Learning Agile Software Engineering Practices using Coding Dojo

Kenny Heinonen, Kasper Hirvikoski, Matti Luukkainen, Arto Vihavainen
University of Helsinki
Department of Computer Science
P.O. Box 68 (Gustaf Hällströmin katu 2b)
Fi-00014 University of Helsinki
{ kennyhei, khirviko, mluukkai, avihavai }@cs.helsinki.fi

Originally appeared as: Kenny Heinonen, Kasper Hirvikoski, Matti Luukkainen, and Arto Vihavainen. 2013. Learning agile software engineering practices using coding dojo. In Proceedings of the 13th annual ACM SIGITE conference on Information technology education (SIGITE '13). ACM, New York, NY, USA, 97-102.

Abstract

Information technology and computer science educators are experiencing an industry-driven change from plan-based software engineering development processes to more people-oriented Agile software engineering approaches. While plan-based software engineering practices have traditionally been taught in lectures, Agile practices can often be best learned by experiencing them in a realistic situation. One approach for bringing Agile practices to the learning community is a coding dojo, where a group of participants solve a programming task together using test-driven development and pair programming. Coding dojo is a form of learning which values concrete experience in a realistic context. In our experiment, we embedded a coding dojo into the Agile practices part of our undergraduate software engineering course. The participating students considered the coding dojo a useful experience, and most of them (82%) would recommend participation in coding dojos for their fellow students, as well.

Categories and Subject Descriptors
K.3.2 [Computers and Education]: Computer and Information Science Education *Computer Science Education*

General Terms
Experimentation

Keywords
coding dojo, software engineering, agile methodologies

1 Introduction

The industry-driven movement towards the use of Agile software engineering practices has caused a stir in higher education. Instructors in software development methodologies need to find a balance between conveying an understanding of the history, traditional plan-based methodologies [12], and giving enough time to cover Agile methodologies. One of the challenges in introducing Agile methodologies is that higher education is traditionally seen as lecture- and self-study-based, where practical experience plays only a small role, while Agile methodologies are often best learned by truly experiencing them.

For example, the so-called *inspect and adapt* effort, where a team continuously monitors their working processes and adapts them in order to become more efficient, requires deep understanding and practical experience of both the team- and process-specific aspects, as well as an understanding of the problem domain, in which the team acts.

Agile methodologies are a collection of best practices, which are often combined as a development team adapts their practices during a task. For example, the Scrum framework [14] and Kanban method [1] are both often used in combination with best practices from Extreme Programming (XP) [4] such as Pair Programming and Test-Driven Development (TDD) [3].

Teaching Pair programming and TDD in a higher education institution is challenging. Knowing in principle what they mean has only little value since just having abstract, theoretical knowledge about a practice does not imply a deep understanding nor a change in mental processes [5], which should be the purpose of higher education.

We have observed that by asking students to practice pair programming and TDD on their own very rarely leads to success. Especially when students are novice programmers, starting by writing a test first seems to be extremely hard, and proceeding in the small steps that the TDD cycle requires often demands explicit guidance. In order for the instruction focused on Agile practices to have an impact on students' behavior, each student should have the opportunity to truly experience pair programming and TDD in a guided setting. Essentially, an experienced instructor, who provides scaffolding, should be present when pair programming and TDD are practiced by students for the first time.

The work presented in this paper is about how a *coding dojo* [13], a safe learning environment for practising various programming-related skills, can be utilised to convey Agile software engineering best practices as part of a course, and how a dojo is viewed from a student's perspective.

This article is structured as follows. In sections 2 and 3, we explain the pedagogical framework in which we base our instruction, and further detail test-driven development and pair programming. Section 4 discusses various types of coding dojos, and outlines some of the challenges in organizing a dojo in a higher education setting. In section 5, the dojo experiment in our Software Engineering course is explained, and in section 6, the outcome of the dojo experiment is analyzed. Finally, we conclude the experiment, and discuss our ongoing work.

2 Pedagogical Framework

The utilization and creation of a realistic environment for learning Agile software engineering practices is backed up by the theory of situated cognition [5], which emphasises that knowing and doing should not be separated. Having a classical classroom setting, where students listen to a teacher discussing and perhaps demonstrating a technique, may lead to theoretical, abstract knowledge on a topic, but not to deeper understanding that could be easily transferable into practice.

In order for learning of a skill or knowledge to be effective, it should happen in an authentic application context and culture with real tools, where both knowing and doing can be combined. As context, culture and the defined activity all play a part in the learning process, some parts of learning can also be unintentional or unanticipated from the facilitator's point of view. This can be seen in students acquiring potentially harmful working habits, if the situation, in which the students study, does not prevent them.

One way to *situate* the learning in an appropriate context is to apply apprenticeship learning such as cognitive apprenticeship [7]. In cognitive apprenticeship a learner starts by first watching a master, i.e. an instructor, engage a problem. An essential part of the process is that the master explicitly speaks out the mental thought process behind his working process, thus giving a model of the task for the learner. This makes it possible for the learner to start understanding the actual problem-solving process and the tasks that the master performs.

Once the learner has a grasp on the task, she can start performing parts of the task by recalling how the master previously worked. Simply engaging in a problem is not optimal as, despite given a model, the student's behavior is heavily influenced by her confidence on her ability to perform a task [2]. Therefore, at the start, the master scaffolds the student by helping her forward in the task by giving temporary help, ensuring that the student is able to proceed and reaches an optimal working process.

It is important that the student does not become dependent on the scaffolding, and as soon as the student is able to proceed on her own, the scaffolding is dismantled. As the learner proceeds with taking on larger and larger tasks, she also discusses the learning process with the master, i.e. articulates the problem and the domain that they both are engaged in.

3 Test-Driven Development and Pair Programming

Test-driven development (TDD) [3] is a discipline that originates from Extreme Programming [4]. In TDD, the usual software development cycle that starts with design, continues with programming, and ends up in testing, is reversed. The developer starts a programming task by writing a test that specifies a small part of the intended behavior of a component under construction. After the test

is done, the developer proceeds to write the code, which makes the test pass. Once the test passes, the developer continues by writing another test, and again the code that makes the test pass. The intention is that the developer does not write code "just in case some feature would be needed in the future", but only if a previously written test requires a feature to be implemented. Writing code only when a test enforces it has a tendency to lead to a codebase that consists of simple and decoupled objects, which have clearly defined interfaces. All of these attributes are usually connected to the extendibility and maintainability of written code [10].

After following the TDD cycle for a while, the developer may notice that the structure of the written code is not optimal, e.g. it may contain redundant parts or lack a needed abstraction. In such a case, the developer must refactor [9] the code to keep its internal quality at a good level. Since refactoring is supposed to be done whenever needed during the TDD-based development, the design of the program evolves as the codebase grows.

TDD is complemented by pair programming [4], where two persons use a single workstation and act together to pursue a goal. The person who possesses the keyboard is often called the *driver* while the other person is called the *navigator*. The driver advances the task by writing the actual code, while the navigator constantly inspects the code that the driver has written and thinks about the problem from a different perspective, perhaps already planning the next step the pair is supposed to take. The driver and the navigator continuously engage in a discussion, where they articulate the next steps and design decisions. Every now and then the persons change roles.

Although pair programming may at first seem overly resource consumptive, it has many benefits. One of the most important benefits is the constant knowledge sharing within the team of programmers that uses pair programming to achieve their goals [6]. In addition, it has been noticed that pair programming produces code that has less defects, and has a more maintainable design than code developed by single developers [6]. Pair programming also improves the discipline of the developers: they become less likely to skip testing, and tend to use their time more efficiently instead of checking email and browsing the web.

Both TDD and test-driven development can be efficiently experienced in a coding dojo, which is a meeting for people willing to practice their coding practices.

4 Coding Dojo

Coding dojo is a meeting where a small group of participants gathers together to practice programming [13]. Its main goal is to act as a safe and non-competitive environment, in which everyone is allowed to learn and make mistakes and share their knowledge so that everyone can improve and become better at their craft. Coding dojos are typically organised by a *dojo master*, whose task is to facilitate the space and time, while the participants engage in solving a problem. Often at least part of the participants acts as an audience, whose task is to monitor

the working process of other participants and give feedback and improvement suggestions.

The problem that the participants engage in is usually not too complex and can even be previously unknown to the participants. This way the participants are able to focus more on honing their programming process, and not spend too much time on planning a solution. Another motivation for not giving out too challenging problems is that each group typically contains both novices and experts. However, if the main focus of the session is to practice problem-solving, more challenging problems may be appropriate.

Various types of coding dojo exist. One of the variants is the *randori kata* dojo, in which the participants solve a problem using pair programming and possibly TDD. In randori kata, one of the participants acts as the driver, and one as the navigator. The rest of the participants act as a silent audience, who observe how the driver and the navigator work together towards a solution. After a specific time-interval or via some other mechanism, the driver moves to the audience, the navigator starts to work as the driver, and a member of the audience moves in to act as the navigator.

The driver and the navigator work towards the solution by communicating with each other, making sure they both know what to do next. Although the observers start forming opinions and suggestions about the progress, they must remain quiet. A possible exception is a situation where the driver and the navigator are completely stuck, and indicate that they need help. Naturally, during the switch of navigator and driver, the new navigator can discuss with the previous navigator, who assumes the position of the driver.

In a version where TDD is practiced, the audience can also voice out their thoughts when all tests pass, or if a timeout is deliberately called. During a timeout, the participants revisit their approach and discuss possible alternative strategies.

Another variant of the coding dojo is the *prepared kata* dojo [13], where one or more of the participants have solved a problem previously, and solve it again in the dojo. The rest of the participants review the working process and the solution, and ask questions whenever they have one or do not understand why something was done. The goal is that the audience understands the solution so that they could later attempt to perform it by themselves.

In a coding dojo, the participants are engaged in learning in several ways; the participants can observe and analyze as others work towards a solution, and ask questions or give suggestions. As a participant works towards a solution either as a driver or a navigator, she receives valuable feedback from her pair as well as others, which helps her to reflect on and improve her own working practices.

4.1 Coding Dojos and Higher Education

Coding dojos have been previously organized in a higher education settings as separate sessions, not belonging to any formal courses (see e.g. [13, 16, 8]). Most of the dojos held in universities have been facilitated by outside experts or the

students themselves, thus being mostly of self-organizing in nature. In contrast to the previous dojos run in a university setting, our approach was to make dojo part of a mandatory course.

Organizing a coding dojo in a university setting has its challenges. If a coding dojo is embedded within a single course, the availability of the activity is constrained by the course duration. On the other hand, if a dojo is created as an extracurricular activity that does not bring study credits or affect course grades, students may choose not to participate at all, as they often prefer activities that affect course outcomes.

When comparing coding dojos to exercise labs, where students work on problems under the guidance of an instructor, coding dojos have both advantages and disadvantages. Dojos create an environment, where training and learning agile practices such as TDD and pair programming, which are difficult to practice on your own, can be practiced more easily.

The benefits are not only limited to learning technical skills, but also include practicing highly valued soft skills [11] such as communication. As students work towards a solution, they articulate both the process and the problem, and simultaneously practice voicing out improvements on both domain-specific problem constructs as well as more generic working processes.

Facilitating a coding dojo in University setting can be easier than facilitating a free-for-all coding dojo, as skill-level, tools, and the social context are typically known already beforehand at University.

The disadvantages and challenges of coding dojos are often related to students' self-confidence, values and communication. Students may feel insecure about their own performance, which may make the meeting an uncomfortable situation. Another problem is that some students value the velocity at which a problem is solved more than code quality and maintainability. The more participants there are, the more opinions there are, which makes it harder to form a consensus on what should be done next and how.

In the worst case scenario, some participants take the meeting too seriously, do not participate properly in a bi-directional communication, and get angry at participants who aren't productive enough, while erasing parts of the previous solution as they act as the driver, forcing their own solutions on others. This can typically be prevented by having a dojo master, i.e. the teacher acting as dojo facilitator present, whose task is to remind the participants that mistakes are supposed to happen and that everyone should take it easy. In addition, the dojo master can limit the number of participants to a relatively small number, e.g. between 4 and 6.

However, limiting the group sizes to a small number brings additional demands on the facilities in which the dojos are held, as well as increases the amount of time required from the available instructors.

5 Coding Dojos at our University

In our experiment, a randori dojo session was added to our 4th semester undergraduate software engineering course. The goal of the session was to have students practice how TDD and pair programming work in a real-life-like setting, and not just in theory, and also give them the understanding needed to organize coding dojos themselves in the future. Each session lasted two hours, and included roughly 1.5 hours of programming, and a 10 to 20 minute retrospective, during which the participants discussed the overall experience and setting, and considered possible future improvements. Each participant in the course took part in at least one dojo session.

5.1 Test Dojo

Experimenting with adding the dojo content to the course started with a 2-hour *test dojo*, which was held for the course staff and a few voluntary students. In the dojo, the participants worked through a problem, while switching roles to experience all the sides in a dojo.

The test dojo was an invaluable way of discovering some of the pitfalls and difficulties associated with dojos, such as having participants compete in the situation or start dominating the solution. It was also an integral part of the learning process for the course staff so that they could understand and experience what it is like to participate in a dojo, and how a person taking the role of dojo master should behave.

In addition, it was valuable to notice that some of the course staff were in a "lecturing mode", even while acting as dojo masters; they had the feeling that they should be instructing the participants how to proceed. However, as the goal of the dojo is to allow the participants to solve the problem, it was emphasized that the interventions should be kept to a minimum. When advice is needed, instead of giving direct advice, instructors should mostly restrain themselves to posing suitable questions that stimulate the students so that they can come to the right conclusions.

During the test dojos, it was also realised that the dojo masters would need to minimise the stressfulness and competitiveness of the dojos, especially when they would be held for the students.

5.2 Dojo Implementation

Each dojo had 4 to 6 students and a member of course staff as dojo master. The sessions were held in a small room equipped with a laptop and a projector, and the student groups were filled based on the enrollment order. This led to a situation, where participants had varying skill sets and knowledge about programming, testing and TDD. The goal was to make the dojos a fun and relaxed, but educational event, where the students would learn practices that they could also apply in the future.

The coding dojos lasted for 2 hours, applied a 5-minute cycle duration, and the programming task that students were supposed to solve was unveiled step by step by the dojo master. The dojo master emphasized that the goal was to implement the given features in the simplest possible way: this was also to force the need of refactoring the code as the task evolved more complex.

The reason for not letting participants know the full scope of the project already at the start was to simulate the iterative nature of an agile software project where the customer's opinions may evolve during the process. This also gave a concrete example of how the design of the software evolves during an iterative process.

During the coding dojos, the students worked towards building a student registry. The task was designed so that there were easier moments as well as refactoring. The outline of the programming task that the students were working on was as follows:

Task: Student Registry

1. A user is able to list students in a register
2. A user can add a course mark to a student
   (a) A student is identified by her name
   (b) A course is identified by its name
   (c) The course mark includes a grade
3. A user can see the number of students that have passed a course
4. The grade for a course is an integer between 0 and 5, and invalid marks should be ignored
5. A user can see the average grades for a given course
6. A user can see the courses that a student has taken
   (a) For each course, only the best grade is shown
7. A user can see the amount of courses that a student has completed
8. A user can see the grade average for a given student
9. As a course mark is added to a student, an old course mark should be replaced if the new course mark is better for the specific course and student
10. When adding a course mark, the number of credits for the course should also be included
11. A user can see the list of courses that a student has taken
12. A user can see the list of students that have taken a course

The system was developed as a console application, deliberately moving the focus away from UI design towards the internal structure of the application. As the "You aren't gonna need it"-principle was enforced throughout the activity, students were essentially required to start the task by implementing a class that contains a collection of strings; doing the simplest thing that could possibly work. Once the class was implemented, the students proceeded with creating a functionality for adding a course mark. All activities were performed by strictly following TDD, and refactoring was encouraged whenever necessary.

Typically, during the two-hour session, the students were able to finish most of the task. While there was no "perfect solution" during the timeframe, future improvements were also discussed after concluding a session.

6 Analysis

A total of 73 students attended 15 sessions, where they were guided and monitored by the instructors. In addition to facilitating the dojo, instructors also observed outbursts, comments, and working practices, which were later collected to form a coherent view of the students' experiences. In addition, the participants were asked to fill in a survey that studied the applicability of a coding dojo to a higher education context.

6.1 Instructors' Analysis

Although the initial expectation was that the least skillful individuals could feel uncomfortable with the practice, even more talented participants who had previous experience in TDD felt the situation was unnatural at first. As the driver was put into the spotlight, the expectations for her success were high. At first, this caused momentary freezes for some of the drivers, and the navigators on the other hand did not always have a good way of helping the driver.

Once the driver and the navigator got used to their roles and got more familiar with the process, they began to proceed at a more rapid pace. In some dojo sessions this led to the discovery of another pitfall. Some sessions started to see a competitive nature between individuals who were acting as an audience, and those who were acting as the driver and the navigator. Although most of the participants knew each other, some didn't find this competitive situation pleasant or helpful for their coding flow.

As the competitiveness was already experienced in the test dojo, dojo masters were quick to react to the situations, and direct the process back on track.

There was no single typical session. As the proficiency of the students varied considerably, and the students themselves were the primary actors in the sessions, the content and direction was heavily influenced by the skill levels. Depending on the skill level of students, the amount of interaction also varied. In sessions, where most of the students were less proficient in programming, students rarely voiced out their concerns about a specific issue, while in sessions, that had students with existing programming background, e.g. hobby projects

Table 1: Likert-scale questions of the survey on student perspectives on coding dojo, and the distributions for the student answers.

Question	% s. disagree	% disagree	% agree	% s. agree
atmosphere and usefulness				
The atmosphere in the dojo was relaxed	0%	8%	42%	42%
The atmosphere in the dojo was competitive	36%	36%	2%	2%
Usefulness of good programming practices became evident in the Dojo	2%	18%	46%	10%
Dojo is useful for improving your own programming practices	4%	8%	46%	14%
learning				
I experienced "Aha!" moments in the Dojo	4%	14%	38%	8%
I learned new programming practices in the Dojo	6%	22%	48%	12%
I learned more about how a group works in the Dojo	2%	6%	52%	18%
participation				
I would encourage a fellow student to participate in a coding dojo	2%	0%	54%	28%
I would participate in coding dojos in the future	4%	4%	48%	22%

or experience from the industry, the students were more active in pointing out improvements and sharing their practices.

Most of the students considered the 5-minute cycle quite short. Sometimes they were spent mostly in understand what the previous pairs had accomplished. Once the tasks got more complex and refactoring was necessary, an intervention was often needed from the dojo master, who advised the pairs to stop tinkering and start to consider and sketch out what changes were needed for the architecture.

6.2 Survey

The survey that the students were asked to fill after the coding dojo contained questions on programming-related background, Likert-scale questions related to the dojo atmosphere, usefulness, learning, and whether the students would suggest participation to their fellow students. In addition, the students were asked to fill in open-ended text questions, where they could assess their learning during the dojo session, and give open feedback and improvement suggestions.

The scale of the Likert-scale questions ranged from *"1 = strongly disagree"* to *"5 = strongly agree"*. No forced choice method was applied, and the participants also had the option of choosing *"3 = neither agree nor disagree"*.

A total of 50 participants answered the survey (68.5% of the participants), and the answers are summarized in Table 1, from which we have left out the option *"3 = neither agree nor disagree"*.

The results of the survey are displayed in Table 1. Most of the students felt that the dojo situation was not competitive, and that the atmosphere was relaxed. A majority of the students felt that the usefulness of good programming became evident in the dojo, and that a dojo is useful for improving their own programming practices.

Almost one half of the students experienced "Aha!" moments in the dojo,

and most of them learned new programming practices and more about how a group works. Over 80% of the students would encourage a fellow student to participate in a dojo, and 70% of them would participate in future coding dojos.

Upon reviewing the textual feedback provided to the open-ended questions, many of the students considered a dojo *"A welcome addition to the teaching practices at the department"*. The most notable objective was that the students felt that they had achieved practical understanding of TDD and pair programming, in addition to avoiding over-engineering of a solution.

Many of the students would have liked to have a clear, textual representation of the problem already at the beginning of the session. We had chosen to reveal the problem step-by-step, which reduces the possibility of over-engineering, forces students to work in small steps and also gave the students a concrete opportunity to experience what the incremental design [15] of software feels like.

Although there were always observers present, the students felt that the situation was relatively relaxing. One of the contributing factors is the relatively fast cycle (5 minutes), which added to the feeling of community, and which some, however, did criticize as being too fast.

Many of the textual comments were also surprising. As the experiment was designed, the goal was to emphasize agile software engineering practices to the students. However, for some students, the dojo introduced them to *"life-improving shortcuts and key-combinations"*, and for some the dojo was the first time they had participated in pair or group programming, and they received positive experiences. As one of the participants put it:

> *This was the first time that I participated in any sort of "group programming". The experience definitely lowered the threshold of participating in future events such as hackathlons etc.*

6.3 Instructors' Unanticipated Findings

From an instructor's perspective, the dojos worked quite much according to our expectations as a mean of teaching pair programming and TDD. Despite the individual flavour in each dojo session, even the least successful sessions gave the intended experience to the participants.

Dojos gave the instructors a unique opportunity to observe how students act together in an actual problem-solving setting. Through the observations it became evident that the working process of quite a remarkable portion of the students is still somewhat non-optimal, even after nearly two years of studies. Also it became evident that many students used the programming environment in a rather suboptimal way, and did not utilize e.g. the embedded support mechanisms such as shortcuts for automatic code generation, refactoring, and the embedded API-descriptions, which ease programmers' mental burden during repetitive tasks.

The extent to which the suboptimal working habits were observed came as a surprise to the instructors, and gave us a clear indication that we need to take measures against the development of bad habits.

7 Conclusions and Current Work

Coding Dojo is a good addition to teaching Agile software engineering practices because it allows the students to experience practices such as pair programming and TDD in a relatively authentic context. The main benefit over traditional labs is the audience in the coding dojo, whose task is to provide comments and suggestions, which can be used for reflection of the students' own study practices.

A majority of the dojo participants saw the dojo as non-competitive and relaxing, and learned new things about both teamwork and programming. Students in general see a dojo as something that many of them could suggest to others, and many would like to participate in a dojo also in the future.

In our context, the experiment with coding dojos in the software engineering course led to a small-scale student movement that has started to create pop-up coding dojos on campus. As an example, a few of our students that participate in organizing K-12 outreach activities, organized a dojo session for an algorithmics club that is attended by high-school students interested in programming competitions.

During the dojo, many types of learning was observed. One spontaneous outburst after the dojo defines the goal quite well:

> *I had learned about TDD already before, but only now I really understand what it is about.*

In addition to students learning about the intended practices, quite a lot of students indicated that they had learned helpful things such as key combinations and shortcuts, which were not planned as part of the learning objectives beforehand. This is in line with the situated view of learning that stresses that some parts of the learning is also unintentional, as it depends heavily on the context and interaction.

In our current activity, we are supporting the students that wish to organize their own dojos by providing facilities whenever needed. In addition, we are starting to embed the coding dojo in other courses such as our server-side web development course.

8 Acknowledgements

The authors would like to thank Juhani Hietikko, Jussi Härkönen, Juho Rautio and Vesa Ylönen from Houston Inc. for hosting a pilot coding dojo, which helped us to smoothen the path for embedding coding dojos into formal education.

References

[1] D. J. Anderson. *Kanban*. Blue Hole, 2010.

[2] A. Bandura. *Self-efficacy*. Wiley Online Library, 1994.

[3] K. Beck. *Test Driven Development: By Example*. Addison-Wesley, 2002.

[4] K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004.

[5] J. Brown, A. Collins, and P. Duguid. Situated cognition culture of learning. *Educational Researcher*, 18(1):32, 1989.

[6] A. Cockburn and L. Williams. The costs and benefits of pair programming. In *the First International Conference on Extreme Programming and Flexible Processes in Software Engineering*. ACM, 2000.

[7] A. Collins, J. Brown, and A. Holum. Cognitive apprenticeship: Making thinking visible. *American Educator*, 15(3):6–46, 1991.

[8] Dojo coding day at Durban University of Technology. http://www.pieterg.com/2012/6/dojo-coding-day-at-durban-university-of-techn.

[9] M. Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[10] R. C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008.

[11] R. L. Meier, M. R. Williams, and M. A. Humphreys. Refocusing our efforts: Assessing non-technical competency gaps. *Journal of Engineering Education*, 89(3):377–385, 2000.

[12] W. Royce. Managing the development of large software systems. In *Proceedings of IEEE WESCON 26*. TeX Users Group, August 1970.

[13] D. Sato, H. Corbucci, and M. Bravo. Coding dojo: An environment for learning and sharing agile practices. In *Proceedings of AGILE '08*. IEEE, 2008.

[14] K. Schwaber and M. Beedle. *Agile Software Development with SCRUM*. Prentice Hall, 2002.

[15] J. Shore. *The Art of Agile Development*. O'Reilly, 2010.

[16] University of Houston Coding dojo. http://www.codedojo.org/.