Teaching Test-First Programming: assessment and solutions

Marcello Missiroli and Daniel Russo and Paolo Ciancarini
DISI, University of Bologna
Email: {marcello.missiroli,daniel.russo,paolo.ciancarini}@unibo.it

***Abstract*—Developing high quality software is a major industry concern, since programs that "just work" may not be suitable to contemporary technological challenges. Agile practices, such as Test-First development (TFD), may help in this direction. However, in our experience this technique is introduced late (if ever), when programmers' habits are already set and difficult to change. Early exposure to TFD in formal education could be an answer to that, but putting the principle into practice poses unexpected challenges. In this work we examine the short- and long-term impact of young programmers' exposure to TFD, highlighting its limits and proposing a reinforced teaching approach.**

***Index Terms*—Test-first development, Education, K-12**

I. Introduction

Test-First Development (TFD), sometimes also called Test Driven Programming or Test Driven Development, is one of the main practices used in Agile development. It is a three-step process: first, write the test-case; then write the (minimal amount of) code that passes the test, then refactor the code. Repeat the process until satisfied with the result. Despite its name, TFD is *not* a testing technique, but a programming/design practice.Though the debate is still open, it is generally assumed [1] that programming Test-first results in more maintainable, flexible, easily extensible code, as well as greater test overage — resulting in better quality; sometimes, but not always, it is counter-balanced by slightly lesser productivity [2].

Due to recent changes to school worldwide (e.g. in the UK [3]), and suggestions from teachers, the age at which kids start coding continues to decrease. Hence, we developed our idea of introducing TFD as early as possible in programming courses for young students to verify its effectiveness.

We performed a series of tests on more than 100 high school students learning to program in various Italian schools. We report the results of the early exposure of this technique, considering the code produced and the subjective point of view of the students, even after a long period of time. We conclude that TFD teaching is not as simple as we envisioned; we therefore propose a teaching strategy to facilitate the introduction of this practice to beginner programmers.

This paper has the following structure: Section II summarizes the related literature on both education and Agile development; Section III outlines the general research methodology we used; Section IV shows the results of code analysis and student surveys; in Section V we present a summarized version of field observations and reflective interviews; in Section VI we collect all the information of our researches in a synthetic way and propose a teaching strategy; finally, in Section VII, we draw our conclusions.

II. Related work

Research on TFD-TDD in the upper education context shows unclear results. Muller [4] notices neither higher productivity nor reliability, even though students were exposed to a full XP course. Kaufmann [5] directly compares two groups with different programming methodologies; TFD seems to produce more code, and all students received better grades. Yaha et al. [6] explain that groups with mixed, moderate and high programming competency produced better quality code whereas lower competency groups did not. Bowyer [7] states that all students succeeded in the TFD experience, but reflection on the process was insufficient and inconclusive. Erdogmus [8] obtained less bugged code, due to the increased number of tests. Therefore, Jones's conclusion [9] that TFD cannot be considered as a "Silver bullet" for improving code quality still holds.

Problems in *teaching* TFD are studied thoroughly by Mugridge. He suggested that more than one semester is needed for teaching test-first [10] and he points out that students are "reluctant to set aside the way they currently program" [11], displaying a sort of Baby-duck syndrome [12].

We found no specific work on TFD in high schools, if we exclude very basic elements of the practice in [13].

III. Research Design

In this section, we describe the methodology used to design, plan and execute our research.

In accordance to the foundational aspect of this study, we chose a research methodology which enabled us to explore several research dimensions: learning outcomes, satisfaction and opinions. We identified **Mixed Methods** (MM) [14] as the research strategy most suited to our needs, particularly due to the educational context. MM allowed us to join qualitative and quantitative results in order to obtain a comprehensive interpretation and is widely used in software engineering [15], [16], [17], [18].

We then formalized our research questions:

- RQ1: Does TFD have any effect on the learning process?
- RQ2: Does it impact the way inexperienced students code?

- RQ3: Will this experience have long-lasting effects on students?

To do so, we used two distinct research methodologies in order to fully exploit the benefits of the methodology. Specifically, we used qualitative and mixed research methodology to address the above questions.

The mixed research section provides the basis of our investigation. The first part consists in a code analysis (IV-A), directly addressing RQ2. It compares code produced by the students during an experiential learning event featuring TFD with that realized for similar projects developed in a traditional manner. Code is evaluated using standard, well-established code metrics. The second part (IV-C) refers to students' feedback on the above experiences, related to RQ1. It took the form of a *post-mortem* online questionnaire, without direct teacher supervision.

In the first part of the qualitative section we report the results of the field observations recorded during the two events (V-A, V-B). The focus was on the general mood, possible tensions and student interaction, also related to RQ1.

The second part of the qualitative section consists in a reflective interview with students after a significant amount of time has passed since the experiment (V-C), to check for long-lasting effects and opinion changes. We call that "post-burial" interviews, and is reltatated to RQ3.

At the end of all the above procedures, we collated and synthesized the results in the Discussion section (VI), answering the research questions.

IV. Case Studies

A. Code analysis

We follow the experiment methodology proposed by Wohlin, et al. [19]: Design, Selection, Procedures, Data collection, Analysis, Plan validity, Study limitations.

1) Design: The object of this study is exploratory in nature. We want to understand if the introduction of a radically different methodology in teaching programming changes not only quantifiable elements, such as basic productivity, and code quality metrics, but also how the methodology is perceived (and possibly accepted) by the learners. We had therefore to acquire codebases of similar size, scope and difficulty developed both with TFD and traditional methodologies. We then examined them according to standard metrics. We also needed data on the opinions of the students on the new methodology. As a selection strategy, we referred to our own experiment [20],[21] and its repetition on other subjects, performed one year later, that offered a reasonably reliable source of the data we needed.

2) Selection: The context of the experiment was the students of the next-to-last year of Computer Science course in Italian Technical School for Informatics (ITIS - Informatics). Our choice of this kind of school is due to the extremely focused curriculum on Computer Science - about 13 hours a week, or 30% of the weekly school time. As a result, students of 12th grade have a programming knowledge comparable to mid-year college freshmen. Overall, we tested 105 students in 5 classes over a period of 2 years.

B. Procedures

We extracted the codebase data from the experiment presented in [20], taking into account only data related to Test-First Development. The Agile codebase was created during a three-hour long experience using Java. First, a training session introduced Agile methodologies and TFD in particular, immediately followed by a hands-on experience. During the next two hours students performed a variant of the *Tennis Game Kata* [22]. They were given a sequence of JUnit tests (see Appendix), used both as a guide and as acceptance tests; they were forced to follow the sequence of 8 JUnit tests exactly, and had to stop after a few steps for refactoring. This method is a variation of the most used strategy to introduce TFD to professional — the teacher provides the test, the learners resolve it. The evaluation was based on the amount of passed tests.

As an added bonus, our study partitioned results according to ability (Good, Average, Poor) and team composition (Pair, Solo) allowing for a detailed and unaggregated analysis. Classes were partitioned by the class teachers using the following guidelines:

- **"Good"** (G) students, whose average grade is 7 or more in the Italian ten-point grading system (roughly equates to US B grade and above, or 71% of the maximum score);
- **"Average"** (A) students, who are "more or less sufficient" (51-70%);
- **"Poor"** (P) students, struggling with the subject. Average grade is 5.5 (50% or less).

We then formed six types of pairs, combining student types: G+G, G+A, G+P, and so on. Solo team were simply labeled by ability (G,A,P).

When the experiment was repeated the following year, some variations were introduced:

- The difficulty of the test was sensibly raised;
- We only had pair teams, instead of both pairs and solo programmers;
- The test was not graded in any way;
- Three schools were involved instead of a single one.

Though the experiment was performed under controlled circumstances, TFD was not the only change that was introduced. We cannot exclude that the effects of multiple changes could interact. For this reasons, we consider the data acquired as second degree data.

The baseline codebase was acquired with the help of the class teachers. A suitable, recently executed lab exercise with characteristics similar to those of the experiment were collected. As the data was acquired without direct interaction, we consider them as second degree data.

Next, we had to define how to measure the quality of the software. This is very difficult in general [23], and especially so in case of elementary algorithms. Since there are several metrics that can be applied to software, we decided to separately analyze three "classic" code metrics: cyclomatic complexity (CC) per source code file, total number of non-commenting lines of code (NCLOC), and programming issues (ISS). Other metrics where collected but later discarded, as they were clearly not suitable for simple programs developed by unexperienced programmers.

As further evaluation parameters we added the number tasks completed by the team (TC), and the overall quality indicator SQALE [24]. With the exception of TC, all values were calculated by *Sonarqube* [25].

1) Data Collection.: Collecting the codebase for the first experiment was already done for our previous work [20] . During the second experiment we simply followed the same protocols used before.

2) Absolute Data Analysis.: Table I shows the overall results of the code analysis. As we were not interested in differences between teams of different size, we grouped Solo programmers' results with the Pair of the same ability group (G with G+G groups, and so on). We then examined the different metrics, one by one.

CC average/file. We have an average Cyclomatic Complexity value of 17.93, almost *twice* the value of the codebase, in all skill groups and pairings.

ISS/project. There is an general *increase* in programming issues, though there are marked differences in skill groups. For example, G+A and G+P display some decrease of issues, whereas A+P groups show a dramatic increase.

TC & NCLOC. The overall number of lines of code produced is generally higher that in the codebase. Especially so in case of students that have at least one "Good" student. Though it might be debatable whether NCLOC is an indication of productivity or inefficiency, the TC metric clearly indicates that teams writing more code also solved the most tasks.

SQALE index average/project. Again, no great difference can be seen overall. There is a small increase of quality in the case of G+P groups.

We were rather surprised with these results. TFD is supposed [26] to produce slightly less code, but of better quality. In fact, our results proved the exact opposite. Our interpretation is that classic code quality indicators are not suited to evaluating quality in elementary programs. For example, an IDE-generated boilerplate program with minimal alterations rates high both as SQALE and CC, but its real contribution to the project is minimal. Conversely, a code that solves the complex TieBreak task has intrinsically a larger CC value, also resulting in worse SQALE value and creating potentially more issues. Refactoring also has limited impact on such simple programs.

It seems that our basic model of applying standard metrics to this type of codebases was flawed.

3) Relative Data Analysis.: Since the above analysis proved inconclusive, we decided to directly compare results of code produced in the two installments of the experiment, since they had very similar requirements and constraints.

In this case the data show a significant *positive trend* related to code quality on two important metrics: CC and SQALE. More specifically, as Table II shows, overall CC was better by about 15%, showing no preferences for ability groupings.

If we exclude A+P and P+P groups, a large percentage of which delivered empty or a low quantity of code that, consequently, had good CC and SQALE scores, the result still shows a significant improvement. Since the main difference between the 2014-15 and 2015-16 run was the introduction of a robustness test, this analysis shows how the experiment is sensitive to how it is built, and at the same time how students respond to problem presentation and conditions.

C. Student Feedback Survey.

As a further element of our investigation, we discuss if the experiment was *deemed* useful in terms of learning experience. We again followed Wohlin's methodology [19].

1) Research Participants and Data Gathering Tools.: As part of our previous work, we designed an **Experiment Feedback** questionnaire for all students who participated in the experiment, aiming at determining how the new methodologies were received at a psychological level. In particular, we wanted to determine if the programming experience was pleasant and useful. In almost all questions we used a standard 5-level Likert-scale. We pointed out that no personal information was required, and in particular no information would be forwarded to the class teachers, to preserve student privacy. The same questionnaire was reused, with minimal differences in wording and eliminating a redundant question, for the second experiment. In both cases, the collected data can be considered as first degree data.

2) Research Process: The experiment feedback questionnaire was proposed to the students after the experiment was over by using an automated survey tool, stating that it should be completed within the week. The questions were the almost the same for the two installment of the experience. The response turnout was surprisingly high, 70% in the first run and 55% in the second run.

3) Data Analysis: Results are displayed in Table III. On the average, students think that TFD offers a good opportunity to code more correctly, and find more bugs, in all cases. However, the perception of usefulness is substantially higher during the second experience, especially in questions related to debugging.

If we analyze this last result further, we notice that student are polarized almost evenly in "lovers" and "haters" of this practice, a fact which is not related to performance but rather to group composition (groups with G students had more lovers that haters).

D. Validity and Plan validity

1) Code Analysis.: The validity of the code comparison depends on how close the problems tackled in the two problems was similar, and how reliable were the conditions under which the coding experience was executed.

The similarity between the baseline and the experiment codebase was assured because the experiment was modeled on standard competences acquired (and tested) as part of the regular curriculum. Moreover, the problem was shown and approved by the class teacher, confirming the suitability of the problem. Regarding the reliability of coding conditions, we concluded that they should not differ much from those used during the experiment; the amount of time, the lab, and the teacher were the same; only some psychological condition posed a sort of social threats. During standard

TABLE I
ABSOLUTE CODE ANALYSIS

	TC /group	CC (AVG)	CC DIFF v. base	ISS /proj	ISS DIFF v. base	NCLOC /proj	NCLOC DIFF v. base	SQALE	SQALE (base)
G+G/G	5.42	17.88	+114.96%	20.57	+98.88%	95,57	+97.32%	A-	A-
G+A	6.33	23.40	+40.48%	14.30	-10.55%	96.70	+44.67%	B+	B+
G+P	2.20	21.20	+161.73%	12.00	-17.05%	50.50	-22.13%	A–	B+
A+A/A	2.80	15.14	+132.51%	11.86	+27.13%	51.00	-13.81%	B-	B+
A+P	2.80	17.55	+164.91%	13.70	+265.33%	51.47	+59.59%	A-	A-
P+P/P	2.00	12.43	+70.77%	6.05	-40.52%	39.50	+6.60%	A–	A-
OVERALL	3.76	17.93	+114.23%	13.08	+53.87%	64.12	+28.71%	A-	A-

TABLE II
RELATIVE CODE ANALYSIS

	Average CC/file	DIFF vs 2015	SQALE (2016)	SQALE (2015)	DIFF vs 2015
G+G/G	23.17	+2.96%	AABBB	ABAA	unchanged
G+A	19.17	-8.73%	AABBB	BBAAA	unchanged
G+P	14.67	-41.33%	AAA	CA	better
A+A/A	14.33	-36.30%	AABC	CAA	worse
A+P	16.92	-9.78%	AAABCC	ACA	better
P+P/P	4.33	-78.33%	AABB	ABC	better
AVERAGE	***15.4***	***-28,58%***	B+	B-	+
AVERAGE (no A+P and P+P)	***17.83***	***-20,85%***	B+	B-	+

TABLE III
STUDENT FEEDBACK ON TFD COMPARISON

	2015-16	**2014-15**
It was a positive experience	Agree (2.51)	Agree (2.71)
Our pair worked well	Agree (2.51)	Agree (2.98)
SOLO produces better code	Disagree (1.48)	Disagree (1.43)
SOLO produces more code	Disagree (1.40)	Disagree (1.51)
I learned something	Slightly agree (2.41)	Undecided (1.88)
Faster development time	No change (2.04)	No change (2.20)
Adherence to specifications	Better (2.5)	Better (2.5)
Code correctness	Better (2.9)	Better (2.52)
More efficient coding	Slightly better (2.47)	Undecided (2.37)
I will use agile practices again	Possibly (2.35)	Undecided(2.00)
I found more bugs	Strongly agree (2.84)	Undecided (2.32)

school activities the students had the pressure of the grade, during the experiment they had the time limitation and the need to use one or more unknown methods. After consulting with the class teachers, we convened that these differences would not produce significant variations in the final results. Consequently, we can also directly compare results of the first and second comparison to one another, since they were obtained under the same conditions.

We now consider plan validity. We understand that it has four aspects: Construct Validity, Internal Validity, External Validity, and Reliability [19].

Construct Validity is under control. We used several, well established and deterministic metrics to evaluate the code, and we considered them separately in our analysis.

Threats to Internal Validity depend on how much the tasks that originated the codebases were similar, the conditions under which the coding experiment was executed, and how the investigated metrics might be affected by some uncontrolled factor. The cooperation with the class teacher assures that both experiments were indeed similar and executed in similar psychological and environmental conditions. The fact that more than one Agile practice was tested at the same time can be a threat to internal validity, but the impact was regarded as limited. Hence, also this aspect was under control.

External validity for data related to the first experiment was under control, since the population was drawn from different schools and cities. In addition, all target classes were, in the opinion of teachers, of average level.

Reliability was under control. Programming tasks were based on standard curricula, all code metrics were well-established and the tools used were open and free software. Both codebases are available at https://bitbucket.org/marcello_missiroli/agileschool_data.

2) Surveys: Construct Validity is under control. We used several clearly defined questions and rated the responses using a standard Likert scale.

Threats to Internal Validity are not high. While there was no direct control on how the student or the teachers submitted the forms, they had no incentive to cheat. So, we assume they responded truthfully.

Threats to External Validity are under control. In the case of the student survey, the population is a sample of the typical student population, and the turnout was significantly high (around 70%). Although we used a non-probabilistic convenience sample, still the sample population is comparable to a probabilistic stratified random sample. All students were equally exposed to the experience and were representative of the class. Moreover, since the population is so small (and well known), the difference between probabilistic and non-probabilistic sample techniques are very narrow. Thus we consider validity under control. Raw analysis results are available at the same URL stated above.

V. Qualitative research

We report the field observations related to the first and second experiment installments and a summary of the reflective interview with students. Our goal was to monitor the effects of the learning experience on student mood, perception and practice acceptance. All field observations were performed by the first author.

A. Field observations during the first event

Impressions were simply observed, since we were mostly interested in the overall class "mood and feel" created by the activity. Additionally, some photos were taken. We noted a significant tension in several cases. Some pairs and solo programmers openly protested that they did not want to participate, and refused to turn in results. In other cases we noticed some arguing and confrontations. Finally, an A+A-type pair even displayed evident bickering.

B. Field observations during the second event

This time we took extra care in observing non-standard pairs, flagged as such by the class teacher. We also had two cases in which, due to absences, the class teacher had to step in and formed an unusual teacher-student pair.

General mood was sensibly more relaxed than the previous run. For example, no pair failed to deliver the code, however bad that might have been (except for technical reasons). No open bickering was observed. Possibly this was due to the clearer declaration of intent given before the start of the test. Even the need to resort to their own laptops to perform the test - due to last-minute lab unavailability - did not have a noticeable impact on the students' attitude.

The increased difficulty of the exercise had a visible impact, especially in groups containing at least one Poor student. We remark a couple "outliers" cases that might be of interest: an A+P pair that had the best performance of all involved, due to intrinsic motivation; a G+A with an exceptionally good student that underperformed due to overcoding.

We note that, beside the cases noted above, students were rather satisfied with the experience and did not feel stressed or pressured, even those that did not perform too well. Teachers confirmed a general "good mood" during the experience.

C. Post-burial interviews

As we were interested in long-term effects of the experience, we performed a reflective interview with a selected number of participants. We wanted to have a more detailed opinion with respect of what we could deduce from online survey, and also see if a one-shot experience could have long-lasting effects.

The interviews were performed in the period September-October 2016, from 5 to 18 months after the experiment. It was too difficult to contact all participants, as many had left school. Therefore, we opted for a non-probabilistic convenience sampling according to how well the person fared during the TFD tests. In all, 3-4 students for each class were interviewed. Results show that:

- *All* students had a vivid recall of the experience, and generally labeled it as 'good' or 'interesting'. Most think it will be useful in their programming job, but only as *"things get complicated, otherwise it becomes useless"*.
- *No* student ever tried to apply TFD in subsequent projects.
- Students that succeeded in at least 40% of the test suite felt confident they could write effective tests if they needed to do so; student that had difficulties during the experience stated they had *"no clue, even on how to begin"*.
- Refactoring was not internalized as an important concept (only one student was able to tell what it was).

VI. Discussion and proposal

In this section, we summarize the results obtained in the previous sections and discuss the multiple perspective offered by the Mixed Method methodology to address the research questions described in section III.

A. Discussion

The answer to RQ1 (*"Does TFD have any effect on the learning process?"*) is yes, but this is not the end of it. If taught properly, in a relaxed atmosphere, Test-First Development has a good potential as a means of writing less buggy code, as several studies in the industry and university level show. The practice is generally welcomed by students and moreover it can become an asset in the job market. It will probably not influence code quality - at least at first.

The key factor is the word **properly**. We note that a few subtle variations in the pilot experience resulted in substantial differences in results (code quality, task completed) and perception. This suggests that particular care should be taken in both the instructional and the exercise phase. In any case, we point out the initial student evaluation on this new programming practices remains divided.

Code quality does not seem to be affected, so we have to answer negatively to RQ2 (*"Does it impact the way young programmers code?"*). This might change in case of more complex programs, but it requires further investigation.

Field observations also showed that psychological and relational aspects are very important dimensions. Generally, pairs with at least a "Good" student performed significantly better than the others. We also note that the practice of refactoring, very important in TFD, plays only a limited role in this approach. One reason is that at this elementary level refactoring does not have a significant impact. It is limited to coding and naming conventions, optimizing loops, sometimes algorithm choice. Major errors, such as repeated IFs, can be easily spotted by the teacher as a part of regular teaching. Moreover, we found that it is more efficient to write/suggest a test that uncovers potential problems (such as code rigidity) than abstractly define an indication of "acceptably good code" based on metrics the students do not really understand and, as we have seen, are not good indicators of elementary program quality.

More troublesome is the negative answer to RQ3 (*"Will this experience have long-lasting effects on students?"*). No student actually tried to use TFD after the experience, even when they they liked it and felt confident in its use. This indicates that a serious rethinking of the way we teach TFD is needed, noting that no established teaching methodology currently exists.

VII. Conclusions and future works

In this work, we presented our research on the exposure of young programmers to the TFD practice. Data collection covered 17 months and involved more than 100 high-school students. Results show that:

- TFD has limited, if any, positive effects on classic code quality metrics - possibly due to the limited complexity of the problem.
- TFD, if properly introduced, is perceived as a useful technique in producing less buggy software and an interesting development methodology, though its acceptance is strongly polarized.
- Its long-term effects are limited to superficial knowledge and a general impression of usefulness, but students never really use the technique again.

We conclude that TFD, while generally useful, requires considerable effort on the teacher part and substantial adaptation to be useful in a school context; this can also apply to undergraduates, since the testing conditions are very similar. Only so can it become an effective technique used with continuity, not just a one-shot school curiosity. In any case, "real" code refactoring is probably not useful in this context and its introduction should be postponed until students are able to design programs of significant complexity.

We are developing and testing a more systematic approach to TFD teaching in schools and universities, which includes Randori-like activities, PBL development, formal grading and feedback.

References

[1] B. 36, "Benefits of TDD." [Online]. Available: http://www.base36.com/2012/07/benefits-of-test-driven-development/

[2] L. Williams, E. M. Maximilien, and M. Vouk, "Test-driven development as a defect-reduction practice," in *Proceedings of the 14th International Symposium on Software Reliability Engineering*, ser. ISSRE '03, 2003.

[3] UK Government, "National Curriculum in England," 2013.

[4] M. M. Muller and O. Hagner, "Experiment about test-first programming," *IEE Proceedings-Software*, vol. 149, no. 5, pp. 131–136, 2002.

[5] R. Kaufmann and D. Janzen, "Implications of test-driven development: a pilot study," in *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM, 2003, pp. 298–299.

[6] N. Yahya and N. Bakar, "The analysis of programming competency in test driven development," in *9th Malaysian Software Engineering Conference (MySEC)*. IEEE, 2015, pp. 290–295.

[7] J. Bowyer and J. Hughes, "Assessing undergraduate experience of continuous integration and test-driven development," in *Proceedings of the 28th Int. Conf. on Software Engineering*. ACM, 2006, pp. 691–694.

[8] H. Erdogmus, M. Morisio, and M. Torchiano, "On the effectiveness of the test-first approach to programming," *IEEE Trans. Softw. Eng.*, vol. 31, no. 3, pp. 226–237, 2005.

[9] C. G. Jones, "Test-driven development goes to school," *Journal of Computing Sciences in Colleges*, vol. 20, no. 1, pp. 220–231, 2004.

[10] R. Mugridge, B. MacDonald, P. Roop, and E. Tempero, "Five challenges in teaching xp," in *Procs. Int. Conf. on Extreme Programming and Agile Processes in Software Engineering*. Springer, 2003, pp. 406–409.

[11] R. Mugridge, "Challenges in teaching test driven development," in *Procs. Int. Conf. on Extreme Programming and Agile Processes in Software Engineering*. Springer, 2003, pp. 410–413.

[12] D. Draheim, "Learning software engineering with ease," in *Informatics and the Digital Society*. Springer, 2003, pp. 119–128.

[13] P. Kastl and R. Romeike, "Towards agile practices in cs secondary education with a design based research approach," in *Proceedings of the 9th Workshop in Primary and Secondary Computing Education*. ACM, 2014, pp. 130–131.

[14] J. Creswell and V. Clark, *Designing and conducting Mixed Methods research*. Wiley Online Library, 2007.

[15] D. Russo, P. Ciancarini, T. Falasconi, and M. Tomasi, "Software quality concerns in the italian bank sector: the emergence of a meta-quality dimension," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17. IEEE, 2017.

[16] P. Ciancarini, D. Russo, A. Sillitti, and G. Succi, "Reverse engineering: a european ipr perspective," in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. ACM, 2016, pp. 1498–1503.

[17] ——, "A guided tour of the legal implications of software cloning," in *Proceedings of the 38th International Conference on Software Engineering Companion*, ser. ICSE '16. ACM, 2016, pp. 563–572.

[18] M. Missiroli, D. Russo, and P. Ciancarini, "Agile for millennials: a comparative study," in *Proceedings of the 1st International Workshop on Software Engineering Curricula for Millennials*, 2017.

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

[20] M. Missiroli, D. Russo, and P. Ciancarini, "Learning agile software development in high school: an investigation," in *Proceedings of the 38th International Conference on Software Engineering Companion*. ACM, 2016, pp. 293–302.

[21] ——, "Una didattica agile per la programmazione," *Mondo Digitale*, vol. 15, no. 64, 2016.

[22] E. Galliot, "Tennis game kata." [Online]. Available: http://codingdojo.org/cgi-bin/index.pl?KataTennis

[23] B. W. Boehm, J. R. Brown, and M. Lipow, "Quantitative evaluation of software quality," in *Proceedings of the 2nd international conference on Software Engineering*. IEEE Computer Society Press, 1976, pp. 592–605.

[24] J.-L. Letouzey, "The SQALE method for evaluating technical debt," in *Proceedings of the Third International Workshop on Managing Technical Debt*. IEEE Press, 2012, pp. 31–36.

[25] G. Campbell, Papapetrou, and P. P, *SonarQube in action*. Manning Publications Co., 2013.

[26] T. Bhat and N. Nagappan, "Evaluating the efficacy of test-driven development: Industrial case studies," in *Procs. ACM/IEEE Int. Symposium on Empirical Software Engineering*, ser. ISESE '06. ACM, 2006, pp. 356–363.