Journal of Education and Vocational Research
Vol. 3, No. 9, pp. 280-290, Sep 2012 (ISSN 2221-2590)

Using Agile Practice for Student Software Projects

[*]Fernando Almeida[1], AntónioCandeias[2]
[1]University of Porto, DEEC/DEI, Portugal
[2]Higher Institute of Gaya, ISP Gaya, Portugal
[*]almd@fe.up.pt

Abstract: Agile methodology as a relatively new approach to software engineering is becoming more popular in both industry and academia. Learning agile software development methodologies will unquestionably increase the capabilities and competences of our students as entry-level software engineers. However, how agile methods and techniques should be taught at the undergraduate level in additional to traditional approaches is still being debated. This study was conducted on a student-programming project, with sample size of 23 students from the Informatics Engineering course. The Scrum methodology was adopted and 28 user stories and 4 sprints were created. The results indicate a significant impact on students' skill improvement and let them to have the first contact with real projects and clients. Besides that, the students agree that the adoption of the Scrum methodology helped them to improve the participation and collaboration. However, some issues were also detected in terms of communication and tasks planning. Therefore, we proposed some polities that could help and boost the software development process inside a classroom.

Keywords: *Agile Software, Software Engineering, Software Development Methodology, Teaching Software, Cooperative Learning*

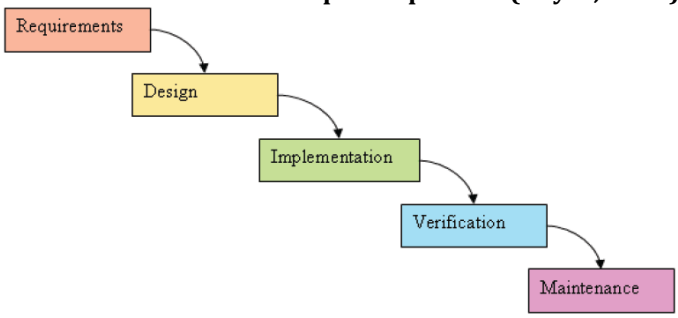
1. Introduction

The field of software development has been characterized by the constant introducing of new methodologies. Indeed, in last twenty five years, a large number of different approaches to software development have been proposed, of which only few have survived to be used today (Jeffery & Scott, 2002). As the industry learned more about developing software, certain techniques for managing and predicting the cost of software development projects came into use. The methodology that has dominated software development projects for decades is called "waterfall." Winston Royce (1970) coined the term in 1970 to describe a serial method for managing software projects through five stages:

- Requirements – during this phase research is being conducted which includes brainstorming about the software, what is going to be and what purpose is it going to fulfil;
- Design – if the first phase gets successfully completed and a well thought out plan for the software development has been laid then the next step involves formulating the basic design of the software;
- Implementation – in this phase the source code of the program is written;
- Verification – at this phase the whole design and its construction is put under a test to check its functionality;
- Maintenance – Integration, maintenance and management is needed to ensure that the system will continue to perform as desired.

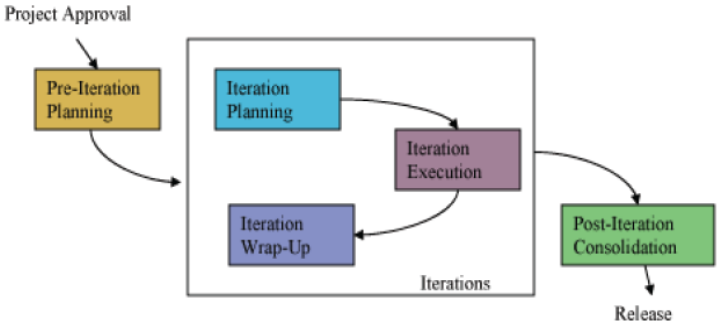
Through the above-mentioned steps, it is clearly shown that the waterfall model was meant to function in a systematic way that takes the production of the software from the basic step going downwards towards dealing just like a waterfall, which begins at the top of the cliff and goes downwards but not backwards. This situation is illustrated in figure 1.

Figure 1: Waterfall software development process (Royce, 1970)



According to Deeb (2010) when done well the waterfall method is excellent for large projects and there are no surprises when the application is finally delivered as all features and even the appearance of the application has been fully specified and understood by the future users of the system. However, if the requirements phase is done badly (and this is often the case when the business confuses shoddy requirements with faster progress) the waterfall method delivers failure as the end result will only ever be as good as the specifications. Adoption of waterfall has helped drive down the failure rate of software development projects, but even with rigorous project management and processes, a full 70 percent of software projects using this methodology fail to meet their objectives (Deeb, 2010) (Serena, 2007). To avoid this situation, organizations have tried to cut the failure rate by insisting on more detail in the requirements and design phases, but this increases the time to market of the solution and typically originates additional delays in the software development process. One of the most important differences between the agile and waterfall approaches is that waterfall features distinct phases with checkpoints and deliverables at each phase, while agile methods have iterations rather than phases. The output of each iteration is working code that can be used to evaluate and respond to changing and evolving user requirements. Waterfall assumes that it is possible to have perfect under-standing of the requirements from the start. However, in software development, stakeholders often do not know what they want and cannot articulate their requirements. With waterfall, development rarely delivers what the customer wants even if it is what the customer asked for. Agile methodologies embrace iterations. Small teams work together with stakeholders to define quick prototypes, proof of concepts, or other visual means to describe the problem to be solved. The team defines the requirements for the iteration, develops the code, and defines and runs integrated test scripts, and the users verify the results. This scenario is illustrated in Figure 2. Verification occurs much earlier in the development process than it would with waterfall, allowing stakeholders to fine-tune requirements while they are still relatively easy to change.

Figure 2: Generic agile development process (Serena, 2007)



This paper analyzes the introduction of the Scrum methodology in the software learning process. Section II makes a revision of literature in terms of agile software principles and computer science education process. Section III details the methodology adopted in this research study. Further, Section IV presents the main results and provides a discussion. Finally, Section V draws conclusions.

2. Literature Review

This section includes a brief summary on literature, which was mainly used as the basis for this study. For this study, a specific focus of agile software principles and motivations was given as they provide the stem of this research. Additionally, an analysis of computer science education was also carried to give an overview of the main existing challenges in the software education process at a university level.

The agile software process: The Agile Manifesto (Vijayasarathy & Turk, 2008) stresses the importance of a) people and interactions over processes and tools, b) working software instead of detailed documentation, c) active customer participation and involvement rather than time and effort expended on negotiating contracts, and d) willingness and ability to take on changes over steadfast commitment to a static plan. Agile software development methods including eXtreme Programming (XP), Scrum, Adaptive Software Development and Feature-Driven Development are based on the principles of the Agile Manifesto and geared towards realizing its goals and objectives. In traditional requirements definition, business analysts are conditioned to believe that they can and should define detailed requirements at the beginning of a project. Traditional requirements methodology assumes the following main principles:

- Assumes that the customer can definitely know, articulate, and functionally define what the system or software should do at the end of the project;
- Assumes that, once documented, the requirements will not change (at least without potential project delays);
- Assumes that the requirements process is confined to a single product owner who sits apart from the development team envisioning the product;
- Does not acknowledge the inherent uncertainty in software development that agile methodologies seek to embrace.

On the contrary, agile project management assumes that the processes required to create high-value working software in today's economy are not predictable: requirements change, technologies change, and individual team member productivity is highly variable. When processes are not static and outcomes cannot be predicted within sufficient tolerance, we cannot use planning techniques that rely on predictability. Instead, we need to adjust the processes and guide them to create our desired outcomes. Agile project management does this by keeping progress highly visible, frequently inspecting project outcomes, and maintaining an ability to adapt as necessary to changing circumstances.

Because agile teams are self-organizing and empowered, the role of "agile project manager" focuses more on leadership than in a traditional development environment. Skills such as facilitation, coaching, and team building become key components for project success. The team leader of an agile project focus less on assigning tasks and managing the plan and more on maintaining the structure and discipline of the Agile team, which includes trusting that through visibility, inspection, and adaptation the team will deliver the desired results. As opposed to traditional requirements gathering, where the Business Analyst (BA) primarily works with the product owner to collect specifications, agile team members from all disciplines are involved in defining project requirements. Technical team members and Quality Assurance (QA) collaborate with the product owner and the BA to develop the project specifications. Both brings their skills and experience into this collaborative process. Increasing the level of interaction ensures the team develops specifications that can be built and tested within the overall project constraints. To effectively deal with scope on an Agile project, specifications must be considered in two dimensions: breadth first and then depth. It is essential that we understand the breadth of what we want to build early in the project. Dealing with the breadth of the solution helps; the team understand scope and cost and will facilitate estimating and release planning. The breadth of a project begins to frame the boundaries of the project and helps to manage the organization's expectations. Looking at the breadth of the requirements is a much smaller investment of time and resources than dealing with the entire depth. The details are most likely to evolve as we progress through the project so defining them early has less value.

Having a solid understanding of the breadth of project requirements early in the lifecycle helps the development team begin to define the set of possible solutions. Once the team has established the breadth of the solutions, it is time to begin incrementally looking at the depth of the solution. The BA will typically take the lead helping the team bring the requirements down to this next level of detail. To incrementally look at the depth of the requirements, the team must abandon traditional notions of the Marketing Requirements Document (MRD), Product Requirements Document (PRD) and the list of "the system shall" specifications. Instead, the team must focus on how the system is going to behave. In general, the feedback from organizations that have implemented agile development is positive. Some of the benefits attributed to agile development are increased productivity, expanded test coverage, improved quality/ fewer defects, reduced time and costs, understandable, maintainable and extensible code, improved morale, better collaboration, and higher customer satisfaction (Abrahamsson et al, 2002). The adoption of agile development has also revealed some challenges such as slow participant buy-in, opposition to pair-programming, lack of detailed cost evaluation, scope creep, reduced focus on code base's technical infrastructure and maintainability, difficulty evaluating and rewarding individual performance, and the

need for significant on-site customer involvement, management support, competent managers and developers, and extensive training (Abrahmsson et al, 2002, Hanssen & Faegri, 2006).

SCRUM Methodology: SCRUM is a popular agile software development methodology. It is iterative, highly collaborative and an effective way of quickly delivering useful software while keeping high quality software. Unlike traditional methods with their protracted processes and lengthy documentation, SCRUM focuses on responding to emerging requirements. The key principles of SCRUM are:

- Collaboration and communication – face to face meetings, co-located teams or for geographically dispersed teams, extensive utilization of communication tools;
- Product backlog – smallest, workable pieces of functionality that translate into tangible business benefits. According to Sims & Johnson (2011), these items must be deliverable live in short iterations of 1-4 weeks. This backlog is then maintained to deliver against incremental iterations;
- Get to market faster – in other words, working software that's delivered in weeks;
- Stakeholder engagement – agile teams view stakeholders as important and visible members of the team and engage them right from the start and throughout the project (Stuart et al, 2008);
- Capacity management – every aspect of capacity is assessed and agreed before delivering any love software;
- Release planning meetings – planning and scheduling of design iterations is done up-front;
- Coordinated sign-offs – agile teams ensure sign-off at every stage of the software development lifecycle from relevant stakeholders;
- Risk management – the product owner actively monitors the software delivered and ensures that stakeholders give early visibility to managing risks and issues;
- Effective change management – agile teams are welcoming and responsive to changes that stakeholders require.

According to Rawsthorne & Shimp (2011) working with the stakeholders helps ensure that these changes are aligned to the business goals. Scrum for software development came out of the rapid prototyping community because prototypes enthusiasts wanted a methodology that would support an environment in which the requirements were not only incomplete at the start, but also could change rapidly during development (Cohn, 2009). According to Sutherland (2001) and Schwaber (2004) there are number of success stories about Scrum implementation, adoption and acceptability. Ambler has performed survey on agile adoption that was published in June 2008. According to this survey, 69%people said that their organizations are somehow practicing some Agile methodology (Ambler, 2009). Secondary research conducted for this paper reveals that Scrum is rapidly growing methodology within IT organizations. Scrum implementation has massively geared in the last four years and many organizations have been found that have expanded Scrum to company-wide levels. The figure 3 illustrates the basic Scrum process for developing software .At the centre of each SCRUM project is a backlog of work to be done. This backlog is populated during the planning phase of a release and defines the scope of the release. After the team completes the project scope and high-level designers, it divides the development process into a series of short iterations called sprints. Each sprint aims to implement a fixed number of backlog items. Before each sprint, the team members identify the backlog items for the print. At the end of a print, the team reviews the sprint to articulate lessons learned and check progress. According to Akhtar et al., (2010) the Scrum development process concentrates on managing sprints. Before each sprint begins, the team plans the sprint, identifying the backlog items and assigning teams to these items. Teams develop, wrap, review, and adjust each of the backlog items. During the development, the team determines the changes necessary to implement a backlog item. After that, the team then writes the code, tests it, and documents the changes. During wrap, the team creates the executable necessary to demonstrate the changes. In review, the team demonstrates the new features, adds new backlog items, and assesses risk. Finally, the team consolidates data from the review to update the changes as necessary. Following each sprint, the entire team (including management, users, and others interested parties) demonstrates progress from the sprint and reviews the backlog progress. Finally, the team then reviews the remaining backlog and add, removes, or reprioritizes items as necessary to account for new information and understanding gathered during the sprint (Serena, 2007).The main SCRUM concepts include the burn down chart, product backlog, sprint backlog and Scrum master. A concise description of these concepts is presented in Table 1.

Figure 3: Scrum workflow (Hicks & Foster, 2010)

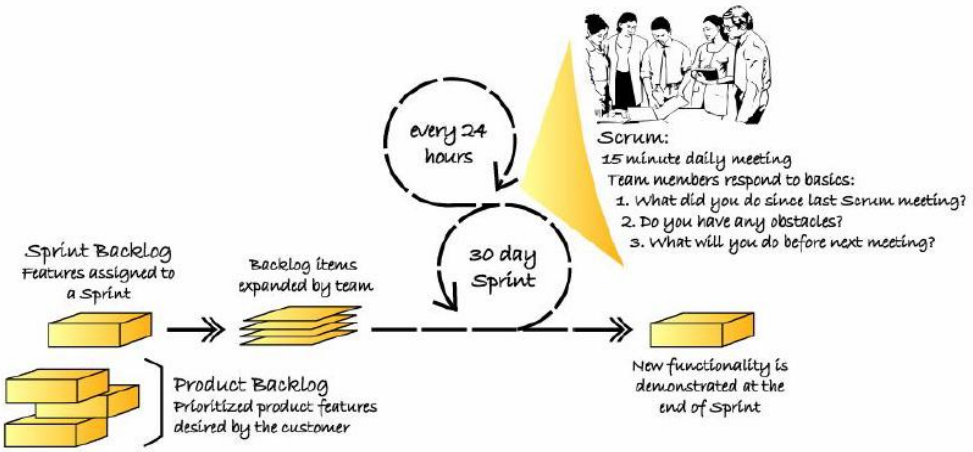


Table 1: Vision of main Scrum concepts (Serena, 2007)

Concept	Description
Burndown chart	This chart, updated every day, shows the work remaining within the sprint. The burndown chart is used both to track sprint progress and to decide when items must be removed from the sprint backlog and deferred to the next sprint.
Product backlog	Product backlog is the complete list of requirements – including bugs, enhancement request, and usability and performance improvements – that are not currently in the product release.
Scrum master	The Scrum master is the person responsible for managing the Scrum project. Sometimes it refers to a person who has become certified as a Scrum master by taking Scrum master training.
Sprint backlog	Sprint backlog is the list of backlog items assigned to a sprint, but not completed. In common practice, no print backlog item should take more than two days to complete. The sprint backlog helps the team predict the level of effort required to complete a sprint.

The software engineering education process: Software engineering (SE) is the profession that creates and maintains software applications applying technologies and practices from computer science, project management, engineering, application domains, and other fields (Wasseman, 1996). IEEE gives a more comprehensive definition. According to IEEE Software engineering is "(1) the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, that is, the application of engineering to software," and "(2) the study of approaches as in (1)" (IEEE, 2012).Over the past three decades, software developers have been educated in traditional ways: undergraduate and graduate programs in colleges and universities, vocational courses and in-house training, and personal initiative in learning new techniques. Most universities now offer undergraduate degrees in computer science, and most provide an extensive selection of software-related courses. These programs typically allow a student to study software design and implementation topics, and they provide a common educational base for entry-level programming positions (Lu & Wang, 2006) (Perera, 2009). For a decade or more, some members of the software education community have advocated under-graduate software engineering degrees separate from computer science. Such programs are intended to provide a better base for a software development career than would a traditional computer science program; the prospects that this will be the case are discussed below. Tomayko and Endres (2002) noted that we are entering in a new era with the introduction of these undergraduate software engineering programs, but they are not yet widespread. As software becomes ubiquitous, the relation between end users and software development is undergoing fundamental changes. Some of these changes have to do with the evolving character of software; others result from increasing pressure for recognized professional credentials. The prevailing model of software development, on which most educational programs is based, involves a team of professional software developers in a single institution working under a well-defined process and product cycle to produce software for a known client and deliver it on known schedule. This closed-shop software development model is increasingly at odds with actual practice.

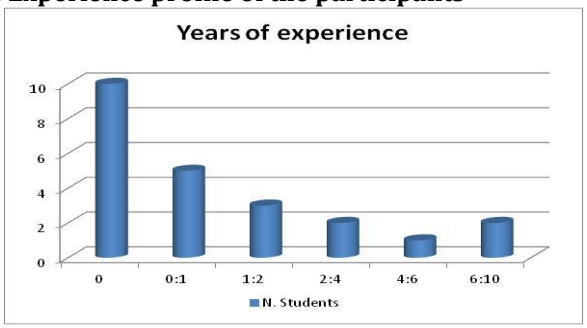
Some of the discrepancies between the closed-shop model and modern software include:

- System requirements emerge as the clients understand better both technology and the opportunities in their own settings, and the clients are intimately involved in this progressive development. This often requires software development to be done concurrently with business reengineering;
- Communities of cooperating volunteers (Buytaert, 2010) are now developing software, especially low-level software. In open-source software, the code is published freely and interested users analyze it and propose changes. Quality arises by an intense, highly parallel social process with rapid feedback rather than by a carefully managed process;
- Software is often developed by creating coalitions of existing resources that are not under control of the software developer (Fowler. 2006). The resources include calculation communication, control, information, and services. Typically, they are often distributed, dynamic, autonomous, and independently managed. They may be modified or decommissioned without notice to users. This open-shop development model (Weiping et al, 2009) is a major departure from the usual closed-shop model, and the uncertainties associated with externally-managed resources require correspondingly more sophisticated analysis;
- Software development is increasingly disintermediated, since it is adapted tailored, composed, or created by its end users rather than professional software developers. These end users need to understand software development in their own terms; they particularly need ways to decide how much faith to have in their creations (Burnett, 2009).
- To respond to these forces, educational institutions must prepare professional software developers to construct and analyse systems that are heavily constrained by nontechnical considerations and that depend on independent distributed re-sources. In addition, professional software developers must learn to create resources that are sufficiently trustworthy to be used and tailored by non-professionals. Therefore, the software development methodology needs to be more dynamic and versatile to adapt to the changes in software products specification and collect feedback from client and end users (Rico & Sayani, 2009). Therefore, agile methodologies appear as an exciting model to be adopted and practically used at software classes.

3. Methodology

The course under study is a one-semester course sequence that encapsulates support decision methodologies and software engineering (SE) practices. This course includes two parallel components. The theoretical part consists of weekly lectures/seminars, reading assignments, and quizzes, which cover core business intelligence models, SE theories (development models and quality assurance) and best practices. The development component involves building a software product for a real customer. Students are required to work at least 8 hours each week on their projects. The goal of this course is to introduce fundamental software engineering concepts and provide an opportunity for students to apply their knowledge in real software development and project management. The projects are real in that the requirements are from real customers for solving real world problems. Project ideas are first solicited from industry partners, such as software companies, non-profit organizations, schools, and individuals who have sponsored projects before. Proposals are evaluated based on their relevance to the faculty's mission statement and the project scope. Each selected proposal is converted into a project overview statement to be handed to the students. All projects are team-based (5 to 7 students). Each team is formed based on the members' SE strength and personalities. After a team is assigned a project, they become responsible for making all decisions in all life-cycle activities, including scheduling meetings with stakeholders. A faculty member, typically the professor, is also assigned to each team as a mentor, who meets with the team regularly to keep the students accountable by viewing their time logs, checking project progress, and asking questions. The student sample was composed by 23 participants. Nearly 45% had no experience in software development and approximately one-third had until 2 years of experience in IT. Only around 20% of the students had more than 2 years of experience. The experience average was 1.48 years. This situation is depicted in Figure 4.

Figure 4: Experience profile of the participants

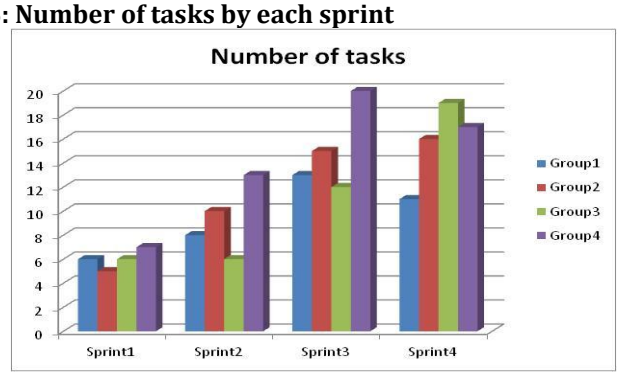


The agile software methodology adopted was based in the SCRUM methodology. Development tasks are divided among several teams, each consisting of approximately 5 to 7 members, one of whom also plays the role of the *scrum master* who organizes the group. Teams implement product features in a series of four-to-six week *sprints*, each of which culminates in a working prototype. At the end of a sprint, the development team and management hold a period of few hours planning meeting to decide what tasks should be carried out during the next sprint. This set of tasks is called the *sprint backlog*. Every day during a sprint, teams hold a *scrum meeting*, in which each team member answers three questions: (1) What did you do since the last scrum meeting?; (2) Do you have any obstacles?; (3) What will you do before the next scrum? For any issues that cannot be immediately resolved, or if a team member seems to be having trouble making progress (whether they realize it or not), the scrum master sets up separate meetings or rakes whatever other actions is appropriate. Scrum meetings should last no more than 15 minutes.

4. Results and Discussion

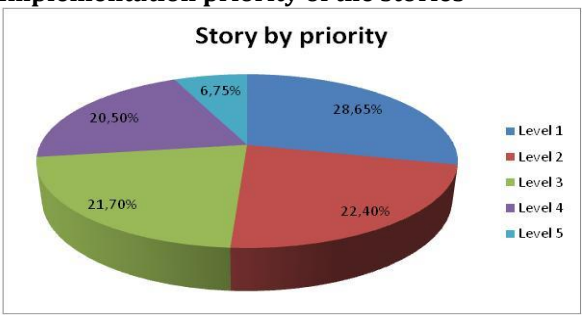
The class of 23 participants grouped together into four teams (two teams with 5 members, one with 6 members, and the last one with 7 members). Initially all teams had a session with clients and the teacher to firstly analyze the project requirements and answer some doubts about the SCRUM methodology. Additional documentation about the SCRUM methodology was given to students. Subsequently, in the next class, a structured brainstorm conducted in class helped prepare the participants for their requirements phase (product backlog) and, subsequently, establishment of the sprint#1 user stories. Figure 5 shows the number of tasks assigned to each sprint by each group. The group 1 and group 2 are composed by 5 members; group 3 by 6 members; and group 4 by 7 members.

Figure 5: Number of tasks by each sprint



These user stories were also organized in five groups according to their priority. The level 1 has very high priority tasks (must be implemented); level 2 has high priority tasks (should be implemented); level 3 has moderate priority tasks; level 4 has low priority tasks (could be implemented); and level 5 has not important tasks. This information was given by clients in the beginning of each user story. The distribution of tasks by each level is depicted in figure 6.

Figure 6: Implementation priority of the stories



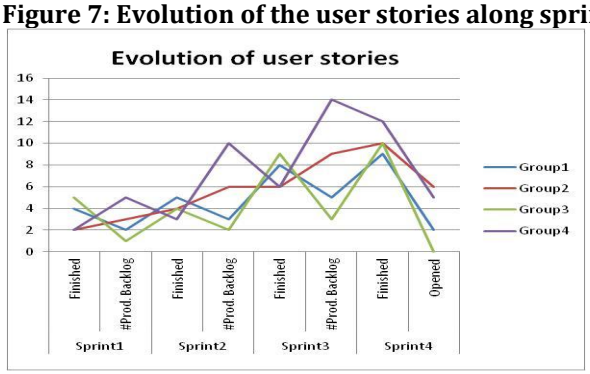
We can also analyze in more detail the number of tasks concluded in sprints by each group. This information is shown in table 2.

Table 2: Analysis of each sprint

	Sprint1		Sprint2		Sprint3		Sprint4	
	Finished	#Prod. Backlog	Finished	#Prod. Backlog	Finished	#Prod. Backlog	Finished	Opened
Group1	4	2	5	3	8	5	9	2
Group2	2	3	4	6	6	9	10	6
Group3	5	1	4	2	9	3	10	0
Group4	2	5	3	10	6	14	12	5

Attending to the table 2, we can realize that group 3 was the only group that implemented all the product backlog items. In last stage (sprint 4) the group 1 had 39, 29% of the stories; group 2 had 57, 14%; group 3 had 35, 71%; and group 4 had 60, 71%. Therefore, it was important to have a uniform distribution of user stories among sprints, to avoid situations of having more than 50% of tasks in last sprint, as it happened with group 4. Typically, in an enterprise environment, we would consider a new sprint iteration to complete the missing tasks. However, in a classroom environment this was not possible and, therefore, the missing tasks were only concluded by students after the final evaluation. This solution had the agreement of all involved companies. A comparative graphical analysis was also made to understand better the behavior of each group. This situation is illustrated in figure 7.

Figure 7: Evolution of the user stories along sprints



Based in figure 7 we can easily realize the different pattern behaviour followed by group 4, particularly in sprints 2 and 3. The high number of 10 product backlog tasks in sprint 2 was a signal that a milestone delay could occur on the deliverable. Curiously, group 4 had 7 members, which could initially suppose that the amount of work needed to be performed by each student would be lower. However, the heterogeneous level of IT competences of the members and the need of a better coordination may justify this peculiar behaviour. Consequently, we analyzed the linear correlation between the number of tasks by sprint and the number of backlog items by iteration. We considered two scenarios: 1) correlation among the first three sprints; 2) correlation among all sprints. The results are depicted in table 3.

Table 3: Linear correlation between number of tasks and product backlog

	Scenario1	Scenario2
Group1	0,999	0,682
Group2	1	0,806
Group3	0,866	-0,439
Group4	0,994	0,619

These results demonstrate that the number of product backlog items tend to increase with the number of tasks for the scenario 1. However, the same situation does not appear in the scenario 2, when we consider all the sprints. Looking to the table 2, we can easily realize that the number of remained tasks in sprint 4 is lower than in sprint 3. This situation can has two causes:

- The students had the care to implement the majority and the most complex tasks in sprint 3. This is visible for group 1 that implemented 46, 43% of the tasks in sprint 3. The same happens for group 3 that implemented 42,85% of the tasks in this stage;
- The students made an increased effort to finish the most number of tasks in the last sprint compared to the previous, because they knew that after the sprint 4 the prototype would be evaluated by the teacher and customer.
- A survey was done with the 23 students who adopted the Scrum methodology for their project. The same survey was done before and after the study. In the survey, eight questions were asked with one of *Yes*, *No*, and *Don't Know* options to be selected as the answer. The responses are shown in the table 4.

Table 4: Survey results about the student perception on using Agile Process

Questionnaire Statements	Before the study			After the study		
	Yes	No	Don't know	Yes	No	Don't know
Scrum helps in the coordination of the team?	8	2	13	12	4	7
Scrum helps to share knowledge?	10	1	12	18	0	5
Scrum helps to improve my skills?	2	3	18	20	1	2
Scrum helps to improve my GPA?	0	2	21	15	5	3
Scrum improves software quality?	6	1	16	15	3	5
Scrum decreases time to market?	9	2	12	12	6	5
I can be part a part of industrial Agile team?	0	18	5	7	8	8
I Recommend Agile practice for others?	2	3	18	18	1	4

These results demonstrated the high motivation of the classroom to learn and adopt agile practices. Before the study, students had the main idea that Scrum has several advantages namely in terms of knowledge share and time to market. On the contrary, all students had several doubts in terms of their competences to be part of an agile industrial team in the short-term. This feeling and high "don't know," answer are justified essentially due to low experience of students in implementing IT software. After the study, students confirmed that Scrum helped them to improve their IT skills and they recommend agile practices to other classes. On the contrary, there are still a significant number of students that believe or have doubts that are ready to participate in an industrial agile team, even if the percentage of students in this situation decreased to 69, 57%.On the same survey, we asked students to freely describe the most positive and negative impact of the course. The most positive comments includes design simplicity, interaction among the group, concurrent programming, use of version control systems (SVN), responsive and committed customers. On the other side, around 15% of the students related some issues in terms of programming knowledge, estimation of user stories, too much testing and no big picture (of the system design) before the end. According to the feedback received, it stands clear the importance of a green-field development and the adoption of a programming language acknowledged by the students. It is a good decision to let students have unit testing, test-driven development, continuous integration; refactoring, responsible software engineering practices early on the program. On the same way, it should be encouraged the adoption of Personal Software Process (PSP) techniques in all programming classes. This would help students to provide better accurate efforts estimations.

5. Conclusion

The introduction of agile methodologies in our software engineering classes has significantly facilitated and improved the acquisition of programming and software management competences among our students. As shown above, not only the agile process helps to increase the student's competencies on a relative difficult study area like programming, but also it does help the work inside a group and promotes the share of knowledge. At the same, it allows students to reach tiny issues with programming as soon as they encounter them in their work, giving a more long lasting learning experience. The main students challenges are typically in communication and planning both stemming from a lack of experience and training. Therefore, some policies are suggested to be implemented, which includes the improvement of soft-skills competence, use of testing techniques earlier in the program, adoption of a version control system for source code and request students to estimate the duration of each activity. As future work, we intend to perform a deeper analysis of the results of this approach in the classroom. For that, we plan to consider the adoption of other agile methodologies such as Extreme Programming, Feature Driven Development and Kanban in the classroom.

References

Abrahamsson, P., Salo, O., Ronkainen, J. & Warsta, J. (2002). Agile software development methods: review and analysis. *VTT Publication*, 478, 9-36.

Akhtar, M., Ahsan, A. & Sadiq, W. (2010). Scrum adoption, acceptance and implementation (a case study of barriers in Pakistan's IT industry and mandatory improvements). *Proceedings of 17th IEEE International Conference on Industrial Engineering and Management*, 1(1), 458-461.

Ambler, S. (2009). Agile adoption rate survey results. Ambysoft White Papers. Available at: http://www.ambysoft.com/surveys/agileFebruary2008.html. [Accessed on 21/02/2012].

Burnett, M. (2009). What is end-user software engineering and why does it matter? *Proceedings of the 2nd International Symposium on End-User Development*, 1, 15-28.

Buytaert, D. (2010). The commercialization of a volume-driven open source project. Available at: http://buytaert.net/the-commercialization-of-a-volunteer-driven-open-source-project. [Accessed on 23/02/2012].

Cohn, M. (2009). Succeeding with agile: software development using SCRUM (1st ed.), Boston: Pearson Education.

Deeb, A. (2010). Software development methodologies in industry. Master thesis in Software Engineering, Australian National University, and ANU College of Engineering & Computer Science.

Endres, A. (2002). Commentary on James E. Tomayko – software as engineering. *Proceedings of the International Conference on History of Computing: software issues*, 1, 83-91.

Fowler, M. (2006). Using an Agile software process with offshore development. Available at: http://www.martinfowler.com/articles/agileOffshore.html. [Accessed on 23/02/2012].

Hanssen, G. & Faegri, T. (2006). Agile customer engagement: a longitudinal qualitative case study. *Proceedings of International Symposium on Empirical Software Engineering (ISESE)*, 1, 1-10.

Hicks, M. & Foster, J. (2010). Adapting Scrum to managing a research group. Technical Report CS-TR-4966, University of Maryland, Department of Computer Science.

IEEE. (2012). Software engineering as defined by IEEE. Available at: http://free-books-online.org/computers/software-engineering-computers/software-engineering-as-defined-by-ieee/. [Accessed on 23/02/2012].

Jeffery, R. & Scott, L. (2002). Has twenty-five years of empirical software engineering made a difference? Proceedings of the Ninth Asia-Pacific Software Engineering, 539-547.

Lu, H. & Wang, X. (2006). A course design and implementation experience on agile software development methodologies. *Proceedings of Software Engineering Research and Practice*, 1, 281-288.

Perera, G. (2009). Impact of using agile practice for student software projects in computer science education. *International Journal of Education and Development using Information and Communication Technology (IJEDICT)*, 5(3), 85-100.

Rawsthorne, D. & Shimp, D. (2011). Exploring Scrum: the fundamentals (people, product and practices) (1st Ed), Seattle: Create Space.

Rico, D. & Sayani, H. (2009). Use of agile methods in software engineering education. *Proceedings of 2009 Agile Conference*, 1, 174-179.

Royce, W. (1970). Managing the development of large software systems. Available at: http://leadinganswers.typepad.com/leading_answers/files/original_waterfall_paper_winston_royce.pdf. [Accessed on 17/02/2012].

Schwaber, K. (2004). Agile project management with Scrum. *Microsoft Press*, Redmon, WA.
Serena. D. (2007). An introduction to agile software development. Serena White Papers. Available at: http://www.serena.com/docs/repository/solutions/intro-to-agile-devel.pdf. [Accessed on 21/02/2012].
Sims, C. & Johnson, H. (2011). The elements of Scrum (1st Ed), New York: Dymaxicon.
Stuart, J., Beede, E., Deville, J. & Rimbey, E. (2008). 10 keys to successful Scrum adoption. Construx Software White Papers, 3-17.
Sutherland, J. (2001). Agile can scale inventing and reinventing Scrum in five companies. *Cutter IT Journal*, 14(1), 5-11.
Vijayasarathy, L. & Turk, D. (2008). Agile software development: a survey of early adopters. *Journal of Information Technology Management*, 19(2), 1-8.
Wasseman, A. (1996). Towards a discipline of software engineering. *IEEE Software*, 13(6), 23-31.
Weiping, L., Weijie, C. & Ying, L. (2009). Call for implementation: a new software development mode for leveraging the resource of open community. *International Journal of Software Engineering and Applications*, 2(1), 34-39.