Development of Auxiliary Functions: Should You Be Agile? An Empirical Assessment of Pair Programming and Test-First Programming

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

***Abstract*—A considerable part of software systems is comprised of functions that support the main modules, such as array or string manipulation and basic math computation. These auxiliary functions are usually considered less complex, and thus tend to receive less attention from developers. However, failures in these functions might propagate to more critical modules, thereby affecting the system's overall reliability. Given the complementary role of auxiliary functions, a question that arises is whether agile practices, such as pair programming and test-first programming, can improve their correctness without affecting time-to-market. This paper presents an empirical assessment comparing the application of these agile practices with more traditional approaches. Our study comprises independent experiments of pair versus solo programming, and test-first versus test-last programming. The first study involved 85 novice programmers who applied both traditional and agile approaches in the development of six auxiliary functions within three different domains. Our results suggest that the agile practices might bring benefits in this context. In particular, pair programmers delivered correct implementations much more often, and test-first programming encouraged the production of larger and higher coverage test sets. On the downside, the main experiment showed that both practices significantly increase total development time. A replication of the test-first experiment with professional developers shows similar results.**

***Keywords*-pair programming; test-first programming; TDD; experimental software engineering; agile methods**

I. Introduction

Auxiliary functions are supportive actions of a software system that comprise 10-200 lines of code (LOC) [1, 2]. Since these functions are relatively simple, they are supposedly less critical with respect to the main development path of a project. As a consequence, they are often assigned to less experienced developers [3–5]. Despite this, these functions tend to be used by important modules of the system and their failures can easily propagate to critical components, thereby affecting the application's overall reliability [6].

There are many examples of failures originated in auxiliary functions causing significant problems throughout the history of software development. To name a few, in 1996, the unmanned Ariane 5 rocket exploded due to a failure caused by a wrong number conversion located at a supportive functionality[1] [7]. More recently, in 2010, Microsoft's Zune Media player presented a bug that caused thousands of devices to crash worldwide for an entire day [8]. The fault was located at a 15-LOC fragment of a day conversion auxiliary function[2] [9]. Apple's iPhone and Sony's PlayStation 3 presented similar problems in two auxiliary functions: daylight savings time update and leap year detection. In Apple's case, several users missed appointments because of incorrect triggering of alarms [10, 11]. For Sony, hundreds of thousands of players were unable to use their consoles for many hours [12].

Some studies have recently evaluated the application of agile practices within different contexts and with respect to a number of variables (*e.g.*, [13–21]). However, none of them have concentrated on auxiliary functionality. According to a recent investigation [21], *system complexity* – and, by extrapolation, its functions' complexity – is one of the not yet well-studied factors that seem to impinge on the effectiveness of *pair programming* [22]. In fact, the same study stresses that subsequent investigations should tackle such an aspect [21]. This also holds for other practices, such as *test-first* programming – *i.e.*, developing test cases prior to and to drive the implementation of functional code [23].

Considering the characteristics of auxiliary functions, an important question to be investigated is whether applying agile practices during their implementation can improve the system's reliability without affecting time-to-market. Pair programming demands *all* code written within a project to be developed by two people working together at a single computer [24]; and test-first programming is suggested to be applied to *every* new piece of code [23]. Therefore, a developer may wonder whether or not to resort to these practices in the development of auxiliary functions.

To obtain evidence about this question, we have conducted two independent experiments, one to compare pair with solo programming, and the other to compare test-first with

[1] In fact, the function was not useful for Ariane 5, but for earlier versions, and was only maintained for commonality reasons [7].

[2] The function splits total days into year, month, and day. Its source code can be found at http://pastie.org/349916.

test-last programming. The experiments considered reliability and effort factors, and involved 85 novice programmers and six auxiliary functions within three domains (namely, array manipulation, basic mathematics, and string manipulation). The novice programmers conservatively represent the *less experienced* developers who are natural candidates to develop supportive functionality [4, 5]. We also intentionally selected narrowly-scoped functionality to conservatively represent auxiliary functions.

To evaluate the reliability of the implemented functions, the subjects' implementations were executed on systematically developed test sets; and to evaluate the effort required to implement functions, for each subject we recorded the time taken to complete tasks. As a complementary reliability measurement, we evaluated the size and coverage of the test sets produced in the test-first experiment. Moreover, this experiment was replicated with professionals in order to obtain more practical evidence. Our goal was also to understand how the programmers' experience would impact the results. We focused the replication on the test-first experiment because previous investigations indicate that less experienced programmers tend to benefit more from pair programming [21].

Results show that while developing auxiliary functions, both agile practices might bring benefits to developers. For instance, compared with solo programmers, more than twice more pair programmers delivered correct implementations, and test-first programming caused the implementation of larger and higher coverage test sets. On the other hand, both agile practices seem to impact negatively on effort in this context. The test-first experiment replication with professionals shows similar results. This evidence shows that, taken the right cautionary measures, the agile practices seem to be a valuable option in the development of auxiliary functionality.

The remainder of this paper is structured as follows. Section II presents fundamental concepts about software testing and the target agile practices, and Section III presents how our study was set up in terms of subjects, experimental design, metrics and statistical procedures. Section IV presents the results and analysis of our experiments. In the sequence, Section V presents our study limitations, and Section VI summarizes some related research. Finally, Section VII concludes the paper.

II. Background

A. Software Testing and Testing Techniques

For the purposes of this paper, a *test case* is a set of inputs, execution conditions, and expected output for a program [2]. The expected output is evaluated based on an *oracle* which determines the correct result of the program given an input [25]. In our case, the oracle is a tester supported by JUnit[3] assertions. A test case can be more formally defined as an ordered tuple: $< (I_1, ..., I_n), EO >$, where EO is the expected output of the program when $I_1, ..., I_n$ is used as the set of inputs.

To evaluate the implementations produced in an experiment, a software testing technique can be applied. Software testing can be defined as the execution of a program against test cases with the intent of revealing faults [26]. The different testing techniques are defined based on the artifact used to derive test cases. Functional – or *black box* – testing derives test cases from the specification or description of a program. In this paper, we use functional testing as the basis to construct test cases to evaluate the correctness of the implemented programs in the experiments. Two of the most well-known functional-based testing selection criteria are *equivalence partitioning* and *boundary-value analysis*. Equivalence partitioning divides the input domain of a program into a finite number of valid and invalid input *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 [26].

Structural – or *white-box* – testing is a technique that complements functional testing. It derives test cases from the internal representation of a program [26]. Some of the well-known structural testing criteria are *statement*, *branch*, or *definition-use* [27] *coverage*, which require that all commands, decisions, or pairs of assignment and use locations of a variable be covered by test cases. In this paper we use the basic statement coverage to evaluate test sets produced while applying test-last and test-first programming. The coverage of 100% of statements of a program is considered to be a minimum requirement for adequate test sets.

B. Pair Programming and Test-First Programming

Agile development methods have been around since the late 1990s [28]. eXtreme Programming (XP) [22], Scrum [29], and Feature-Driven Development [30] are some examples of methods that call themselves *agile*. XP, one of the most popular agile methods [31], focuses on developmental practices. Pair programming is one of the core practices of XP and requires that all code developed within a project be created by two people working together at a single computer. Pair programming allegedly increases software quality without impacting time to deliver [24]. Another popular agile development technique is Test-Driven Development (TDD) [23]. TDD guides programmers to write test cases before production code.

With respect to TDD, the practice of writing the tests *before* production code is our focus in this paper (referred to as *test-first* from now on). It is important to note that test-first does not require the use of any testing technique: test cases are developed only to drive the implementation. However, a consequence of this practice is ensuring that the source code

[3] http://junit.org/ - 09/02/2011

is thoroughly unit tested [32]. The reference technique to be compared to test-first in this paper is the more conventional practice of writing the tests *after* production code (referred to as *test-last* from now on).

III. Study Setup

Our goal is to investigate the impact of using agile practices against traditional practices to develop auxiliary functions. We evaluate such practices in terms of reliability and effort factors. In this endeavor, we are interested in the following research questions:

Reliability-related. **In the development of auxiliary functions**:

R1. Can pair programming help obtain more correct implementations than solo programming?

R2. Can test-first programming help obtain more correct implementations than test-last programming?

R3. Does test-first programming encourage the implementation of more test cases than test-last programming?

R4. Do test cases produced by test-first programmers attain higher coverage than the ones produced by test-last programmers?

Effort-related. **In the development of auxiliary functions**:

R5. Do pair programmers spend more time than solo programmers?

R6. Do test-first programmers spend more time than test-last programmers?

Note that R4 complements R3, because larger test sets do not necessarily imply higher coverage test sets (*i.e.*, there can be redundant tests). Our investigation develops in terms of six hypotheses derived from these research questions. The null (0) and alternative (A) definitions of each hypothesis are described in Table I. Note that we have one hypothesis for each research question: hypothesis H_i is related to research question R_i.

Table I
HYPOTHESES FORMULATED FOR OUR EXPERIMENTS.

	Null hypothesis (0)	**Alternative Hypothesis (A)**
H_1	$\mathrm{Correctness}_{\mathrm{PP}} = \mathrm{Correctness}_{\mathrm{SP}}$	$\mathrm{Correctness}_{\mathrm{PP}} > \mathrm{Correctness}_{\mathrm{SP}}$
H_2	$\mathrm{Correctness}_{\mathrm{TF}} = \mathrm{Correctness}_{\mathrm{TL}}$	$\mathrm{Correctness}_{\mathrm{TF}} > \mathrm{Correctness}_{\mathrm{TL}}$
H_3	$\mathrm{TestSize}_{\mathrm{TF}} = \mathrm{TestSize}_{\mathrm{TL}}$	$\mathrm{TestSize}_{\mathrm{TF}} > \mathrm{TestSize}_{\mathrm{TL}}$
H_4	$\mathrm{TestCoverage}_{\mathrm{TF}} = \mathrm{TestCoverage}_{\mathrm{TL}}$	$\mathrm{TestCoverage}_{\mathrm{TF}} > \mathrm{TestCoverage}_{\mathrm{TL}}$
H_5	$\mathrm{Effort}_{\mathrm{PP}} = \mathrm{Effort}_{\mathrm{SP}}$	$\mathrm{Effort}_{\mathrm{PP}} > \mathrm{Effort}_{\mathrm{SP}}$
H_6	$\mathrm{Effort}_{\mathrm{TF}} = \mathrm{Effort}_{\mathrm{TL}}$	$\mathrm{Effort}_{\mathrm{TF}} > \mathrm{Effort}_{\mathrm{TL}}$

Legend: H = Hypothesis, SP = Solo Programming, PP = Pair Programming, TF = Test-First, TL = Test-Last

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

Subjects. The main study involved Computer Science undergraduates. 46 were in the second semester of the program and 39 were in the sixth. The second semester students were invited to perform the pair programming experiment, while the sixth semester students were invited to perform the test-first experiment. The choice was motivated by the fact that auxiliary functions tend to be natural candidates to be assigned to less experienced developers [3–5]. Moreover, a meta-analysis of several pair programming experiments indicates that junior programmers tend to benefit more from the agile practice in terms of increased correctness [21]. With respect to the test-first experiment, more knowledge about software engineering and software testing was required, and these skills are typically acquired at later stages of a Computer Science degree program. In our case, the sixth semester students had a fair amount of knowledge of agile practices, software testing, and software development. Such knowledge was acquired during the Software Engineering and Software Testing courses.

The second semester students received basic training in pair programming that comprised a 50-minutes module given during an object-oriented programming course. The sixth semester students received advanced training in test-first that comprised two JUnit/TDD modules of around 100 minutes each, and several exercises. These exercises comprised programming tasks in which students should drive implementations with test cases. It is also important to note that the sixth semester students had good knowledge about software testing techniques.

Since the second semester students did not have knowledge about software testing, the pair programming experiment did not take into account the testing approach. Also, for the test-first experiment, only solo programming was applied. Therefore it should be clear that the experiments and analysis reported here were independent, *i.e.*, they do not take into account the compound effect of pair programming and test-first programming.

We also invited professionals that were taking a course on advanced Software Engineering topics to participate in a replication of the test-first experiment. A tally of 7 subjects completed all assigned tasks and were considered in the replication. The sample included professionals in different programming and agile development levels: 4 professionals had 1-3 years of experience with Java development, whereas the other 3 had basic Java programming skills; 2 professionals had 1-3 years of experience in agile development in general, 2 professionals had 1-3 experience with pair programming, whereas the other 3 had basic knowledge about agile development. The professionals were working for companies developing software in different domains, including financial, military, and health systems.

Target Functions. To select representative functions for our study – functions that fall into the category of *auxiliary functions* as commented in Section I –, we looked into the Apache Commons project, which provides libraries of reusable Java components[4]. We also selected functionality that could easily be found through searches issued to Google Code Search [33]; that is, we tried to identify commonly used auxiliary functions that were not readily available in the Java API. We categorized these functions into three domains: array manipulation (Array), basic mathematics

[4] http://commons.apache.org/ - 23/01/2011

(Math), and string manipulation (String). 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 our functions is that they are intentionally narrowly scoped. The idea here is to perform a conservative evaluation: if a practice can impact on the development of simpler auxiliary functions, we can expect them to impact on more complex ones. With respect to size and scope, the selected functions are similar to the day conversion function that presented a fault in Microsoft's Zune media player[5][9]. Another advantage is that this type of function enables the application of more systematic test case selection techniques such as functional testing.

Test Sets. To evaluate the programs implemented by the subjects, we developed a full functional test set for each of the selected functions. The last column of Table II shows the number of test cases developed for each. To construct the test sets we applied the equivalence partitioning and boundary-value analysis criteria. 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.

Tools. Eclipse[6] was the IDE used to develop functions, and JUnit was the framework used to develop test cases. The students received some instructions in order to make sure they would concentrate their effort only on realizing the intended auxiliary functionality using the assigned techniques. For instance, the students 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. To evaluate the coverage attained by the developed test sets we used the Cobertura tool[7].

B. Experimental Design and Procedure

For the conducted experiments, we adopted the *repeated measures* with cross-over experimental design, where each subject implemented functions using both traditional and agile techniques. Such type of design supports more control to the variability among subjects [34]. To minimize the variability of the difference among functions and approaches, we randomized the assignments among students. Finally, to cancel function asymmetry, each function was assigned to be implemented using both the traditional and agile approaches.

All experiments were conducted in two sessions. In the first session, part of the students applied the traditional approaches – solo or test-last programming – and the other part applied the agile practices – pair or test-first programming; and in the second session they switched. This was done to cancel order effects; *e.g.*, it might be that applying *first* solo programming and *second* pair programming could benefit one of the approaches; similarly for the test-first experiment. Moreover, students had to implement functions from different domains in the first and second sessions. For instance, a student implementing a_1 in the first session would implement a Math or String function in the second session. This also applies for the pair programming experiment. We did this to cancel the impact of function domains on each other while implementing the functionalities in the first and second sessions.

To help understanding the adopted experimental design, Table IV present part of the assignments used for the pair programming and test-first programming experiments. Note that the subjects are not the same for the two experiments; we only maintained the first column to save space. The same procedures were applied for the test-first experiment with professionals.

C. Metrics

We adopted metrics to evaluate the *reliability* of the developed functions and the *effort* of the subjects while each approach was being applied.

Reliability Measurement. One of the metrics we adopted to measure reliability in our experiments was the correctness in terms of the *Functional Test Set Success Level* (FTSSL). FTSSL grades implemented functions in accordance with a scale composed of three values: 0 (**I**ncorrect – all test cases fail), 0.5 (**N**either correct nor incorrect – some test case fail, but not all), and 1 (**C**orrect – all test cases pass). We adopt such scale because in our context a failure is very significant: since we are using functional test sets, each test case covers an important part of the functionality. In this way, we are more interested in functions that do not fail at all, but we also want to look out for functions that are completely incorrect. Functions that fail one or more test cases, but not all, receive a middle score, since they can not be classified as correct, but at least implement correctly some part of the functionality. We then assign the highest score only to functions that pass all test cases. The FTSSL is an ordinal variable, since there is a natural ordering among the values (having all test cases passing is better than having less test cases passing, which, in turn, is better than having all test cases failing). A similar scale was adopted by Canfora et al. [13] in a pair designing experiment.

[5] We mention this function because its source code is available online. However, one can expect that the other functions mentioned in Section I are also similar in size. For instance, a leap year detection function such as PlayStation's does not require more than a few lines of code to be implemented.

[6] http://eclipse.org/ - 09/02/2011

[7] http://cobertura.sourceforge.net/ - 23/08/2011

Table II
AUXILIARY FUNCTIONS USED IN THE EXPERIMENT.

Domain	F	Description	Sample Test Case	# Test Cases
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

Legend: F = Function

Table III
EQUIVALENCE CLASSES AND BOUNDARY VALUES CONSIDERED FOR TESTING *Array Equality* (A_1).

Input Cond.	Valid Classes	Invalid Classes	Boundary Values
$\lvert$ar1$\rvert$	$\lvert$ar1$\rvert$ > -1 (C1)		$\lvert$ar1$\rvert$ = 0 (B1)
ar1 is *null*	No (C2)	Yes (C3)	
$\lvert$ar2$\rvert$	$\lvert$ar2$\rvert$ > -1 (C4)		$\lvert$ar2$\rvert$ = 0 (B2)
ar2 is *null*	No (C5)	Yes (C6)	
$\lvert$ar1$\rvert$, $\lvert$ar2$\rvert$	$\lvert$ar1$\rvert$ > $\lvert$ar2$\rvert$ (C7) $\lvert$ar2$\rvert$ > $\lvert$ar1$\rvert$ (C8)		$\lvert$a1$\rvert$ - $\lvert$a2$\rvert$ = 1 (B3) $\lvert$a2$\rvert$ - $\lvert$a1$\rvert$ = 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)

Table IV
PARTIAL TASK ASSIGNMENTS TO SUBJECTS.

S	Pair Programming Exp.		Test-First Exp.	
	1^{st} Session	2^{nd} Session	1^{st} Session	2^{nd} Session
	A + F	A + F	A + F	A + F
1	solo + s_1	pair + a_1	test-first + a_1	test-last + s_1
2	solo + s_2		test-first + a_2	test-last + s_2
3	pair a_1	solo + s_1	test-last + a_1	test-first + s_2
4		solo + m_1	test-last + a_2	test-first + s_1
⋮	⋮	⋮	⋮	⋮

Legend: S = subject, A = approach, F = function

As an additional reliability measure, we evaluate the test sets developed while using test-last and test-first programming. The two metrics we adopted were *size* and *coverage*. The size is calculated as the number of test cases contained in the test set. For code coverage, we adopted the well-known *statement coverage* criterion, which requires every statement in the code to be executed by at least one test case.

Effort Measurement. The effort metric adopted in our experiment was the *Total Development Time* (TDT), which is defined as the total number of programmers-minutes taken to develop a given function. Thus, such as in Arisholm et al.'s investigation [20], in the pair programming experiment the effort was equal to the duration for the individuals and twice the duration for the pair programmers. This is a common measure that was also used in other studies [20, 21, 35, 36].

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 applied each approach separately to develop the functions. In this case, the *paired* statistical hypothesis tests compare measures within subjects rather than across them. Paired tests are considered to greatly improve precision when compared to unpaired tests [34]. Before choosing the more adequate test to be applied, we must verify the normality of our observations. A Shapiro-Wilk test indicated that most of the continuous data observations in our experiments did not follow a normal distribution. Therefore, considering that we are also dealing

Table VI
RESULTS OF THE MAIN TEST-FIRST EXPERIMENT.

Subj.	FTSSL		TDT		Cov		# TC		Subj.	FTSSL		TDT		Cov		# TC		Subj.	FTSSL		TDT		Cov		# TC	
	TL	TF	TL	TF	TL	TF	TL	TF		TL	TF	TL	TF	TL	TF	TL	TF		TL	TF	TL	TF	TL	TF	TL	TF
1	N	N	35	45	0%	100%	0	8	14	N	N	24	24	100%	100%	5	5	27	N	N	40	20	100%	100%	5	3
2	N	N	37	44	0%	100%	0	3	15	N	I	16	60	0%	75%	0	1	28	N	N	25	30	100%	100%	2	3
3	N	N	22	33	0%	75%	0	3	16	N	N	10	15	0%	100%	0	5	29	N	C	8	25	100%	89%	4	4
4	N	N	15	35	0%	100%	0	4	17	N	N	46	60	0%	81%	0	5	30	C	N	26	48	0%	91%	0	1
5	N	N	13	25	0%	100%	0	4	18	N	N	50	23	0%	100%	0	7	31	N	N	20	37	100%	100%	4	5
6	N	N	20	21	100%	88%	3	4	19	N	N	47	21	100%	100%	2	3	32	N	N	39	12	100%	100%	4	7
7	N	N	18	16	100%	100%	6	2	20	N	N	20	27	0%	100%	0	1	33	N	N	20	32	70%	88%	4	4
8	N	N	15	14	100%	100%	5	5	21	C	N	18	37	88%	100%	2	6	34	N	N	11	12	89%	100%	5	4
9	N	N	12	15	100%	100%	2	3	22	N	N	11	19	0%	100%	0	3	35	N	I	32	40	0%	100%	0	1
10	N	C	33	33	100%	100%	4	8	23	N	N	8	23	0%	100%	0	4	36	N	N	17	33	100%	100%	7	5
11	C	C	29	25	100%	100%	5	5	24	N	N	23	27	100%	100%	4	5	37	N	N	38	33	100%	100%	1	5
12	N	N	45	15	100%	100%	0	7	25	C	C	18	21	100%	100%	6	14	38	I	N	55	35	100%	100%	1	5
13	I	N	15	10	84%	100%	2	3	26	N	N	40	40	0%	100%	0	4	39	C	N	30	50	0%	83%	0	7
Mean (considering all subjects)																					*25.67*	*29.10*	*57%*	*97%*	*2.13*	*4.51*

Table V
RESULTS OF THE PAIR PROGRAMMING EXPERIMENT.

Subj.	FTSSL		TDT		Subj.	FTSSL		TDT		Subj.	FTSSL		TDT	
	SP	PP	SP	PP		SP	PP	SP	PP		SP	PP	SP	PP
1	N	N	25	50	17	N	**C**	13	18	33	**C**	N	10	26
2	N	N	10	40	18	N	N	8	22	34	N	N	30	120
3	N	N	20	20	19	N	N	20	46	35	N	**C**	22	50
4	*I*	N	10	12	20	**C**	N	20	50	36	N	N	10	18
5	N	N	5	18	21	N	N	31	20	37	N	N	13	120
6	N	N	10	60	22	N	**C**	13	8	38	N	**C**	5	16
7	N	N	29	50	23	N	N	21	38	39	N	N	19	44
8	*I*	N	5	38	24	N	N	11	60	40	N	N	5	24
9	N	N	9	22	25	N	N	26	16	41	N	N	12	44
10	N	N	5	26	26	N	N	5	44	42	N	N	60	42
11	N	N	25	46	27	N	**C**	22	8	43	N	N	5	40
12	*I*	**C**	23	44	28	N	N	110	50	44	**C**	N	4	12
13	N	N	15	120	29	N	N	6	18	45	N	N	5	34
14	N	N	50	50	30	N	N	27	42	46	N	N	44	44
15	N	N	15	8	31	N	**C**	33	44					
16	N	**C**	30	24	32	N	N	5	34					
Mean time (considering all subjects)													*19.59*	*37.91*

with ordinal data in some cases, we decided to apply the Wilcoxon/Mann-Whitney non-parametric signed-rank paired test, which does not assume normal distributions [37]. For statistical significance we adopted the traditional confidence level of 95%; thus, our analyses consider p-values below 0.05 significant. For all statistical tests we used the R language and environment[8].

IV. RESULTS AND ANALYSIS

Tables V and VI present the results of our experiments with novice programmers, and Tables VII and VIII present the statistics for the pair programming and test-first experiments with respect to correctness. These last tables show the compared outcomes of subjects while using the traditional and agile approaches. For instance, the cell in row SP I, column PP I of Table VII contains the number of subjects that produced an *incorrect* function while using solo programming and another *incorrect* function while using pair programming. The *Frequency* columns and rows show the number of implementations produced at each Functional Test Set Success Level (I, N, or C), for each approach. For instance, the cell in row Freq., column PP I of Table VII shows that a total of 0 pair programmers produced *incorrect* functions in the experiment. With respect to the test-first experiment replication with professionals, Table IX shows the results.

Table VII
CORRECTNESS STATISTICS FOR THE PAIR PROGRAMMING EXPERIMENT.

		PP			
		I	**N**	**C**	**Freq.**
SP	**I**	0	2	1	3 (6.5%)
	N	0	33	7	40 (87%)
	C	0	3	0	3 (6.5%)
	Freq.	0 (0%)	38 (82%)	8 (18%)	

Table VIII
CORRECTNESS STATISTICS FOR THE TEST-FIRST EXPERIMENT.

		TF			
		I	**N**	**C**	**Freq.**
TL	**I**	0	2	0	2 (5%)
	N	2	28	2	32 (82%)
	C	0	3	2	5 (13%)
	Freq.	2 (5%)	33 (87%)	4 (7%)	

A. Reliability Evaluation

For the pair programming experiment, with respect to correctness, soloists implemented 3 functions that were completely correct and 3 that were incorrect (6.5%). On the other hand, pair programmers implemented 8 functions that were correct (18% – more than twice as more as solo programmers), and 0 incorrect ones. The remainder

[8] http://www.r-project.org/ - 28/01/11

Table IX
RESULTS OF THE TEST-FIRST EXPERIMENT WITH PROFESSIONALS.

Subj.	TDT		FTSSL		Cov		# TC	
	TL	TF	TL	TF	TL	TF	TL	TF
01	31	48	N	N	100%	100%	1	1
02	13	35	N	N	86%	100%	1	1
03	50	50	I	N	0%	0%	0	1
04	31	35	N	N	0%	0%	0	0
05	23	28	N	N	100%	100%	2	3
06	37	23	N	N	100%	100%	3	5
07	50	35	N	N	100%	100%	7	6
Mean	*33.57*	*36.28*	–	–	*69%*	*71%*	*2*	*2.43*

implementations failed on at least one test case, but not all. We believe this happened mainly because few students implemented functions that handled exceptional inputs, such as *null* objects. Since our test cases also covered such inputs, most implementations failed on at least one of them.

To check whether the observed difference in terms of correctness was significant, we ran the Wilcoxon test, which indicated a statistically significant difference at 95% confidence level (df = 45, p-value = 0.04982). Such result favors the *alternative* hypothesis (H_1-A) that pair programming outperforms solo programming with respect to the correctness of auxiliary functions. We believe this is a key finding, since even for supportive functionality it appears that pair programming can bring benefits to developers. Moreover, the higher number of completely correct implementations for pair programmers shows that they tend to be more careful while implementing auxiliary functions, 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 very 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 application of pair programming could even prevent problems like the ones reported by major companies, discussed in Section I.

With regard to the test-first main experiment, note that both approaches performed almost exactly equally with respect to correctness. The only difference was that test-last programmers delivered an additional correct implementation, comparing to test-first programmers. At 95% confidence level, the Wilcoxon test did not indicate a statistically significant difference (df = 38, p-value = 0.6554). This result favors the *null* hypothesis (H_2-0) that test-last and test-first programming impact similarly on the correctness of auxiliary functionality. Such evidence indicates that developers should be cautious here. Even though test-first motivated subjects to come up with more and higher coverage test sets – as will be discussed in hypothesis H5 and H6 –, functional correctness was not significantly influenced by this agile practice. Other studies have reported that even though test-first might increase development time, it also generally increases correctness [17, 38, 39].

The development of auxiliary functions with lower complexity might also have impacted this time. It might be the case that since developers perceive functions as easy to be implemented, they tend to overlook their subtleties. This problem, however, occurred less in the context of pair programming, as one of the developers might be more careful than the other about such nuances. It must also be noted that test-first programming does not prescribe any testing technique or criterion to be followed, and even though the programmers already had knowledge about functional testing and other testing techniques, they might have developed test cases only to drive the implementation. Also note that other properties of the implemented functions were not analyzed (*e.g.*, design quality factors, which are sometimes pointed out to be enhanced by test-driven development [16]). It should also be noted that our results are conservative in the sense that the functions selected for our experiments are narrowly scoped. Test-first programming might be more effective for more complex functionality.

Test Set Size and Coverage. With respect to the size of the produced test sets, note that the means are 53% higher for test-first programming. At 95% confidence level, the statistical test shows a significant difference (df = 38, p-value = 0.00001708). Such result favors the alternative hypothesis (H_3-A) that test-first outperforms test-last programming with respect to test set size. It is interesting to mention that the difference continues to be significant even when we remove results from test-last programmers that did not develop any test cases (test-last mean test set size = 3.77; test-first mean test set size = 4.91; Wilcoxon test: p-value = 0.03216).

This is an important result in favor of the agile approach, since it encouraged the implementation of more test cases than test-last programming. The fact that some test-last programmers did not test their implementations can be seen as a negative effect of that approach. In fact, Beck [23] even affirms that "*any program feature without an automated test simply does not exist*". In the long run, the mindset of having to develop at least a minimum set of test cases might compensate the extra effort required to develop them, even when there are no directly observable improvements on correctness. The absence of regression tests for some functions, for instance, can break the confidence feel targeted by TDD. This promoted "safety net" [40] helps alleviating the fear that added code might have broken other parts of the system and can also improve its reliability.

Our results are also consistent with the argument of test-first promoters that if you leave the task of testing programs to the end of the developmental cycle, you might end up not testing them at all. It is interesting to see that this can also hold for the development of auxiliary functions, such as the ones selected for our experiments.

With respect to test coverage, note that the agile approach outperformed the traditional approach, covering 40% more statements. The statistical test shows a significant difference at 95% confidence level (df = 38, p-value = 0.00003341). Such result favors the alternative hypothesis (H_4-A) that test-first outperforms test-last programming with respect to test set coverage.

Note that even though the test sets produced with test-first programming were significantly larger and attained better coverage, they did not seem to impact directly on the correctness of functions (*i.e.*, test-last and test-first programming performed similarly with respect to that factor). However, having larger and higher coverage test sets might be beneficial for other reasons. For instance, in regression testing, better test sets can improve the chance of finding faults at the integration of auxiliary functions with other parts of the system, when these parts are changed later in a project. Lower coverage test sets can fail to reveal the introduced faults, because the paths in the program that could be sensitized by the changes might not be executed.

B. Effort Evaluation

With respect to effort, note that the mean total development time for pair programming was 48% higher than for solo programming. One of the reasons that might explain such outcome is that subjects were novice programmers and target functionalities were narrowly scoped. It might be the case that pair programming starts to be more productive after some development time, improved, for instance, by *pair jelling* [19]. Since the target functions required little time to develop, subjects might not have had time to take advantage from the technique, from an effort perspective. For these outcomes, at 95% confidence level, the Wilcoxon test confirms a statistically significant difference between the means (df = 45, p-value = 0.000009278). Such result favors the *alternative* hypothesis (H_5-A) that solo programming outperforms pair programming with respect to effort.

Some of the previous pair programming investigations [13, 19] reached different conclusions, with pairs developing sometimes 40-50% faster than solo programmers. In Williams et al.'s experiment [19], however, only programmers with long-standing industry experience were included, and not novice programmers. Canfora et al.'s experiment [13], on the other hand, addressed *pair designing* and maintenance tasks of models such as use cases and class diagrams, and not programming. These differences and, more importantly, the fact that we targeted narrowly-scoped functionality might explain the results reached in our experiment.

For the test-first experiment, the mean value for the total development time while using test-first programming was 12% higher than for the test-last approach. At 95% confidence level, the Wilcoxon test again showed a statistically significant difference (df = 38, p-value = 0.0463). Such result favors the *alternative* hypothesis (H_6-0) that test-last outperforms test-first programming with respect to effort. Our conclusion is similar to other studies that have also reported that test-first programming seems to impact negatively on effort [15, 17, 39]. This is probably due to the additional effort of having to develop test cases first. We noticed that, even though we encouraged subjects to test their implementations while using the test-last approach, many did not develop any test cases. In fact, when we remove their results from the data set, the difference between the means becomes non-significant (test-last mean development time = 24.77; test-first mean development time = 24.22; Wilcoxon test: p-value = 0.5446). Since test-first programming requires developers to drive the implementation with tests, differently from test-last programming, all subjects developed test cases while applying that approach. As discussed earlier, this advantage might compensate the extra effort observed.

C. Test-First Experiment Replication and Further Discussions

For the replication with professionals, since the sample was small, we do not present statistical tests, but only analyze the outcomes descriptively. Note that, with respect to correctness, the approaches again performed similarly. However, the sensible difference at this time is that only a test-last programmer produced an incorrect function, while no test-first programmers produced such outcome. With respect to effort, the approaches also performed very similarly, with test-last programmers taking, on average, only 3 additional minutes to complete their tasks. In terms of test set completeness, test-first programmers performed slightly better, both with respect to size and coverage. These results are consistent with all outcomes produced by the novice programmers, so we believe they reinforce the evidence for the favored hypotheses in the main experiment.

It should be clear that our experiments take into account two main factors while developing software: reliability in terms of functional correctness and test set size and coverage, and effort in terms of total development time. The agile practices might also impact positively on other aspects, such as quality in terms of design metrics, knowledge dissemination, and programmer satisfaction [41]. This should also be taken into account while considering the application of pair programming and test-first programming to develop auxiliary functionality.

One might argue that the functions targeted in our experiments could be reused from libraries such as Apache Commons (see Section III), instead of being developed from scratch. We agree that this is a feasible solution when auxiliary functions come up in a given project. However, there are some auxiliary functions that are harder to be adapted to a specific context. These should not be easily found in general-purpose code libraries or adapted from them, and so reuse would not be an option. Moreover, it is clear that a function like Sony's Playstation leap year detection discussed in Section I, for instance, could also be found at some general-purpose code library, but Sony decided to develop it themselves. This is an evidence that the situation simulated in our experiment is not unrealistic.

For pair programming, a point for further investigation is whether pairing novice and experienced programmers would yield better results in our context. In the case of auxiliary functionality, the different backgrounds might help to better address the involved nuances. For instance, novice programmers might help softening the impact of programming vices presented by experienced programmers; and experienced programmers might help overcoming skill deficiencies presented by novice programmers.

Since pair programming seems to impact positively in the correctness of auxiliary functions and test-first programming on the test set quality, the use of both practices in conjunction might be a good choice in this context. Moreover, our results also indicate that it might be interesting to apply some formal testing technique along with test-first programming. Since the agile practice appears to produce larger and higher coverage test sets that do not impact directly on the correctness of functions, a testing technique such as functional testing might be of help in this case.

V. Study Limitations

This section discusses the study limitations based on three categories of threats to validity described by Wohlin et al. [42]. Each category includes several possible threats for an experimental study. For each category, we list all possible threats, measures taken to reduce each risk, and suggestions for improvements in future evaluations.

A. Internal Validity

An internal validity threat that may have affected our experiments was the lack of control of the following variables: (1) the subjects' skill (other than all being in the same semester of the course); and (2) how the pairings took place (they were simply randomized). With respect to (1), the repeated measures design decreases the probability of this threat affecting our outcomes, because the same students played the roles of solo and pair programmers, and performed both test-last and test-first. Threat (2) has possibly affected the pair programming experiment. In regard to it, we believe the sample size – which was not small (46 subjects performing solo and pair programming) – reduces the extent of this threat's effect (*i.e.*, different types of pairings with respect to skill might have occurred). In this case we followed the rule of thumb "block what you can, randomize what you cannot" [43], since if we were to block pairs by skill, we might not have had sufficient subjects. In any case, for future evaluations we suggest that pairings should be blocked with respect to skill – such as in a recent study by Arisholm et al. [20] – to cancel this threat (even though this might require larger samples).

Another internal validity aspect to be considered in our experiments is the *mortality*. Since we had 123 students invited to participate in the main experiments in the beginning, the actual tasks that took place did not follow the initial assignments. This could affect the balance of the assignments that was taken care in the experiment design. However, since in both experiments we had a considerable sample size (46 – pair programming experiment; 39 – test-first experiment), we believe an adequate balance could still be maintained. This is reinforced by the fact that, to help in the event of some drop outs, the initial assignment set contained some redundancies.

B. External Validity

A characteristic of our experiment that might reduce its external validity is the use of students as subjects for the larger experiments. In fact, some studies have shown opposite trends for students and professionals (*e.g.*, Arisholm and Sjoberg [44]). However, according to other authors, students can play an important role in experimentation in the field of software engineering [45, 46] (specially for an initial evaluation like ours). For instance, Canfora et al. [13] have conducted a pair programming experiment in academia and replicated the same experiment in industry. According to the authors, the experiments produced similar results from both samples. Since the experiment evaluated aspects of agile practices similar to ours, we believe our results might also be generalizable in that sense. In any case, to minimize the influence of such nuisance factor, we also replicated the test-first experiment with a sample of 8 professionals, which provided results similar to the main experiment.

Another threat to the external validity of our experiments is the representativeness of the selected functions. One might argue that the functionalities do not represent the population of auxiliary functions, due to their lack of complexity. However, as commented earlier, it was our intention to select narrow-scoped functionality to conduct a conservative evaluation. Since in some cases the agile practices did impact on the development of the selected functions, we can expect them to impact on more complex ones as well. In any case conducting more experiments using larger auxiliary functions is one way to further reduce this threat. In the future we plan to replicate our experiments with more complex auxiliary functionality present in open source systems.

C. Construct Validity

A characteristic of our pair programming experiment that might have affected its construct validity is that students had little or no previous experience with pair programming and, in most cases, had not programmed with their partners before. Therefore, similarly to Arisholm et al.'s study [20], our results might be conservative with respect to the effects of pair programming. Subsequent experiments should consider programmers that have more experience with this development approach. Moreover, it should be noted that we analyzed the agile practices separately, that is, we do not study the compound effect of pair programming and test-first programming.

VI. Related Work

In a recent work, Hannay et al. [21] carried out a meta analysis of the effectiveness of pair programming when compared to solo programming. All selected primary studies presented quantitative data regarding at least one of the three measures: quality and reliability (mostly in terms of correctness), duration, and effort. The subjects could be either students or professional software developers. In total, 18 studies were selected to compose the dataset to which a series of statistical procedures was applied. The results indicate that pair programming (i) slightly outperforms solo programming with respect to quality, (ii) has a medium positive overall effect regarding duration, and (iii) has a medium negative overall effect on effort.

Some other interesting observations made by Hannay et al. regard the expertise of the subjects involved in the selected studies. Considering junior, intermediate and senior pairs, the most noticeable gains and losses compared to individual subjects are: (gain) 73% increase in quality and (loss) 111% increase in effort for junior pairs, (gain) 28% decrease in duration and (loss) 43% increase in effort for intermediate pair, and, (loss) 83% increase in effort and, surprisingly, no overall gains for senior pairs.

Salleh et al. [41] presents the results of another systematic literature review concerning pair programming effectiveness. Differently from Hannay et al. [21], Salleh et al. analyzed compatibility factors (*e.g.*, the *feel-good*, personality, and skill level factors) and their effect on pair programming effectiveness as a pedagogical tool in Computer Science and Software Engineering education. Four measures were analyzed: academic performance, technical productivity, program/design quality and learning satisfaction. The general findings of Salleh et al. [41] are that, when compared to solo programming, pair programming is more effective in terms of technical productivity, learning satisfaction and academic performance, while no significant differences were found in regard to program/design quality. Studies that applied internal and external quality metrics revealed a slight positive effect of pair programming over solo programming.

Desai et al. [47] investigated TDD experiments in the academic environment. In general, they observed that when the control group in a controlled experiment used iterative test-last programming (*i.e.*, continuous testing), no significant differences were noticed regarding the quality of the produced software. However, when all code was written before the tests started to be implemented, test-first programming outperformed the test-last approach in terms of fault counts (reduction between 35% and 45%). Slight gains (5% – 10%) in productivity in favor of test-first were also reported in the analyzed studies.

To the best of our knowledge, the study presented in this paper is the first that evaluates agile practices in the development of auxiliary functions. Moreover, compared to other similar studies, ours is the first to apply the repeated measures with cross-over experimental design. Most investigations make use of *two-group* experimental designs where a control group applies, for instance, a traditional approach, and a treatment group applies the agile approach. This kind of design is less powerful than repeated measures because comparisons are made across – and not within – subjects. With repeated measures extraneous error variance is reduced because each subject serves as his/her own control [48]. Our study is also one of the few that applies systematic test case design to evaluate the correctness of implementations produced by the subjects. Most studies use test cases developed in an ad hoc manner, which might introduce bias to the analysis.

With respect to the achieved results, previous evidence regarding the effectiveness of pair programming and test-first is generally contradictory. The aforementioned surveys [21, 41, 47] show that varied scenarios lead to different effectiveness measures, sometimes favoring the agile practices, sometimes not. Our results show that, to implement auxiliary functionality, both agile practices required substantial increase in the development effort, but offered a counterpart in terms of significant correctness improvement, and larger and higher coverage test sets.

VII. Conclusions

Recent studies have reported on evaluations of pair programming and test-first programming with respect to a number of characteristics [13–21]. However, none of them have tackled the development of auxiliary functionality with respect to the correctness gain and impact on time-to-market. Our study investigated the following factors: *reliability* in terms of functional correctness and test set size and coverage, and *effort* in terms of total development time. Results show that pair programming tends to increase reliability in terms of correctness, while test-first programming tends to increase reliability in terms of test set size and coverage. Our study also shows that the agile practices seem to impact negatively on effort. The replication of the test-first experiment with professionals shows similar results.

We believe these results may aid developers in choosing among practices while implementing auxiliary functionality and considering their impact on the overall reliability of the system. In particular, pair programming seems to be effective in improving the system's correctness. Even though it seems to impact negatively on effort, the benefit might compensate in the end. As shown by recent events involving companies such as Microsoft and Apple (see Section I), problems in auxiliary functions can bring damages to the rest of the system, affecting users' lives.

Acknowledgements

The authors thank CNPq, CAPES, FAPERJ, and FAPESP for financial support, and E. Guerra for experimental support. Otavio is supported by FAPESP (2010/15540-2); and Alessandro is supported by FAPERJ: dist. scientist (E-26/102.211/2009) and DANSis project (E-26/111.152/2011), CNPq: productivity scholarship (305526/2009-0) and Universal Project (483882/2009-7, 483699/2009-8, 485348/2011-0), CAPES: international collaboration scheme (5688-09), PROCAD-NF (720/2010), and PUC-Rio (productivity grant).

References

[1] O. A. L. Lemos, S. Bajracharya, J. Ossher, P. C. Masiero, and C. Lopes, "A test-driven approach to code search and its application to the reuse of auxiliary functionality," *Inf. Softw. Technol.*, vol. 53, pp. 294–306, 2011.

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

[3] A. Begel and B. Simon, "Novice software developers, all over again," in *Proc. of the ICER '08*. New York, NY, USA: ACM, 2008, pp. 3–14.

[4] B. Dagenais, H. Ossher, R. K. E. Bellamy, M. P. Robillard, and J. P. de Vries, "Moving into a new software project landscape," in *Proc. of the ICSE '10*. New York, NY, USA: ACM, 2010, pp. 275–284.

[5] L. Williams and R. Kessler, *Pair Programming Illuminated*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002, ch. 15, p. 71.

[6] B. Littlewood and L. Strigini, "Software reliability and dependability: a roadmap," in *Proc. of the ICSE '00*. New York, NY, USA: ACM, 2000, pp. 175–188.

[7] Lions, Jacques-Louis et al., "ARIANE 5 Flight 501 Failure, Report by the Inquiry Board," 1996, available from: http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf (accessed 06/09/2011).

[8] BBC News, "Microsoft zune affected by 'bug'," 2008, available from: http://news.bbc.co.uk/2/hi/technology/7806683.stm (accessed 17/08/2011).

[9] W. Weimer, S. Forrest, C. Le Goues, and T. Nguyen, "Automatic program repair with evolutionary computation," *Commun. ACM*, vol. 53, pp. 109–116, May 2010.

[10] Apple Inc., "iOS 4.1: Repeating alarms may trigger incorrectly before or after DST change," 2011, available from: http://support.apple.com/kb/TS3542 (accessed 12/08/2011).

[11] The Telegraph, "Apple iPhone 4 clock bug strikes again," 2011, available from: http://www.telegraph.co.uk/technology/apple/8380346/Apple-iPhone-4-clock-bug-strikes-again.html (accessed 16/08/2011).

[12] R. Cellan-Jones, "Sony's leap year bug," 2010, available from: http://www.bbc.co.uk/blogs/thereporters/rorycellanjones/2010/03/sonys_millennium_bug.html (accessed 06/09/2011).

[13] 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.

[14] B. George, "Analysis and quantification of test driven development approach," Master's thesis, North Carolina State University, 2002.

[15] A. H. P. Abrahamsson and J. Jäälinoja, "Improving business agility through technical solutions: A case study on test-driven development in mobile software development," in *Business Agility and Information Technology Diffusion*. Springer, 2005, pp. 227–243.

[16] D. Janzen and H. Saiedian, "Does test-driven development really improve software design quality?" *IEEE Softw.*, vol. 25, pp. 77–84, 2008.

[17] B. George and L. Williams, "An initial investigation of test driven development in industry," in *Proc. of the ACM SAC 2003*, ser. SAC '03. New York, NY, USA: ACM, 2003, pp. 1135–1139.

[18] S. H. Edwards, "Using test-driven development in the classroom: Providing students with concrete feedback on performance," in *Proc. of the EISTA '03*, August 2003.

[19] L. Williams, R. R. Kessler, W. Cunningham, and R. Jeffries, "Strengthening the case for pair programming," *IEEE Softw.*, vol. 17, pp. 19–25, 2000.

[20] E. Arisholm, H. Gallis, T. Dyba, and D. I.K. Sjoberg, "Evaluating pair programming with respect to system complexity and programmer expertise," *IEEE Trans. on Softw. Eng.*, vol. 33, pp. 65–86, 2007.

[21] J. E. Hannay, T. Dybå, E. Arisholm, and D. I. K. Sjøberg, "The effectiveness of pair programming: A meta-analysis," *Inf. Softw. Technol.*, vol. 51, pp. 1110–1122, 2009.

[22] K. Beck, *Extreme programming explained: embrace change*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

[23] ——, *Test Driven Development: By Example*. Addison-Wesley, 2002.

[24] D. Wells, "Extreme Programming: a gentle introduction," 1999, available from: http://www.extremeprogramming.org (accessed 09/02/2011).

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

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

[27] 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.

[28] C. Larman and V. R. Basili, "Iterative and incremental development: A brief history," *Computer*, vol. 36, pp. 47–56, 2003.

[29] K. Schwaber and M. Beedle, *Agile Software Development with Scrum*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2001.

[30] S. R. Palmer and M. Felsing, *A Practical Guide to Feature-Driven Development*, 1st ed. Pearson Education, 2001.

[31] M. Pikkarainen, J. Haikara, O. Salo, P. Abrahamsson, and J. Still, "The impact of agile practices on communication in software development," *Empirical Softw. Engg.*, vol. 13, pp. 303–337, 2008.

[32] S. Ambler, "Introduction to test driven design (TDD)," 2010, available from: http://www.agiledata.org/essays/tdd.html (accessed 17/02/2011).

[33] Google Inc., "Google code search," 2011, available from: http://www.google.com/codesearch/ (accessed 09/02/2011).

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

[35] H. Erdogmus, M. Morisio, and M. Torchiano, "On the effectiveness of the test-first approach to programming," *IEEE Trans. on Softw. Eng.*, vol. 31, pp. 226–237, 2005.

[36] K. M. Lui, K. C. C. Chan, and J. Nosek, "The effect of pairs in program design tasks," *IEEE Trans. Softw. Eng.*, vol. 34, pp. 197–211, 2008.

[37] F. Shull, J. Singer, and D. I. Sjøberg, *Guide to Advanced Empirical Software Engineering*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007.

[38] E. M. Maximilien and L. Williams, "Assessing test-driven development at ibm," in *Proc. of the ICSE '03*. Washington, DC, USA: IEEE Computer Society, 2003, pp. 564–569.

[39] S. H. Edwards, "Using software testing to move students from trial-and-error to reflection-in-action," *SIGCSE Bull.*, vol. 36, pp. 26–30, 2004.

[40] R. Jeffries and G. Melnik, "Guest editors' introduction: Tdd–the art of fearless programming," *IEEE Softw.*, vol. 24, pp. 24–30, 2007.

[41] N. Salleh, E. Mendes, and J. Grundy, "Empirical studies of pair programming for CS/SE teaching in higher education: A systematic literature review," *IEEE Trans. on Softw. Eng.*, vol. 99, no. PrePrints, 2010.

[42] 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.

[43] G. E. P. Box, W. G. Hunter, and J. S. Hunter, *Statistics for Experimenters*. John Wiley & Sons, 2005.

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

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

[46] 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.

[47] C. Desai, D. Janzen, and K. Savage, "A survey of evidence for test-driven development in academia," *ACM SIGCSE Bulletin*, vol. 40, pp. 97–101, 2008.

[48] S. E. Maxwell and H. Delaney, *Designing Experiments and Analyzing Data: A Model Comparison Perspective*, 1st ed. Routledge Academic, 2003.