Software Quality Journal (2024) 32:1137–1174
https://doi.org/10.1007/s11219-024-09682-4

RESEARCH

Unraveling the code: an in-depth empirical study on the impact of development practices in auxiliary functions implementation

Otávio Lemos[1] · Fábio Silveira[1] · Fabiano Ferrari[2] · Tiago Silva[1] · Eduardo Guerra[3] · Alessandro Garcia[4]

Accepted: 24 May 2024 / Published online: 25 June 2024

Abstract
Auxiliary functions in software systems, often overlooked due to their perceived simplicity, play a crucial role in overall system reliability. This study focuses on the effectiveness of agile practices, specifically the pair programming and the test-first programming practices. Despite the importance of these functions, there exists a dearth of empirical evidence on the impact of agile practices on their development, raising questions about their potential to enhance correctness without affecting time-to-market. This paper aims to bridge this gap by comparing the application of agile practices with traditional approaches in the context of auxiliary function development. We conducted six experiments involving 122 participants (85 novices and 37 professionals) who used both traditional and agile methods to develop six auxiliary functions across three different domains. Our analysis of 244 implementations suggests the potential benefits of agile practices in auxiliary function development. Pair programming showed a tendency towards improved correctness, while test-first programming did not significantly extend the total development time, particularly among professionals. However, these findings should be interpreted cautiously as they do not conclusively establish that agile practices outperform traditional approaches universally. As indicated by our results, the potential benefits of agile practices may vary depending on factors such as the programmer's experience level and the nature of the functions being developed. Further research is needed to fully understand the contexts in which these practices can be most effectively applied and to address the potential limitations of our study.

Keywords Pair programming · Test-first programming · TDD · Experimental software engineering · Agile

Otávio Lemos, Fábio Silveira, Fabiano Ferrari, Eduardo Guerra and Alessandro Garcia authors contributed equally to this work.

Extended author information available on the last page of the article

1 Introduction

In the context of software engineering, auxiliary functions play a vital role as supportive actions within a system or component, aiding in the execution of primary functions. These auxiliary functions, typically comprising a relatively small number of lines of code, serve to assist other functions in fulfilling their tasks efficiently and effectively (Lemos et al., 2011; IEEE, 1990).

Since these functions are relatively simple and usually self-contained, they are supposedly less critical with respect to the main development path of a project. Consequently, they are often assigned to less experienced developers (Begel & Simon, 2008; Dagenais et al., 2010). Nevertheless, these functions tend to be used by important modules of the system, and their failures can easily propagate to critical components, thus affecting the overall reliability of the application. Their considerable economic impact underscores the significance of such failures.

In fact, a study estimated that software defects cost the U.S. economy $2.08 trillion in 2020 (Krasner, 2021). Even though catastrophic software errors are fortunately scarce nowadays, the possibility of chaos remains. Inadequate testing is one of the most critical factors contributing to poor software quality. Throughout software development history, many examples of failures originating from auxiliary functions caused significant problems. In 2010, Microsoft's Zune Media player presented a bug that caused tens of thousands of devices to malfunction for a full day. The fault was discovered in a 15-LOC fragment of an auxiliary conversion function of the time (Weimer et al., 2010). Apple's iPhone and Sony's PlayStation3 both had issues with two auxiliary functions: daylight savings time update and leap year detection. In Apple's case, many users missed appointments due to incorrect alarm triggers (Spence, 2011). For Sony, hundreds of thousands of players could not use their consoles for extended periods (Cellan-Jones, 2010).

Given the characteristics of auxiliary functions and their potential impact on system reliability, an essential question arises: can the application of agile practices during their implementation enhance system reliability without compromising time-to-market? Two of the most popular agile practices are pair programming and test-first programming. The concept of pair programming requires integrating two people on a single computer to produce the code written for a project; concurrently, it is suggested that test-first programming is implemented with each new snippet of code being developed (Beck, 2002). However, developers may wonder whether to adopt these practices in developing auxiliary functions.

Over the past years, numerous studies have examined the application of agile principles across different contexts, taking into account a multitude of variables (*e.g.*, Canfora et al., 2007; Abrahamsson et al., 2005; George & Williams, 2003; Arisholm et al., 2007; Hannay et al., 2009; Fucci et al., 2017; Munir et al., 2014; Fucci et al., 2018; Sun et al., 2016; Rafique & Mišić, 2013; L. Salge & Berente, 2016; Saltz & Shamshurin, 2017; Romano et al., 2019; Tosun et al., 2021; Xu & Correia, 2023). However, none of them have directly emphasized auxiliary functions. Some studies (Hannay et al., 2009; Demir & Seferoglu, 2021) show that system complexity (and, by extrapolation, its functions' complexity), flow experience, and coding quality are ones of the not yet well-studied factors that seem to impinge on the effectiveness of pair programming. 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 (Beck, 2002).

We focus on pair and test-first programming because these practices have become popular with the agile movement, even though their benefits are not self-evident. In

fact, both have been considered the flagship and most influential practices of eXtreme Programming, but also the most controversial ones (Madeyski, 2010; Zhong & Li, 2020; Zieris & Prechelt, 2021). Since they may require more effort when compared to their traditional counterparts (solo and test-last programming), many do not advocate their application (Meyer, 2014), or suggest they be used eventually (*e.g.*, for the development of complex parts of the system (Trikha, 2014)). At the same time, both practices allegedly improve software quality: pair programming through its live code inspection aspect (Williams & Kessler, 2002), and test-first programming by requiring the continuous creation and execution of automated test cases (Nagappan et al., 2008). While most studies that evaluate agile practices focus on a single technique, we believe the investigation of two key practices – which are, at the same time, controversial – can provide a more general idea about the application of the agile philosophy to software development (in our case, in the specific scenario of auxiliary functions). The investigation of two practices with the same experimental design can also support a more general comparison between them.

We conducted two independent experiments to obtain evidence regarding the application of agile practices in developing auxiliary functions: i) comparing pair programming with solo programming; and ii) comparing test-first programming with test-last programming. First, we carried out these experiments with students. Then, we replicated these experiments, including 37 professionals from diverse backgrounds. Adding a professional sample to our study allows us to better generalize our results to a broader population of developers.

To compare results more effectively, we conducted our experiments using the repeated measures – or within-subject – design (Montgomery, 2006), where each subject applies both target approaches at two different times. Thus, comparisons can be made within – rather than across – subjects, and paired hypothesis tests can be applied, providing stronger statistical evidence.

The experiment examined the reliability and effort factors of 85 novice programmers who performed experiments involving six auxiliary functions within three domains: array manipulation, basic mathematics, and string manipulation. We also intentionally selected narrowly-scoped functionality to represent auxiliary functions conservatively. To understand how the programmers' experience would impact results in a more extensive analysis, each experiment was replicated twice with professionals, resulting in a total of six experiment instances (two with novices and four with professionals).

The subject's implementations were systematically executed on developed test sets to evaluate the implemented functions' reliability. To determine the effort required to implement functions, we recorded the time required to execute tasks for each subject. Further, we evaluated the size and coverage of the test sets generated in the test-first programming experiments as an additional reliability measurement.

Upon analyzing the data collected from these methods, we obtained insights into how the use of agile practices affected the development of auxiliary functions. According to the results, in our context, adopting these practices during the development of auxiliary functions might benefit developers. For instance, compared to solo programmers, approximately twice more pair programmers delivered correct implementations for novices and professionals. Test-first programming, conversely, caused the implementation of more extensive and higher coverage test sets (for novices and professionals), and more correct implementations (for professionals). An interesting observation about pair programming is that such a practice made programmers more cautious with subtle bugs, causing the implementation of more robust code compared to solo programming. However, it must be noted that both practices can sometimes negatively impact the effort in this context. The available evidence suggests that implementing agile practices in the development of auxiliary

functionalities can be a valuable approach. It indicates that it might even be the case of using the agile practices 100% of the time since they seem effective even for functions that would not be initial candidates to be developed with their use. This is especially true for pair programming, which in several companies is applied only when dealing with complex parts of the system. In any case, developers should always be aware of the additional effort that the practices might bring, considering the trade-offs.

With respect to the difference between novices and professionals, we observed that professionals benefited more from agile practices. For example, code implemented with test-first programming was significantly more correct when compared to test-last programming for such a sample. Additionally, for professionals, there was no significant difference in terms of effort between test-first and test-last programming. Both effects – improved correctness and lack of additional effort – were not observed in the sample of novice programmers for test-first programming. Our investigation, which considers the perspectives of both novice programmers and professionals, seeks to contribute to the body of knowledge in the field of agile practices.

The remainder of this paper is organized as follows. Section 2 describes the fundamentals of software testing and the target agile practices, and Sect. 3 summarizes some related work. Next, Sect. 4 presents ethical implications, the subjects, experimental design, metrics, and statistical procedures of our study. Section 5 presents and analyzes the results of the experiments comparing the software development techniques performed with novices and professionals. Section 6 provides an in-depth analysis of these results, summarizing the outcomes in relation to the initially formulated hypotheses and discussing the potential implications of these findings. Next, Sect. 7 presents our study limitations. Finally, Sect. 8 concludes the paper.

2 Background

This section presents basic background on software testing (Sect. 2.1) and on the agile practices addressed in this research, in particular, pair programming and test-first programming (Sect. 2.2).

2.1 Software testing and testing techniques

The software testing activity aims to ensure the best possible quality of software products. One of the reasons why testing has gained such denotative importance is that it consumes about 50% of the expended effort on software development (Pressman & Maxim, 2020). In recent years, the rise of Agile and DevOps methodologies has led to a shift towards continuous testing, where testing is integrated throughout the software development lifecycle. This way, software testing becomes a more crucial part of the development process, ensuring the software is thoroughly tested and validated before it is released to the end users.

For this paper, a test case is a collection of inputs, execution conditions, and expected output for a program. Given an input, the expected output is evaluated using an oracle that determines the correct program result. In our case, the oracle is implemented as JUnit[1] assertions.

[1] http://junit.org/ – accessed in July, 2023.

A test case can be formally defined as the following ordered tuple: $(I_1, ..., I_n), EO >$, where EO is the expected output of the software when inputs are $(I_1, ..., I_n)$.

According to Myers et al. (2004), software testing is the execution of a program against test cases to reveal faults in the software. Testing techniques differ from each other according to the artifact from which test cases are derived. The test cases for functional testing are derived from the program's specifications. In this paper, functional testing serves as the foundational technique for constructing test cases, intending to evaluate the accuracy of the implemented programs. Equivalence partitioning and boundary value analysis are the most widely recognized functional-based testing selection criteria. Equivalence partitioning divides a program's input domain into a finite number of valid and invalid input classes. Then, it is presumed that evaluating any other value within the same class is equivalent to a test case with a representative value. This criterion requires either common or individual test cases to cover valid classes, and requires individual test cases for each invalid class. Boundary value analysis supplements equivalence partitioning by requiring test cases to include values at the boundaries of equivalence classes (Myers et al., 2004). In this paper, we used equivalence partitioning and boundary value analysis to create the test sets that were used to evaluate the reliability of functions implemented by experimental subjects.

2.2 Pair programming and test-first programming

Agile development methodologies, including eXtreme Programming (XP), Scrum, and Feature-Driven Development, have emerged since the late 1990s (Williams, 2012). XP, recognized as one of the most prevalent agile methods (Dingsøyr et al., 2012), accentuates development practices wherein pair programming (PP) assumes a pivotal role. Pair programming is a practice that requires two developers working together on the same task, sharing one computer. Alternatively, in the current era of remote work and distributed teams, this can also be carried out via distributed pair programming (DPP), where both developers collaborate virtually using shared coding environments and communication tools to work on the same section of the computer code. One of the main advantages of pair programming is that it promotes knowledge sharing and learning between team members. By working together, developers can share their expertise and learn from each other, which can help to improve the overall quality of the code (Swamidurai & Umphress, 2012). Additionally, pair programming can allegedly reduce errors and improve the speed and efficiency of development.

Test-Driven Development (TDD), another widespread agile development practice (Beck, 2002), advocates the creation of test cases before the actual implementation of production code. This paper focuses on such a practice, which will also henceforth be referred to as test-first. Importantly, test-first does not require the use of a particular testing technique; test cases are developed solely to drive the implementation. A consequence of this practice is ensuring that the source code is thoroughly unit tested. This work compares test-first to the more conventional practice of writing tests after production code (henceforth referred to as test-last).

TDD can be used as a software design technique, where tests are used for defining APIs and class relationships, or it can be used only as a development technique (Guerra & Aniche, 2016). In the latter case, tests defined before the production code are used to incrementally guide the introduction of functionality. In the experiments described in our study, we evaluated test-first programming only as a development technique since the functions' signatures were previously specified (see Sect. 4).

3 Related work

This section presents studies that are related to ours. First, we focus on pair programming (Sect. 3.1) and on test-first programming and TDD (Sect. 3.2). We then refer to other related investigations (Sect. 3.3), preparing the groundwork for discussing our study's main findings in subsequent sections.

3.1 Studies related to pair programming (PP)

A comprehensive meta-analysis evaluating the effectiveness of PP in comparison to solo programming was conducted by Hannay et al. (2009). The study analyzed quantitative measures of quality, duration, and effort, drawing upon data from 18 studies involving student and professional developers. The findings indicate a slight quality improvement and a moderately positive effect on task duration[2] with PP, despite a medium negative impact on effort.[3] The research also notes variances based on developer experience levels: junior pairs showed a 73% quality increase at the cost of 111% more effort, and intermediate pairs achieved a 28% reduction in duration. However, they expended 43% more effort, whereas senior pairs experienced an 83% effort increase with no noticeable gains.

Salleh et al. (2010) reported the outcomes of a systematic literature review (SLR) focused on the effectiveness of PP. In contrast to Hanny et al. (2009), Salleh et al. examined compatibility factors (e.g., the *feel-good*, personality, and skill level factors) and their impact on the effectiveness of PP as an educational tool in Computer Science and Software Engineering education. Four aspects were assessed: academic performance, technical productivity, program/design quality, and learning satisfaction. The general findings suggest that, in comparison to solo programming, PP proves to be more effective in terms of technical productivity, learning satisfaction, and academic performance. However, no significant differences were observed concerning program/design quality. Studies employing internal and external quality metrics indicated a marginally positive effect of PP over solo programming (Salleh et al. 2010).

Sun et al. (2016) carried out a survey with software professionals considering their views regarding the effectiveness of PP practices versus the traditional solo programming approach. The authors pointed out that pair composition and the project complexity influence PP effectiveness concerning efforts, defect rate, and overall cost of the project. Moreover, previous PP experience leads to a more positive view of this practice than those who never experienced it.

Bella et al. (2013) conducted an experiment to evaluate the effect of PP on the quality and efficiency of defect corrections and its impact on the overall development process of a developer team. The study was based on a 14-month dataset collected from a team of professional developers working for an IT department of a large Italian manufacturing company in an agile software development project. The analysis showed that new defects tend to decrease when PP is practiced, even though the nonparametric statistical tests did not confirm the significance of this behavior. While these results

[2] Duration typically refers to the total time required for individuals, pairs, and teams of developers to complete tasks.

[3] Effort is generally calculated by summing the time spent by each individual in a pair or team of developers, akin to our method of measuring effort, i.e., considering twice the time spent for pairs of programmers.

may not mirror the statistical significance found in our study, they resonate with our findings by demonstrating that PP potentially contributes to improve reliability.

Sillitti et al. (2012) presented an investigation on how PP practices affect how developers write code, analyzing the effects of the agile practice on developers' attention and productivity. By studying a team of 17 developers over ten months, they observed that developers working in pairs: (a) spend more time in directly productive activities; (b) switch less often between tools; (c) have longer permanence in a tool before switching to another one; and (d) tend to focus more on productive activities. Such results align with ours since, in our experiments, the agile practice also improved the developers' performance.

Expanding on the concept of PP, the Global Software Engineering (GSE) concept has emerged in recent years due to the globalization of IT, which has led companies to distribute their software development globally. In this context, PP began to be adopted by distributed teams, resulting in the emergence of Distributed Pair Programming (DPP). In DPP, two programmers collaborate to create software using tools that enable screen sharing and communication via audio, text, and video. An SLR conducted by da Silva Estácio and Prikladnicki (2015) examined DPP, highlighting the increasing industrial adoption and the corresponding lack of empirical research. The authors highlighted the need for more professional-oriented DPP research that bridges the theoretical and practical domains. Key research opportunities identified include investigating DPP effects on coordination, communication, and cultural diversity and analyzing the function of particular DPP-supporting tools. da Silva Estácio and Prikladnicki provided a comprehensive overview of the current state of research on DPP, including its benefits, challenges, and tools. It also identified areas where further research is needed. However, one potential weakness is that the review focused primarily on studies exploring DPP from a teaching perspective, with less emphasis on its use in industry.

A very recent SLR Xu and Correia (2023) provides a comprehensive analysis of DPP, underscoring its growing importance in education and industry, particularly in the context of the post-COVID-19 digital learning trend. The review reveals that individual characteristics such as prior programming experience, perceived skill, gender, personality, and pair compatibility significantly impact DPP effectiveness. The study suggests further exploration of how task structures influence DPP effectiveness and how this relates to computational thinking education.

While both SLRs provide comprehensive analyses of DPP, their scope, context, and focal point are distinct. The former, by da Silva Estácio and Prikladnicki (2015), highlights the industrial application of DPP and the lack of empirical studies in this context. In contrast, the latter, by Xu and Correia (2023), focuses more on the educational implications of DPP, particularly in the digital learning environment post-COVID-19; it considers the individual characteristics that influence the efficacy of DPP in a learning context. In this study, we focused solely on local PP. However, the same methodological approach can be applied to Distributed Pair Programming, allowing for comparative analysis and a thorough comprehension of the dynamics between local and distributed settings.

3.2 Studies related to test-first programming and TDD

Desai et al. (2008) examined TDD experiments within an academic context. They generally observed that, in a controlled experiment, when the control group employed

iterative test-last programming (*i.e.*, continuous testing), no substantial differences were detected in the quality of the resulting software. Conversely, when all code was composed prior to the implementation of tests, signifying a strict test-last programming approach, test-first programming surpassed the test-last method in terms of fault counts (reduction ranging from 35% to 45%). Their experiment also reported modest gains (5% to 10%) in productivity, favoring the test-first approach.

Erdogmus et al. (2005) performed an experiment focused on the test-first aspect of TDD using 24 third-year undergraduate Computer Science students. They observed that TDD increased productivity, even though they found no difference in code quality. Compared to the control group (test-last programming), the subjects using test-first programming produced a considerably larger set of tests (52% larger, on average). In particular, such results are consistent with our outcomes since test-first programming also caused the production of more tests in our experiments. Regarding code quality, both groups performed very similarly (the mean of the control group was only 2% higher). Finally, the productivity of the test-first programming group was higher (28% higher mean). With respect to these variables – quality and productivity – our results with professionals were somehow different: subjects using test-first programming produced significantly more correct implementations without additional cost in terms of development time. This might be due to the expertise of programmers, as professionals tend to benefit more from the agile practice.

In an SLR, Munir et al. (2014) classified the main studies into categories based on two factors: relevance and rigor. Relevant studies, that is, those involving realistic settings with industrial applicability, demonstrate that TDD benefits students and professional developers in terms of external quality at the expense of productivity. Precisely, these outcomes correspond to our own. However, the authors suggest that industry experiments involving real-world systems and long-term studies are necessary.

Bissi et al. (2016) presented a systematic review to identify publications that compare the effects of TDD on internal and external software quality and productivity, comparing TDD with Test Last Development. The review found that most studies have identified an increase in internal and external software quality when using TDD. However, there was an increase in productivity in the academic environment but a decrease in an industrial scenario when using TDD.

Latorre (2014) conducted a quasi-experiment to investigate the impact of developers' experience levels on their ability to learn and apply unit test-driven development (UTDD). The primary objective was to assess the difficulty experienced by professionals in learning UTDD and to evaluate the feasibility of employing this agile practice in real-world projects. The results suggest that experienced developers can correctly apply UTDD after a brief practice period, retaining the knowledge for use in their companies within an industrial setting. Conversely, junior developers required additional self-learning throughout the process. This finding aligns with our test-first study, as our experiments also demonstrated that professional developers might benefit more from the agile practice when compared to novices.

Fucci et al. (2018) conducted a TDD quantitative cohort study with 30 undergraduate students (third-year) in Computer Science at the University of Bari, Italy, aiming to measure the TDD effects on the external quality of software products and developers' productivity. Even though the authors stated that non-significant statistical results were observed, they recognized that the use of the TDD has produced significantly more tests than the non-TDD process, confirming one of our findings.

Baldassarre et al. (2021) also confirm two of our findings: it was observed that participants applying TDD produced significantly more tests, with a higher fault-detection capability than those using a non-TDD approach.

On the other hand, results obtained in study by Fucci et al. (2017) revealed little difference between the test-first behavior of TDD and the test-last behavior. More specifically, they tried to verify whether TDD affected the external quality of code, the number of tests written, or the productivity of software developers. They have observed no statistical difference between test-first and test-last. The authors argue that productivity and quality improvements are more associated with granularity and uniformity than with the order in which testing and code production is applied. Similar results are described in a study performed by Karac et al. (2019), in which authors observed that the ability to break down tasks into smaller parts and the practitioners' familiarity with the tasks are highly coupled to the impact of the TDD approach.

Tosun et al. (2021) performed an experiment with industry professionals using the TDD and Incremental Test-Last Development (ITLD) approaches. They found that the type of task significantly impacts on quality in TDD, with TDD being more suitable for smaller tasks. The choice between TDD and ITLD depends on task size, developer experience, and project goals. Further research is needed to understand these approaches' pros and cons in various contexts.

Santos et al. (2021) conducted experiments to compare TDD with control approaches, primarily the waterfall model, regarding software quality. The findings suggest that TDD generally yields higher quality. However, the degree of this advantage varies based on factors such as research methods, programming environments, evaluation lengths, units of analysis, types of tasks, and participant types. The authors recommend further experiments to investigate these variables. They also suggest future studies to consider the impact of the development task and the order of TDD application.

Ghafari et al. (2020) argued the existence of some factors that cause disparities in TDD research results and limit the application of this approach to practitioners. They pointed out that the TDD definition, participants, tasks, type of projects, and comparisons are among the factors identified in the literature. In this study, we tried to overcome some of these factors, for example, by including more experienced professionals in the experiments.

Romano et al. (2019) conducted a study on the affective reactions of beginner developers towards various development approaches. They discovered that novices tended to prefer non-TDD development approaches over TDD and that the testing phase tended to make developers using TDD less content.

3.3 Other related work

Lemos et al. (2018) evaluated the impact of software testing education on code reliability. The authors used a very similar experimental design as the one applied in this paper: students implemented an analogous set of auxiliary functions before and after learning software testing concepts and techniques. The correctness of the produced code was then compared. Results showed that exposure to testing knowledge can, in fact, impact on code quality without a significant additional cost in terms of lines of code. An interesting evaluation that could be made in the future is to measure the combined effect of software testing knowledge and the use of the agile practices addressed in this paper.

To the best of our knowledge, the study presented in this paper and a previous study conducted by the same authors (Lemos et al., 2012) represent the first effort to specifically

evaluate agile practices in the context of developing auxiliary functions. In addition, compared to other studies of a similar nature, ours is one of the first to use repeated measurements with a cross-over experimental design. Most studies tend to employ two-group experimental design in which a control group employs a traditional approach, for example, and a treatment group employs an agile approach. This kind of design is less powerful than repeated measures because comparisons are made across – and not within – subjects (differences observed in two-group experiments might be due to the variability of subjects and not an effect of the evaluated approach). With repeated measures, extraneous error variance is reduced because each subject serves as his/her own control. 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. Moreover, many experiments involved only students, while ours also included a sample of 37 professional developers in a single extensive analysis. In this sense, we believe our results are thus more generalizable. Another particularity of our study is that it tackles two agile practices (pair and test-first programming), while the majority of related work target only a single practice.

Concerning the achieved results, previous evidence regarding the effectiveness of pair and test-first programming is somehow contradictory. Some of the aforementioned surveys (Hannay et al., 2009; Salleh et al., 2010; Desai et al., 2008; Sun et al., 2016; Fucci et al., 2018; Ghafari et al., 2019; Karac et al., 2019; Kazerouni et al., 2019) 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 a substantial increase in the development effort (except for professionals when using test-first programming), but offered a counterpart in terms of significant correctness improvement and larger and higher coverage test sets.

4 Study setup

This study aims to examine the impact of pair versus solo programming and test-first versus test-last programming on the development of auxiliary functions. We evaluate these practices concerning their reliability and effort factors. As discussed in Sect. 1, both practices are controversial within the software development community, so their benefits are unclear (all the more when considering the development of auxiliary functions). As also discussed in Sect. 1, auxiliary functions are an important target for investigation as they might bring severe problems to software systems (applying good developmental practices in their implementation might help prevent such problems).

We are thus interested in assessing whether agile practices can benefit developers in terms of improved reliability of auxiliary functions when compared to traditional approaches. We also want to look into the possible additional cost involved in terms of effort. Based on such goals, we focus our study in the following research questions:

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

R1 Does pair programming help obtain more correct implementations than solo programming?

Table 1 Hypotheses formulated for our experiments

	Null hypothesis (0)	Alternative Hypothesis (A)
H_1	$Correctness_{PP} = Correctness_{SP}$	$Correctness_{PP} > Correctness_{SP}$
H_2	$Correctness_{TF} = Correctness_{TL}$	$Correctness_{TF} > Correctness_{TL}$
H_3	$TestSize_{TF} = TestSize_{TL}$	$TestSize_{TF} > TestSize_{TL}$
H_4	$TestCov_{TF} = TestCov_{TL}$	$TestCov_{TF} > TestCov_{TL}$
H_5	$Effort_{PP} = Effort_{SP}$	$Effort_{PP} > Effort_{SP}$
H_6	$Effort_{TF} = Effort_{TL}$	$Effort_{TF} > Effort_{TL}$

H Hypothesis, *SP* Solo Programming, *PP* Pair Programming, *TF* Test-First, *TL* Test-Last

R2 Does 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 questions. **In the development of auxiliary functions**:

R5 Do pair programmers spend more time in the development of functions than solo programmers?
R6 Do test-first programmers spend more time in the development of functions than test-last programmers?

It should be observed that R4 complements R3, as larger test sets do not inherently guarantee increased coverage test sets (i.e., redundancy in tests is possible). Our exploration progresses through six hypotheses, which derive from these research questions. The null (0) and alternative (A) delineations for each hypothesis can be found in Table 1. It is important to note that we have a hypothesis corresponding to each research question: hypothesis H_i is associated with research question R_i

4.1 Ethical implications

Ethical considerations are paramount in academic research, especially when involving human subjects or collecting personal data. It is standard protocol for researchers to assess whether their study might introduce any significant ethical risks. When such risks are identified, the study typically requires an independent ethical review before its commencement.

After thoroughly examining the ethical guidelines from the participating universities for this study, our team ascertained that our research introduced either no risks or only minimal risks to the participants. Given this determination, we did not deem it necessary to seek approval from a central Institutional Review Board (IRB). Nonetheless, we ensured that all requisite steps were taken to maintain participant data's anonymity, protection, and confidentiality. To further clarify the context and our rationale: i) The core participants of this study were students from the university's software engineering and software testing courses. ii) The tasks they engaged in for the purposes of this research were integral components of the course curriculum, ensuring that no extraneous demands were placed upon

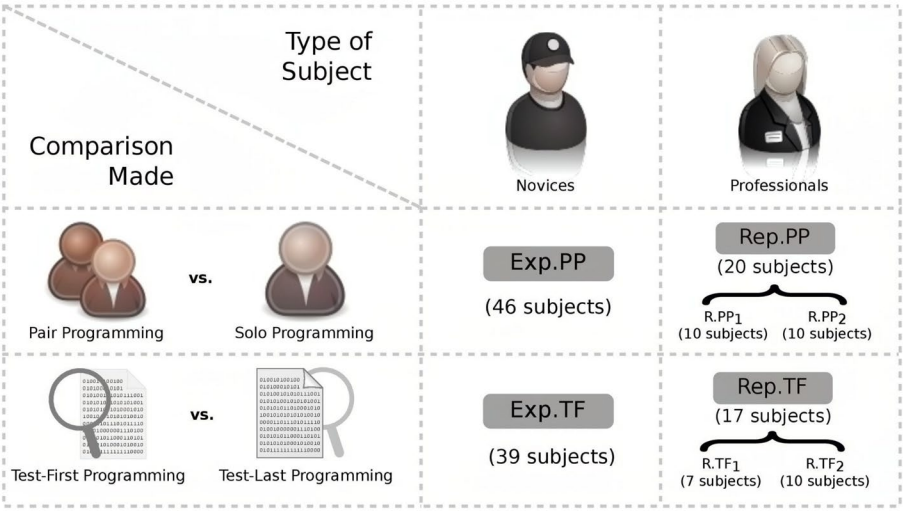

Fig. 1 Diagram showing the six experiment instances conducted for our study (Exp = Experiment; R, Rep = Replication; PP = Pair programming; TF = Test-first programming)

them. iii) Importantly, we neither offered nor provided any form of monetary compensation or other incentives for their involvement. iv) The recruitment process for the study was invitation-based, with all participants giving their consent. They were well-informed of the voluntary nature of their involvement and the academic intent behind collecting their input. v) This process of obtaining consent was transparently witnessed by the entire cohort of participants and faculty members, irrespective of whether the setting was a classroom or an industry environment. vi) Our data collection was strictly limited. We gathered only the function's source code and details regarding the participants' years of experience in specific domains like Java, agile development, pair programming, and test-first programming. We refrained from collecting any other personal data, ensuring the impossibility of participant identification. vii) An interesting note is that some participants proactively indicated their eagerness to view the study's results. Considering all these factors, we are confident that the study was conducted with utmost ethical integrity, thus obviating the need for formal ethical approval.

4.2 Subjects, target functions, test sets, and tools

Figure 1 depicts the six experiment instances we conducted with their respective number of subjects. Throughout the paper, we use the names used in the figure to refer to each experiment. In a nutshell, *Exp.PP* refers to the pair programming experiment with students, while *Exp.TF* refers to the test-first programming experiment with students. *Rep.PP* refers to the pair-programming replication with professionals, while *Rep.TF* refers to the test-first programming replication with professionals.

Subjects Since auxiliary functions tend to be natural candidates to be assigned to less experienced developers (Dagenais et al., 2010; Williams & Kessler, 2002), we believe

students can form a good conservative sample for evaluating the agile practices in our context. Therefore, in our first investigations we decided to invite Computer Science undergraduates to participate in the experiments.

For the pair programming experiment (*Exp.PP*), only basic Java and pair programming skills were required for the evaluation. Therefore, we selected 46 students that were in the second semester of the program to participate in the pair programming experiment. On the other hand, the test-first experiment (*Exp.TF*) required more knowledge about software engineering and software testing (*e.g.*, developers have to know what test cases are and how to implement them). Since these skills are typically acquired at later stages of a Computer Science degree program, and that was the case for our target program, we selected 39 students that were in the sixth semester of the program to participate in the test-first programming experiment. The experiments with novices thus involved 85 undergraduate students.

In an object-oriented Java programming course, the second-semester students received a 50-minute module on pair programming instruction. The advanced training in test-first programming for sixth-semester students consisted of two 100-minute JUnit/TDD modules and multiple exercises. Students were required to demonstrate implementations with test cases as part of the programming tasks. It is also essential to observe that the sixth-semester students had prior experience in software testing techniques (as a result of a 72-hour mandatory undergraduate course).

Although PP and TDD are theoretically simple and widely taught, our empirical observations during the experiments revealed that many second-semester students lacked practical familiarity with these concepts. Despite being aware of the theory behind these practices, they often faced challenges when implementing them in real-world scenarios. As a result, the pair programming experiment did not specifically focus on the testing approach (i.e., test-first or test-last). In addition, only solo programming was used during the test-first experiment. Consequently, it should be evident that the experiments and analyses reported in our study are independent, *i.e.*, they do not account for the compound effect of agile practices.

As commented in Sect. 1, we wanted to understand how the programmers' experience would impact results in a more extensive analysis. Therefore, we decided to replicate the initial experiments with professional developers. The replication with professionals (*Rep.PP* and *Rep.TF*) involved 37 subjects. A tally of 20 participated in the pair programming replication, while 17 participated in the test-first programming replication. In fact, to ensure we had a more significant sample, both replications combined results from two experiment instances run with a smaller number of subjects. For the pair programming replication, the first run involved 10 professionals from different companies ($R.PP_1$); while the second run involved other 10 professionals working for a single company with experience with agile practices ($R.PP_2$). For the test-first programming replication, a similar combination took place: the first run involved 7 professionals from different companies ($R.TF_1$), and the second run involved 10 professionals from a single company with experience with agile practices ($R.TF_2$).

Since the experiments were conducted with the exact same design, it was possible to consider the combinations as two single replications (*Rep.PP* and *Rep.TF*). Before we could combine results, we had to make sure they were consistent across the two sets of subjects for each replication. We did this by simulating an additional variable to the experiments – *run* – and by looking at whether there was a significant difference in results when taking into account such a variable. We then applied the non-parametrical Kruskall-Wallis test,

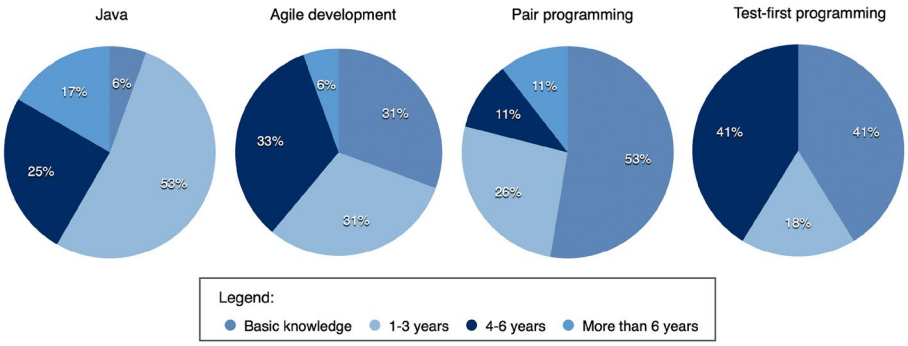

Fig. 2 Proportions of professional developers that participated in our study at each experience level

which verifies the probability that samples originate from the same distribution. With 95% confidence level, the test indicated a non-significant difference for both replications when we consider the variation of our main metric – correctness – across runs (p-values = 1 and 1 – PP/SP, and p-values = 0.3243 and 0.9508 – TF/TL programming). Therefore, we decided to aggregate results from the smaller experiments in order to have more significant sets of subjects.

Figure 2 shows the proportion of subjects at each level of experience in (1) Java, (2) agile development, (3) pair programming, and (4) test-first programming ("Basic knowledge" means that the subject received only training in the related practice/technology but has not practiced it professionally).[4] Note that our group of subjects has varied experience in the related practices/technology. This is important to improve the generalization of our conclusions to a broader population. Moreover, although the sample contains less experienced developers – some have only basic knowledge in the practices/technology – almost half of them have one to four or more years of professional experience using such approaches. Also, part of the subjects has more than six years of experience in some of the practices/technology (17% of the subjects for Java, 6% of the subjects for agile development, and 11% of the subjects for pair programming).

Target functions To choose representative functions for our study – functions that come under the category of auxiliary functions as defined in Sect. 1 – we examined the Apache Commons project, which provides libraries of reusable Java components.[5] Moreover, we chose functionality that could be easily retrieved via code search engines (*e.g.*, Open Hub (Black Duck Software Inc., 2014)), *i.e.*, we attempted to identify commonly used auxiliary functions that were not readily available in the Java API. These functions have been divided into three categories: array manipulation (Array), fundamental mathematics (Math), and string manipulation (String). We took two functionalities from each domain to attain a more robust set. The auxiliary functions utilized in our research are detailed in Table 2.

[4] Only a single subject that participated in our study did not respond to our survey and was thus not considered in the graphs.

[5] http://commons.apache.org/ – accessed in July, 2023.

Table 2 Auxiliary functions used in the experiment

Domain	Function	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

Table 3 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)		|a1| - |a2| = 1 (B3)
	|ar2| > |ar1| (C8)		|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)

Our functions are also deliberately narrowly scoped, which is an additional characteristic. The objective here is to perform a conservative evaluation: if a practice affects the development of simpler auxiliary functions, we can expect that it will also have an effect on more complex auxiliary functions. The designated functions are comparable in size and scope to the day conversion feature that was defective in the already mentioned Microsoft's Zune media player[6] (Weimer et al., 2010). A further benefit is that this type of function enables the implementation of more systematic test case selection techniques, such as functional testing.

Test sets To evaluate the subject's programs, we created a comprehensive set of boundary-value functional tests for each of the selected functions.[7] The last column of Table 2 indicates the number of test cases created for each function. To build the test sets, we applied the equivalence partitioning and boundary-value analysis criteria.

These criteria were applied to select representative test cases for each test set, attempting to cover as many of the functional specificities of the functions as possible. To provide an illustrative example of the test case development process, Table 3 shows the equivalence classes and, where applicable, boundary values for the functionality. *ar1* and *ar2* are the input arrays; |*ar*| represents the array *size*; and ar_x[i] represents an element of the array. *Int.MIN* and *Int.MAX* correspond to the minimum and maximum integer values. Given the utilization of the Java programming language in the study, the highest and lowest possible integer values were employed as boundary values for this particular data type. A similar principle was also applied to other data types concerning other functions. Here, specific values are not used to represent test cases independently of the programming language.

It is important to note that, for the professional sample, two types of test cases developed for the String manipulation functions were not taken into account for the presented

[6] This function is cited because its source code is available online. Nonetheless, it is reasonable to presume that the other functions listed in Sect. 1 are also of equivalent size. For example, a PlayStation-like leap year detection function requires only a small number of code lines to be implemented.

[7] All experimental artifacts, encompassing subjects' programs, test sets, function descriptions, and more, are accessible via the following link: http://www.ict.unifesp.br/fsilveira/data/SoftwareTestingExperiment_data.zip.

(a) Pair programming. (b) Test-first programming.

Fig. 3 Basic subject/treatment configurations used in our experiments

results (namely, the ones that tested for *null* strings and the ones that tested for *empty* strings). We did this to normalize outcomes because these tests were failing for both agile and traditional practices, for all subjects. This happened because these subjects were developing functions against contracts different from the ones we anticipated. It appears that the professional sample expected that the client functions would have handled such exceptional inputs and not by the String manipulation functions, which is a pretty reasonable contract. In any case, it should be noted that, in this sense, some implementations from the students were more resilient since they also handled those boundary values.

Tools The functions were implemented using the Eclipse[8] IDE, while JUnit served as the framework for developing test cases. The students were instructed to concentrate their efforts solely on implementing the requested auxiliary functionality using the designated techniques. Students were instructed, for instance, to implement functions as static methods within a class with a predefined name. This was done because auxiliary functions typically rely solely on parameter values to carry out their responsibilities. We used the Cobertura[9] tool to evaluate the coverage attained by the developed test sets.

Test implementation For the subjects that needed to implement the function using the test-first approach, it was required to create automated tests before the implementation. However, creating automated tests was not a requirement for those that used the test-last approach. Nothing about creating automated tests was mentioned in the pair programming and solo programming experiment specifications.

4.3 Experimental design and procedure

We used repeated measurements with a cross-over experimental design for the conducted experiments, with each subject implementing functions using both traditional and agile methods. Such a design permits more control over the variation between subjects (Montgomery, 2006). We randomized the students' assignments in order to reduce the variance in the differences between functions and approaches. To eliminate function

[8] http://eclipse.org/ – accessed in July, 2023.
[9] http://cobertura.github.io/cobertura/ – accessed in July, 2023.

Table 4 Partial task assignments to subjects

S	Pair Programming Exp.		Test-First Exp.	
	1st Session	**2nd Session**	**1st Session**	**2nd 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
...	...	...	...	...

asymmetry, it was determined that each function would be implemented using both traditional and agile methods.

All investigations were conducted throughout two sessions. In the first session, a portion of the students applied traditional approaches – solo or test-last programming – while the remaining students applied agile practices – pair or test-first programming. In the second session, the students switch approaches. This was done to eliminate order and learning effects; *e.g.*, applying first solo programming and later pair programming may have benefited one of the approaches; similarly, the test-first experiment utilized the same methodology. In addition, subjects were required to implement functions from various domains during the first and second sessions. In the second session, a subject who implemented an Array Function in the first session would implement a Math or String function. When implementing functionalities in the initial and subsequent sessions, we took this approach to minimize the influence of one function domain over the others.

To facilitate understanding the adopted experimental design, Fig 3 depicts the basic subject/treatment configurations adopted and Table 4 presents part of the assignments used for the experiments. Note that the subjects are not the same for the two experiments; we only maintained the first column for simplification. The same procedures were applied for the replications with professionals.

4.4 Metrics

In this study, metrics were utilized to assess the reliability of the functions developed and the effort exerted by the subjects during the application of each approach.

Reliability measurement Functional Test Set Success Level (FTSSL) correctness was one of the metrics we used to measure reliability in our experiments. FTSSL assigns grades to functions based on a scale with three values: 0 (Incorrect – all test cases fail), 0.5 (Neither correct nor incorrect – some test cases fail, but not all), and 1 (Correct – all test cases pass). We use this scale because, in our context, a failure is highly significant: since we are utilizing functional test sets, each test case encompasses a significant portion of the functionality. In this manner, we are more concerned with functions that never fail, but we also want to identify entirely incorrect functions. Functions that fail one or more test cases, but not all, receive a score in the middle because they cannot be considered correct but at least implement a portion of the functionality accurately. Then, we only grant the highest score to functions that pass every test case. The FTSSL is an ordinal variable because there is a

natural ordering between the values (having all test cases pass is preferable to having fewer test cases pass, which is preferable to having all test cases fail).

As an additional measure of reliability, we evaluate the test sets generated by test-last and test-first programming using the metrics of size and coverage. The size is calculated by the number of test cases in the test set. We adopted the ordinary statement coverage criterion for code coverage, which demands that every statement in the code be executed by at least one test case.

Effort measurement Total Development Time (TDT) is the total number of minutes programmers require to develop a given function. This metric was utilized in our experiment. Thus, the individual effort was equivalent to the duration in the pair programming experiment, whereas the effort for the pair programmers was double the duration. This metric is a standard and straightforward method for evaluating effort, being also utilized in other investigations.

4.5 Statistical analysis

A mere observation of the means or medians of sample observations is statistically insufficient to infer about the actual populations. This occurs because the observed differences may be a result of random sampling. Statistical hypothesis tests can be used to determine whether the observed differences are actually significant.

In our investigation, each subject independently utilized each method to develop the functions. In this instance, the paired statistical hypothesis tests compare measurements within subjects instead of between subjects. Compared to unpaired tests, paired tests are thought to improve precision Montgomery (2006) significantly. Prior to selecting the most appropriate test to apply, we must confirm the normality of our observations. Most continuous data observations in our experiments did not follow a normal distribution, as determined by the Shapi-ro-Wilk test. We chose the Wilcoxon/Mann–Whitney non-parametric signed-rank paired test, which does not presume normal distributions because we also deal with ordinal data in certain instances Shull et al. (2008). For statistical significance, we adopted the conventional confidence level of 95%; thus, p-values below 0.05 are considered significant in our analyses. We utilized the R language[10] for all statistical analyses.

5 Results and analysis

5.1 Experiment with novice programmers

Tables 5 and 6 present the results of our experiments with novice programmers (*Exp. PP* and *Exp.TF*), and Tables 7 and 8 present the statistics for the pair programming and test-first experiments with respect to correctness. The first two tables present results for each subject, for each metric, and also mean values for the non-ordinal metrics (for *Exp. PP*: Total Development Time – TDT, and for *Exp.TF*: TDT, Test Coverage, and Number of Test Cases). Tables 7 and 8, on the other hand, are contingency tables that show the compared outcomes of subjects while using the traditional and agile approaches,

[10] http://www.r-project.org/ - 07/06/2021.

Table 5 Results of the pair programming experiment with novices – *Exp.PP*

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*

FTSSL Functional Test Set Success Level, *TDT* Total Development Time, *SP* Solo Programming, *PP* Pair Programming, *I* Incorrect, *N* Neither correct nor incorrect, *C* Correct

for *correctness*. For instance, the cell in row *SP-I*, column *PP-I* of Table 7 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 7 shows that a total of 0 pair programmers produced *incorrect* functions in the experiment.

5.1.1 Reliability evaluation

For pair programming experiments with novices, the soloists implemented three completely correct functions and three incorrect functions (6.5%). Conversely, pair programmers implemented 8 correct functions (18% – more than twice as many as solo programmers) and 0 incorrect functions. The remaining implementations failed at least one test case, but not all. We believe this was primarily because few students implemented functions that handled exceptional inputs, such as null objects. Since our test cases also covered such inputs, the majority of implementations failed on at least one of them.

Moving on, the Wilcoxon test revealed a statistically significant difference at the 95% confidence level (df = 45, p-value = 0.04982) regarding the observed difference in terms of correctness. This result supports the alternative hypothesis (H_1-A) that pair programming

Table 6 Results of the test-first experiment with novices – *Exp. TF*

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

Table 7 Correctness statistics for the pair programming experiment with novices – *Exp.PP*

		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%)	

SP Solo Programming, *PP* Pair Programming, *I* Incorrect, *N* Neither correct nor incorrect, *C* Correct

outperforms solo programming in terms of auxiliary function correctness. This is a significant finding, as it appears that developers can benefit from pair programming even for supportive functionality. In addition, the higher number of completely correct implementations for pair programmers demonstrates that they tend to be more cautious when implementing auxiliary functions, considering exceptional inputs included in the test sets.

Regarding the test-first experiment with novices, it should be noted that both approaches performed nearly identically in terms of correctness. The only difference between test-last and test-first programmers was that test-last programmers delivered one additional correct implementation. The Wilcoxon test did not indicate a statistically significant difference at the 95% confidence level (df = 38, p-value = 0.6554). This result supports the null hypothesis (H_2-*0*) that test-last programming and test-first programming have equivalent effects on the correctness of auxiliary functionality. Such evidence suggests that developers should proceed with caution in this area. As discussed in Hypotheses H5 and H6, although test-first encouraged participants to generate more comprehensive test sets, this agile practice did not substantially affect functional correctness. Other studies have shown that although test-first may lengthen the development process, it also generally improves correctness (Baldassarre et al., 2021; Edwards, 2004).

With respect to the size of the produced test sets, however, note that the means are around 112% higher for test-first programming (see Table 6). The statistical test reveals a significant difference at the 95% confidence level (df = 38, p-value = 0.00001708). This result supports the alternative hypothesis (H_3-*A*) that test-first programming is superior to test-last programming concerning the size of the test set. Notably, the difference remains significant after excluding results from test-last programmers who did not implement test cases (test-last mean test set size = 3.77; test-first mean test set size = 4.91; Wilcoxon test: p-value = 0.03216).

Table 8 Correctness statistics for the test-first experiment with novices – *Exp.TF*

		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%)	

TL Test Last, *TF* Test First, *I* Incorrect, *N* Neither correct nor incorrect, *C* Correct

Regarding test statement coverage, observe that the agile approach outperformed the traditional approach by 40%. At a confidence level of 95%, the statistical test reveals a significant difference (df = 38, p-value = 0.00003341). This finding confirms the alternative hypothesis (H_4-*A*) that test-first programming is superior to test-last programming in terms of test set coverage.

However, at this time, when we remove results from test-last programmers that produced no test cases, the difference becomes non-significant (test-last mean test set coverage = 97%; test-first mean test set coverage = 98%; Wilcoxon test: p-value = 0.1459). We believe that this happens mostly because the functions targeted in our experiments are small, in which case few test cases can easily reach a high coverage.

On the other hand, also note that similarly to the interpretations for H1, it might also be the case that pair programmers produce higher quality software later in the evolution of a project, even considering the more fine-grained metric adopted in this paper. Since, in our experiment, the subjects only developed a single functionality, it may be that if they had produced more code, pair programming would increase system correctness in the end. In any case, our study indicates that developers should be cautious while resorting to pair programming in the development of auxiliary functionality.

5.1.2 Effort evaluation

Regarding effort, observe that the mean total development time for pair programming was approximately 94% longer than for solo programming (Table 5, "Mean time"). The Wilcoxon test confirms a statistically significant difference between the means at the 95% confidence level (df = 45, p-value = 0.000009278). This result supports the alternative hypothesis (H_5 -*A*) that, in terms of effort, solitary programming outperforms pair programming.

The mean total development time for the test-first experiment was approximately 13% longer than for the test-last approach. The Wilcoxon test again revealed a statistically significant difference at the 95% confidence level (df = 38, p-value = 0.0463). Such a result supports the alternative hypothesis (H_6-*0*) that test-last programming outperforms test-first programming in terms of effort. Similar to the findings of other studies, we find that test-first programming appears to have a negative effect on effort.

5.2 Replications with professionals

Tables 9 and 10 present the results of our experiments with professional programmers, and Tables 11 and 12 present the statistics for the pair programming and test-first experiments with respect to correctness, for the same subjects.

5.2.1 Reliability evaluation

For the ***pair programming replication with professionals***, with respect to correctness, note that soloists implemented 6 functions that were completely correct and 2 that were incorrect (10%). On the other hand, pair programmers implemented 12 functions that were correct (exactly twice more than solo programmers), and 0 incorrect ones. All other implementations failed on at least one test case, but not all. So again, we see that pair programming yields better results, also for professionals. It is noteworthy that in both experiments – with novices and professionals –, none of the pairs implemented completely incorrect functions.

Table 9 Results of the pair programming experiment with professionals – *Rep.PP*

Subj.	FTSSL		TDT		Subj.	FTSSL		TDT	
	SP	PP	SP	PP		SP	PP	SP	PP
1	C	C	19	32	11	C	C	4	10
2	N	C	20	32	12	N	C	6	10
3	N	N	26	14	13	N	N	8	48
4	N	N	15	14	14	N	N	18	48
5	I	N	8	40	15	I	N	10	4
6	N	N	30	40	16	N	N	3	4
7	N	C	5	58	17	N	C	6	10
8	N	C	9	58	18	N	C	6	10
9	C	C	16	20	19	C	C	4	14
10	C	C	10	20	20	C	C	2	14
Mean time (all subjects)								*11.00*	*25.00*

FTSSL Functional Test Set Success Level, *TDT* Total Development Time, *SP* Solo Programming, *PP* Pair Programming, *I* Incorrect, *N* Neither correct nor incorrect, *C* Correct

We ran the statistical test to check whether the observed differences were significant. At 95% confidence level, the Wilcoxon test does indicate a significant difference in terms of correctness (df = 19, p-value = 0.00298100). Such a result also favors the *alternative* hypothesis (H_1-A) that pair programming outperforms solo programming with respect to the correctness of auxiliary functions. This confirms our key finding that even for

Table 10 Results of the test-first experiment with professionals – *Rep.TF*

Subj.	FTSSL		TDT		Cov		# TC	
	TL	TF	TL	TF	TL	TF	TL	TF
1	N	N	31	48	100.00%	100.00%	1	1
2	N	N	13	35	86.00%	100.00%	1	1
3	I	N	50	50	0.00%	0.00%	0	1
4	N	N	31	35	0.00%	0.00%	0	0
5	N	N	23	28	100.00%	100.00%	2	3
6	N	C	37	23	100.00%	100.00%	3	5
7	N	C	50	35	100.00%	100.00%	7	6
8	I	N	10	22	0.00%	85.71%	0	2
9	N	N	12	24	0.00%	0.00%	0	0
10	C	C	30	15	0.00%	85.71%	0	3
11	N	N	16	12	0.00%	83.33%	0	2
12	N	C	14	24	0.00%	100.00%	0	8
13	N	C	2	4	0.00%	83.33%	0	3
14	N	N	10	6	0.00%	85.71%	0	2
15	N	N	12	11	0.00%	83.33%	0	1
16	C	N	19	19	0.00%	100.00%	0	2
17	N	N	8	13	0.00%	85.71%	0	5
Mean			*21.65*	*23.76*	*29%*	*76%*	*0.82*	*2.65*

Table 11 Correctness statistics for the pair programming experiment with professionals – *Rep.PP*

		PP			
		I	**N**	**C**	**Freq.**
SP	**I**	0	2	0	2 (10%)
	N	0	6	6	12 (60%)
	C	0	0	6	6 (30%)
	Freq.	0 (0%)	8 (40%)	12 (60%)	

SP Solo Programming, *PP* Pair Programming, *I* Incorrect, *N* Neither correct nor incorrect, *C* Correct

supportive functionality, it appears that pair programming can bring benefits to developers, alike for professionals.

With regard to the ***test-first replication with professionals***, note that at this time, we had a deviation from the novices experiment: professional programmers performed much better when using test-first programming than when using test-last programming. Note that incorrect implementations were only produced while test-last programming was being used (2 subjects, 11.7%). Also, test-first programmers produced more than twice as many correct implementations as test-last programmers: five test-first programmers (29.4%) against two test-last programmers (11.8%). This is an important finding, as it indicates that professionals can benefit more from the agile technique than novices.

At 95% confidence level, the Wilcoxon test indicates a statistically significant difference (df = 16, p-value = 0.03630000). This result favors the *alternative* hypothesis (H_2 -*A*) that test-first programming outperforms test-last programming with respect to the correctness of auxiliary functions.

When we look at test set size and coverage, we also find interesting results. Test-first programmers produced, on average, 2.65 test cases per function, while test-last programmers produced only 0.82. 12 subjects produced no test cases while using test-last programming. This once again provides evidence that when tests are left for later, they might not be written at all. The Wilcoxon test confirms a significant difference at 95% confidence level (df = 16, p-value = 0.00135600).

Tests produced when test-first programming was being used got higher coverage than when test-last programming was being used. On average, tests covered 47% more statements with the agile technique. At 95% confidence level, the Wilcoxon test confirms a statistically significant difference (df = 16, p-value = 0.00272400).

Table 12 Correctness statistics for the test-first experiment with professionals – *Rep.TF*

		TF			
		I	**N**	**C**	**Freq.**
TL	**I**	0	2	0	2 (11.8%)
	N	0	9	4	13 (76.4%)
	C	0	1	1	2 (11.8%)
	Freq.	0 (0%)	12 (70.6%)	7 (29.4%)	

TL Test Last, *TF* Test First, *I* Incorrect, *N* Neither correct nor incorrect, *C* Correct

5.2.2 Effort evaluation

Regarding the effort evaluation for the ***pair programming replication with professionals***, results were similar to the experiment with novices. On average, pair programmers took approximately 2.3 times more development time to develop functions than solo programmers. With respect to the statistical analysis, at 95% confidence level, the Wilcoxon test confirms a statistically significant difference between the means (df = 19, p-value = 0.00101700). Such a result favors the *alternative* hypothesis (H_5-A) that solo programming outperforms pair programming with respect to effort, also for professionals.

For the ***test-first replication with professionals***, results were quite surprising and diverse from the experiment with novices. At this time, test-first programmers performed similarly to test-last programmers in terms of development time, with only a 10% increase in total development time, on average. The Wilcoxon statistical test confirms a non-significant difference at 95% confidence level (df = 16, p-value = 0.06746000). This outcome indicates that more experienced professionals may take more advantage from test-first programming than novices. Note that professionals produced more reliable implementations, both with respect to correctness and test case completeness, but taking approximately the same time required when test-last programming was used.

5.3 Summary of results

Table 13 summarizes our results in terms of the hypotheses formulated for our study, described in Table 1. Note that, except for two hypotheses, results were consistent across novice and professional subjects. The deviations are in favor of the agile approaches. In particular, test-first professional programmers benefited more from the practice since they did not take significantly more time when compared with test-last programming. Moreover, their implementations were also improved in terms of correctness, whereas for novices, only test case completeness was improved by test-first programming. In the next section, we discuss our findings in more detail.

6 Discussions

In this section, we conduct further analysis of our findings and discuss some of their possible implications. Discussions are grouped by subject.

Pair programming Our results with respect to pair programming were quite interesting. In particular, for both samples – novices and professionals –, the agile practice yielded significantly better results with respect to correctness when compared with solo programming. An important implication of such a result is the following. Since our test cases also covered exceptional inputs (*e.g.*, *null* and boundary values) and pair programmers were more successful at producing code that passed such tests, it can be argued that the agile practice supports the production of more robust code. This is very important, particularly in the case of auxiliary functions, because such modules tend to be used by several parts of the system and should thus be reliable.

Table 13 Summary of the results of our experiments in terms of the formulated hypotheses for both samples

	Experiments with novices		**Replications with professionals**	
	Null hypothesis (0)	**Alternative Hypothesis (A)**	**Null hypothesis (0)**	**Alternative Hypothesis (A)**
H_1	$Correctness_{PP} = Correctness_{SP}$	$Correctness_{PP} > Correctness_{SP}$	$Correctness_{PP} = Correctness_{SP}$	$Correctness_{PP} > Correctness_{SP}$
H_2	$Correctness_{TF} = Correctness_{TL}$	$Correctness_{TF} > Correctness_{TL}$	$Correctness_{TF} = Correctness_{TL}$	$Correctness_{TF} > Correctness_{TL}$
H_3	$TestSize_{TF} = TestSize_{TL}$	$TestSize_{TF} > TestSize_{TL}$	$TestSize_{TF} = TestSize_{TL}$	$TestSize_{TF} > TestSize_{TL}$
H_4	$TestCoverage_{TF} = TestCoverage_{TL}$	$TestCoverage_{TF} > TestCoverage_{TL}$	$TestCoverage_{TF} = TestCoverage_{TL}$	$TestCoverage_{TF} > TestCoverage_{TL}$
H_5	$Effort_{PP} = Effort_{SP}$	$Effort_{PP} > Effort_{SP}$	$Effort_{PP} = Effort_{SP}$	$Effort_{PP} > Effort_{SP}$
H_6	$Effort_{TF} = Effort_{TL}$	$Effort_{TF} > Effort_{TL}$	$Effort_{TF} = Effort_{TL}$	$Effort_{TF} > Effort_{TL}$

H Hypothesis, *SP* Solo Programming, *PP* Pair Programming, *TF* Test-First, *TL* Test-Last

Since system reliability is the quality driver, a failed test case in our scenario is paramount, especially true for our experimental environment setting. Since we applied functional testing, each test covers a significant portion of the functionality (i.e., an input or output equivalence class or a boundary value). Therefore, a failed test case significantly impacts the system's validity. Again, in this sense, our results indicate that pair programming could prevent problems in auxiliary functions such as those reported by significant corporations, as discussed in Sect. 1.

The production of more robust code by pair programmers might be related to the live code inspection aspect of such a practice. While one programmer is writing code, the other performs an online parallel code review of the implementation at hand. This can improve the chances of revealing bugs even before they are introduced. In fact, code review seems to be an effective practice to reveal faults, as evidenced by other empirical studies (Bavota & Russo, 2015). The application of pair programming enables code review earlier, possibly without the need for additional review sessions later on in the project.

Another possible implication of our results is related to the use of pair programming by the industry. In fact, the application of such a practice inside companies comes in different flavors. As commented in Sect. 1, pair programming is a controversial practice: some strongly advocate its use while others are more skeptical about it (e.g., some managers believe it's inefficient (Matheny, 2015; Williams & Kessler, 2002); also, in a book by Bertrand Meyer, it is regarded as one of the *new and not good* practices put forward by some agile proponents (2014)). A recent survey involving more than 300 developers working with agile development shows that the use of such a practice is not consistent among practitioners (Williams, 2012) (it appeared with a higher standard deviation when respondents were asked to evaluate its importance within agile development).

Although some companies practice pair programming always, most seem to apply it only when dealing with complex code (also known as *situational* or *moderate* pair programming (Trikha, 2014)). In this sense, our study indicates that *pure* pair programming, that is, the use of the agile practice 100% of the time, could be an option to be considered by developers. This is because it appears to be effective even for the implementation of auxiliary functions, that is, parts of the system that would not be initial candidates for being coded with a moderate approach to pair programming. Another option would be to apply the practice periodically, regardless of the type of function. In fact, some companies have a policy of having at least two engineers look at every piece of code that goes to production. Pairing is one way to comply with such a policy, and pre-merge code review is another (Brock, 2015). However, while the latter requires an additional step in the development process, the first does not because it embeds inspection into the coding task itself.

With respect to effort, it is important to note that pair programmers required significantly more time to implement functions when compared with solo programmers. One of the reasons that might explain such an outcome is that the target functionalities in our experiments were narrowly scoped. Some studies show that pair programming starts to yield better productivity after some development time, caused, for instance, by pair jelling or when applied to high-complexity software projects (Sun et al., 2016). Since the target functions required little time to develop, subjects did not have time to take advantage of the technique from an effort perspective. Another factor that played a role here is that pair programmers usually require an initial setup time before starting to program. For instance, pairs tend to discuss the task at hand before starting to code (which can be seen as a very positive practice). Moreover, since, in some cases, pairs were not even acquainted, they sometimes had to introduce themselves and talk for a while before starting to work. Even

though these actions might only take a few minutes, since we are targeting small functions, they probably had an impact on the effort outcomes.

Some of the previous studies (Williams et al., 2000; Canfora et al., 2007; L. Salge & Berente, 2016) reached different conclusions with respect to effort, with pairs developing sometimes 40–50% faster than solo programmers. However, only programmers with long-standing industry experience were included, whereas ours also included a sample of novice programmers. Canfora et al.'s experiment (2007), on the other hand, addressed *pair designing* and maintenance tasks of models such as use cases and class diagrams, and not programming. L. Salge and Berente's studies (2016) indicated that even though pairs seem to code programs faster than individuals, such an effect is not statistically significant. Pairs generally produce higher-quality code.

These differences and, more importantly, the fact that we targeted narrowly-scoped functionality might explain the results reached in our experiment.

Demir and Seferoglu (2021) investigated the impact of pair programming on the state of flow (a state of heightened focus and immersion in activities) and the quality of code in coding education. The study found that pair programming significantly improved the flow state, leading to better concentration, enjoyment, and intrinsic motivation. This enhanced flow state resulted in higher code quality compared to solo programming due to real-time feedback and knowledge sharing. However, the paper also highlighted the need for more research in different educational settings and with varied programming tasks to fully understand the implications of pair programming. This call for further research, as well as the observed improvement in code quality through pair programming, aligns with the findings and recommendations of our paper.

Zieris and Prechelt (2021) explored the dynamics of pair programming (PP), arguing that proficiency is not directly linked to experience. The authors recognized that pair programming can have many benefits in the industry but also acknowledge that it involves some skills that might take time to learn and improve. They aimed to provide a better understanding of what makes pair programming successful by identifying key elements of pair programming skills and problematic behavioral patterns that can affect its success. They also mentioned open questions such as how PP novices manage to have good PP sessions and which elements of PP skill can be acquired through what types of experience.

For pair programming, an additional 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 address the involved nuances better. For instance, novice programmers might help soften the impact of programming vices present in experienced programmers (*e.g.*, as will be discussed, more professional programmers tended not to test their implementations when using test-last programming); and experienced programmers might help overcoming skill deficiencies present in novice programmers (*e.g.*, novice test-first programmers did not benefit as much from the practice as professionals).

Test-first programming Our results with test-first programming were quite interesting as well. In particular, both novices and professionals produced larger and higher-coverage test sets when using such a practice. This is an important result in favor of the agile approach since it encouraged the implementation of better and more test cases than test-last programming. Test-first programming also significantly improved the correctness of implementations for professional programmers.

The fact that many test-last programmers did not write a single test for their implementations – twelve professionals (around 70%) and sixteen novices (around 40%) – can be seen as a negative effect of the traditional approach. It is 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. It is also noteworthy that such an occurrence was more consistent among professional developers, as 70% did not test their implementations with test-last programming (against 40% from the novices sample).

Having at least a minimum set of tests for each function is generally seen as a positive characteristic of a software system. Indeed, two decades ago, Beck (2002) presciently asserted that "*any program feature without an automated test simply does not exist*". This proclamation, still strikingly relevant today, emphasizes that the absence of regression tests for certain functions can significantly undermine the assurance typically sought in the practice of TDD. Such promoted "safety net" might help alleviating the fear that added code might have broken other parts of the system and can also improve its reliability. Moreover, while regression testing the system, better test sets can improve the chances of finding faults at the integration of auxiliary functions with other parts of the system when these parts are changed later on in a project. Lower coverage test sets can fail to reveal the introduced faults because the paths in the program that the changes could sensitize might not be executed.

Note that although test-first programming promoted the implementation of larger and higher-coverage test sets, it did not yield better results in terms of correctness for novices. The development of auxiliary functions with lower complexity could have been impacted this time. It might be the case that since developers perceived functions as easy to be implemented, they tended to overlook their subtleties. However, this problem occurred less in the context of pair programming, as one of the developers might be more careful than the other about such nuances. It also occurred less for professional developers, as this sample was more successful with test-first programming than with test-last programming.

It must also be noted that test-first programming does not prescribe any testing technique or criterion to be followed, and even though the novice 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 that might have been affected by the practices were not analyzed (*e.g.*, design quality factors, which are sometimes pointed out to be enhanced by test-driven development).

Novice programmers also took significantly more time to develop functions when using test-first programming. This is mainly due to the additional effort of having to develop test cases first. As commented before, we noticed that, even though we encouraged subjects to test their implementations while using the test-last approach, many did not develop any tests. In fact, when we remove their results from the data set, the difference between the mean times becomes non-significant. For test-last programming, it becomes 24.77, and for test-first programming, 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, the majority of subjects developed test cases while applying that approach. It must be noted, however, that for professionals, no significant difference in terms of effort was observed, even when we consider programmers that did not test their implementations while using test-last programming.

General comments The outcomes of our experiments seem to indicate two important insights. For pair programming, it appears that indeed *four eyes are better than two* Bavota and Russo (2015). That is, having two programmers working on a same implementation helps avoiding faults that would otherwise go undetected by a single developer. This can be related to the live code inspection aspect of the agile practice, as commented before. It is also interesting to see that the agile practices performed very similarly: both improved the reliability of auxiliary functions in terms of correctness and/or test set completeness, and both sometimes required more effort in terms of development time.

Our results suggest that pair programming and test-first programming are thus important practices to improve code quality, in particular by avoiding the introduction of subtle bugs. One can even argue that the *Goto Fail* and *HeartBleed* Bland (2014) security bugs that have appeared in the media, although not necessarily in auxiliary functions, could have been avoided if these practices were used. For instance, it is reasonable to suppose that the extra `goto` statement in the *Goto Fail* bug would be detected if two programmers were developing the code. On the other hand, as discussed by Bland (2014), if unit tests were being used to guide the design of such a block of code, the extra `goto` statement could also be avoided.

It should be clear that our experiments take into account two main considerations while developing software: reliability in terms of (1) functional correctness and (2) test set size and coverage; and effort in terms of total development time. Agile practices might also have an impact on other aspects, such as quality in terms of design metrics, knowledge dissemination, and programmer satisfaction Romano et al. (2019). This should also be taken into account while considering the application of pair programming and test-first programming to develop auxiliary functions.

In reference to the five categories of factors that were identified as contributing to inconclusive TDD experiments by Ghafari et al. (2020), we present arguments that establish correlations between these categories and our study.

1. TDD Definition: In this research, we adopted a specific interpretation of TDD, focusing on the initial stage where tests precede function implementation. This approach allowed us to investigate the impacts of this test-first phase in a granular manner. However, our study did not encompass the full spectrum of TDD practices, particularly iterative code writing, and refactoring. This limitation may restrict the comprehensiveness of our insights, emphasizing only the benefits and drawbacks of the test-first stage rather than the entire TDD process. This caveat should be considered when interpreting our results.
2. Participants Selection: On the positive side, our participant selection process was designed to include a varied mix of individuals, with different levels of programming experience. This ensured a rich set of perspectives, potentially making our results more representative of a broader population of developers, thus enhancing the external validity of our findings. On the other hand, a limitation lies in the fact that not all participants were seasoned TDD practitioners. While this allowed us to capture the learning curve and potential difficulties faced by beginners, it also meant that we may not have fully leveraged the benefits of TDD as would be done by seasoned practitioners. Therefore, our results may lean towards the challenges and hurdles of TDD adoption rather than its potential benefits when executed by seasoned practitioners. Consequently, the generalizability of our findings to contexts involving experienced TDD practitioners could be limited.

3. Task Selection: The selection of tasks for TDD is pivotal for the outcome of the research. In our studies, we did not limit ourselves to synthetic tasks (easily comparable, for example, in terms of complexity). We used a variety of short tasks, including both synthetic and real-world tasks, thus ensuring our results were not confined to a specific type of task. This diversity in tasks used for our research contributes to enhancing the strength of our findings.
4. Type of Project: Our study primarily engaged in "greenfield" projects, where auxiliary functionalities were developed from scratch. This approach allowed us to effectively control the project environment and isolate the impacts of TDD practices. However, the lack of examination in "brownfield" projects involving pre-existing codebases might hinder the extrapolation of our findings to such contexts, which are common in the software development industry.
5. Comparisons: We focused on comparing the test-first approach with the test-last approach. This decision was based on our primary interest in the sequence of testing and coding rather than the broader scope of the entire TDD process. Positively, this allowed us to provide a clear and direct contrast between these two practices, thereby generating specific insights that contribute to the existing body of knowledge on the subject. The simplicity of our comparison also helped eliminate potential confounding factors that might have clouded our findings.

In conclusion, despite potential ambiguities in TDD research due to five identified factors, our studies have strived to address these issues. Our aim was to enhance the reliability and validity of our findings, potentially offering significant insights into TDD application.

7 Study limitations

A study by Siegmund et al. (2015) discusses the trade-offs between internal and external validity in software engineering experiments. According to their results, we can argue that our study combines a good balance between these validity aspects. In particular, internal validity was increased by controlling several variables (*e.g.*, by using a set of well-defined auxiliary functions and systematically developed test sets); and external validity was increased by including a sample of 37 professional developers. Despite having a smaller sample of professionals compared to novices, this discrepancy does not undermine our study's validity. It is important to note that our approach did not aim for a direct comparison between these groups, given that the level of experience could significantly affect the effectiveness of the programming methodologies examined.

Therefore, issues with our experiments still need to be resolved. The following subsections examine these limitations in light of the three kinds of validity threats listed by Wohlin et al. (2000). There are various potential threats to an experiment in each area. We provide a list of potential threats for each category, measures taken to decrease each risk, and recommendations for enhancements in subsequent assessments.

7.1 Internal validity

The lack of control over certain variables, such as the subjects' skill (beyond being in the same semester of the course) and the method of pairings (which was random), may have posed a threat to our experiments' internal validity. However, the repeated measures

design employed in the study decreased the likelihood of skill level influencing our findings, as the same students performed both test-last and test-first in solo and pair programming roles. Nonetheless, the random pairing approach may have affected the pair programming experiment. We believe that the sample size of 46 subjects performing solo and pair programming mitigated the impact of this threat, as various skill-based pairings may have taken place. We followed the dictum "block what you can, randomize what you cannot" because blocking pairings based on skills could have resulted in a small sample size. To eliminate this threat in future evaluations, we suggest blocking pairings based on skills, as done in some studies. However, a larger sample size may be required for this method.

Mortality, which refers to dropouts from the experiment as discussed by Wohlin et al. (2000), is another aspect of internal validity in our experiments with novices. Since 123 students were invited to participate in the primary experiments, the tasks did not correspond to the initial assignments. This could affect the balance of the assignments that were considered during the experiment's design. Due to the large sample sizes in both investigations (46 for the pair programming experiment and 39 for the test-first experiment), an adequate balance can still be maintained. This is strengthened by the initial assignment set that included redundant tasks in case some students dropped out.

7.2 External validity

Our participant selection process brought together a diverse group, enhancing the external validity and broadening the potential applicability of our results. The representativeness of the chosen functions, however, poses a threat to the external validity of our experiments. Due to their simplicity, one could claim that the functionalities do not represent the population of auxiliary functions. However, as stated previously, we intended to select functions with a limited scope to conduct a conservative evaluation. Since agile practices have had an impact on the development of specified functions in some instances, we can anticipate that they will also have an impact on more complex functions. In any event, undertaking additional experiments with larger auxiliary functions is one way to reduce this threat further. In the future, we intend to replicate our experiments using open-source systems with more complex auxiliary functions.

7.3 Construct validity

Construct validity is the extent to which the operationalization of a study's measures corresponds to the constructs in the real world. The construct validity of our pair programming experiment involving novice participants was likely undermined by their minimal or nonexistent prior experience with pair programming. Furthermore, most students had not previously collaborated with their respective partners in a programming context. A potential deficiency of our study lies in not comparing our approach with Iterative Test Last (ITL). ITL, much like TDD, fosters an iterative process, but the order of operations differs. In ITL, tests are written immediately after implementing a small code change, as opposed to before the code, as in TDD. Consequently, our findings regarding the effects of pair programming may be conservative. It should also be noted that we separately analyzed the agile practices. In other words, we did not study the combined effect of pair programming and test-first programming.

8 Conclusion

The effectiveness and impact of agile practices on software engineering remain a topic of ongoing research and debate in the software development community. Although most studies claim that further research is needed to better understand the advantages and disadvantages of Test-Driven Development (TDD), they have not yet directly tackled the development of auxiliary functionality with respect to the correctness gain and impact on time-to-market.

Our empirical investigation involved a total of 37 professional developers and 85 students and investigated the following factors: *reliability* in terms of functional correctness, test set size and coverage, and *effort* in terms of total development time. In the pair programming experiment with novices, the mean total development time was found to be 19.59 for solo programming and 37.91 for pair programming. This indicates that pair programming might require more time, but it also tends to increase reliability in terms of correctness.

In the test-first replication with professionals, results were quite surprising and diverse from the experiment with novices. Test-first programmers performed similarly to test-last programmers in terms of development time, with only a 10% increase in total development time, on average. This outcome indicates that more experienced professionals may take more advantage of test-first programming than novices. Note that professionals produced more reliable implementations with respect to correctness and test case completeness but took approximately the same time required when test-last programming was used.

These findings have exciting implications for the software engineering field, and we believe our results are solid because our experiments balanced internal and external validity aspects. In particular, internal validity was increased by controlling several variables, such as using a set of well-defined auxiliary functions and systematically developed test sets. Results suggest that adopting pair programming and test-first programming can lead to the production of more robust and reliable code. This is particularly relevant for developing auxiliary functions, which are often used by several system parts and thus need reliability. However, our findings indicate that the experience level of the developers can influence the effectiveness of agile practices such as PP and TDD. This underscores the need for additional research to assess how these practices can be most effectively deployed within diverse contexts and team structures. Moreover, these findings should be interpreted cautiously due to the inconclusive studies in the literature. For example, others, highlighted numerous factors potentially contributing to variations in TDD research outcomes, and these factors subsequently impact the practice's applicability to software practitioners.

In terms of future research, further investigation could focus on the impact of these practices on more complex functions. Additionally, it would be beneficial to explore how these practices can be combined for maximum effectiveness. For instance, how does PP influence the effectiveness of test-first programming and vice versa? Furthermore, we will replicate our study in the context of DPP (Distributed Pair-Programming) to see how it fares when working with programmers in remote locations. This extension would enrich our understanding of how agile practices adapt and contribute to a virtual work scenario, which has become increasingly common in the software development landscape. Our research contributes with a distinct empirical perspective to the ongoing discussions in software engineering, subtly affirming the potential advantages of pair programming and test-first programming in developing auxiliary functions to enhance code correctness and

reliability. It enriches the existing body of knowledge, providing a fresh lens to the literature and gently nudging toward unexplored paths for future research, particularly in the realm of agile development practices.

Author contributions All authors contributed equally to this work (study conception, design, material preparation, data collection and analysis, and writing).

Funding Partial financial support was received from Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP) and Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq).

Availability of data and materials The authors declare that the data (programs, test sets, function descriptions, and more) supporting the findings of this study are available within the paper and are accessible via the following link: http://www.ict.unifesp.br/fsilveira/data/SoftwareTestingExperiment_data.zip.

Declarations

Conflicts of interest The authors have no relevant financial or non-financial interests to disclose.

References

Abrahamsson, P., Hanhineva, A., & Jäälinoja, J. (2005). Improving business agility through technical solutions: a case study on test-driven development in mobile software development. In R. Baskerville, L. Mathiassen, J. Pries-Heje, & J. DeGross (Eds.), *IFIP Advances in Information and Communication Technology* (pp. 227–243). Germany: Springer.

Arisholm, E., Gallis, H., Dyba, T., & Sjoberg, D. I. (2007). Evaluating pair programming with respect to system complexity and programmer expertise. *IEEE Transactions on Software Engineering, 33*, 65–86. https://doi.org/10.1109/TSE.2007.17

Baldassarre, M. T., Caivano, D., Fucci, D., Juristo, N., Romano, S., Scanniello, G., & Turhan, B. (2021). Studying test-driven development and its retainment over a six-month time span. *Journal of Systems and Software, 176*, 110937. https://doi.org/10.1016/j.jss.2021.110937

Bavota, G., & Russo, B. (2015). Four eyes are better than two: On the impact of code reviews on software quality. *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (pp. 81–90). https://doi.org/10.1109/ICSM.2015.7332454

Beck, K. (2002). *Test Driven Development: By Example*. Inc, USA: Addison-Wesley Longman Publishing Co.

Begel, A., & Simon, B. (2008). Novice software developers, all over again. *Proc. of the ICER '08* (pp. 3–14). New York, NY, USA: ACM.

Bella, E., Fronza, I., Phaphoom, N., Sillitti, A., Succi, G., & Vlasenko, J. (2013). Pair programming and software defects-a large, industrial case study. *IEEE Transactions on Software Engineering, 39*(7), 930–953. https://doi.org/10.1109/TSE.2012.68

Bissi, W., Serra Seca Neto, A. G., & Emer, M. C. F. P. (2016). The effects of test driven development on internal quality, external quality and productivity: A systematic review. *Information and Software Technology, 74*, 45–54. https://doi.org/10.1016/j.infsof.2016.02.004

Black Duck Software Inc. (2014). Open Hub Code Search. https://www.openhub.net/. Accessed 07 Jun 2021

Bland, M. (2014) Goto Fail, Heartbleed, and Unit Testing Culture. http://martinfowler.com/articles/testing-culture.html. Accessed 07 Jun 2021

Brock, Z. (2015) What tech companies do a lot of pair programming? http://qr.ae/Rkh8Gl. Accessed 07 Jun 2021

Canfora, G., Cimitile, A., Garcia, F., Piattini, M., & Visaggio, C. A. (2007). Evaluating performances of pair designing in industry. *Journal of Systems and Software, 80*, 1317–1327. https://doi.org/10.1016/j.jss.2006.11.004

Cellan-Jones, R. (2010). Sony's leap year bug. http://www.bbc.co.uk/blogs/thereporters/rorycellanjones/2010/03/sonys_millennium_bug.html. Accessed 07 Jun 2021

da Silva Estácio, B. J., & Prikladnicki, R. (2015). Distributed pair programming: A systematic literature review. *Information and Software Technology, 63*, 1–10. https://doi.org/10.1016/j.infsof.2015.02.011

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

Demir, Ö., & Seferoglu, S. S. (2021). A comparison of solo and pair programming in terms of flow experience, coding quality, and coding achievement. *Journal of Educational Computing Research, 58*(8), 1448–1466. https://doi.org/10.1177/0735633120949788

Desai, C., Janzen, D., & Savage, K. (2008). A survey of evidence for test-driven development in academia. *ACM SIGCSE Bulletin, 40*, 97–101. https://doi.org/10.1145/1383602.1383644

Dingsøyr, T., Nerur, S., Balijepally, V., & Moe, N. (2012). A decade of agile methodologies: Towards explaining agile software development. *Journal of Systems and Software, 85*, 1213–1221. https://doi.org/10.1016/j.jss.2012.02.033

Edwards, S. H. (2004). Using software testing to move students from trial-and-error to reflection-in-action. *SIGCSE Bulletin, 36*, 26–30. https://doi.org/10.1145/1028174.971312

Erdogmus, H., Morisio, M., & Torchiano, M. (2005). On the effectiveness of the test-first approach to programming. *IEEE Transactions on Software Engineering, 31*, 226–237. https://doi.org/10.1109/TSE.2005.37

Fucci, D., Erdogmus, H., Turhan, B., Oivo, M., & Juristo, N. (2017). A dissection of the test-driven development process: Does it really matter to test-first or to test-last? *IEEE Transactions on Software Engineering, 43*(7), 597–614. https://doi.org/10.1109/TSE.2016.2616877

Fucci, D., Romano, S., Baldassarre, M. T., Caivano, D., Scanniello, G., Turhan, B., & Juristo, N. (2018). A longitudinal cohort study on the retainment of test-driven development. *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. ESEM '18* (pp. 18–11810). New York, NY, USA: ACM. https://doi.org/10.1145/3239235.3240502

George, B., & Williams, L. (2003). An initial investigation of test driven development in industry. *Proc. of the ACM SAC 2003. SAC '03* (pp. 1135–1139). New York, NY, USA: ACM. https://doi.org/10.1145/952532.952753

Ghafari, M., Eggiman, M., & Nierstrasz, O. (2019). Testability first! *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)* (pp. 1–6). https://doi.org/10.1109/ESEM.2019.8870170

Ghafari, M., Gross, T., Fucci, D., & Felderer, M. (2020). Why research on test-driven development is inconclusive? *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). ESEM '20.* New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/3382494.3410687

Guerra, E., & Aniche, M. (2016). Chapter 9 - achieving quality on software design through test-driven development. In I. M. S. A. G. Tekinerdogan (Ed.), *Software Quality Assurance* (pp. 201–220). Boston: Morgan Kaufmann. https://doi.org/10.1016/B978-0-12-802301-3.00009-0

Hannay, J. E., Dybå, T., Arisholm, E., & Sjøberg, D. I. K. (2009). The effectiveness of pair programming: A meta-analysis. *Information and Software Technology, 51*, 1110–1122. https://doi.org/10.1016/j.infsof.2009.02.001

IEEE. (1990). Ieee standard glossary of softw. eng. terminology. *IEEE Standard Glossary of Softw. Eng. Terminology.* New York: IEEE Computer Society Press.

Karac, E. I., Turhan, B., & Juristo, N. (2019). A controlled experiment with novice developers on the impact of task description granularity on software quality in test-driven development. *IEEE Transactions on Software Engineering*. https://doi.org/10.1109/TSE.2019.2920377

Kazerouni, A. M., Shaffer, C. A., Edwards, S. H., & Servant, F. (2019). Assessing incremental testing practices and their impact on project outcomes. *Proceedings of the 50th ACM Technical Symposium on Computer Science Education. SIGCSE '19* (pp. 407–413). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/3287324.3287366

Krasner, H. (2021). The cost of poor software quality in the us: A 2020 report. Technical Report CISQ-TR-2021-01, Consortium for Information & Software Quality. Accessed 26 Mar 2021. https://www.it-cisq.org/the-cost-of-poor-software-quality-in-the-us-a-2020-report/

L. Salge, C.A., Berente, N. (2016). Pair programming vs. solo programming: What do we know after 15 years of research? In: 2016 49th Hawaii International Conference on System Sciences (HICSS), pp. 5398–5406. https://doi.org/10.1109/HICSS.2016.667

Latorre, R. (2014). Effects of developer experience on learning and applying unit test-driven development. *IEEE Transactions on Software Engineering, 40*(4), 381–395. https://doi.org/10.1109/TSE.2013.2295827

Lemos, O. A. L., Bajracharya, S., Ossher, J., Masiero, P. C., & Lopes, C. (2011). A test-driven approach to code search and its application to the reuse of auxiliary functionality. *Information and Software Technology, 53*, 294–306.

Lemos, O. A. L., Ferrari, F. C., Silveira, F. F., & Garcia, A. (2012). Development of auxiliary functions: Should you be agile? an empirical assessment of pair programming and test-first programming. *2012*

34th International Conference on Software Engineering (ICSE), (pp. 529–539). https://doi.org/10.1109/ICSE.2012.6227163

Lemos, O. A. L., Silveira, F. F., Ferrari, F. C., & Garcia, A. (2018). The impact of software testing education on code reliability: An empirical assessment. *Journal of Systems and Software, 137*, 497–511. https://doi.org/10.1016/j.jss.2017.02.042

Madeyski, L. (2010). *Test-Driven Development: An Empirical Evaluation of Agile Practice* (1st ed.). Berlin: Springer.

Matheny, K. (2015). Why is it that when pair programming produces better code, almost no company practices it? http://qr.ae/RkhnlG. Accessed 07 Jun 2021

Meyer, B. (2014). *Agile!: The Good, the Hype and the Ugly* (1st ed.). Berlin: Springer.

Montgomery, D. C. (2006). *Design and Analysis of Experiments*. New York: John Wiley & Sons.

Munir, H., Moayyed, M., & Petersen, K. (2014). Considering rigor and relevance when evaluating test driven development: A systematic review. *Information and Software Technology, 56*(4), 375–394. https://doi.org/10.1016/j.infsof.2014.01.002

Myers, G. J., Sandler, C., Badgett, T., & Thomas, T. M. (2004). *The Art of Software Testing* (2nd ed.). New York: John Wiley & Sons.

Nagappan, N., Maximilien, E. M., Bhat, T., & Williams, L. (2008). Realizing quality improvement through test driven development: Results and experiences of four industrial teams. *Empirical Software Engineering, 13*(3), 289–302.

Pressman, R. S., & Maxim, B. R. (2020). *Software Engineering: A Practitioner's Approach*. Berlin: McGraw-Hill Education.

Rafique, Y., & Mišić, V. B. (2013). The effects of test-driven development on external quality and productivity: A meta-analysis. *IEEE Transactions on Software Engineering, 39*(6), 835–856. https://doi.org/10.1109/TSE.2012.28

Romano, S., Fucci, D., Baldassarre, M. T., Caivano, D., & Scanniello, G. (2019). An empirical assessment on affective reactions of novice developers when applying test-driven development. In X. Franch, T. Männistö, & S. Martínez-Fernández (Eds.), *Product-Focused Software Process Improvement* (pp. 3–19). Cham: Springer.

Salleh, N., Mendes, E., & Grundy, J. (2010). Empirical studies of pair programming for CS/SE teaching in higher education: A systematic literature review. *IEEE Transactions on Software Engineering*. https://doi.org/10.1109/TSE.2010.59. PrePrints.

Saltz, J.S., Shamshurin, I. (2017). Does pair programming work in a data science context? an initial case study. In: 2017 IEEE International Conference on Big Data (Big Data), pp. 2348–2354. https://doi.org/10.1109/BigData.2017.8258189

Santos, A., Vegas, S., Dieste, O., Uyaguari, F., Tosun, A., Fucci, D., Turhan, B., Scanniello, G., Romano, S., Karac, I., Kuhrmann, M., Mandić, V., Ramač, R., Pfahl, D., Engblom, C., Kyykka, J., Rungi, K., Palomeque, C., Spisak, J., … Juristo, N. (2021). A family of experiments on test-driven development. *Empirical Software Engineering, 26*(3), 42. https://doi.org/10.1007/s10664-020-09895-8

Shull, F., Singer, J., & Sjøberg, D. I. K. (2008). *Guide to Advanced Empirical Software Engineering*. Secaucus, NJ, USA: Springer.

Siegmund, J., Siegmund, N., & Apel, S. (2015). Views on internal and external validity in empirical software engineering. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering* (Vol. 1, pp. 9–19). https://doi.org/10.1109/ICSE.2015.24

Sillitti, A., Succi, G., & Vlasenko, J. (2012). Understanding the impact of pair programming on developers attention: A case study on a large industrial experimentation. *2012 34th International Conference on Software Engineering (ICSE)* (pp. 1094–1101). https://doi.org/10.1109/ICSE.2012.6227110

Spence, N. (2011). Apple offers advice on iPhone alarm bug. http://www.macworld.com/article/1155509/dst_bug_iphone.html. Accessed 07 Jun 2021

Sun, W., Marakas, G., & Aguirre-Urreta, M. (2016). The effectiveness of pair programming: Software professionals' perceptions. *IEEE Software, 33*(4), 72–79. https://doi.org/10.1109/MS.2015.106

Swamidurai, R., & Umphress, D. (2012). Collaborative-adversarial pair programming. ISRN. *Software Engineering*. https://doi.org/10.5402/2012/516184

Tosun, A., Dieste, O., Vegas, S., Pfahl, D., Rungi, K., & Juristo, N. (2021). Investigating the impact of development task on external quality in test-driven development: An industry experiment. *IEEE Transactions on Software Engineering, 47*(11), 2438–2456. https://doi.org/10.1109/TSE.2019.2949811

Trikha, R. (2014). Pair Programming: Yay or Nay? http://www.cybercoders.com/insights/pair-programming-yay-or-nay. Accessed 07 Jun 2021

Weimer, W., Forrest, S., Le Goues, C., & Nguyen, T. (2010). Automatic program repair with evolutionary computation. *Communications of the ACM, 53*, 109–116.

Williams, L. (2012). What agile teams think of agile principles. *Communications of the ACM, 55*(4), 71–76. https://doi.org/10.1145/2133806.2133823

Williams, L., & Kessler, R. (2002). *Pair Programming Illuminated* (p. 71). Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. Chap. 15.

Williams, L., Kessler, R. R., Cunningham, W., & Jeffries, R. (2000). Strengthening the case for pair programming. *IEEE Software, 17*(4), 19–25. https://doi.org/10.1109/52.854064

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., & Wesslén, A. (2000). *Experimentation in Software Engineering: an Introduction*. Norwell, MA, USA: Kluwer Academic Publishers.

Xu, F., & Correia, A.-P. (2023). Adopting distributed pair programming as an effective team learning activity: a systematic review. *Journal of Computing in Higher Education*. https://doi.org/10.1007/s12528-023-09356-3

Zhong, B., & Li, T. (2020). Can pair learning improve students' troubleshooting performance in robotics education? *Journal of Educational Computing Research, 58*(1), 220–248.

Zieris, F., & Prechelt, L. (2021). Two elements of pair programming skill. *Proceedings of the 43rd International Conference on Software Engineering: New Ideas and Emerging Results. ICSE-NIER '21* (pp. 51–55). New York: IEEE Press. https://doi.org/10.1109/ICSE-NIER52604.2021.00019

Authors and Affiliations

Otávio Lemos[1] · Fábio Silveira[1] · Fabiano Ferrari[2] · Tiago Silva[1] · Eduardo Guerra[3] · Alessandro Garcia[4]

✉ Fábio Silveira
fsilveira@unifesp.br

Otávio Lemos
otavio.lemos@unifesp.br

Fabiano Ferrari
fcferrari@ufscar.br

Tiago Silva
silvadasilva@unifesp.br

Eduardo Guerra
guerraem@gmail.com

Alessandro Garcia
afgarcia@inf.puc-rio.br

[1] Science and Technology Institute, Federal University of São Paulo, São Paulo, Brazil

[2] Computing Department, Federal University of São Carlos, São Carlos, Brazil

[3] Faculty of Engineering, Free University of Bolzen-Bolzano, Bolzano, Italy

[4] Informatics Department, Pontifical Catholic University of Rio de Janeiro, Rio de Janeiro, Brazil