I.J. Modern Education and Computer Science, 2018, 5, 17-26
Published Online May 2018 in MECS (http://www.mecs-press.org/)
DOI: 10.5815/ijmecs.2018.05.03

Continuous Delivery Pipelines for Teaching Agile and Developing Software Engineering Skills

Héctor F. Cadavid
Decanatura de Ingeniería de Sistemas, Escuela Colombiana de Ingeniería, Bogotá, Colombia
Email: hector.cadavid@escuelaing.edu.co

Received: 08 December 2017; Accepted: 15 January 2018; Published: 08 May 2018

Abstract—The amount of research reports on how to properly teach, in conjunction with technical topics, agile skills in undergraduate courses is a good indicator of how important are such skills in academy and industry nowadays. Such investigations have addressed challenges like how to engage students with agile principles and values without getting distracted by technology, or how to balance theory and practice to get students to meet learning objectives through practical experience. This paper intends to contribute to this research topic by describing new strategies for our particular needs for teaching agile in an introductory software engineering course, including better evaluation criteria for agile values and practices, and higher quality projects. The described strategies include a new approach for theoretical, laboratory, and project sessions arrangement, as well as a 'Continuous Delivery Pipeline' adapted to our educational context, with very promising results.

Index Terms—Software Engineering, Agile Teaching, Software Engineering Education, SDLC Teaching.

I. INTRODUCTION

The differences between academy and industry when it comes to working settings (schedules, workload, working environments, etc.), have motivated the development of a significant body of research on how to achieve meaningful learning processes of Software Development Life Cycles (SDLC) in the former, in order to ease the incorporation processes of new Software Engineers in the latter.

Such research works have evolved together with the SDLC themselves, in particular between the early 2000s when most programs taught highly prescriptive methodologies like RUP/UP or Extreme Programming (XP) [1, 2, 3], and a decade later, when agile SDLC like SCRUM had already become one of the most widespread agile methodologies in the industry and academia slowly began to embrace it (near 100 references in Google Scholar for "teaching agile software development" between 2010 and 2017).

An important consensus that has been reached with regards to teaching SCRUM is that the strong orientation towards values and the hands-on nature of agile values and practices mean that traditional approaches to teaching -through theoretical communication- are incapable of leading to a real adoption of such principles and values [4]. In contrast, in our own experience, these agile elements are difficult to assess objectively in the end of a software engineering course, mostly when the work is group-based and individual grades are expected.

This paper describes a proposed methodological approach a for the first software engineering course at the *Escuela Colombiana de Ingeniería*, built upon ideas of previously published works and a novel strategy based on an extension of the concept of Continuous Delivery (a popular practice in IT industry) that aims to:

- Gather data during project development process for more objective evaluations of practices and values.
- Encourage agile values through continuous semi-automated feedback of such assessments and other software metrics.
- Achieve a realistic software project scenario within the limited time of a course and the rigorous schedule of an undergraduate course.

This paper is organized as follows: section 2 describes the related works on which this methodological proposal is based; section 3 provides an overview of the traditional approach of the software engineering courses at the *Escuela Colombiana de Ingeniería*, including new problems identified in the learning of software engineering concepts; sections 4 and 5 describe the proposed methodological approach, including the course outline, contents, and methodology. Section 6 describes and analyzes the results obtained so far, and conclusions are then presented in section 7.

II. RELATED WORK

Starting in the 2010s, and given the growing popularity in the industry of management-oriented agile

development frameworks such as Kanban, Lean, and especially Scrum, research efforts on the topic of software engineering teaching methodologies were focused on how to create appropriate environments and contents for the learning process of these agile approaches. Mahnic and Viljan Mahnic [5] proposed a project-oriented software engineering course focused exclusively on Scrum by performing four sprints (one theoretical and three practical) in a quasi-real project. This course had previous software engineering courses as a pre-requisite where all the required technical concepts for basic software development were covered using a 'traditional' software development process.

Kropp et at Kropp and Meier [6] went one step further by going into more detail on this topic and proposing the pyramid of competencies for software engineering course design: software engineering practices, management practices, and agile values. They proposed a project-oriented software engineering course where the agile values are expected to be taught in lectures and workshops, but more importantly, through the practice in the form of a project (a 2D game) and some collaboration-related activities: sprint-retrospective, common code ownership, or pair programming. On the other hand, as engineering practices, topics such as eXtreme Programming (XP), software versioning, and Continuous Integration (CI) were covered; whereas Scrum roles and principles, pair programming and planning poker—among others— were studied for management practices. In this case, there is no real 'stakeholder', as the teacher defines the product and assumes the role of product owner.

In the same year, Václav Rajlich [7], besides proposing a similar competencies-based course design, documented what is known as the 'deadly sins' of software engineering education. These can be summarized as follows:

1. Unrealistic projects and unrealistic quality expectations.
2. Courses focused on outdated, out of the current mainstream practices.
3. Course practices or roles too advanced for the average student.
4. Too much time spent on software development processes surveying.
5. Course projects in unfamiliar and difficult domains.
6. Course projects that are aimed at producing a 'pretty' final product, without emphasizing the process.
7. Group evaluation based only on the project outcome and fail to consider individual contributions (unfair grades).

Another important milestone in this research topic is the concept of agile games, which consists of methodological proposals for teaching -in the practice- agile principles in a small time-scale. Given that software development is a heavy time-demanding task, previous approaches to software engineering teaching allow for reflection about the agile process only at the end of the course. The agile games challenge the students to create simpler non-software artifacts that meet certain acceptance criteria, by using the same 'ceremonies' and artifacts of a software-oriented Scrum project. Some examples of such games include SCRUMIA [8], a game focused on the building of origami artifacts; SCRUMI [9], an electronic board serious game; and Scrum4Lego [10], a game focused on the building of cities with Lego blocks, which has become the most popular of these activities thanks to its creative-commons licensed material.

Kropp et al. [11], who participated in some of the early investigations mentioned above, proposed an updated version of the agile competency pyramid by performing a survey of the most relevant practices in today's Swiss IT-industry (for technical and collaborative practices), and including Scrum4lego as a key activity for teaching agile values. On the other hand, Matthies et al [12] proposed a set of metrics to improve the evaluation process in Software Engineering courses by quantifying the outcome of an agile process, including collective code ownership (GIT Statistics), untested complexity, committing rate, and Agile User Stories quality. More recently, Ochodek [13] proposed an approach to synchronize theoretical and practical sessions of a regular Software Engineernig course towards an incremental understanding of Scrum principles.

III. PROPOSED APPROACH BACKGROUND

The computer and systems engineering program at the *Escuela Colombiana de Ingeniería* has trained hundreds of highly skilled professionals since the 1980s, most of them with a strong background in software engineering. Starting with the thirteenth curriculum upgrade process in 2009, the program committee defined three consecutive courses as the core of the software engineering component of the curriculum. Preceded by computer programming and database design courses, and inspired by the unified process phases (see Fig. 1), the following learning objectives were defined for such courses:

- Software Engineering 1 (Inception): software development life cycles, with a strong emphasis on unified process. Requirements for software gathering and analysis.
- Software Engineering 2 (Elaboration): Software Architectures, Quality Attributes, and Architectures documentation.
- Software Engineering 3 (Construction): Software Construction, with an emphasis in Software Configuration Management.

In the subsequent program upgrades, during the transition to agile methodologies as the base approach for software engineering courses, the three courses were reformulated as a set of transversal competencies, with different levels of detail in each one. With this approach,

Copyright © 2018 MECS

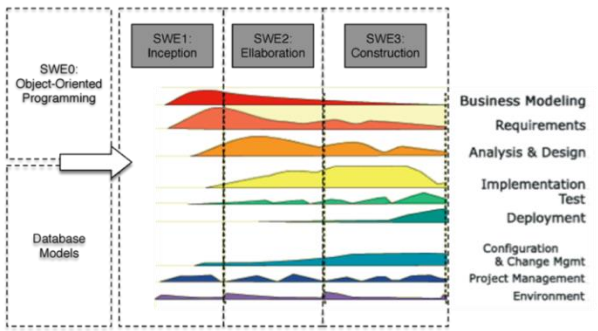


Fig.1. Sequence of courses in the Software Engineering line. Adapted from an illustration at [14].

the first software engineering course aims to the development of competencies in SCRUM, including values, principles, and how to follow SCRUM master and team member roles while covering the minimal amount of technical and architectural concepts in the process as depicted in Fig. 2. In the courses that follow it competencies in software architecture/development, user-centered design, and software entrepreneurship are developed in greater depth, applying and extending the methodological elements of the first course (for example, by taking on the role of Product Owner).

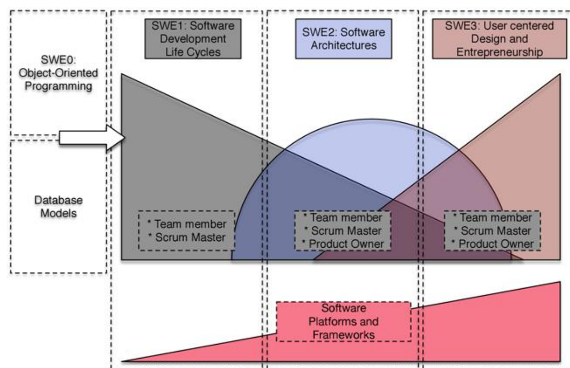


Fig.2. Sequence of courses in the Software Engineering Line revisited: now defined as a set of overlapping competencies, which we cover to varying degrees in courses.

Through the transition described above, the program's staff identified a particular set of 'deadly sins' of agile teaching, complementary to the ones previously reported by authors like Rajlich [7]. These methodological mistakes or 'sins', described below, were what motivated us to seek and propose the strategies described in this paper:

1. Trying to encourage the students to follow an agile approach (e.g., SCRUM) in a project, while teaching all the required technical concepts for its architecture and technology stack. In our experience (unlike the one described by Ochodek [13]), this causes two problems: first, this sometimes makes it impossible to follow the agile principle of using working software as measurement of progress, and second—but most importantly—the novelty and the challenges of the newly presented technologies tend to shift the students' focus from the methodological elements of the software development to the technical ones.
2. Not identifying 'cowboy' coders and 'negative solidarity' opportunely. These are two variations of the same problem. In the first case, some 'skilled' but rule-aware students known as cowboy coders (a term coined by Janes and Succi [15]) tend to take on most of his/her team's work (in most cases, because of a lack of confidence in their teammate's skills). In the second case a project team—motivated by a false sense of 'solidarity'—allow one of its members not to contribute to the project, sometimes due to a recognition of a lack of software development knowledge or skills on the part of the non-contributing team member. Besides the problem of passing students who lack key skills to more advanced classes, this also leads to some team members feeling that the process is unfair when a group evaluation is performed (Rajlich's seventh 'deadly sin').
3. Missing opportunities to learn from mistakes due to late feedback. Time is a very limited resource for a teacher, mainly when he/she has large groups of students on his hands. This situation sometimes leads to only a few shallow and/or mediocre reviews of the projects (e.g., code quality), which may hide fundamental problems until final reviews. Given that at the end of a course students tend to be more concerned about their grades than on reflecting on errors that they have no time or reason to fix, late feedback is a lost opportunity for students to learn from their mistakes.
4. Allowing unjustified postponement (procrastination) due to lack of progress tracking. Starting work close to the deadline seems to be a cultural issue for many students, but last minute 'coding marathons' simply mean —besides a mediocre outcome—a lost opportunity to learn, through the practice, the benefits of applying Software Management Principles.

IV. Introduction to Agile: Course Structure.

To address the methodological mistake #1, we propose the rearrangement of the 16-week course activities in two big phases: the first, for theoretical/technical foundation phase, and the second, a hands-on agile principles interiorization phase as depicted in Fig. 3 and detailed in Fig. 4. The theoretical/technical foundation phase, which takes approximately two weeks, aims to develop the minimum required skills of software design principles, configuration management, and the project's technology stack. In the remaining six weeks, after evaluating the skills mentioned above and forming the SCRUM teams, three SCRUM sprints are performed.

The theoretical/technical foundation phase follows a problem-based approach, where software design concepts

are deepened through hands-on software refactoring laboratories, gradually including software engineering and management practices (configuration administration). For example, the third-week laboratory [1] deepens in S.O.L.I.D. principles and GoF (Gang of Four) behavioral patterns, introducing at the same time elements of test-driven development and test design techniques. The more advanced -and latest- laboratories, by following the same approach, aims to the development of a proof of concept of the architecture, application stack, and configuration management environment expected to be used in the final project. In order to achieve this, each laboratory is aimed at the development of a different layer (presentation/security, middleware, and persistence), with a final integration exercise [2], where the benefits of loosely-coupled/layered architectures are demonstrated.

The agile principles interiorization phase, on the other hand, is focused on the development of agile skills through a near-real SCRUM process, with a real problem and real stakeholders. In order to achieve a more realistic working environment—where people usually are focused on fewer tasks simultaneously—, during this phase the students are allowed to devote 10.5 hours/week out of the 12 hours/week (defined by the course's academic credits) to the project. The remaining 1.5 hours are devoted to reflection activities to introduce and compare alternative software development life cycles (Waterfall, UP, Lean, etc), activities that are highly benefited from the student's new perspective as active members of an agile (SCRUM) team. To create a bank of real projects, inspired by the experience documented by Viljan Mahnic [5], we perform a survey of information systems needs within several institutional administrative departments and academic programs. Such needs were prioritized in accordance with the feasibility implementation taking into account the technical limitations of the architecture and technology stack selected for the course. Each selected, and the first version of the product vision and product backlog are defined between the stakeholder and semester, therefore, a new project and stakeholder (a representative of the selected program or department) are the product owner, a role played by the teacher in question.

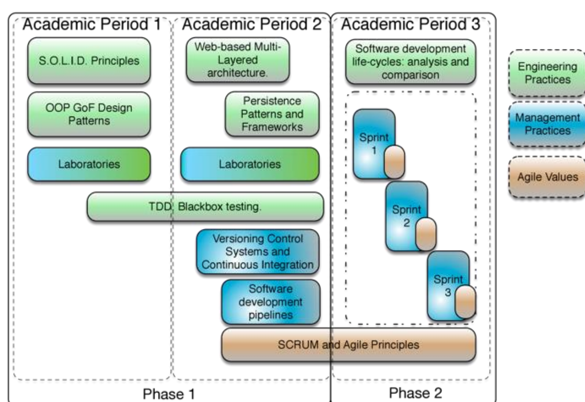


Fig.3. First Software Engineering Course: Proposed Course schedule and Phases distribution.

Agile games are the key activities that link the two phases of the course described above. Although there are several variations of such games, the Scrum4Lego workshop [10] has been repeatedly used in recent years given the excitement and engagement observed (Steghöfer et al. [16]) amongst students during its application.

Week	Theoretical session	Lab Session
1	Introduction	Study case: GoF and SOLID
2	GoF patterns (cont)	Study case: GoF and SOLID.
3	GoF patterns and TDD	Study case: Test case design techniques.
4	GoF patterns and TDD. Introduction to Internet and Web concepts.	Introduction to Web development.
5	MVC Web frameworks.	Introduction to Web development.
6	MVC Web frameworks.	**EXAM**
7	Configuration Management Concepts. Continuous Integration and Continuous Delivery.	Study case: Web development with CI / Versioning Control System (GIT)
8	Persistence patterns. ACID principles.	Study case: GoF and persistence patterns. Object-relational mapping.
9	Web Applications Layered Architectures. Persistence Patterns and Frameworks.	Study case: Persistence Frameworks, Persistence Testing. Layers integration , Continuous Integration and Delivery.
10	Agile SDLC. Agile games.	Agile games - activity: user stories, planning poker, product and sprints backlogs.
11	Agile SDLC and SCRUM. Stakeholder introduction and product backlog overview.	**EXAM**
12	Sprint 1 - Sprint planning.	Daily Scrum. In-class technical support.
13	Daily Scrum. SDLC Analysis and Comparison.	Sprint review/Sprint retrospective/Sprint 2 planning
14	Daily Scrum. SDLC Analysis and Comparison.	Daily Scrum. In-class technical support.
15	Daily Scrum. SDLC Analysis and Comparison.	Sprint review/Sprint retrospective/Sprint 3 planning
16	Daily Scrum. SDLC Analysis and Comparison.	Daily Scrum. In-class technical support.
	Final EXAM	Sprint Review, Sprint Retrospective. Stakeholder evaluation.

Fig.4. First Software Engineering Course: detailed schedule and contents.

With this activity, shown in Fig. 5, the students are exposed, for a first time, to a small-scale SCRUM sprint, where the goal is to build a Lego city based on a product vision and a set of user stories. Activities such as user stories' estimation and prioritization, sprint planning, sprint retrospective and sprint review—previously presented in class—, are experimented through an entertaining but meaningful group dynamic. For this kind

Fig.5. Agile games: SCRUM4Lego activity in a classroom specially designed for collaborative work.

[1] https://github.com/PDSW-ECI/GoF-Testing-BehavioralPatterns-CADTool_Rotation

[2] https://github.com/PDSW-ECI/MyBatisGuice_Integration_VideoRental

Copyright © 2018 MECS

of activities, our program has designed what we call 'the interactive classroom' (also shown in Fig. 5), a classroom that encourages teamwork, with a special distribution of roundtables, whiteboards, and computers (one for each team).

V. Continuous Delivery Pipelines and Evaluation Metrics.

Methodological mistakes 2, 3, and 4 are all, at the end, related to a lack of individual and continuous progress tracking. To address these issues we propose the inclusion of the concept of Continuous Delivery pipeline (CDP), a popular software strategy that enables the delivery of new features to users as quickly and efficiently as possible, by including elements that also enable a continuous delivery of progress and evaluation metrics. The proposed production pipeline is depicted in Fig. 6, and can be summarized with the following activities (not necessarily sequential), assuming an in-progress sprint (after sprint planning):

1. Based on the selected user stories for the current sprint, the development team (students) start creating test cases and software artifacts and committing them to a central repository.
2. When a piece of code is committed to the repository, a continuous integration platform applies the test cases to the developed artifacts and performs a static code quality analysis with PMD (identifying common coding bad practices) and code coverage reports (measuring, partially, test cases quality).
3. The teaching assistant (and sometimes the teacher himself/herself) assumes the role of quality assurance advisor and periodically performs more advanced code quality analysis. The quality assurance advisor receives, as review criteria, our own database of hard-to-find—but common—bad design and coding practices, like bad separation of concerns through layers, or the inefficient use of computational resources. When such poor practices are identified, an 'issue report' is included in the GitHUB repository.
4. The development team, through the continuous integration platform and the versioning control system, automatically receive all the reports mentioned above in order to address all the related problems in the following sprints.
5. When the software artifacts pass all the tests, the continuous integration platform automatically deploys a new version of the application in the cloud, making it available as a working software for the teacher and the stakeholder.
6. The deployed application is used during the sprint reviews with the stakeholder and the product owner.
7. The teacher generates statistics from the versioning control system (GIT) in order to identify: a) The consistency of teams' sprint backlogs, and b) teams or team members with irregular commitment rates.

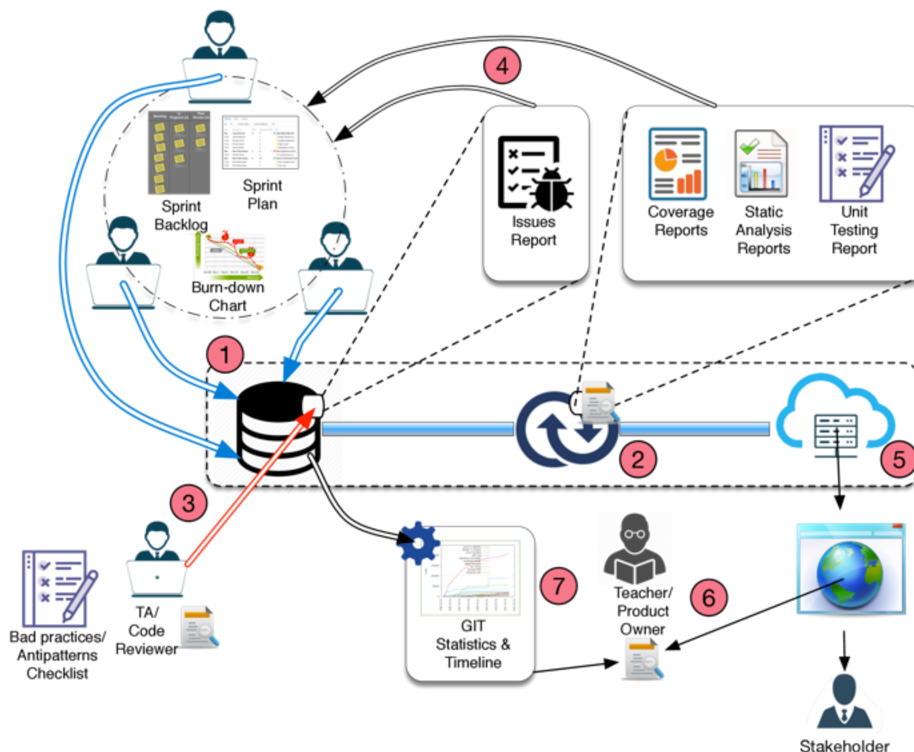


Fig.6. An overview of the Continuous Delivery Pipeline introduced in the first Software Engineering Course.

A. Evalation Metrics

On the basis of the Continuous Delivery Pipeline described above, we proposed the evaluation criteria for early problems detection (process) and final evaluations (product) listed in Table 1.

Table 1. Evaluation Criteria

Criteria	Metrics
Sprint Performance	Planned story points vs Accepted Story Points.
Internal Quality	Final reports of the static code analysis tools, and the rate of solved/reported issues opened by the Quality Assurance Advisor or the Teacher, as shown in Fig. 7.
External Quality / Usability	Stakeholder: User interface Usability.

Fig.7. Issues Tracking through GITHub's interface.

B. Process

Application of Continuous Integration Principles.

One of the principles of CI practice states that "Everyone commits to the baseline every day". This is a key principle when it comes to work-distribution problems previously discussed, as its validation ensures the identification the real contribution of each team member. Unlike the approach followed by authors like Matthies et al [12], where CI metrics were calculated automatically using GIT's history, we propose an automatic generation of timeline graphics of the individual contributions (commits/lines of code vs time). The reason for this is that graphical information is easier to follow and get analyzed by the teams (when compared to a single result of an automated calculation), and gives more discussion elements with the teacher at early stages of the sprints.

Figs. 8 and 9 show two opposing examples of GIT's committing timelines. The first, with a similar, regular slopes from the beginning of the project; and the second, with extremely high slopes near the end of the project and heterogeneous contributions through the time (even with a team member that reports only a single, insignificant, contribution).

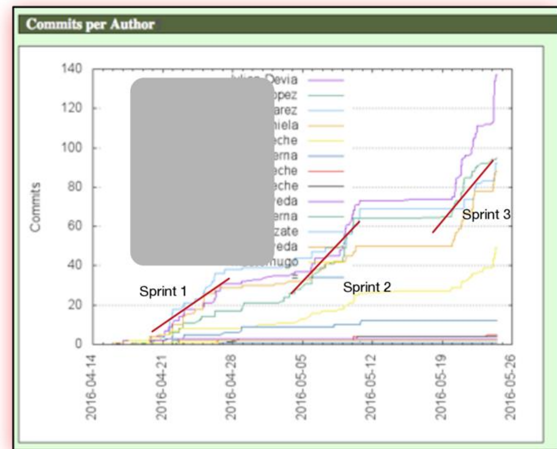


Fig.8. A good example of the expected source code repository timeline for a team committed with Continuous Integration, and with fair work distribution. Although there are duplicate authors (due workstations misconfigurations, a problem we expect to solve), all the team members show a similar progress slope through the time.

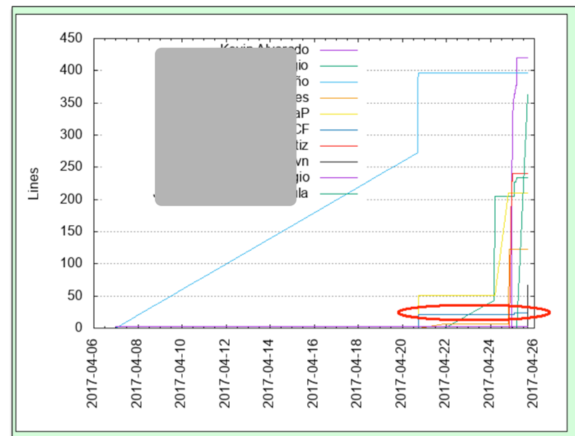


Fig.9. This source code repository timeline shows (1) a lack of Continuous Integration -most of the code was committed to the baseline near the end of the project-, and (2) some team members with minimal contributions (red circle).

Application of Test-Driven Development, and Test Design Principles.

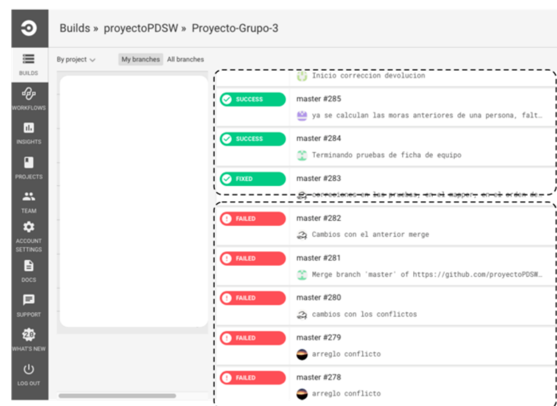


Fig.10. Automatic project building performed by Circle.CI, the Continuous Integration Platform chosen for the course. Each time a new commit is performed, the platform executes unit tests, marking with green the absence of errors, and with red its presence.

In order to evaluate the application of the test-first principle, it is validated that (1) the timeline of the tests begins before the timeline of the target source code, and (2) the CI (Continuous Integration) Platform -see Fig. 10- reports failed test cases from the earliest builds.

Sprint planning and progress tracking for estimation skills improvement.

Keeping tracking of the time spent on each sprint is essential to identify the improvements in estimation accuracy as more and more Sprints are performed. To encourage this practice, at certain points of each Sprint, the correspondence between the provided Sprint backlog template shown in Fig. 11 (tasks assignments and time reports) and the GIT logs is verified.

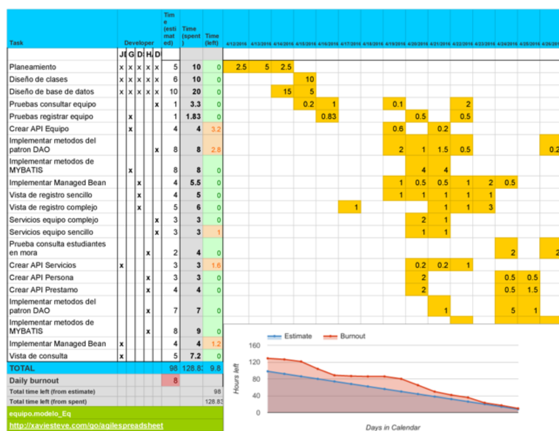


Fig. 11. Open source Sprint Backlog format with planning and time tracking elements. The Sprint-burndown chart is generated automatically.

VI. PRELIMINARY RESULTS

This methodological approach has been applied for three consecutive semesters, with a total of 57 students. Given the size of the students' population at the time of the definition of this pedagogical approach, it was impossible to create an experiment with a second 'control' group. However, preliminary results from the final products point of view, and the students' feedback are presented below.

A. Course Outcomes

A selection of the best projects (from each semester) is available at *https://github.com/LIS-ECI*, including links to their CI environments and Cloud-based hosting. Table 2 shows part of the history of 'real projects' created through course's production lines.

Table 2. Projects created between 2016 and 2017.

Semester	Project and Product Backlog Link	Stakeholder
2017-1	Scheduling System for the Graduate Programs of Project's Unit *https://goo.gl/Fr6DD7*	Projects Unit Director
2016-2	Graduates Association System: *https://goo.gl/vdh43A*	Graduates Association Leader
2016-1	Electronic Engineering Lab: *https://goo.gl/oSQcfH*	Dean of Electric Engineering program.

When we asked potential stakeholders to take part in the course process, in order to motivate them we offered them the possibility of achieving a good base-line product for the needs of their departments or programs. Table 3 presents the quantitative product evaluations given by the respective stakeholder each semester, and the number of projects that he/she considered to be good candidates—because of their quality— for a baseline of a real product. Although there are cases of final products with very low grades in the earliest version of the course, it is worth mentioning that the overall evaluations improved over time. Likewise, it is noteworthy that two projects from 2016-1 and 2016-2 are now officially baselines projects for the products defined for the electrical engineering program and the graduate association, waiting for further funding to continue their development (hopefully, with the collaboration of the students).

Table 3. Final evaluation of the products by the Stakeholder

Semester	Projects count	Final grades	Baseline candidates
2016-1	4	3.0, 3.8, 4, 2.9	50%
2016-2	3	4.1, 4.4, 4.8	66%
2017-1	4	3.5, 3.8, 4.3, 4.3	50%

Finally, Table 4 shows an average evaluation of two key practices: test-driven development and continuous integration, through the aforementioned metrics. These results show an improvement of such practices adoption over the time, which may be related to the continuous evolution of course materials and study cases:

Table 4. Final evaluation of key practices

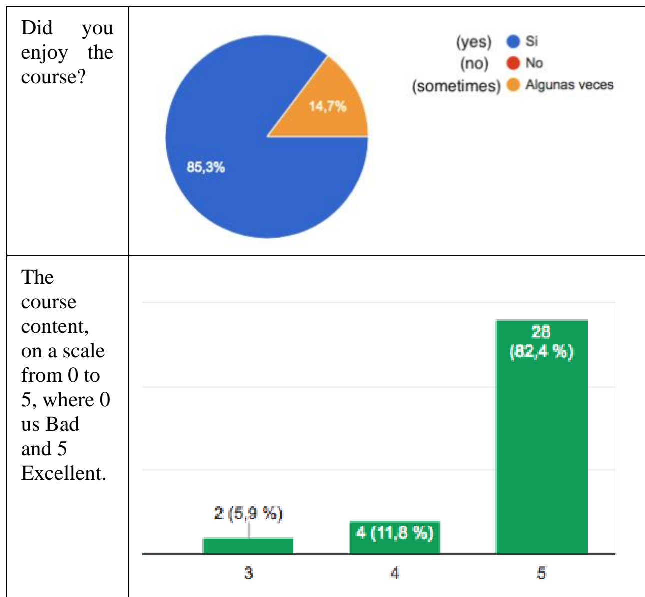
Semester	TDD	CI/CD	Product
2016-1	4.3	3.6	3.9
2016-2	4.3	4.0	4.0
2017-1	4.5	4.2	3.9

B. Students Survey

Of the 57 students, 34 agreed to take part in an anonymous, online survey. According to the results, there has been an excellent perception of the course, as can be seen in Table 5.

Copyright © 2018 MECS

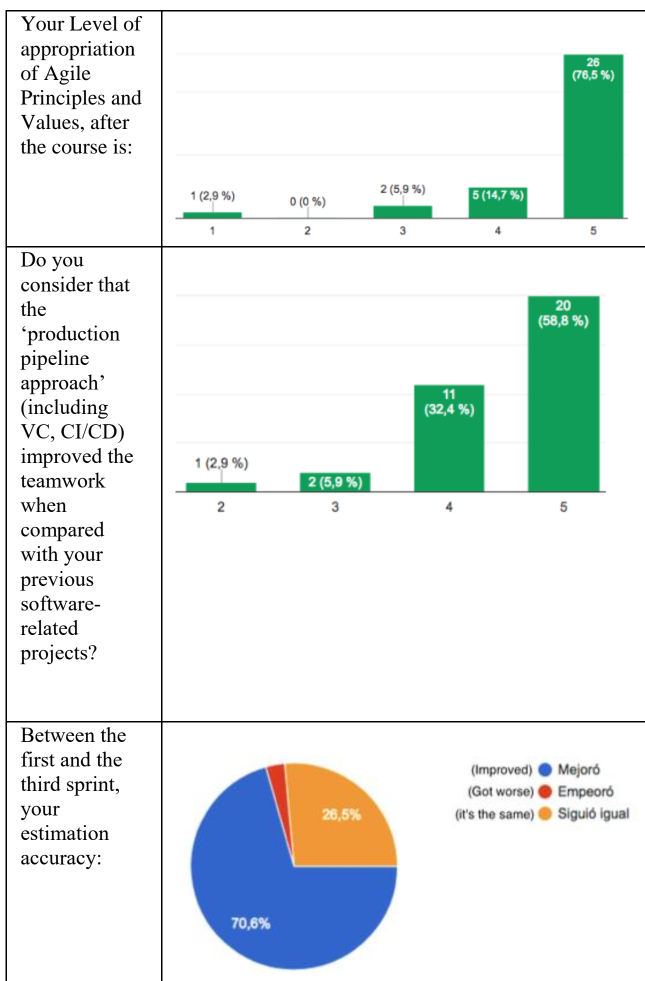
Table 5. Global perception of the course.



Question	Responses
Did you enjoy the course?	(yes) Si: 85,3% (no) No (sometimes) Algunas veces: 14,7%
The course content, on a scale from 0 to 5, where 0 us Bad and 5 Excellent.	3: 2 (5,9 %) 4: 4 (11,8 %) 5: 28 (82,4 %)

On the other hand, most of the students recognized the acquisition of new skills and the appropriation of key principles and values, as shown in Table 6.

Table 6. Course outcomes perception.



Question	Responses
Your Level of appropriation of Agile Principles and Values, after the course is:	1: 1 (2,9 %) 2: 0 (0 %) 3: 2 (5,9 %) 4: 5 (14,7 %) 5: 26 (76,5 %)
Do you consider that the 'production pipeline approach' (including VC, CI/CD) improved the teamwork when compared with your previous software-related projects?	2: 1 (2,9 %) 3: 2 (5,9 %) 4: 11 (32,4 %) 5: 20 (58,8 %)
Between the first and the third sprint, your estimation accuracy:	(Improved) Mejoró: 70,6% (Got worse) Empeoró (it's the same) Siguió igual: 26,5%

Students also reflected on the benefits of the proposed educational approach in their answers to open questions asked.

Question(translated):

What is the most important knowledge or skill you acquired in this course for your professional life?

Answers (translated):
"Working in big groups and getting used to appropriate frameworks"
"Everything. In this course everything is important. Mostly living a Scrum Process, and patterns application"
"The different methodologies that exist for a project development… View a software product from a Client point of view".

Additionally, according to the following responses, although there are still specific teamwork problems, students recognized the last third of the course as a realistic and fruitful experience:

Question(translated):

In general, what advantages or disadvantages did you find in dedicating most of the last third of the course to a real SCRUM process?

Answers (translated):
"It allows us to know what a real work environment is like."
"Advantages: experiencing an agile software development method (the assigned project). Disadvantages: the number of assigned story points. Sometimes these didn't correspond to the real required work."
"Working in groups with members who did not devote time to the project."
"Advantage: ... the sprint reviews were pretty good because these allowed us to take actions in our groups, evaluating what worked and what needed to be improved."
"Advantage: completing a real, high-level project, identifying the concepts applied through the course."
"When team members don't work at the same speed it is sometimes impossible to finish tasks successfully."
"Disadvantages: selfish teammates. Advantages: a better understanding of the concepts, given that when putting them in practice, it is easier to 'digest' them better."

VII. Conclusions and Future Work

It is already well known that software engineering students tend to pay more attention to technical elements than to methodological ones. This study finds that the last segment of the course dedicated to a realistic and more focused process—after all technical elements have been introduced—, could be more useful as they enable students to experience, appreciate, and appropriate software methodologies. The inclusion of real problems and real stakeholders in the course process drastically improves motivation and, therefore, the quality of the process the final products. It is noteworthy that there has been a high rate of well-scored (by teachers and real stakeholders) software projects that were created by students on early stages of their careers (in their sixth semester of studies, on average). This study also shows a new point of view of the concept of Continuous Delivery Pipeline for educational purposes in software engineering

Copyright © 2018 MECS

education. In the presented study case, the frequent feedback (regarding teamwork and product quality) provided by such pipelines drastically improved the identification of the methodological mistakes described, including a reduction in team members procrastination, and more frequent discussions –and reflections—between the teacher and the students (team members), at early stages of each sprint, when coding and design issues are detected. Planned future work includes a deeper review and integration of emerging platforms like ScrumLint [17] and INGinious [18]. With the former, we hope to automate the grading of agile practices and to use the scores obtained as complementary feedback in our continuous delivery pipeline. The expectation for the latter is to improve the theoretical/technical foundation phase described before by automating part of the evaluation/feedback process for hands-on laboratories.

REFERENCES

[1] Orit Hazzan and Yael Dubinsky. Teaching a software development methodology: the case of extreme programming. In *Software Engineering Education and Training, 2003.(CSEE&T 2003). Proceedings. 16th Conference on*, pages 176–184. IEEE, 2003.

[2] Joe Bergin, James Caristi, Yael Dubinsky, Orit Hazzan, and Laurie Williams. Teaching software development methods: the case of extreme programming. In *ACM SIGCSE Bulletin*, volume 36, pages 448–449. ACM, 2004.

[3] Görel Hedin, Lars Bendix, and Boris Magnusson. Teaching extreme programming to large groups of students. *Journal of Systems and Software*, 74(2):133–146, 2005.

[4] Jan-Philipp Steghöfer, Eric Knauss, Emil Alégroth, Imed Hammouda, Håkan Burden, and Morgan Ericsson. Teaching agile: addressing the conflict between project delivery and application of agile methods. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 303–312. ACM, 2016.

[5] Viljan Mahnic. A capstone course on agile software development using scrum. *IEEE Transactions on Education*, 55(1):99–106, 2012.

[6] Martin Kropp and Andreas Meier. Teaching agile software development at university level: Values. *Management, and Craftsmanship, CSEE&T*, 2013.

[7] Václav Rajlich. Teaching developer skills in the first software engineering course. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 1109–1116. IEEE, 2013.

[8] Christiane Gresse Von Wangenheim, Rafael Savi, and Adriano Ferreti Borgatto. Scrumia an educational game for teaching scrum in computing courses. *Journal of Systems and Software*, 86(10):2675–2687, 2013.

[9] Adler Diniz De Souza, Rodrigo Duarte Seabra, Juliano Marinho Ribeiro, and Lucas E da S Rodrigues. Scrumi: a board serious virtual game for teaching the scrum framework. In *Proceedings of the 39th International Conference on Software Engineering Companion*, pages 319–321. IEEE Press, 2017.

[10] Maria Paasivaara, Ville Heikkilä, Casper Lassenius, and Towo Toivola. Teaching students scrum using lego blocks. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 382–391. ACM, 2014.

[11] Martin Kropp, Andreas Meier, and Robert Biddle. Teaching agile collaboration skills in the classroom. In *Software Engineering Education and Training (CSEET), 2016 IEEE 29th International Conference on*, pages 118–127. IEEE, 2016.

[12] Christoph Matthies, Thomas Kowark, Matthias Uflacker, and Hasso Plattner. Agile metrics for a university software engineering course. In *Frontiers in Education Conference (FIE), 2016 IEEE*, pages 1–5. IEEE, 2016.

[13] Miroslaw Ochodek. A scrum-centric framework for organizing software engineering academic courses. In *Towards a Synergistic Combination of Research and Practice in Software Engineering*, pages 207–220. Springer, 2018.

[14] Ivar Jacobson, Grady Booch, James Rumbaugh, James Rumbaugh, and Grady Booch. *The unified software development process*, volume 1. Addison-wesley Reading, 1999.

[15] Andrea A Janes and Giancarlo Succi. The dark side of agile software development. In Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software, pages 215–228. ACM, 2012.

[16] Jan-Philipp Steghöfer, Håkan Burden, Hiva Alahyari, and Dominik Haneberg. No silver brick: Opportunities and limitations of teaching scrum with lego workshops. *Journal of Systems and Software*, 131:230–247, 2017.

[17] Christoph Matthies, Thomas Kowark, Keven Richly, Matthias Uflacker, and Hasso Plattner. Scrumlint: identifying violations of agile practices using development artifacts. In *Cooperative and Human Aspects of Software Engineering (CHASE), 2016 IEEE/ACM*, pages 40–43. IEEE, 2016.

[18] Guillaume Derval, Anthony Gego, Pierre Reinbold, Benjamin Frantzen, and Peter Van Roy. Automatic grading of programming exercises in a mooc using the inginious platform. *European Stakeholder Summit on experiences and best practices in and around MOOCs (EMOOCS'15)*, pages 86–91, 2015.

Author's Profile

Héctor F. Cadavid (MSc) is an Assistant Professor in the Department of Computer and Systems Engineering, Escuela Colombiana de Ingeniería, Bogotá, Colombia. Currently, he is the leader of the Software Engineering Research Center in his department and has been continuously involved in interdisciplinary projects with several research groups including GIMECI, ECITRONICA, CTG-Informática, NASA – LFM (Langley Formal Methods) and Unesco.

How to cite this paper: Héctor F. Cadavid, "Continuous Delivery Pipelines for Teaching Agile and Developing Software Engineering Skills", International Journal of Modern Education and Computer Science(IJMECS), Vol.10, No.5, pp. 17-26, 2018.DOI: 10.5815/ijmecs.2018.05.03

Copyright © 2018 MECS