2021 ASEE ANNUAL CONFERENCE
Virtual Meeting | July 26–29, 2021 | Pacific Daylight Time
Paper ID #32787

Using Agile and Active Learning in Software Development Curriculum

Prof. Ben Tribelhorn, University of Portland

Ben Tribelhorn teaches Computer Science at the University of Portland. His research includes machine learning for chaos in Lorenz systems, dynamic obstacle avoidance algorithms for unmanned aerial vehicles, improving software engineering pedagogy, and ethical concerns in artificial intelligence.

Dr. Andrew M. Nuxoll, University of Portland

Andrew began his career as a software engineer. Lately (since 2007) he has been teaching computer science at the University of Portland. He is an active researcher in artificial general intelligence and computer science pedagogy. He also loves playing bridge and being outdoors.

Using Agile and Active Learning in Software Development Curriculum

Abstract

This paper introduces a novel implementation of an agile software development process within an active learning paradigm in a semester long junior-level course for Computer Science majors. This Software Engineering course is centered around experiential learning of the entire software development lifecycle (SDLC) and applying a modified version of Scrum throughout. In order to mirror the real-world practice in a twice a week 85 minute class, we present many adjustments to Scrum for use in the classroom. We describe the implementation of the top six agile techniques used in industry (daily standup, sprint planning, retrospectives, sprint review, short iterations, planning poker) which focuses the learning experience on the most important components of agile development in addition to including top engineering practices used in industry. Additionally, we report extensions and variants for adapting this design to existing software engineering courses at other universities. Among these variants we propose adopting class-wide teams which is atypical at other universities for junior-level project courses.

Introduction

The agile software development is widespread in industry, however it is not widespread in computer science education. According to a corporate survey, the 13th Annual State of Agile Report[1], "97% of respondents report their organizations practices agile development methods." This development process is a missing skill set for most computer science undergraduates entering industry. Given the nature and scope of most coursework, undergraduate computer science students naturally learn to develop using a waterfall design process which is an antithesis to agile software development processes. Studies have shown that active learning promotes learning[2], so this work integrates active techniques to completely focus the course on the real-world engineering process using agile methods. Ultimately, there is a need to incorporate this into the curriculum as a focal point of teaching software development to undergraduate computer science students.

Background

Variants of agile development have been slowly integrated with traditional approaches into courses, most commonly in capstone courses and project management courses. (e.g., [3, 4, 5, 6]). However, it is rarely a focal point as most textbooks provide minimal content on Agile[7]. There is a need to incorporate Agile into both the content and pedagogy of courses[8]. Agile has been combined with service-learning[9], but there has not been a lot of work integrating Agile with active learning techniques. A specific implementation of agile that is inherently and especially active is Scrum. Scrum or a hybrid Scrum are the most common versions of applied Agile

“methodologies”[1]. Scrum is used in some courses, but often not applied holistically and in the manner of industry[10, 7, 9]. The authors posit that the difficulty of replicating a complete set of agile practices in the form of a process such as Scrum limits the adoption of this pedagogical focus in curricula outside of software engineering programs. Many adaptions are needed to approximate these techniques in an undergraduate classroom setting.

Scrum is a variant of agile software development that aims to be lightweight and subscribes to the standard agile practice of breaking up the work schedule into blocks, called sprints. These sprints are typically units of time ranging from one to four weeks. At the end of each sprint, the development team should be delivering some incremental value to the end customer[11]. Within Scrum, the sprint is managed with a number of key process-based events. Each sprint begins with a sprint planning meeting where the entire Scrum team generates tasks in various forms, often in the form of user stories (user-centered statements of requirements and acceptance criteria) which are placed in the product backlog. These are refined by the development team and then confirmed for the sprint and added to the sprint backlog. Each future work day within a sprint includes a short stand-up or daily scrum meeting where each team member addresses three issues: recent work, immediate work plan, impediments. These meetings are time-limited so issues are addressed by smaller groups and not the entire team. At the end of a sprint, the work is reviewed, often via demos, and finally a retrospective meeting is completed. In the retrospective, the team reflects on how to improve the application of the specific agile process. Scrum specifies three roles: product owner, Scrum master, development team. An important distinction between Scrum and other agile methodologies is that it has no particular software emphasis which allows for individual teams to specify specific engineering practices. This also makes it particularly suitable to academic settings. The table below summarizes the key components of the Scrum process.

Scrum Components

Roles	Events	Artifacts
Product Owner Scrum Master Development Team	Sprint Sprint Planning Daily Scrum (standup) Sprint Review Sprint Retrospective	Product Backlog Sprint Backlog

Scrum is used in some curricula, but often not applied holistically and in the manner of industry[10, 7, 9]. Some of the most recent work looks specific components of the agile process to assess and validate their effectiveness in a software engineering course[12]. Given the success and popularity of this developmental methodology, software engineering education could be improved by reducing this delta. This paper introduces a novel implementation of an agile software development process within an active learning paradigm in a semester long junior-level course at our university. This course, “Software Engineering,” is centered around experiential learning of the entire software development lifecycle (SDLC) and applying Scrum throughout. In order to mirror the real-world practice in a twice a week 85 minute class, some adjustments to Scrum are necessary. This course implements the top six agile techniques used in industry (daily standup, sprint planning, retrospectives, sprint review, short iterations, planning poker)[1] which focuses the learning experience on the most important components of agile development in

addition to including top engineering practices used in industry. The course has been taught in its described form since 2017 over six semesters with a total of eleven sections. Within the description we offer a commentary on variations to allow this to be adapted for use at other institutions. We believe that the combined focus on professional practices and tools in this setting allows our graduates to rapidly integrate into their software teams during both internships and jobs. Finally, we include some novel project-based learning interventions to increase student engagement with the agile methodologies and engineering practices taught in the course.

In order to achieve technical learning objectives (such as skills in web technologies, writing unit/acceptance tests, creating detailed use cases and requirements) and experience of agile principles in practice, the authors must note that much of the strategic level processes and context are omitted from this course due to the time constraints.

Methods and Design

Our Software Engineering course is scheduled as a semester long (15 week) course, required for computer science majors and commonly taken by minors. There is a single semester long project that the students complete in groups of four to five students. The large group size mirrors the larger teams found in industry and encourages development of communication skills. The course project has included various primary programming languages including Java, Python, and JavaScript. Past course projects include creating a web browser and creating web-based applications. In most offerings the assigned work has been scaffolded with increasing requirements provided by the instructor in addition to students self-directing the final product based on a written requirements specification that evolves during the semester and successive sprint planning. Having the instructor provide new and/or changing requirements during the semester helps the students avoid a waterfall design process. In three offerings we have had externally directed projects as discussed in the variations. This course has the following learning objectives:

- Students will specify, design, build, and test real world engineering systems.
- Students will increase their ability to understand the ramifications of design decisions.
- Students will increase their ability to identify and evaluate sources of information.
- Students will learn and apply the software development lifecycle.
- Students will learn real world development technologies.
- Students will use Agile development practices.
- Students will reflect on professionalism in software development.
- Students will increase their ability to work in teams and communicate technical information.

Course Design

Scrum Roles: The biggest modification of Scrum in the classroom is the application of the three roles: product owner, Scrum master, and development team. Naturally, the student teams comprise the development team, however there are many methods to assign product ownership and the Scrum master. In most sections we have split the roles of product owner and Scrum master between the instructor and the teams. To this end, the instructor plays the role of product

owner as it comes to specifying releases (in the form of assigned milestones to complete during sprints) and encourages the teams to focus on the stakeholders especially the potential end users. The students however, have product ownership in that they are required to generate requirements and priorities for development. The grading of a sprint reflects this division. For the role of Scrum master, the instructor educates and facilitates the application of Scrum with each group during class time. This would require teaching assistants for larger sections, we find that a single instructor has more than enough time to coach five teams during 85 minute class periods. Later we discuss a variation in this application of roles.

Scrum Events: The course is divided into approximately five to six sprints (see sample schedule below). The initial weeks are where the students learn about the Agile process, how to elicit and document requirements, and various software tools. As the content taught at this point is all new to the students this "sprint zero" is not actually a sprint as we are rapidly teaching students Scrum components (in context of Agile principles), development tools, and methods of requirements elicitation.[1] This often includes programming separate from the course project; we have found that short individual programming assignments are most effective during this requirements gathering sprint to aid mastery of new programming languages, libraries, and tools. This rapid exposure to new technologies and tools gives the students a real world experience as they must learn these tools with limited instructor-provided scaffolding. The following sprints run through the software development lifecycle (SDLC): creating a minimum viable product, feature refinement/refactoring, deployment and risk management, and quality assessment via maintenance. As a model of real software teams, a sprint is often two weeks long and begins with a sprint planning meeting using class time.

Sprint Schedule

Weeks	Sprint	Topics
1-3	Tools	Agile, Requirements, UIs
4-5	Prototype	Testing
6-7	Alpha/MVP	Technologies
9-10	Beta/Feature refinement	
11-12	Deploy	Deployment, QA
13-15	QA/Maintenance	QA

The sprint planning meeting uses class time, students must take features assigned by the instructor and features that were self-generated and create the implementable tasks in the form of user stories or use cases which do include acceptance criteria. We require students to use Trello (`www.trello.com`) to track task progress, task assignments, and task sizes (modeled on the Scrum product and sprint backlogs). Students are taught estimation to assign units of work to their tasks using simplified planning poker where each finger revealed is one hour of work. The appropriate use of task tracking is evaluated by the instructor. Students will tend to volunteer for tasks during estimation based on their confidence and interest in tasks. Left over, unassigned tasks are then assigned through a second round of volunteering until at about two thirds of the sprint tasks (by units of work) are assigned.

[1] Industry teams often designate these tasks as Scrum spikes to ensure technology and design decisions are separated from product requirements. This distinction is clarified during their senior capstone projects rather than in this course.

In most class periods, we have the students run through a standup meeting where they discuss what work has been done since their last meeting, what they will work on next, and any problems they are having. As students meet twice a week in class and usually at least one to two times outside of class as a team, they experience a standup three to four times a week. Students are often tempted to problem solve during these meetings and must be guided by the instructor to stay on task to focus on process so that this meeting is effective.

The final class period of every sprint is used for each team to present a live demo of their software, review sprint progress, and discuss testing. As part of the sprint review, a demo allows the instructor to provide feedback and to assign specific tasks to each team for the next sprint (especially important as projects differ). These tasks are often feature improvements or additions, and this practice is not a model of industry but required academically to ensure weaker or less-motivated teams continue to make appropriate progress. In fact, these instructor-assigned tasks are how the instructor informs students that they may have missed appropriate acceptance criteria. Students are encouraged to report their code coverage during the demo which incentivizes a continued focus on unit and acceptance testing. In some cases students will showcase code, and a more detailed code review could be conducted. During this demo, acceptance criteria are reviewed to ensure that the team has deployed code that satisfies the sprint's requirements and constraints. We also prohibit students from demoing from multiple machines or codebases. All code must have been successfully deployed at the end of the sprint.

After all teams have presented (typically 5-15 minutes per team), a sprint retrospective occurs. The first retrospective is done anonymously to encourage constructive comments (i.e. using ideaboardz). The students tend to focus on programming challenges so the instructor must coach them to focus on process improvements. It is during these meetings that students realize the importance of good communication and effective meetings.

At the end of this class period, the students will then start their sprint planning for the next sprint. Usually, there is not enough time for them to complete this meeting, so they will meet outside of class to complete their sprint planning.

Each sprint is graded as a team for each sprint as follows: primary feature implementation (as instructor defined) 35%, appropriate generation and implementation of team-defined features 20%, appropriate tool use (such as GitHub `www.github.com`, Trello, etc.) 25%, unit testing and code coverage reports 15%, communication & teamwork 5%. This score for all sprints is modified per individual as necessary to reflect their actual participation and counts for 50% of the course grade. Alerting the students to the fact that almost a quarter of the course grade is determined by conformance to the tools and agile methodology makes them focus on engaging with the development practices. Although it seems that the teamwork grade is unreasonably low, this is because communication and teamwork are evaluated holistically over the entire course. The instructors require students to complete surveys twice: mid-semester and end-of-semester which ask for their contributions and the contributions of each team member. Additionally, the scope of each member's commits, activity on trello, and external feedback if any is incorporated as an adjustment to the project grade. In practice, this means that poor teamwork (such as missing meetings, failure to complete tasks, failure to communicate tasks being worked on, etc.) have reduced grades to the point of requiring retakes. The methodology here matches our departmental grading of senior capstone projects.

Project-Based Learning

SDLC topics are covered with a variety of active learning activities including requirements elicitation, creating user stories, software testing, and quality attributes. These activities are inspired by project-based learning.[13] The course content is heavily front-loaded, so most of these activities are conducted in the first half of the course. Here we briefly detail some of the activities.

As most of our projects are student-driven we need to encourage the students to generate a wide and deep set of project requirements. To achieve this we have the students complete an interview activity about a theoretical business idea. We assign students to an even number of teams for this activity, each team must come up with a new software idea (and may use existing software for inspiration). They are told they are business students and they are going to hire a group of software engineers to build their idea. They have a small amount of class time to discuss, and then in a future class period teams are paired and then one to one pairings separate and interview each other for their business idea using context free interview questions. After separately interviewing, the teams come back and integrate the software requirements received. Finally, each team presents to the class what they think they would build for this team. Most students find that this activity improves their ability to elicit requirements and integrate information from multiple sources.

Unit testing is the number one engineering practice cited according to the 13th Annual State of Agile Report[1] used by 69% of teams and Continuous Integration is third with 53% of teams reporting its use. We find that it helps to show students the value of testing to increase their commitment to these industry standard processes. We use a day of class to read (for approx. 10 min) a short real case study of an incident where there was economic damage (& possibly loss of life) from a failure to run tests before deploying software. Students are assigned away from their project teams to new teams to represent different groups 1) Legal/Public Relations for the company, 2) Developers, 3) Customers/Users, 4) Government. They are asked to prepare a statement that might address one or more of the following: lessons learned, responsibility, actions, ethical concerns, impacts. These groups then present informally this statement or set of statements and the instructor is sure to bring up in discussion issues including: who was hurt, litigation, liability, job security, how it happened, what it means to be a whistleblower, governmental regulation or policy interventions. After all teams have presented they return to their project teams to discuss and decide the likely outcomes in terms of fault, liability, costs, etc. based on what each group presented. Students can present this to the class or just present to the instructor. Finally the instructor leads a whole class discussion of the value of testing, and possible ethical issues related to their course project. This can focus on the tradeoffs in terms of costs and risk. This activity can fill a single 85-minute period, or two 55-minute periods. After completing this activity, we have found that students are more motivated to actually write unit tests.

This course also mandates the use of a large number of software tools based on what is commonly used in industry and available for use academically (currently many options through the GitHub Student Developer Pack). As is becoming common, we use GitHub Classrooms to create repositories with instructor access. These repositories are then linked with a Continuous Integration (CI) system (we use Travis-CI, `www.travis-ci.com`) and a code coverage tool

(we use Codecov, `www.codecov.io`). This integration can be the most difficult part due to language choice as testing systems can be challenging to understand quickly. The benefit of including a limited amount of CI is that the tests are then run at every commit and reinforces the importance of testing and a focus on proper integration of new code into the code base. CI and cloud-based code coverage is an easy elimination in adopting this course as students can still, at a minimum, determine testing coverage through simpler tools locally. One of the important methods here is our requirements for code reviews using pull requests. Pull requests are often overlooked in the classroom, but have been shown to be effective in encouraging successful collaboration and is an essential industry skill.[14, 15] In our experience, requiring personal branches, pull requests, and review vastly reduces merge conflicts and improves communication. In contrast to industry, students may end up having feature-level branches independently while working on the same feature. We encourage this as it tends to avoid many common frustrations. Requiring appropriate, reviewed pull requests is our strongest recommendation for courses that use GitHub and larger teams.

Quality assurance is a key component of this course as most agile methodologies prioritize delivering high quality software at each sprint. This course incorporates a number of components to encourage a quality focus by the students. In the later sprints the students have to "deploy" their software and evaluate the quality at this point. This can include an analysis of performance, security, aesthetics, and other relevant quality attributes. In the final sprint there is a feature freeze so that coding efforts are working on quality assurance. In class, the students use each other's software and submit issues on GitHub. Then the teams are asked to identify target quality attributes and to craft sprint goals designed to improve their product in these chosen aspects. Class time is also used to review existing software products and compare their software quality for specific quality attributes using a Jigsaw[16] style expert groups and ranking activities followed by discussions. The instructors anecdotally observed that overall product quality did improve as a result. Furthermore, customers were happier with the final result.

Many variations are possible with the structure of this course. Below is a brief summary of some that we have used successfully.

International Collaboration: In the Spring 2019 semester, we collaborated with Kaunas Technological University (KTU) in Kaunas, Lithuania. The student team was larger: consisting of six students from one university and five from the other. Both universities used the same high-level structure for their courses though individual assignments were different. This approach presented some predictable pros and cons.

- Pro: Students learned about the difficulties associated with remote collaboration
- Pro: Students gained exposure to another culture
- Con: Communication and collaboration were much more difficult both for cultural and logistical reasons. The different time zones were particularly problematic.
- Con: The difference in semester schedules meant that one university had to begin work a full three weeks before the other which, subsequently, continued for four weeks after the first. The universities also had mid-semester breaks at different times. The net result was that collaboration could only occur during half the 15-week semester.

Class-Wide Project: One of our most unique contributions was to experiment with large groups

as most reported agile project based learning courses tend to utilize small teams[17, 18]. In the Fall 2019, Spring 2020, and Fall 2020 semesters, we experimented with assigning a single project to the entire class. Additional roles were assigned to students in order to better approximate an actual software company: a CEO (the instructor), CTO (student elected), Scrum masters, software engineers and quality engineers. These roles were rotated on a sprint-by-sprint basis. This makes a number of differences compared to smaller teams, mostly leading to a better immersive experience:

- When the team is a least 15 students, communication/coordination suddenly gets a lot harder. The value of tools like Trello boards and stand up meetings become much more apparent. We require the student teams to meet once per week outside of class but we find that they often voluntarily meet more frequently. Certainly the leadership team (CTO and team leads) meets even though it is not required.
- Students get taste of being a mid-level manager (the "CTO" role). They are elected to this position by their peers and not allowed to write any code while they are in the role. Even so, they quickly realize that it's a lot of work. This is much more of a true leadership position than just a team lead. The CTO selects sprint goals and team composition. "I was class CTO" looks good on a resume too.
- Progress is faster. Having one team of 20 isn't four-times faster than one team of five would be because of the social/learning overhead. Still, they make progress about twice as fast. So, about twice as much functionality is implemented in a semester.
- Since there is different CTO every sprint who must shuffle the teams, they effectively they get to experience multiple re-orgs.
- The team changes mean they spend a lot of time eating their own dog food (e.g., adding features and fixing bugs in code they didn't write). So, the value of comments and good code design becomes much more apparent. The value of code reviews becomes very apparent too.
- It's much easier to find a real customer for the team since the instructor only need to find one (see next section). That customer is often particularly engaged and, due to Scrum, comes to class often (about once every 2-3 weeks).

This approach created more realism at the expense of adding additional communication and organizational hurdles for the students. In both cases, the instructor anecdotally observed a much stronger sense of camaraderie and enthusiasm for the project. This class-wide approach allows the instructors to stay truer to Scrum within a classroom setting as all of the roles can be filled especially when we recruit a real world client.

Real-World Client: We have also experimented with introducing a real-world client in the Fall 2019, Spring 2020, and Fall 2020 semesters. Our university has an entrepreneurial certificate program wherein students learn to conceive, assess, create a business plan for and pitch an idea. By partnering with this program, we were able to recruit students whose business ideas require a web application. In the Scrum fashion, this client becomes part of the team.

This approach gave the students an opportunity to understand how a software product fits in a business context. Students got more exposure to the reason for an agile software development model: the desired product requirements change as the project progresses. These customers change their minds and change is a critical thing to cope with. The value of agile methodology is

clearer in this context. The students had to identify, negotiate and, ultimately, adjust their design for changes in client needs. Finally, the fact that the product had a real user in mind meant that students were less inclined to compromise on the quality of their work.

Instructors who try this variant should be mindful of intellectual property. In the Spring 2020, we introduced a legal agreement where the students agreed to give up their intellectual property to the customer.

Discussion

The course presented here has increased the process focus beyond the level that is normally included in a typical software engineering curriculum. This course design is highly flexible and applicable at other institutions. In addition to the possible variations discussed above, we found that the choice of the primary programming language(s) has not impacted the learning outcomes; although when introducing a new languages the initial progress towards the stated project goals is slower. Ultimately, this course achieves its learning outcomes as an ABET benchmark course for for program outcomes #2 and #5, and students respond positively to this real world experience.[2] Unlike many other Computer Science programs, this implementation of agile practices achieves a high level of immersive industry-like experience.

This course design works well for a semester long project. Students find that their initial software designs are not well thought out, yet using the agile process enables them to improve iteratively unlike traditional course projects. We believe that this application of the agile process counteracts the common waterfall paradigm used by students due to the limited scope of most undergraduate course assignments. Additionally, the agile methodology is a natural fit with active learning techniques that promote deeper learning by increasing engagement in each class period. Specifically, we described how five of the top six agile techniques in industry were adapted use class time to flip the classroom experience (daily standup, sprint planning, retrospectives, sprint review, planning poker). Throughout the course students are required to use the software tools and processes that are commonly used in industry. This experience is unique and leads to excellent achievement of learning outcomes in real world engineering, professionalism, and teamwork.

Acknowledgments

The authors would like to thank Dr. Tomas Blazauskas (KTU) for support of the international development project and Ignas Paplauskas of Devbridge for working as a client and mentor of the international project. The funding for this work was provided by the Kern Entrepreneurial Engineering Network (KEEN) as part of a grant to the University of Portland and by a grant from the Donald P. Shiley School of Engineering.

[2] ABET Performance Indicator 2: Design, implement, and evaluate a computing-based solution to meet a given set of computing requirements in the context of the program's discipline.
ABET Performance Indicator 5: Function effectively as a member or leader of a team engaged in activities appropriate to the program's discipline.

References

[1] 13th Annual State Of Agile Report., 2019. URL https://www.stateofagile.com/.

[2] M. Prince. Does active learning work? a review of the research. *Journal of Engineering Education*, 93(3): 223–231, 2004.

[3] A. Baird and F. J. Riggins. Planning and sprinting: Use of a hybrid project management methodology within a cis capstone course. *Journal of Information Systems Education*, 23(3):243–257, 2012.

[4] J. P. Landry and R. McDaniel. Agile preparation within a traditional project management course. *Information Systems Education Journal*, 14(6):27–33, 2016.

[5] P. A. Laplante. An agile, graduate, software studio course. *IEEE Transactions on Education*, 49(4):417–419, 2006.

[6] S. Ramakrishnan. Innovation and scaling up agile software engineering projects. *Issues in Informing Science and Information Technology*, 6:557–575, 2009.

[7] J. May, J. York, , and D. Lending. Play ball: Bringing scrum into the classroom. *Journal of Information Systems Education*, 27(2):87–92, 2016.

[8] J. H. Sharp and G. Lang. Agile in teaching and learning: Conceptual framework and research agenda. *Journal of Information Systems Education*, 29(2):45–52, 2018.

[9] S. Robinson and M. Hall. Combining agile software development and service-learning: A case study in experiential is education. *In SIGCSE '18: 49th ACM Technical Symposium on Computer Science Education*, 2018.

[10] O. Jimenez and D. Cliburn. Scrum in the undergraduate computer science curriculum. *J. Comput. Sci.*, Coll. 31 (4):108–114, 2016.

[11] M. Cohn. *Agile estimating and planning*. Prentice Hall, 1 edition, 2005. ISBN 9780131479418.

[12] An Ju, Adnan Hemani, Yannis Dimitriadis, and Armando Fox. What Agile Processes Should We Use in Software Engineering Course Projects? *In SIGCSE '20: 51th ACM Technical Symposium on Computer Science Education*, 2020.

[13] Brigid JS Barron, Daniel L Schwartz, Nancy J Vye, Allison Moore, Anthony Petrosino, Linda Zech, and John D Bransford. Doing with understanding: Lessons from research on problem-and project-based learning. *Journal of the learning sciences*, 7(3-4):271–311, 1998.

[14] Arie van Deursen, Mauricio Aniche, Joop Aee, Rogier Slag, Michael de Jong, Alex Nederlof, and Eric Bouwers. A collaborative approach to teaching software architecture. *SIGCSE '17: Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, pages 591–596, 2017.

[15] L. Haaranen and T. Lehtinen. Teaching git on the side: Version control system as a course platform. *In Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*, page 87–92, 2015.

[16] Elliot Aronson. The Jigsaw Classroom, 2000. URL http://www.jigsaw.org/index.html.

[17] Pasquale Salza, Paolo Musmarra, and Filomena Ferrucci. Agile Methodologies in Education: A Review. In Kathryn MacCallum David Parsons, editor, *Agile and Lean Concepts for Teaching and Learning*. Springer, Singapore, 1st edition, 2019. ISBN 978-981-13-2750-6.

[18] Michael J. Lutz, J. Fernando Naveda, and James R. Vallino. Undergraduate software engineering. *Commun. ACM*, 57(8):52–58, August 2014. ISSN 0001-0782. doi: 10.1145/2632361.