Design Thinking and Agile Practices for Software Engineering: An Opportunity for Innovation

2 authors:

Luis Corral
Tecnológico de Monterrey
42 PUBLICATIONS   **386** CITATIONS

Ilenia Fronza
Free University of Bozen-Bolzano
64 PUBLICATIONS   **389** CITATIONS

Some of the authors of this publication are also working on these related projects:

Project   COCONATS View project

Project   Edge Cloud Architecture View project

Design Thinking and Agile Practices for Software Engineering

An Opportunity for Innovation

Luis Corral
Monterrey Institute of Technology and Higher Education
Queretaro, Mexico
lrcorralv@itesm.mx

Ilenia Fronza
Free University of Bozen-Bolzano
Bolzano, Italy
ilenia.fronza@unibz.it

ABSTRACT

Commonly, the instruction of Software Engineering implements processes that are inherent to the theory and practice of software development. Traditional and Agile methods lay the foundation for building "functional software products" that meet the requirements of a system of a larger scope. However, if we consider software as a product that frequently has the mission of satisfying the needs of human users, we can go beyond the typical *"analysis - design - implementation - testing"* process, to reinterpret it with the *"empathize - define - ideate - prototype - testing"* proposed by Design Thinking, a development methodology commonly used in creative and innovative professional settings. In this work, we study the use of Design Thinking as a methodological approach for the instruction of Software Engineering at undergraduate level, in courses that have the particular aim of creating innovative software products from scratch. We describe the similarities and differences between Design Thinking and Software Development Processes, taking as instance Agile Practices. We compare evidence on methods and deliverables produced by students in their learning path using Agile Practices and Design Thinking in two different educational environments. Finally, we discuss coincidences, weaknesses, and opportunities to keep investigating in this topic as a research subject, toward finding practices to promote in students both creativity and technical discipline to develop innovative software solutions.

CCS CONCEPTS

• **Applied computing** → **Education**; • **Software and its engineering** → **Software creation and management**;

KEYWORDS

Agile; Creativity; Design Thinking; Education; Software Engineering.

1 INTRODUCTION

University instructors who teach Software Engineering promote knowledge and experiences that attempt to emulate, methodologically and practically, the common environment of the software industry. The selection of a procedural approach to teach Software Engineering is a considerable responsibility for the teaching staff, considering that the differences between software development processes can introduce important variations in the methods, interactions, intermediate products, and general practices to develop software, and in consequence, in the final product. The instruction of Software Engineering typically utilizes widespread development methodologies, for instance, *traditional* development processes (e.g., waterfall, V-model, Spiral, and others) or *Agile* development practices (e.g., Scrum, XP) [18]. Both traditional and Agile methods lay the foundation for building "functional software products" that meet the requirements of a system of a broader scope.

In this work, we discuss a methodological strategy that permits the incorporation of Design Thinking as a development process that is efficient to meet the Software Engineering principles taught at undergraduate level, and compare it with a similar work setting that uses instead Agile practices. As Design Thinking claims to be a methodology to create solutions with a human-centric approach [22], we based our strategy in the fact that software can be as well a product that has the mission of satisfying the needs of human users. In this way, we can go beyond the typical *"analysis - design - implementation - testing"* software process, re-interpreting it as the *"empathize - define - ideate - prototype"* proposed by Design Thinking.

The rest of the paper is organized as follows: Section 2 covers the state of the art and current status on methodological approaches to teach Software Engineering at undergraduate level. Section 3 discusses in detail Design Thinking and Agile practices, and maps activities and artefacts considered by them. Section 4 compares the experience in teaching Software Engineering courses in two educational settings, the first one applying principles of Agile development practices, and the second one using Design Thinking. Section 5 discusses coincidences, weakness and opportunities to study more in depth this topic as a research subject, and Section 6 closes this work drawing conclusions.

2 STATE OF THE ART

The educational setting has as an imperative to promote an environment in which students receive the principles of a discipline; however, the educational environment should provide as well a setting open enough to try, fail and adjust as necessary, to mentor students in their educational journey through mastering the subject at hand. When teaching Software Engineering, it is necessary to

choose a development model that is easy to understand, adopt and execute, while giving students theoretical foundations and practical experiences that emulate the common environment of the software industry [18]. Any model chosen by the teaching staff should include a set of activities applied in sequence during a typical software development cycle: requirements elicitation, analysis, design, implementation, test and delivery.

Software Engineering and Education have been the focus of extensive research, especially on the value that can be acquired in a classroom setting toward having professional skills that enable practitioners to develop successful products. In a seminal study, Shaw established challenges and aspirations for educators in Software Engineering, including, among others, fostering current practices in an ever-changing arena, and having a clear focus on practical skills [24]. Much research has been done in terms of understanding the current status of curriculum, practices and tools. We identified works that present a research outlook [14, 24], classroom experiences efficient in mapping software processes to course sessions and deliverables [3, 10], research that discusses practices, behaviours and interactions among students [9, 15], and opportunities to stop and look back at the practical experience teaching Software Engineering [4, 19].

Published experiences and studies include methodological strategies in which instructors and students follow a development process (traditional or Agile) that was designed solely for the purpose of creating a software product. Those methodologies are foundational to foster Software Engineering skills in students, thanks to their ability to elicit customer requirements, create designs that address them, and finally implement, test and deliver software. In the end, those practices can be associated with the skills that are needed in industry, academia and entrepreneurship to accomplish technically in the software development business.

Very little research is published on the adoption of non-software focused methodologies as an efficient attempt to create better software products. An example is the use of the statistical quality processes of Six Sigma as a metaphor for software development [25], the Lean methodology [23], or Design Thinking [16]. From those instances, the first two are inspired by manufacturing processes; thus, strong principles of those methodologies can be beneficial when looking for efficiency and productivity but may become a roadblock when promoting human-centricity, creativity and innovation in students and their final products.

With this vision at hand, we looked for opportunities of implementing Design Thinking as an alternative to Agile development practices in a Software Engineering course that interprets software as a product that may have the mission of satisfying the needs of human users.

3 AGILE PRACTICES AND DESIGN THINKING

To utilize Design Thinking as a *software development process*, first it is necessary to provide an analysis between this methodology and an actual software development process. In this way, it could be visible upfront if Design Thinking has the risk of falling short when used to manage a software development process. It is not difficult to find in scientific literature examples of comparisons between software development processes [6, 8]. Also, Design Thinking has

been compared with Lean Startup [20]; nonetheless, there is no evidence of published research with the focus of mapping or comparing Agile Practices and Design Thinking. With this goal, we implemented a process mapping between Design Thinking and Agile practices as an instance of a software development process. Agile practices are an approach for software development widely used in industry and academia. Its philosophy, practical approach and customer centricity suggests in principle a number of similarities with Design Thinking.

3.1 Agile Practices

Agile practices for software development aim to offer a lightweight, product-oriented solution in a fast changing environment. Agile practices have as high-level philosophy simplicity and speed, concentrating only on the functions needed, delivering them fast, collecting feedback and reacting rapidly to business and technology changes [1]. From that baseline, different well-known strategies to implement Agile practices have emerged: Scrum, eXtreme Programming, Lean-Startup, and others.

Agile practices are motivated by the product and driven by people: software development projects have a clear mission, but the specific requirements can be volatile due to uncertainty. That can be, in practice, faced by [11]:

- *Collaborating closely with customers* to understand their needs;
- *Constantly releasing software* of limited features, focusing on working products;
- *Developing adaptively*, being responsive to change; and
- *Walking an iterative process* that is repeated until accomplishing the final product release.

Agile practices, in summary, provide a framework for software development, standing out from traditional practices because of a dynamic strategy that involves requirements evolution, frequent face-to-face communication, collaboration; continuous delivery, customer interaction, and failure-resilient approach. The context of a team implementing Agile practices is typically small, within time and budget constraints, with processes that can be repetitive, adaptive, and minimally defined [12]. The context of the product developed in an Agile environment deals with the variety of target platforms, and with a traditional assumption a small-sized, non-critical end product [7], but it is important to note that Agile has proven its value also in contexts of large organizations and complex products [2, 17].

3.2 Design Thinking

The expression *Design Thinking* traces its roots to a book of Architecture and gained acceptance in other Design and artistic disciplines. The coined term inspired the creation of a "solution development framework", that has gone beyond the boundaries of its original domain, and is now positioned for implementation in a variety of disciplines. Even though there is no full consensus on a single, formal definition of what Design Thinking is, it can be understood as an approach to create solutions with a human-centric focus [22].

Much like Agile, Design Thinking is typically defined from an abstract, high-level perspective, but in practice, there are diverse

Table 1: Mapping between Agile Practices and Design Thinking.

Agile Practices	Design Thinking
Analysis: Lean management, system metaphor, user stories	*Empathize*: Market research, interviews, user persona, user journey
Design: Agile modeling, CRC Cards, story-driven development, customer on site	*Define*: Point-of-view statement, Analysis/synthesis, problem statement
-	*Ideate*: Brainstorming, point-of-view analysis, *How might we* templates
Implementation: Backlog management, Extreme Programming, Feature-driven development	*Prototype*: Storyboard, sketching, non-functional prototyping, functional prototyping
Testing: Test-driven development, continuous integration, sprint retrospective	*Testing*: Minimum viable product rollout, user feedback grid

strategies and flavours to implement it "hands on". The following are the stages generally accepted as a framework to Design Thinking [5], although specific steps and nomenclature may change depending on the source:

- *Inspiration:*
  - *Empathize*: understanding what the user or customer needs, in terms of what the customer feels.
  - *Define*: setting down the needs of the customer in terms of a problem that can be solved.
- *Ideation*: the proposal of a solution (product, service, experience) that meets the needs of the customer.
- *Implementation:*
  - *Prototype*: creating a solution that covers a selection of the features required by the user, so that the final customer can experience it.
  - *Test*: actual utilization of the solution, in a way that the customer can provide timely feedback and the process can iterate.

The keystone of Design Thinking is that products, services, and processes are complex, and people's ideas about those products are quite subject to changes. With this focus, Design Thinking proposes that through empathy, experimentation, prototyping and prompt feedback, designers are more prone to succeed when solving a need, simplifying and humanizing the solution [13].

Design Thinking thus can be flexible enough as to be taken as a general solution development framework that makes it suitable to be utilized in software development processes, which is indeed a field in which it is required to have an open, creative and flexible approach to solve problems and find opportunities for improvement.

3.3 Mapping Agile and Design Thinking

While Design Thinking and Agile Practices were first conceived for very distinct purposes, we can identify that both of them aim to deliver a fast solution in partial fulfilment of a particular need. Moreover, both of them propose a framework that can be associated to the general development process that, through many interpretations, is used by Software Engineering (that is, *analysis - design - implementation - testing*).

The difference relies on the practical approach that is taken when facing and addressing the issue of each phase, fostering fast

failure towards rapid success, which translates into simple deliverables of enhanced value. For instance, when *analyzing* a need, traditional Software Engineering will go for a long, comprehensive **requirements elicitation** effort, Agile Practitioners will construct a **system metaphor**, and Design Thinkers will develop **empathy** for the final user. These stages can be broken down in specific actions and deliverables for each one: requirements elicitation will conduct *customer interviews*, system metaphor will evolve into *user stories*, and empathy will be settled in concepts like *user persona* and *user journey*.

To provide a better understanding between high-level stages of the two approaches, and how they relate to software development, we defined a general mapping in Table 1. It is important to mention that given the breadth and complexity of each methodology, not all the practices may be included in the mapping, but only a representative sample of practices that relate to each development phase.

4 CLASSROOM EXPERIENCE: A COMPARATIVE VIEW

In this section, we explain separately the working setting, development and results yielded by two different classes, one implementing Agile practices, and the other implementing Design Thinking. Our comparison is made in two mandatory Software Engineering courses taught in an undergraduate degree in Computer Science. Depending on the plan of study of each participant, this course can be taken in the second or third year. In both cases, the following conditions are met:

- To enroll, participants have completed at least two courses of programming;
- The course is focused on the design and implementation of a software-driven system, and not in the instruction of a specific programming language;
- Based on the previous point, students may use the programming language of their choice;
- The educational goal of each course is to cover a software development process with a hands-on approach;
- Students team up in small groups (about three or four members);
- Teams are randomly sorted by instructors;
- Software projects are proposed by students;

Table 2: Mapping between sessions and activities of each course.

Week	Group 1 *(Agile Practices)*	Week	Group 2 *(Design Thinking)*
0	Introduction	0	Introduction
1	Teamwork fundamentals	1	Teamwork fundamentals
2	*Analysis*: system metaphor, user stories	2	*Empathizing*: Identification of a relevant need
-	-	3	*Empathizing*: Target user identification
-	-	4	*Empathizing*: user persona, user journeys
3	*Design*: Product outline, UML diagrams	5	*Definition*: High-level goals
-	-	6	*Definition*: Goal setting, system objectives
4	*Design*: Product outline, database modelling	7	*Ideate*: Brainstorming
5	*Implementation*: 1st iteration: code, document	8	*Ideate*: Short-list of solutions: First project pitch
6	*Implementation*: 2nd iteration: code, document	9	*Prototyping*: 1st week
7	*Implementation*: 3rd iteration: code, document	10	*Prototyping*: 2nd week
8	*Implementation*: 4th iteration: code, document	11	*Prototyping*: Second project pitch
9	*Implementation*: 5th iteration: code, document	12	*Prototyping*: 3rd week
10	*Implementation*: 6th iteration: code, document	13	*Prototyping*: 4th week
-	-	14	*Test*: Third project pitch
11	*Test*: thorough testing and documentation	15	*Test*: Project closure
12	Final evaluation: Functional product pitch	16	Final evaluation: Minimum viable product pitch

- Software teams are managed by students, assigning roles to themselves;
- As course outcome, students shall deliver a functional software solution, which is a must-have requisite to obtain a passing grade.

The two classes have the following characteristics. A first group was observed in a state-funded university in Italy. There, software projects are managed using Agile Practices. The second group was observed in a private university in Mexico. This group managed the software projects using Design Thinking. In both cases, the size of the class was around 20 students. The teaching staff was common in the two settings, that means that the instructors that planned, taught and evaluated the course were the same in both working settings. In addition to their capacity of instructors, the teaching staff took as well the role of project customers. For team management, in the two settings teams were self-managed and were required to have a consistent communication pace with course instructors. For team management and to assure a disciplined means of collaboration, it was fostered the team management practices recommended in [21].

4.1 First Setting: Software Engineering with Agile Practices

The first observed group consisted of a class of second-year undergraduates, majoring in Computer Science at an Italian university. The class was formed by 21 students, coming from 5 different countries. The group was organized in teams: five teams formed by three students, and three teams were formed by two students. The course is organized in 12 weeks. Each week, a total of 6 hours are dedicated to the course: four hours of lectures, and two hours of laboratory work. To follow Agile Practices, the class implemented Scrum to manage workload and progress, Kanban boards to distribute work, and other Agile deliverables. The breakdown of the sessions is shown in Table 2.

Analysis: In this stage, teams are asked to think about a potential software product, and to consider its feasibility. The software product can be a new idea or an improvement on an existing idea. In the latter case, teams have to specify how the idea is better than the existing one. Once the idea has been found, teams need to provide a description of how the system works (*system metaphor*), and short, simple descriptions of each feature (*user stories*). Those products are the main deliverable of this phase.

Design: Teams adopt a CRC (class responsibility collaborator) modelling process for identifying user requirements. To create *CRC cards*, teams begin by writing out a scenario which identifies the major actors (classes) and their actions (responsibilities). CRC cards provide a direct way to translate high-level abstractions into clear designs that can be eventually coded. This said, the set of CRC cards is the key deliverable of this stage.

Implementation: Teams work to deliver working software based on six iterations (*sprints*) distributed in a time span of six weeks. The goal of each sprint is to construct a *working product* to be shown by the end of the sprint. Features are built, tested, and demonstrated to the teaching staff. With the working product and the feedback from the instructors, new requirements are gathered. Customers (i.e., instructors) feedback is used to adjust the plan for the forthcoming iteration.

Test: The testing effort is conducted in parallel with the implementation phase, and is documented every sprint. Teams are required to complete thorough testing and documentation of results before final delivery. The final evaluation includes a *poster presentation* and a *system live demo* in front of a panel of experts invited by the teaching staff.

The result of this coursework was the delivery of eight functional software systems. Examples of the delivered systems included (1) a system to run sentiment analysis for social media, (2) an automation system for library management, (3) an application to ease garbage

separation, (4) a system to manage the inventory of a vending machine, and other transactional systems.

The open opportunity to choose a project led the class to select ideas from a vast range of applications and domains. There is no clear convergence in the type and style of projects delivered by this class, yet all the delivered projects aim to serve a human user. Also, it is important to say that at least four projects followed the traditional "add - delete - modify" structure of a software system. Moreover, the systems were delivered together with a documentation package, that described the high-level goal of the system, architecture, designs, and deliverables of each phase (metaphor, user stories, CRC cards, etc.).

4.2 Second Setting: Software Engineering with Design Thinking

The second observed group consisted of a class of 17 third-year undergraduates coming from three different countries, majoring in Computer Science in a major private university in Mexico. The group was organized in five teams of three members, and two teams formed by four students. The course is organized in 16 weeks. Each week, a total of 4 hours are dedicated to the course, typically divided into two hours of lectures, and two hours of laboratory work. The breakdown of the sessions is shown in Table 2.

Empathize: Teams are first asked to find a need that can be satisfied through a software system. The teaching staff suggests that teams ask classmates, fellow students and friends what are common needs they have that can be solved through a software product. With a *need* clearly identified, teams are requested to identify the target group the product is directed to, and to create an archetypical definition of who is the person who faces the problem (*user persona*) and the situation she commonly finds the problem (*user journey*). The definition of the need, user persona and user journey are the required deliverables of this phase.

Define: Once the need and the target group are identified, the next step is to explore more through interviews to outline a better definition of the situation they want to solve. This is then set down in the *high-level objective* of the software system, as well as low-level goals. Goals and objectives are typically written in the form of *user story-inspired sentences*, which make it easier for students to map out the user's need with concrete actions. The high-level goal and user stories are mandatory deliverables.

Ideate: The creative and innovative journey is ramped up in this phase, with the urgency that teams not only identify a real-world problem, but also deliver a solution that satisfies the needs of the persona. Teams brainstorm solutions, verify feasibility, compare technologies and submit *an idea to develop the rest of the course*. In this phase, it was observed that after exercising the definition of the problem and brainstorming solutions, a team might pivot the original idea into a different one. For example, a team that wanted to implement an application for home security of doors and windows, pivoted into a safety system to prevent toddlers from opening or closing doors and drawers in a kitchen. The idea of the solution is the key deliverable of this phase.

Prototype: For the actual elaboration of a *working solution*, teams are free to use the technology of their choice, which leads to a variety of programming languages and a very diverse technology

stack. Students also go beyond the boundaries of software to extend their solutions to basic hardware components like sensors or simple actuation systems operated by a Raspberry Pi. As the main deliverable of this phase, and for the project as a whole, a working product shall be demonstrated by the end of the course.

Test: It is required that teams pitch three project presentations through the course. Those presentations have the aim to (1) strategize the original idea, (2) check progress and announce relevant changes, and (3) ensure that the product satisfies the needs stated by the beginning of the course. The final evaluation includes an *executive presentation* and a *system live demonstration* in front of a panel of experts invited from industry, that provide comments on several fronts like technical, creativity, business and entrepreneurship.

The result of this coursework was the delivery of software systems that responded to needs of a person that can be identified as a student of the institute where this course took place. Examples of the delivered systems are: (1) a system to implement a rear/reverse camera in cars that are not equipped with a built-in one, (2) a system to open the door in an apartment when somebody in a list of party invitees "rings" the doorbell using a counterpart application, (3) a system to lock and unlock a padlock to release the bicycles to move across the campus. In those cases, the archetypical user is always a young student, much like the class was able to empathize among themselves, with a specific need that can be solved through a mobile application or gadget.

5 DISCUSSION

The two courses were completed as planned and scheduled within the time span given for the course. No major incidences (for instance, course drop-outs) were identified. In both working settings, every team was able to deliver a working software solution. However, several commonalities and differences were identified between the two working settings. To discuss them in detail, we collected a list of general traits:

- Both groups directed their efforts to the development of human-centric software;
- The group implementing Agile was more effective leveraging iterations to improve the quality of the final product, while the Design Thinking group concentrated on the construction of the solution, reducing the scope of the final product to a minimum viable product;
- The Agile group implemented a software-driven delivery pace that included a number of intermediate products that were used to build documentation. The Design Thinking group did not deliver documentation;
- The Agile group developed a series of divergent solutions whose value can be challenged by multiple segments of users. Design Thinking groups converged naturally to implement solutions more relevant to the needs of their own environment and age;
- Regardless of the methodology used, teams were able to manage their work independently with minimum intervention of the teaching staff;
- Projects developed under the Agile methodology can be regarded as software-centric (that is, *functional software for the sake of high quality software*), while Design Thinking

teams delivered human-centric solutions (that is, *functional software for the sake of solving a problem*, yet prototypes could be perfectible).
- Both groups were able to provide a professional pitch to an independent panel of experts to discuss and provide clear arguments to sustain their ideas;

A common trend of the two courses is the identification of opportunities of software systems that solve the needs of a human user. On the one hand, the group implementing Agile gave preference to the software development rigor, meeting the educational needs of a Software Engineering course. On the other hand, the class using Design Thinking used a methodology of creativity and innovation to supply a software solution. While the teams met the expectations of the course by delivering a minimum viable product, aspects of Software Engineering discipline like software design or documentation are not considered.

Through the observation of the two classes, and with the certainty that in both working settings students were able to deliver functional solutions, we can conclude that both methodologies deliver value in different aspects. The Agile methodology can be suggested in a fast-paced, software-driven development project, while Design Thinking could be preferred in more uncertain, innovative projects.

A limitation of this study is that, even though it is acknowledged that the course goals and the teaching staff were common in the two observed settings, the methodologies were taught at two different Universities in two different countries. Further research may identify whether the differences in the results are attributable to those variables in addition to the traits of the methodologies taught.

Based on this discussion, we recommend that additional research needs to be done to collect additional evidence, with a particular focus on the quality and effectiveness of the products developed (for instance implementing source code analysis). Also, it is advisable to repeat the observation in further terms, for repeatability and reproducibility reasons, identify trends or divergences between similar cases (for instance, asking the two independent classes to develop the same project, each class following a different methodology).

6 CONCLUSIONS

In this paper, we presented an initial study to map out two distinct methodologies, Agile and Design Thinking, as convergent methodological approaches for the instruction of Software Engineering at undergraduate level. We compared evidence on methods, artefacts, and final products delivered by students in their learning path using separately Agile and Design Thinking in two different and independent educational environments.

In the two working settings, teams were successful implementing software systems under two different methodologies, but it was as well observed that the utilization of each discipline introduces separate traits in the way the software is developed, and the kind of product delivered by the teams. Additional research shall confirm if the different traits delivered by the two separate groups is indeed a direct influence of each methodology, as well as isolate other sources of variation that can be attributable to the two different working environments.

Both methodologies proved to be effective to manage software projects, and both deliver value in different aspects. Agile was observed as more rigorous in strict Software Engineering aspects, while Design Thinking led the outcome to more innovative, outside-the-box products. Further research shall be conducted to collect additional evidence, with a particular focus on the quality, effectiveness and satisfaction of the products developed under the two methodologies.

REFERENCES

[1] Pekka Abrahamsson, Juhani Warsta, Mikko T. Siponen, and Jussi Ronkainen. 2003. New Directions on Agile Methods: A Comparative Analysis. In *Proc. of the 25th Int. Conf. on Software Engineering (ICSE '03)*. IEEE, 244–254.
[2] Julian M Bass. 2015. How product owner teams scale agile methods to large distributed enterprises. *Empirical Software Engineering* 20, 6 (2015), 1525–1557.
[3] M Brian Blake. 2003. A student-enacted simulation approach to software engineering education. *IEEE Trans. on Education* 46, 1 (2003), 124–132.
[4] Narasimha Bolloju and Sujit Kumar Chakrabarti. 2017. Designing Software Engineering Courses for Effective Teaching and Learning. In *Proc. of the 10th Innovations in Software Engineering Conf. (ISEC'17)*. ACM, 220–220.
[5] Tim Brown and Jocelyn Wyatt. 2010. Design thinking for social innovation. *Development Outreach* 12, 1 (2010), 29–43.
[6] Vishal Chandra. 2015. Comparison between Various Software Development Methodologies. *Int. J. of Computer Applications* 131, 9 (2015), 7–10.
[7] Luis Corral, Alberto Sillitti, and Giancarlo Succi. 2015. Software assurance practices for mobile applications. *Computing* 97, 10 (2015), 1001–1022.
[8] A. M. Davis, E. H. Bersoff, and E. R. Comer. 1988. A strategy for comparing alternative software development life cycle models. *IEEE Trans. on Software Engineering* 14, 10 (Oct 1988), 1453–1461.
[9] Ilenia Fronza, Nabil El Ioini, Andrea Janes, Alberto Sillitti, Giancarlo Succi, and Luis Corral. 2014. If i had to vote on this laboratory, i would give nine: Introduction on computational thinking in the lower secondary school: Results of the experience. *Mondo Digitale* 13, 51 (2014), 757–765.
[10] Ilenia Fronza, Nabil El Ioini, and Luis Corral. 2017. Teaching computational thinking using agile software engineering methods: a framework for middle schools. *ACM Transactions on Computing Education (TOCE)* 17, 4 (2017), 19.
[11] Jim Highsmith and Alistair Cockburn. 2001. Agile software development: The business of innovation. *Computer* 34, 9 (2001), 120–127.
[12] Samireh Jalali and Claes Wohlin. 2012. Global software engineering and agile practices: a systematic review. *J. of software: Evolution and Process* 24, 6 (2012), 643–659.
[13] Jon Kolko. 2015. Design thinking comes of age. *Harvard Business Review* 93, 9 (2015), 66–71.
[14] T. C. Lethbridge, J. Diaz-Herrera, R. J. J. LeBlanc, and J. B. Thompson. 2007. Improving software practice through education: Challenges and future trends. In *Future of Software Engineering, 2007. FOSE '07*. 12–28.
[15] Janet Liebenberg, Magda Huisman, and Elsa Mentz. 2015. The relevance of software development education for students. *IEEE Trans. on Education* 58, 4 (2015), 242–248.
[16] Tilmann Lindberg, Christoph Meinel, and Ralf Wagner. 2011. Design thinking: A fruitful concept for it development? In *Design thinking*. Springer, 3–18.
[17] Mikael Lindvall, Dirk Muthig, Aldo Dagnino, Christina Wallin, Michael Stupperich, David Kiefer, John May, and Tuomo Kahkonen. 2004. Agile software development in large organizations. *Computer* 37, 12 (2004), 26–34.
[18] Maíra R Marques, Alcides Quispe, and Sergio F Ochoa. 2014. A systematic mapping study on practical approaches to teaching software engineering. In *IEEE Front. in Education Conf.* 1–8.
[19] Nancy R. Mead. 2009. Software engineering education: How far we've come and how far we have to go. *J. of Systems and Software* 82, 4 (2009), 571 – 575.
[20] Roland M Müller and Katja Thoring. 2012. Design thinking vs. lean startup: A comparison of two user-driven innovation strategies. *Leading through design* 151 (2012).
[21] Barbara Oakley. 2002. It Takes Two to Tango: How good students enable problematic behaviors in teams. *J. of Student Centered Learning* 1, 1 (2002), 19–28.
[22] Hasso Plattner, Christoph Meinel, and Larry Leifer. 2010. *Design thinking: understand – improve – apply*. Springer.
[23] Mary Poppendieck and Michael A Cusumano. 2012. Lean software development: a tutorial. *IEEE software* 29, 5 (2012), 26–32.
[24] Mary Shaw. 2000. Software engineering education: a roadmap. In *Proc. of the Conf. on the Future of Software Engineering*. ACM, 371–380.
[25] Christine B Tayntor. 2007. *Six Sigma software development*. Crc Press.