0.83	0.71	0.77	70	16.00	3.00
9	0.92	0.83	0.88	35	16.00	3.00	1.00	0.86	0.93	110	21.50	5.50
10	0.55	0.67	0.61	90	19.50	4.50	0.92	0.71	0.82	120	18.00	4.50
11	1.00	0.82	0.91	75	23.00	6.50	0.83	0.83	0.83	70	21.50	4.50
12	0.43	0.71	0.57	40	9.50	1.50	0.85	0.81	0.83	60	13.50	3.00
13	0.92	0.06	0.49	30	18.50	3.50	1.00	1.00	1.00	56	18.50	5.00
14	0.75	0.06	0.41	45	14.50	2.00	1.00	0.82	0.91	45	16.50	5.00
15	0.63	0.71	0.67	35	28.50	5.00	0.95	1.00	0.98	25	24.00	6.00
16	0.50	0.43	0.46	30	14.00	3.00	0.67	1.00	0.83	37	20.00	6.50
17	0.85	0.57	0.71	22	9.50	4.50	0.56	0.86	0.71	60	22.00	4.00
18	0.83	0.92	0.88	50	14.00	2.50	1.00	0.64	0.82	63	20.50	6.00
19	1.00	0.82	0.91	33	15.50	5.00	0.42	0.92	0.67	22	15.50	3.00
20	1.00	0.86	0.93	31	17.00	4.00	0.85	0.83	0.84	18	15.00	3.00
μ	0.78	0.62	0.70	38.45	15.75	3.40	0.82	0.87	0.84	57.85	17.78	4.58

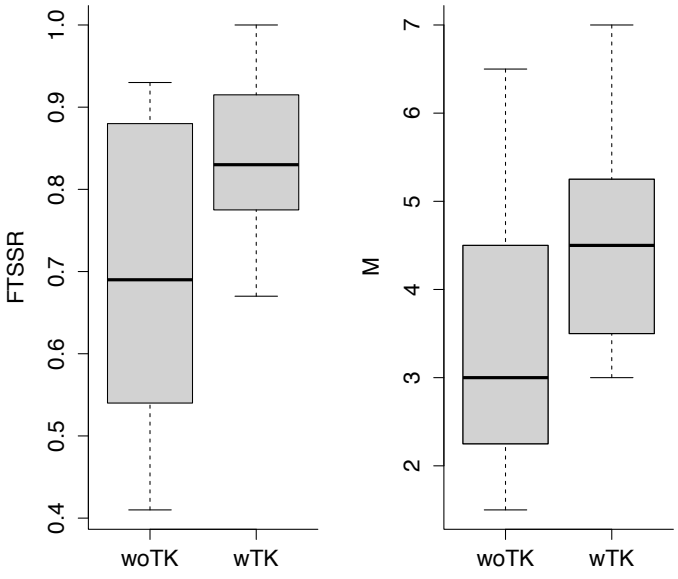

Fig. 1. Boxplot of the Functional Test Set Success Rate (FTSSR) and Cyclomatic Complexity (M) outcomes of our experiment, for the treatment group. Legend: woTK = without Testing Knowledge; wTK = with Testing Knowledge.

The same subject that developed such a function – subject #1 – developed another one that passed all tests in the second session.

To check whether the observed difference in terms of correctness was significant – the difference in average FTSSR –, we ran the t-test, which indicated a statistically significant difference at 99% confidence level (df = 19, p-value = 0.005168). Such result favors the *alternative* hypothesis (H_1-*A*) that subjects perform better after learning the ST concepts and techniques. We believe this is a key finding, since even for supportive functionality it appears that learning ST can bring benefits to developers. Moreover, the larger number of completely correct implementations for subjects in the second session shows that they tend to be more careful while implementing auxiliary functions after being exposed to ST knowledge, probably taking into account exceptional inputs that were included in the test sets.

Because we are considering reliability as the system's quality driver, a failing test case in our scenario is critical. This is particularly true for our experimental setting: since we applied functional testing, each test covers an important part of the functionality (*i.e.*, either an input or output equivalence class, or a boundary value), therefore a failing test case impacts significantly on the correctness of the system. In this sense, our results indicate that the exposition to ST knowledge could even help prevent serious problems caused by auxiliary functions, like the ones reported by major companies, as discussed in Section III and by Lemos et al. [15].

Effort and Complexity Investigation

In our additional investigation into the impact of learning ST theory on effort and complexity, we also reached interesting results. With respect to duration, subjects invested considerably more time in the second session: 19.4 minutes more, on average (more than 50%). To check whether the observed difference in terms of time was significant, we ran the t-test, which indicated a statistically significant difference at 99% confidence level (df = 19, p-value = 0.002406). Such result favors the *alternative* hypothesis (H_2-*A*) that subjects invest more time implementing functions after learning the ST concepts and techniques. This is an indication that they tend to be more cautious after the exposition to ST theory, either by investing more time in the application code itself, or in the tests.

On the other hand, when we look into the produced lines of application code, we did not find a significant difference: subjects produced, on average, 2.03 more LoC in the second session (15%). This is quite surprising as the implementations produced after learning the ST theory were more reliable than the ones produced before, but not significantly larger in terms of LoC. This indicates that ST theory might improve programming skills not only in terms of producing more correct code, but also in terms of producing leaner code. To confirm our intuition, we ran the t-test, which in fact did not indicate a statistically significant difference at 99% confidence level (df = 19, p-value = 0.03048). Such result favors the

null hypothesis (H_2-*0*) that subjects produce similar number of lines of code after learning ST.

As commented in Section III, some studies have found a positive correlation between the size of program modules in LoC and fault-proneness. Since implementations produced in the second session were not significantly larger in LoC, we believe this is an additional evidence that reliability did not decrease from the first to the second session.

With respect to complexity, code produced in the second session was sensibly more complex than code produced before, in terms of the McCabe's complexity metric: 1.18 points higher, on average (34.70%). Such difference indicates that the functions implemented after learning ST contained more conditional statements. This is expected, as those functions were more successful against the established test suites, which covered exceptional inputs. The t-test indicated a statistically significant difference at 99% confidence level (df = 19, p-value = 0.004649). Such result favors the alternative hypothesis (H_3-*A*) that subjects tend to produce more complex implementations after learning the ST concepts and techniques.

Analysis of the Control Group

Table VI presents the results for our control group. We included this group in our study mainly to reduce threats to the internal validity of our experiment, in particular related to history (*e.g.*, students in the treatment group could have been exposed to other types of knowledge that might also have had an impact on reliable programming).

TABLE VI
OUTCOMES FOR THE CONTROL GROUP. S = SUBJECT; FTSSR = FUNCTIONAL TEST SET SUCCESS RATE; F_n = FUNCTION N; μ = AVERAGE; Σ = TOTAL; T = DEVELOPMENT TIME (IN MINUTES); L = LINES OF CODE; M = CYCLOMATIC COMPLEXITY.

S	1^{st} Session						2^{nd} Session					
	FTSSR			ΣT	μL	μM	FTSSR			ΣT	μL	μM
	F_1	F_2	μ				F_1	F_2	μ			
1	1.00	0.75	0.88	67.00	13.50	2.50	0.60	1.00	0.80	23.00	19.50	6.00
2	0.58	0.29	0.43	35.00	12.00	3.00	0.00	0.75	0.38	11.00	17.00	3.00
3	0.71	0.75	0.73	27.00	19.50	3.50	0.82	0.56	0.69	39.00	24.00	5.00
4	0.95	0.06	0.51	49.00	12.50	1.00	0.92	0.75	0.83	30.00	14.00	2.50
5	0.92	0.43	0.67	41.00	18.00	5.00	1.00	0.85	0.93	11.00	10.50	3.50
6	0.43	0.58	0.51	18.00	17.50	5.00	0.56	0.83	0.70	53.00	15.50	3.00
7	0.50	0.29	0.39	45.00	16.00	3.00	0.67	0.29	0.48	28.00	13.50	2.00
8	0.92	0.29	0.60	8.00	11.00	3.00	1.00	0.57	0.79	28.00	17.00	5.50
μ	0.75	0.43	0.59	36.25	15.00	3.25	0.70	0.70	0.70	27.88	16.38	3.81

Although the sample size was much smaller for the control group – only 8 subjects –, some interesting outcomes are worth noting. In particular, results reached for this group reinforce the evidence that the ST theory can have more impact on reliable programming than other types of knowledge. Note that although FTSSR also improved in the second session for this group, the difference was not as significant and consistent as for the treatment group. In fact, the t-test shows that the difference was not statistically significant at 99% confidence value (df = 7, p-value = 0.04415). It is worthwhile noting that this group was taking a Java OO design course, so it was expected that their programming skills would be improved. On the other hand, it is quite surprising that the ST course, which does not focus directly on programming, had a more significant impact on reliable programming.

Other outcomes are worth noting. For instance, similarly to the treatment group, there was a single implementation that did not pass any of the tests. However, for the control group, such implementation was produced in the second session (recall that for the treatment group no subject implemented a function that did not pass any tests in the second session). Also note that, differently from the treatment group, cyclomatic complexity was not significantly higher in the second session (df = 7, p-value = 0.2251). This is an evidence that the students taking the ST course were probably more cautious about exceptional inputs, which made them add conditionals to their implementations, whereas the same did not happen here. On the other hand, for the control group the total development time reduced from the first to the second session (although, according to the t-test, such difference was not statistically significant – df = 7, p-value = 0.2083). This is an interesting deviation from the treatment group, which took significantly more time to develop functions after learning ST (probably because they were more cautious and developed tests for their implementations). With respect to LoC, similarly to the treatment group, there was no significant difference from the first to the second session.

In summary, with respect to reliability, it can be noticed that the ST knowledge seems to have had more significant impact on the students than the OO design knowledge. Also, it seems that the subjects that were exposed to ST theory became more cautious afterwards, by taking more time to implement functions and tests, and by adding conditionals to cover borderline cases.

V. DISCUSSION

An interesting insight provided by our experiment is that subjects in the treatment group seemed to have performed better even when no specific testing technique was explicitly applied. As discussed in Section III, no testing tools were readily available to the students when they performed the experimental tasks. This was done because we were more interested in the impact that the testing *knowledge* itself had on programming skills, not as much in the impact of the *application* of formal testing techniques.

Although students were also trained to write JUnit test cases, some wrote their tests on `main` programs. For instance, Figure 2 shows a code snippet of tests implemented by one of the subjects on the second session in a `main` method. This student was one of the ones that obtained good results after learning ST theory. Note that nothing implies that a specific testing technique was being explicitly applied. Although it appears that a strategy similar to the one established by functional testing was followed (*i.e.*, some even and odd numbers were tried, the number one was tried), it does not seem that a formal partitioning of the input domain was performed, neither

that all boundary values were considered. However, the student did perform better in the second session, indicating that the ST theory improved her/his programming skills. In fact, in one of the first session implementations by the same subject, none of the tests passed, indicating that significantly better coding was happening in the second session.

```
public static void main(String[] args) {
    //m1 - power of two
    boolean r;
    r=Util.isPowerOfTwo(5);
    System.out.print(r);
    System.out.println("");
    r=Util.isPowerOfTwo(1);
    System.out.print(r);
    System.out.println("");
    r=Util.isPowerOfTwo(4);
    System.out.print(r);
    System.out.println("");
    r=Util.isPowerOfTwo(8);
    System.out.print(r);
    System.out.println("");
    r=Util.isPowerOfTwo(16);
    System.out.print(r);
    ...
}
```

Fig. 2. A code snippet from tests developed by one of the subjects in the second session of our experiment.

On the other hand, it must be noted that some of the students did write tests following a specific technique. For instance, Figure 3 shows a partial JUnit class with tests developed by another subject that also had better outcomes on the second session. Note that, in this case, it appears that functional testing was being followed more thoroughly: there are additional tests for zero, a negative number, and a large number.

```
public class UtilTest extends TestCase {
 ...
 public void testIsPowerOfTwo(){
  assertTrue(Util.isPowerOfTwo(4));
 }
 public void testIsPowerOfTwoWrong(){
  assertFalse(Util.isPowerOfTwo(9));
 }
 public void testIsPowerOfTwoZero(){
  assertFalse(Util.isPowerOfTwo(0));
 }
 public void testIsPowerOfTwoNegative(){
  assertFalse(Util.isPowerOfTwo(-2));
 }
 public void testIsPowerOfTwoLarge(){
  assertTrue(Util.isPowerOfTwo(4096));
 }
 ...
}
```

Fig. 3. Code snippet from tests developed by another subject in the second session of our experiment.

In their classic *The Art of Software Testing* [2], Myers et al. argue that "the software tester needs the proper attitude [..] to successfully test a software application". They go on to say that such *psychology of testing* establishes the most important considerations in ST. Our results indicate that the exposition to the principles behind such testing attitude, along with the techniques that derive from it, might produce positive effects on programmers themselves, with respect to reliable programming.

Agile development proponents seem to have identified the effect of such *testing attitude*, by encouraging testing activities along the development process (*e.g.*, the *test infected* condition [20]). In fact, in a recent survey with 326 agile developers, two of the top 5 most important agile principles identified by developers involved testing (*i.e.*, "Automated tests run with each build", and "Automated unit testing") [21]. Although ST activities in the agile community are more on the lines of *practical* testing as opposed to more formal testing, both agile philosophy and classical testing education do share important principles with regards to ST. For instance, both traditional testing literature [22] and test-driven development proponents [23] put forward the idea that *successful* test cases are the ones that find faults, and not the ones that simply "pass".

VI. Threats to Validity

It is common knowledge that all empirical studies have limitations [24]. On the other hand, we believe our study had several characteristics that made it more rigorous and thus with improved validity. In particular, the fact that it was conducted in an academic setting made us have more control over confounding effects. In any case, there are still limitations that are worth mentioning. In this section we discuss such limitations based on three types of threats to validity described by Wohlin et al. [25]. For each type, we list possible threats, measures taken to reduce each risk, and suggestions for improvements in future studies.

A. Internal Validity

The lack of control of the subjects' skills on programming and testing (other than all being in the same year of the program) might have affected the internal validity of our experiment. However, the repeated measures design decreases the probability of this threat affecting our outcomes, because the same subjects implemented functions before and after learning ST and OO Design.

Another aspect to be considered is *mortality*. Since we had more students invited to participate in the experiment in the beginning (some of them did not complete the two sessions and were therefore excluded from the study), the actual tasks that took place did not follow the exact initial assignments. This could affect the balance of the assignments that was taken into consideration in the experiment design. However, we believe that since our sample was not too small for the treatment group, an adequate balance could still be maintained. Moreover, the initial assignment set contained some redundancies which helped circumvent such threat.

Another threat to the internal validity of our experiment was the removal of some subjects that produced outliers with respect to the FTSSR metric in the treatment group[6]. In fact, when the same statistical test is run in the presence of the outliers, it indicates a non-significant difference at 99% confidence level. In order to justify the removal of such outliers, we investigated what could have affected these particular students in making them produce such extraneous results. In three cases, the FTSSR score was far below the average in the second session (0.21, 0.29, and 0.30), and in the remaining case, far below the average in the first session (0.31). The three subjects of the first group – results which could affect our conclusions – had strong reasons not to benefit well from the ST course: one of them was a foreigner with difficulty in understanding the language in which the course was taught; the second had a poor record of scores in many other courses taken; and the third was a part-time student.

B. External Validity

The use of students as subjects for our experiment might have reduced its external validity. In fact, some experiments have shown opposite trends for students and professionals (*e.g.*, Arisholm and Sjoberg [26]). However, according to other authors, students can play an important role in experimentation in the field of Software Engineering [27, 28]. For instance, Canfora et al. [29] have conducted a pair programming experiment in academia and replicated the same experiment in industry. According to the authors, the experiments produced similar results for both samples.

Another more recent study that involved Test-Driven Development (TDD) has also shown similar outcomes for both students and professionals. The same study points with more in-depth analysis that students can in fact sometimes be representatives of professionals in Software Engineering experiments, in particular when trying new approaches [30]. Since in our case, students were being exposed to testing knowledge for the first time, it is likely that professionals with little testing background would perform similarly. Carver et al. [31] also analyze characteristics in which empirical studies with students can make them more valid. According to the authors, when studying issues related to a technology's learning curve or the behavior of novices, students are exactly the right test population. In our case, since we wanted to evaluate how learning ST could impact on the behavior of programmers, we believe our experiment falls into such a category of study.

The students involved in our experiment – except for the single one mentioned before – were all Brazilians. It might be the case that students from other nationalities might perform differently. Moreover the experiment only involved students from a single school, the Federal University of São Paulo. Replications with students from other countries and schools would be required to be able to further generalize our results.

As commented in Section IV, *history* might also have affected our experiment. In particular, subjects of the treatment group could have gathered other knowledge that would also impact on the FTSSR metric. However, the fact that outcomes were not as significant for the control group – with students at the same level but who were not taking the ST course – helps increase confidence that the gathered ST knowledge explains the improvement in reliable programming. In any case, further experiments are required to confirm such evidence. In particular, our control group was much smaller than the treatment group. However, it must be noted that they were taking a course that involved programming, which was expected to improve programming skills. As commented before, we plan to replicate our experiment with larger groups of subjects, including a control group with students taking courses unrelated to software development, and with professionals.

[6]In all cases the outliers were identified by box-plotting our results in R.

C. Construct Validity

A characteristic of our experiment that might have affected its construct validity is related to the metrics we have chosen to evaluate our results. For instance, we have used functional testing to develop the set of test cases to measure the reliability of the produced code. However, since we have taught such technique to the subjects, results might only indicate the extent to which they have applied it, but not the impact of the ST knowledge as a whole. However, as discussed in Section V, some students did not appear to have explicitly used this technique, which indicates that at least for some subjects, other principles and techniques might have played a role in the final outcomes. For instance, mutation testing theory stresses common types of faults made by developers, modeled as mutation operators. Such theory might also have stimulated students to omit faults that were present in the first session implementations in their second session code. Moreover, as commented in Section IV, the non-significant increase in LoC in implementations from the first to the second session was also a secondary evidence that reliability has not decreased afterwards.

VII. Related Work

To the best of our knowledge, there are no studies that directly measure the impact of software testing education on programming skills, in particular with respect to the production of more reliable software. However, there are investigations into the testing proficiency of CS students, and the improvement of ST education in CS curricula. Also, there are studies that generally highlight the importance of ST education. Next, we sample some of these studies in comparison to ours.

In regard to the testing proficiency of CS students, Carver and Kraft [32] conducted an empirical study to determine the testing ability of senior-level CS students. In particular, they wanted to evaluate whether students were able to create small, complete test suites for simple programs. Results show that a coverage tool can significantly help students produce better tests. While this investigation is related to ours, it is significantly different in that Carver and Kraft's experiment evaluated the testing ability of students, while our study investigates the impact of ST education on programming skills.

With respect to improving software testing education, particularly in CS programs, some authors report on a positive "side-effect" in programming skills. Nevertheless, to the best of our knowledge, these studies do not report on controlled experiments to assess such "side-effect". For instance, Jones [7] has explored the integration of testing into introductory CS courses through testing labs and diverse forms of courseware, including tool support for automated program grading. The author reports on the experience of an elective testing course compound by 80% of practice and 20% of theory. Beyond ordinary testing-related practices such as test case design, students have performed reverse engineering tasks to derive system specifications, have created test drivers and have written test scripts. According to the author, one major benefit obtained from this experience was the general improvement of students' software design and programming skills. Despite this, Jones did not report on objective measurement of the gains related to programming skills improvement and quality of produced code, that is, conclusions are more qualitative than quantitative. Our study, on the other hand, involved a quantitative controlled experiment, which provided more concrete evidence about such skill improvement.

Initiatives like Test Driven Learning (TDL) have also yielded gains in students' programming ability. For instance, Janzen and Saiedian [33] introduced a combined, simultaneous testing-programming learning approach that can help both novice and experienced programmers improve their comprehension and ability, and hence produce high quality code design with reduced defect density. To verify their assumptions, the authors reported on an experiment that involved first-year CS students. The students have run four 50-minute lectures whose topics were related to arrays and object manipulation. While one group of students was taught using a TDL approach, another group was exposed to non-TDL (*i.e.* traditional) teaching manner. A posterior evaluation revealed that the first group performed 10% better than the second on a quiz that covered the concepts and syntax from the experiment topics. Although our experiment did not consider a simultaneous testing-programming learning process, our results are well-aligned with Janzen and Saiedian's.

More recently, Offutt et al. [34] also presented an approach to teach students to test better. The authors present an in-depth teaching experience report on how to successfully teach criteria-based test design using abstraction and publicly accessible web applications. Clarke et al. [1], on the other hand, proposed a collaborative learning environment to integrate testing education into Software Engineering courses. Although these studies are important to improve the level of testing education, none of them investigate its impact on students' programming skills.

In the literature, we can also find several studies that highlighted the importance of testing education, and reported the lack of adequate ST training in CS curricula. For instance, Astigarraga et al. [8] showed that most CS academic programs tend to emphasize *development* at the expense of *testing* as a formal engineering discipline. Wong [9], on the other hand, argued that software testers are generally not adequately trained because most CS programs offer ST only as elective courses. Clarke et al. [1] also argued that due to the large number of topics to be covered in SE courses, little or no attention is given to software testing, resulting in students entering industry with little to no testing experience. Since our experiment provides evidence that ST education can lead to the production of more reliable software, it presents a new argument to further motivate better testing education.

VIII. Conclusions

In a recent article, Vinton Cerf, one of the fathers of the Internet, called attention to the growing need for *responsible programming* in industry [35]. Such term is defined as a clear sense of responsibility programmers should have for their systems' reliable operation and resistance to compromise and error. We also believe programmers should be more responsible for the code they produce, specially today when so many depend heavily on software to work as advertised.

However, Cerf suggests that in order to achieve responsible programming, we need better tools and programming environments. While we agree with him, we believe that better *training* of professionals could also improve the quality of the produced software, in particular by the teaching of classical software testing theory. Other authors have argued that more exposure to software testing, practices, and tools is required for better training of software developers [1, 9].

In this paper we presented scientific evidence for such a claim. We conducted an experiment to verify the impact that testing knowledge has on programming skills, in terms of reliability. Our results suggest that after learning basic testing principles and techniques, developers are more than twice more likely to produce correct implementations. Moreover, our data suggests that although the code produced afterwards is more reliable, it does not tend to be significantly larger (in terms of lines of code). This indicates that the exposure to testing knowledge can make developers produce more reliable implementations with approximately the same amount of code.

Future work includes the replication of our experiment with professional developers and larger groups of students to shed more light into this subject. Moreover, it would be interesting to conduct experiments to further analyze how training in each separate testing technique can impact on programming skills (*e.g.*, by performing several programming tasks along the course).

Acknowledgements

The authors would like to thank for the following financial support: *Otávio Lemos*: FAPESP (grant 2013/25356-2); *Fabiano Ferrari*: CNPq/Universal (grant 485235/2013-7); *Fábio Silveira*: CNPq/Universal (grant 455080/2014-3); *Alessandro Garcia*: FAPERJ (distinguished scientist grant E-26/102.211/2009), CNPq Productivity (grant 305526/2009-0) and Universal (grant 483882/2009-7), and PUC-Rio (productivity grant).

The authors would also like to thank the students who participated in the experiment.

References

[1] P. J. Clarke, D. Davis, T. M. King, J. Pava, and E. L. Jones, "Integrating testing into software engineering courses supported by a collaborative learning environment," *ACM Trans. Comput. Educ.*, vol. 14, no. 3, pp. 18:1–18:33, Oct. 2014.

[2] G. J. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing*, 3rd ed. Wiley Publishing, 2011.

[3] M. J. Harrold, "Testing: a roadmap," in *Proceedings of the Conference on The Future of Software Engineering*, ser. ICSE 2000. ACM, 2000, pp. 61–72.

[4] 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)*. IEEE, 2007, pp. 688–697.

[5] P. Ammann and J. Offutt, *Introduction to Software Testing*, 1st ed. New York, NY, USA: Cambridge University Press, 2008.

[6] A. Patterson, M. Kölling, and J. Rosenberg, "Introducing unit testing with BlueJ," in *Proceedings of the 8th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '03)*. ACM, 2003, pp. 11–15.

[7] E. L. Jones, "Integrating testing into the curriculum – arsenic in small doses," in *Proceedings of the 32th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '01)*. ACM, 2001, pp. 337–341.

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

[9] E. Wong, "Improving the state of undergraduate software testing education," in *Proc. of the 2012 Annual Conference of American Society for Engineering Education (ASEE)*. IEEE, 2012.

[10] L. Madeyski and N. Radyk, "Judy: a mutation testing tool for Java," *IET Software*, vol. 4, pp. 32–42, 2002.

[11] IEEE, "IEEE Standard Glossary of Softw. Eng. Terminology." New York: IEEE Computer Society Press, 1990.

[12] R. V. Binder, *Testing object-oriented systems: models, patterns, and tools*. Boston, MA, USA: Addison-Wesley, 1999.

[13] S. Rapps and E. J. Weyuker, "Selecting software test data using data flow information," *IEEE Trans. Softw. Eng.*, vol. 11, no. 4, pp. 367–375, 1985.

[14] W. E. Howden, "Weak mutation testing and completeness of test sets," *IEEE Trans. Softw. Eng.*, vol. 8, no. 4, pp. 371–379, Jul. 1982.

[15] O. A. L. Lemos, F. C. Ferrari, F. F. Silveira, and A. Garcia, "Development of auxiliary functions: should you be agile? an empirical assessment of pair programming and test-first programming," in *Proc. of the ICSE 2012*. IEEE, 2012, pp. 529–539.

[16] D. C. Montgomery, *Design and Analysis of Experiments*. John Wiley & Sons, 2006.

[17] T. Gyimothy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE Trans. Softw. Eng.*, vol. 31, no. 10, pp. 897–910, Oct. 2005.

[18] R. Subramanyam and M. S. Krishnan, "Empirical analysis of CK metrics for object-oriented design complexity: Implications for software defects," *IEEE Trans. Softw. Eng.*, vol. 29, no. 4, pp. 297–310, Apr. 2003.

[19] T. J. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. 2, no. 4, pp. 308–320, Jul. 1976.

[20] K. Beck, "JUnit test infected: Programmers love writing tests," 2000, available from: http://junit.sourceforge.net/doc/testinfected/testing.htm (accessed 07/jun/2015).

[21] L. Williams, "What agile teams think of agile principles," *Commun. ACM*, vol. 55, no. 4, pp. 71–76, Apr. 2012.

[22] G. J. Myers, C. Sandler, T. Badgett, and T. M. Thomas, *The Art of Software Testing*, 2nd ed. John Wiley & Sons, 2004.

[23] S. Ambler, "Introduction to test-driven development (TDD)," 2002, available from: http://www.agiledata.org/essays/tdd.html (accessed 07/jun/2015).

[24] L. Briand and Y. Labiche, "Empirical studies of software testing techniques: Challenges, practical strategies, and future research," *SIGSOFT Softw. Eng. Notes*, vol. 29, no. 5, pp. 1–3, 2004.

[25] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering: an introduction*. Norwell, MA, USA: Kluwer Academic Publishers, 2000.

[26] E. Arisholm and D. I. K. Sjoberg, "A controlled experiment with professionals to evaluate the effect of a delegated versus centralized control style on the maintainability of object-oriented software," Simula Research Laboratory, Tech. Rep. 6, June 2003.

[27] V. R. Basili, F. Shull, and F. Lanubile, "Building knowledge through families of experiments," *IEEE Trans. Softw. Eng.*, vol. 25, pp. 456–473, 1999.

[28] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. E. Emam, and J. Rosenberg, "Preliminary guidelines for empirical research in software engineering," *IEEE Trans. Softw. Eng.*, vol. 28, pp. 721–734, 2002.

[29] G. Canfora, A. Cimitile, F. Garcia, M. Piattini, and C. A. Visaggio, "Evaluating performances of pair designing in industry," *J. Syst. Softw.*, vol. 80, pp. 1317–1327, 2007.

[30] I. Salman, A. T. Misirli, and N. Juristo, "Are students representatives of professionals in software engineering experiments?" in *Proc. of the ICSE 2015*. IEEE, 2015, pp. 666–676.

[31] J. C. Carver, L. Jaccheri, S. Morasca, and F. Shull, "A checklist for integrating student empirical studies with research and teaching goals," *Empirical Softw. Engg.*, vol. 15, no. 1, pp. 35–59, Feb. 2010.

[32] J. Carver and N. Kraft, "Evaluating the testing ability of senior-level computer science students," in *Proc. of the 24th IEEE-CS Conference on Software Engineering Education and Training*, May 2011, pp. 169–178.

[33] 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)*. Houston, Texas, USA: ACM, 2006, pp. 254–258.

[34] J. Offutt, N. Li, P. Ammann, and W. Xu, "Using abstraction and web applications to teach criteria-based test design," in *Proc. of the 24th IEEE-CS Conference on Software Engineering Education and Training*. IEEE, May 2011, pp. 227–236.

[35] V. G. Cerf, "Responsible programming," *Commun. ACM*, vol. 57, no. 7, pp. 7–7, Jul. 2014.