ΟΙΚΟΝΟΜΙΚΟ
ΠΑΝΕΠΙΣΤΗΜΙΟ
ΑΘΗΝΩΝ

ATHENS UNIVERSITY
OF ECONOMICS
AND BUSINESS

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΜΕΤΑΠΤΥΧΙΑΚΟ ΔΙΠΛΩΜΑ ΕΙΔΙΚΕΥΣΗΣ
(MSc)

στα ΠΛΗΡΟΦΟΡΙΑΚΑ ΣΥΣΤΗΜΑΤΑ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

"Η διδακτική της διαχείρισης έργων πληροφορικής με χρήση ευέλικτων τεχνικών"

Αλέξανδρος Λεκάτος

ΜΜ4140026

ΑΘΗΝΑ, ΣΕΠΤΕΜΒΡΙΟΣ 2016

ABSTRACT

While software is so important for all facets of the modern world, teaching software development itself is not a perfect process. Agile software engineering methods have recently emerged as a new and different way of developing software as compared to the traditional methodologies. Agile methods are becoming commonplace in the workplace, the effectiveness of teaching agile methods in the classroom has been disputed and many academics have taken a different approach.

This research study was a survey on the on the success of different methods of agile teaching, their efficiency as perceived by academics and students, as well as their ability to provide future software developers with the skills now required by information technology businesses.

An extensive literature review was done to identify the various teaching methods employed at institutions around the world, including recent researches on the subject as well as a case study from the industry. The aim of the literature review was to establish what is now considered best practice to distinguish any known problems or gaps in knowledge regarding the teaching of agile methodologies at universities.

A survey was conducted among current and former computer science students, gathering data from 200 students that attended university in the past 5 years. The students have previously attended a course meant to teach agile methods through the completion of a capstone project and they were required to provide feedback regarding their views on the agile methods used as well as the method of teaching. Students with relevant work experience after the completion of their studies were also required to reflect on the parallels of using agile at university and workplace and its perceived effect on their employability.

ACKNOWLEDGEMENT

I would first like to thank my supervisor Dr Panos Fitsilis of the Athens University of Economics and Business . He allowed this paper to be my own work, but steered me in the right the direction whenever he thought I needed it and I could approach him whenever I had a question about my research or writing.

I would also like to thank Mr Aggelos Thanos, Mr Nikolaos Drosos and Mr Dionisis Adamopoulos who were involved in the distribution of the survey questionnaire for this research. Without their participation, the research could not have been successfully conducted.

Finally, I must express my gratitude to my parents for providing me with support and encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them.

Thank you.

Alexandros Lekatos

TABLE OF CONTENTS

1. INTRODUCTION

In recent years, “Course design has developed from a craftsmanship-like process to a structured production, which involves interdisciplinary teams and requires more complex communication skills”. [Botturi, 2006] This makes methods and modeling languages increasingly important.

Many instructional design methods have been developed in the last years, but they seem to be inadequate if applied in the context of the 21th century school. In fact, nowadays, new skills for students are required, like the ability to perform in social activities or to collaborate with each other to solve problems [De Vincentis, 2007], which in turn, make it necessary that learning embodies and encompasses these new activities. That yields other problems to be faced, like the students’ satisfaction, the administrative transparency, the effectiveness of the documentation and the need for cooperation among the members of a team.

As agile methods are becoming main−stream in the software industry, there has been an increase in the demand for them to be taught in the classroom. Annual Surveys have shown that the use of Scrum has increased in the last few years. In 2007, 37% of respondents used Scrum, and nowadays, more than 50% of the surveyed companies are adopting Scrum as an effective vehicle toward agility. The knowledge of Scrum is crucial to most companies because it emphasizes the importance of team effort and social activities in the development of software. Scrum concentrates on project management practices and includes monitoring and feedback activities that ensure transparency.

This emerging use of Scrum has opened a gap between the skills taught in academic contexts and the ones required by the software industry. As a result, in undergraduate as well as postgraduate courses, there is an increasing need for effective ways of teaching the fundamentals of Agile Methods; yet constraints within educational premises demand creative use of resources.

In the past decade, agile development has begun to see widespread industrial acceptance. A 2009 survey found that 35% of Information Technology companies were using agile teams for at least some of their software development projects. A more recent survey, conducted in 2014, found that 75% of surveyed companies were using agile development for some projects and nearly 50% were using agile development practices for the majority of their software projects.

Despite industrial adoption of agile methodologies, their acceptance and incorporation into academic curricula has been limited. Most popular software engineering texts, have been updated to include some coverage of agile methodologies but are still organized around traditional waterfall methodology.

In order to solve the problems arising from classical software engineering methods, in 2001 a group of software developers and designers formed the Agile Alliance, which published the Manifesto for Agile Software Development based on the assumption that individuals and interactions, working software, customer collaboration and responding to change are, respectively, more important than process and tools, documentation, contract negotiation and following a plan. [Manifesto for Agile Software distribution, 2001]

Agile methodologies aim to satisfy the customer, to welcome changing requirements, to deliver working software frequently, to make customers and developers collaborate, to motivate individuals by suitable environment and to support, considering dialogue essential to exchange information about the project.

These methods can be seen as a reaction to plan-based or traditional methods, which emphasize an engineering-based approach in which it is claimed that problems are fully specifiable and that optimal and predictable solutions exist for every problem. The traditionalists are said to advocate extensive planning, codified processes, and rigorous reuse to make development an efficient and predictable activity.

By contrast, agile processes address the challenge of an unpredictable world by relying on “people and their creativity rather than on processes”.

Ericksson et al. define agility as follows: “Agility means to strip away as much of the heaviness, commonly associated with the traditional software-development methodologies, as possible to promote quick response to changing environments, changes in user requirements, accelerated project deadlines and the like”.

Williams and Cockburn state that agile development is about feedback and change and that agile methodologies are developed to embrace, rather than reject, higher rates of change.

Many have tried to explain the core ideas in agile software development, some by examining similar trends in other disciplines. Conboy and Fitzgerald, describe agility as what is known in other fields as flexibility and leanness. They refer to several sources of inspiration, primarily:

Agile manufacturing, which was introduced by researchers from Lehigh University in an attempt for the USA to regain its competitive position in manufacturing. Key concepts in agile manufacturing are integrating customer-supplier relationships, managing change, uncertainty, complexity, utilizing human resources, and information.

Lean development, which is rooted in the Toyota Production System from the 1950s. Some of the core ideas in this system were to eliminate waste, achieve quality first time, and focus on problem solving.

Meso and Jain have compared ideas in agile development to those in Complex Adaptive Systems by providing a theoretical lens for understanding how agile development can be used in volatile business environments. Turk et al. have clarified the assumptions that underlie processes of agile development and also identifies the limitations that may arise from these assumptions. In the literature, we also find articles that trace the roots of agile development to the Soft Systems Methodology of Peter Checkland.

Nerur and Balijepally compare agile development to maturing design ideas in architectural design and strategic management: ‘‘the new design metaphor incorporates learning and acknowledges the connectedness of knowing and doing (thought and action), the interwoven nature of means and ends, and the need to reconcile multiple world-views’’.

2. LITERATURE REVIEW

2.01 Agile methods

Scrum

Scrum is one of the most popular frameworks for implementing agile. So popular, in fact, that many people think scrum and agile are the same thing. Scrum is an iterative and incremental agile software development framework for managing product development. It defines a flexible, holistic product development strategy where a development team works as a unit to reach a common goal, challenges assumptions of the traditional, sequential approach to product development, and enables teams to self-organize by encouraging physical co-location or close online collaboration of all team members, as well as daily face-to-face communication among all team members and disciplines involved. [Foegen, 2010]

A key principle of Scrum is its recognition that during product development, the customers can change their minds about what they want and need, and that unpredicted challenges cannot be easily addressed in a traditional predictive or planned manner. As such, Scrum adopts an evidence-based empirical approach accepting that the problem cannot be fully understood or defined, focusing instead on maximizing the team's ability to deliver quickly, to respond to emerging requirements and to adapt to evolving technologies and changes in market conditions.

Scrum focuses on project management in situations where it is difficult to plan ahead, with mechanisms for empirical process control where feedback loops constitute the core element.

With scrum, the product is built in a series of fixed-length iterations called sprints that give teams a framework for shipping software on a regular cadence. Milestones come frequently, bringing with them a feeling of tangible progress with each cycle that focuses and energizes everyone. Short iterations also reinforce the importance of good estimation and fast feedback from tests — both recurring struggles in waterfall projects.

Scrum calls for four ceremonies that bring structure to each sprint: sprint planning, daily stand-up, sprint demo and sprint retrospective.

During a sprint, visual artifacts like task boards and burn-down charts, visible to the team and spectators alike, are powerful motivators. Having the opportunity to show off new work at the sprint demo is equally motivating, and the consistent, incremental feedback the team gets from stakeholders at each demo creates a powerful way to develop products.

A scrum team has a slightly different composition than a traditional waterfall project, with three specific roles: product owner, scrum master, and the development team. And because scrum teams are cross-functional, the development team includes testers, designers, and ops engineers in addition to developers.

Product owners are the champions for their product. They are focused on understanding business and market requirements, then prioritizing the work to be done by the engineering team accordingly. Effective product owners build and manage the product backlog, closely partner with the business and the team to ensure everyone understands the work items in the product backlog, give the team clear guidance on which features to deliver next and decide when to ship the product with the predisposition towards more frequent delivery.

Scrum masters coach the team, the product owner, and the business on the scrum process and look for ways to fine-tune their practice of it. An effective scrum master deeply understands the work being done by the team and can help the team optimize their delivery flow. As the facilitator-in-chief, they schedule the needed resources for sprint planning, stand-up, sprint review, and the sprint retrospective.

Scrum masters also look to resolve impediments and distractions for the development team, insulating them from external disruptions whenever possible.

Part of the scrum master's job is to defend against an anti-pattern common among teams new to scrum: changing the sprint's scope after it has already begun.

Scrum masters are commonly mistaken for project managers, when in fact, project managers don't really have a place in the scrum methodology. A scrum team controls its own destiny and self-organizes around their work. Agile teams use pull models where the team pulls a certain amount of work off the backlog and commits to completing it that sprint, which is very effective in maintaining quality and ensuring optimum performance of the team over the long-term. Neither scrum masters nor project managers nor product owners push work to the team.

The most effective scrum teams are tight-knit, co-located, and usually 5 to 7 members. Team members have differing skill sets, and cross-train each other so no one person becomes a bottleneck in the delivery of work. All members of the team help one another to ensure a successful sprint completion.

As mentioned above, the scrum team drives the plan for each sprint. They forecast how much work they believe they can complete over the iteration using their historical velocity as a guide. Keeping the iteration length fixed gives the development team important feedback on their estimation and delivery process, which in turn makes their forecasts increasingly accurate over time. [Berczuk, 2007]

Feature-driven development

Feature-driven development is an iterative and incremental software development process. It blends a number of industry-recognized best practices into a cohesive whole. These practices are all driven from a client-valued feature perspective. Its main purpose is to deliver tangible, working software repeatedly in a timely manner. [Felsing & Palmer, 2002]

Feature driven development is claimed to be suitable for the development of critical systems.

FDD combines model-driven and agile development with emphasis on initial object model, division of work in features, and iterative design for each feature.

Jeff De Luca delivered a set of five processes that covered the development of an overall model and the listing, planning, design and building of features. The first process is heavily influenced by Peter Coad's approach to object modeling. The second process incorporates Peter Coad's ideas of using a feature list to manage functional requirements and development tasks. The other processes and the blending of the processes into a cohesive whole is a result of Jeff De Luca's experience.

In FDD, the building of an object model is not a long, drawn-out activity. Instead, building an initial object model in FDD is an intense, highly iterative, collaborative and generally enjoyable activity involving ‘domain and development members under the guidance of an experienced object modeler in the role of Chief Architect'.

The idea is for both domain and development members of the team to gain a good, shared understanding of the problem domain. It is important that everyone understands the key problem domain concepts, relationships, and interactions. In doing so, the team as a whole learn to communicate with each other and start to establish a shared vocabulary. The object model developed at this point concentrates on breadth rather than depth; depth is added iteratively through the lifetime of the project. Throughout the project, the model becomes the primary vehicle around which the team discusses, challenges, and clarifies requirements.

With the first activity being to build an object model, some may conclude FDD is a model-driven process. It is not. While the model is central to the process, an FDD project is like a Scrum or XP project in being requirement-driven. Small, client-valued requirements referred to as features drive the project; the model merely helps guide. Formally, FDD defines a feature as a small, client-valued function expressed in the form: <action> <result> <object> (e.g., “"calculate the total of a sale"”).

Unlike Scrum and XP that use a flat list of backlog items or user stories, FDD organizes its features into a three level hierarchy that it unimaginatively calls the feature list. Larger projects/teams need this extra organization. It helps them manage the larger numbers of items that are typically found on an FDD features list than on a Scrum-style backlog.

To define the upper levels in the feature list hierarchy, the team derives a set of domain subject areas from the high-level breakdown of the problem domain that the domain experts naturally used during the object modeling sessions. Then within these areas, the team identifies the business activities of that area and places individual features within one of those activities. Therefore, in the features list we have areas containing activities that in turn contain features.

In practice, building the features list is a formalization of the features already discussed during the development of the object model. For this reason, lead developers or Chief Programmers can perform this task using the knowledge they gained during the modeling. Other members of the modeling team including the domain experts provide input to, and verification of the list as necessary.

Not only does this avoid the problems often encountered when customers/domain experts that are unused to doing this sort of formal decomposition try to do it, it provides another level of assurance that the Chief Programmers do understand what is required.

In addition, the ubiquitous language the model provides helps phrase features consistently. This helps reduce frustration in larger teams caused by different domain experts using different terms for the same thing or using the same terms differently.

The third and last of the FDD processes involves constructing an initial schedule and assigning initial responsibilities. The planning team initially sequence the feature sets representing activities by relative business value. Feature sets are also assigned to a Chief Programmer who will be responsible for their development. At the end of this process, each Chief Programmer effectively has a subset of the features list assigned to them. For a Chief Programmer this is their backlog or ‘virtual inbox’ of features to implement.

The planning team may adjust the overall sequence of feature sets to take into account technical risk and dependencies where appropriate. In larger development efforts, the dependencies that have an impact on the sequence may be purely technical in nature but are just as likely to revolve around which feature sets are assigned to which Chief Programmers, and as we shall see, which classes are owned by which developers.

FDD also departs from traditional agile thinking, in that it chooses not to adopt collective ownership of source code. Instead, it assigns individual developers to be responsible for particular classes. The initial assignment of developers to classes takes place during this planning process.

The advantages of individual class ownership include the following:

There is someone responsible for the conceptual integrity of that class. As enhancements are made, the class owner ensures that the purpose and design of the class is not compromised.

There is an expert available to explain how a specific class works. This is especially important for complex or business-critical classes.

The class owner typically implements a required change faster than another developer that is not as familiar with the class.

The class owner has something of his or her own that he or she can take personal pride in.

In addition, it can become tricky to maintain true collective ownership of code as team sizes increase. In my experience, over time, the same developers naturally gravitate to working with the same parts of the code again and again and effectively take ownership of them.

In FDD though, class ownership implies responsibility not exclusivity. A class owner may allow another developer to make a change to a class they own. The big difference is that the class owner is aware of, and approves of, the change and is responsible for checking that the change is made correctly. [Felsing & Palmer, 2002]

DSDM

Dynamic systems development method (DSDM) is an agile project delivery framework, primarily used as a software development method. DSDM is an iterative and incremental approach that embraces principles of Agile development, including continuous user involvement.

DSDM fixes cost, quality and time at the outset and uses the MoSCoW prioritization of scope into musts, shoulds, coulds and won't haves to adjust the project deliverable to meet the stated time constraint.

DSDM is one of a number of Agile methods for developing software and non-IT solutions, and it forms a part of the Agile Alliance.

Dynamic Software Development Method (DSDM) divides projects in three phases: pre-project, project life-cycle, and post project.

Nine principles underlie DSDM: user involvement, empowering the project team, frequent delivery, addressing current business needs, iterative and incremental development, allow for reversing changes, high-level scope being fixed before project starts, testing throughout the lifecycle, and efficient and effective communication.

Within DSDM a number of factors are identified as being of great importance to ensure successful projects:

First there is the acceptance of DSDM by senior management and other employees. This ensures that the different actors of the project are motivated from the start and remain involved throughout the project. [Abrahamsson et al, 2002]

The second factor follows directly from this and that is the commitment of management to ensure end-user involvement. The prototyping approach requires a strong and dedicated involvement by end user to test and judge the functional prototypes.

Then there is the project team. This team has to be composed of skillful members that form a stable union. An important issue is the empowerment of the project team. This means that the team has to possess the power and possibility to make important decisions regarding the project without having to write formal proposals to higher management, which can be very time-consuming. In order for the project team to be able to run a successful project, they also need the right technology to conduct the project.

Finally DSDM also states that a supportive relationship between customer and vendor is required. This goes for both projects that are realized internally within companies or by outside contractors. [Abrahamsson et al, 2002]

XP

Extreme programming (XP) is a software development methodology which is intended to improve software quality and responsiveness to changing customer requirements. As a type of agile software development, it advocates frequent "releases" in short development cycles, which is intended to improve productivity and introduce checkpoints at which new customer requirements can be adopted. [Beck, 2000]

Other elements of extreme programming include: programming in pairs or doing extensive code review, unit testing of all code, avoiding programming of features until they are actually needed, a flat management structure, simplicity and clarity in code, expecting changes in the customer's requirements as time passes and the problem is better understood, and frequent communication with the customer and among programmers. The methodology takes its name from the idea that the beneficial elements of traditional software engineering practices are taken to "extreme" levels. As an example, code reviews are considered a beneficial practice; taken to the extreme, code can be reviewed continuously, i.e. the practice of pair programming.

XP focuses on best practice for development. Consists of twelve practices: the planning game, small releases, metaphor, simple design, testing, refactoring, pair programming, collective ownership, continuous integration, 40-hour week, on-site customers, and coding standards.

The revised XP2 consists of the following primary practices: sit together, whole team, informative workspace, energized work, pair programming, stories, weekly cycle, quarterly cycle, slack, 10-minute build, continuous integration, test-first programming, and incremental design. [Beck, 2000]

Crystal Methodologies

Crystal methods are a family of methodologies that were developed by Alistair Cockburn in the 1990s. The methods come from years of study and interviews of teams by Cockburn. Cockburn's research showed that the teams he interviewed did not follow the formal methodologies yet they still delivered successful projects.

The Crystal family is Cockburn's way of cataloguing what they did that made the projects successful.

Crystal methods are considered and described as “lightweight methodologies”. The use of the word Crystal comes from the gemstone where, in software terms, the faces are a different view on the “underlying core” of principles and values. The faces are a representation of techniques, tools, standards and roles.

Crystal methodologies is a family of methods for teams of different sizes and criticality: Clear, Yellow, Orange, Red, Blue.

The most agile method, Crystal Clear, focuses on communication in small teams developing software that is not life-critical.

Between all the methods in the Crystal family, there are seven prevailing common properties. Cockburn found that the more of these properties that were in a project, the more likely it was to succeed.

The seven properties are: frequent delivery, reflective improvement, close or osmotic communication, personal safety, focus, easy access to expert users and a technical environment with automated tests, configuration management, and frequent integration. [Abrahamsson et al, 2002]

Lean software development

An adaptation of principles from lean production and, in particular, the Toyota production system to software development.

Consists of seven principles: eliminate waste, amplify learning, decide as late as possible, deliver as fast as possible, empower the team, build integrity, and see the whole. [Mary & Tom, 2003]

Adaptive software development

Adaptive software development is a design principle for the creation of software systems. The principle focuses on the rapid creation and evolution of software systems. There is never a period where the software is finished; there are just stable periods between new releases.

The adaptive development method grew out of the rapid application development method. These two methods are similar in structure, but rapid application development allows for a time when the project is finished, while adaptive software development doesn't.

Adaptive software development replaces the traditional waterfall cycle with a repeating series of speculate, collaborate, and learn cycles. This dynamic cycle provides for continuous learning and adaptation to the emergent state of the project. The characteristics of an ASD life cycle are that it is mission focused, feature based, iterative, time-boxed, risk driven, and change tolerant.

The word ‘speculate’ refers to the paradox of planning – it is more likely to assume that all stakeholders are comparably wrong for certain aspects of the project’s mission, while trying to define it. During speculation, the project is initiated and adaptive cycle planning is conducted. Adaptive cycle planning uses project initiation information—the customer’s mission statement, project constraints (e.g., delivery dates or user descriptions), and basic requirements—to define the set of release cycles (software increments) that will be required for the project.

Collaboration refers to the efforts for balancing the work based on predictable parts of the environment (planning and guiding them) and adapting to the uncertain surrounding mix of changes caused by various factors, such as technology, requirements, stakeholders, software vendors. The learning cycles, challenging all stakeholders, are based on the short iterations with design, build and testing. During these iterations the knowledge is gathered by making small mistakes based on false assumptions and correcting those mistakes, thus leading to greater experience and eventually mastery in the problem domain. [Highsmith, J., 2013]

2.02 Agile practices

Acceptance test-driven development

Acceptance test–driven development (ATDD) is a development methodology based on communication between the business customers, the developers, and the testers. ATDD encompasses many of the same practices as specification by example, behavior-driven development (BDD), example-driven development (EDD), and story test–driven development (SDD).

All these processes aid developers and testers in understanding the customer's needs prior to implementation and allow customers to be able to converse in their own domain language.

ATDD is closely related to test-driven development (TDD). It differs by the emphasis on developer-tester-business customer collaboration. ATDD encompasses acceptance testing, but highlights writing acceptance tests before developers begin coding.

Acceptance tests are from the user's point of view the external view of the system. They examine externally visible effects, such as specifying the correct output of a system given a particular input.

Acceptance tests can verify how the state of something changes, such as an order that goes from paid to shipped. They also can check the interactions with interfaces of other systems, such as shared databases or web services.

In general, they are implementation independent, although automation of them may not be. [Downs, 2011]

Agile modeling

Agile modeling (AM) is a methodology for modeling and documenting software systems based on best practices. It is a collection of values and principles, that can be applied on an (agile) software development project. This methodology is more flexible than traditional modeling methods, making it a better fit in a fast changing environment. It is part of the agile software development tool kit.

An important concept to understand about AM is that it is not a complete software process. AM's focus is on effective modeling and documentation. It doesn't include programming activities, although it will tell you to prove your models with code. It doesn't include testing activities, although it will tell you to consider testability as you model. It doesn't cover project management, system deployment, system operations, system support, or a myriad of other issues. Because AM's focus in on a portion of the overall software process you need to use it with another, full-fledged process such as eXtreme Programming (XP), DSDM, SCRUM, the Agile Unified Process (AUP), or the Rational Unified Process (RUP). You start with a base process, such as XP or RUP or perhaps even your own existing process, and then tailor it with AM (hopefully adopting all of AM) as well as other techniques as appropriate to form your own process that reflects your unique needs. Alternatively, you may decide to pick the best features from a collection of existing software processes to form your own process. For XP projects, AM explicitly describes how to improve productivity through addition of modeling activities whereas with for RUP projects it describes how to streamline modeling and documentation efforts to improve productivity.

Agile modeling is a supplement to other agile methodologies such as Scrum or extreme programming (XP). It is explicitly included as part of the disciplined agile delivery (DAD) framework.

As per 2011 stats, agile modeling accounted for 1% of all agile software development. [Wiley, J. and Sons, A.M., Effective Practices for Extreme Programming and the Unified Process]

Backlog

The backlog comprises an ordered list of requirements that a scrum team maintains for a project. It consists of features, bug fixes, non-functional requirements, etc.—whatever must be done to successfully deliver a viable product. The project owner orders the Product Backlog Items (PBIs) based on considerations such as risk, business value, dependencies, and date needed.

Items added to a backlog are commonly written in story format. The product backlog is what will be delivered, ordered into the sequence in which it should be delivered. It is visible to everyone but may only be changed with the consent of the Product Owner, who is ultimately responsible for ordering Product Backlog Items for the Development Team to choose.

The Product Backlog contains the project owner's assessment of business value and the Development Team's assessment of development effort, which are often, but not always, stated in story points using a rounded Fibonacci sequence. These estimates help the project owner to gauge the timeline and may influence ordering of Product Backlog Items; for example, if the "add spellcheck" and "add table support" features have the same business value, the Product Owner may schedule earlier delivery of the one with the lower development effort (because the return on investment is higher) or the one with higher development effort (because it is more complex or riskier, and they want to retire that risk earlier).

The Product Backlog and the business value of each Product Backlog Item is the responsibility of the project owner. The size of each item is, however, determined by the developers, who contribute by sizing in story points or estimated hours. [Deemer et al, 2012]

Behavior-driven development

Behavior-driven development (BDD) is a software development process that emerged from test-driven development (TDD). Behavior-driven development combines the general techniques and principles of TDD with ideas from domain-driven design and object-oriented analysis and design to provide software development and management teams with shared tools and a shared process to collaborate on software development.

Behavior-driven development is an extension of test-driven development: [1] development that makes use of a simple, domain-specific scripting language. These DSLs convert structured natural language statements into executable tests. The result is a closer relationship to acceptance criteria for a given function and the tests used to validate that functionality. As such it is a natural extension of TDD testing in general.

Although BDD is principally an idea about how software development should be managed by both business interests and technical insight, the practice of BDD does assume the use of specialized software tools to support the development process. Although these tools are often developed specifically for use in BDD projects, they can be seen as specialized forms of the tooling that supports test-driven development. The tools serve to add automation to the ubiquitous language that is a central theme of BDD.

BDD is largely facilitated through the use of a simple domain-specific language (DSL) using natural language constructs (e.g., English-like sentences) that can express the behavior and the expected outcomes. Test scripts have long been a popular application of DSLs with varying degrees of sophistication. BDD is considered as an effective technical practice especially when the "problem space" of the business problem to solve is complex. [Solis et al, 2011]

Continuous integration

In XP, Continuous Integration (CI) was intended to be used in combination with automated unit tests written through the practices of test-driven development. Initially this was conceived of as running all unit tests in the developer's local environment and verifying they all passed before committing to the mainline. This helps avoid one developer's work-in-progress breaking another developer's copy. If necessary, partially complete features can be disabled before commit, such as by using feature toggles.

Continuous Integration is backed by several important principles and practices that include maintaining a single source repository, automating the build, make the build self-testing, every commit should build on an integration machine, keeping the build fast, testing in a clone of the production environment, making it easy for anyone to get the latest executable, letting everyone see what is happening and automating deployment. [Fowler & Foemmel, 2006]

Continuous Integration is a development practice that requires developers to integrate code into a shared repository several times a day. Each check-in is then verified by an automated build, allowing teams to detect problems early.

By integrating regularly, they can detect errors quickly, and locate them more easily.

Later elaborations of the concept introduced build servers, which automatically ran the unit tests periodically or even after every commit and reported the results to the developers. The use of build servers had already been practised by some teams outside the XP community. Nowadays, many organisations have adopted CI without adopting all of XP.

In addition to automated unit tests, organisations using CI typically use a build server to implement continuous processes of applying quality control in general — small pieces of effort, applied frequently. In addition to running the unit and integration tests, such processes run additional static and dynamic tests, measure and profile performance, extract and format documentation from the source code and facilitate manual QA processes. This continuous application of quality control aims to improve the quality of software, and to reduce the time taken to deliver it, by replacing the traditional practice of applying quality control after completing all development. [Fowler & Foemmel, 2006]

Domain-driven design

The philosophy of domain-driven design (DDD) is about placing our attention at the heart of the application, focusing on the complexity that is intrinsic to the business domain itself. It also distinguishes h the core domain from the supporting sub-domains, and place appropriately more of the design efforts on the core.

Domain-driven design consists of a set of patterns for building enterprise applications from the domain model out. In your software career you may well have encountered many of these ideas already, especially if you are a seasoned developer in an OO language. But applying them together will allow you to build systems that genuinely meet the needs of the business.

Domain-driven design (DDD) is an approach to software development for complex needs by connecting the implementation to an evolving model. The premise of domain-driven design is the following placing the project's primary focus on the core domain and domain logic, basing complex designs on a model of the domain and initiating a creative collaboration between technical and domain experts to iteratively refine a conceptual model that addresses particular domain problems. [Evans, 2004]

Pair programming

Pair programming is an agile software development technique in which two programmers work together at one workstation. One, the driver, writes code while the other, the observer or navigator, reviews each line of code as it is typed in. The two programmers switch roles frequently.

While reviewing, the observer also considers the strategic direction of the work, coming up with ideas for improvements and likely future problems to address. This frees the driver to focus all of his or her attention on the tactical aspects of completing the current task, using the observer as a safety net and guide.

Pair programming is considered to offer benefits such as:

Increased discipline because by pairing partners they are more likely to "do the right thing" and are less likely to take long breaks.

Better code as pairing partners is claimed to produce higher quality designs.

Improved team morale as programmers agree that it is much more enjoyable than programming alone.

Collective code ownership as pairs rotate frequently and everybody gains a working knowledge of the entire code.

Improved mentoring as everyone, even junior programmers, has knowledge that others don't.

Better team cohesion as people get to know each other more quickly when pair programming. [Cockburn & Williams, 2000]

Test-driven development

Test-driven development (TDD), is an evolutionary approach to development which combines test-first development where you write a test before you write just enough production code to fulfill that test and refactoring. The primary goal of TDD is specification and not validation. In other words, it’s one way to think through your requirements or design before your write your functional code. [Ambler, 2003]

Test-driven development (TDD) is a software development process that relies on the repetition of a very short development cycle: requirements are turned into very specific test cases, then the software is improved to pass the new tests, only. This is opposed to software development that allows software to be added that isn't proven to meet requirements. [Beck, 2000]

A significant advantage of TDD is that it enables you to take small steps when writing software. This is a practice that is far more productive than attempting to code in large steps.

Kent Beck, who is credited with having developed or ‘rediscovered' the technique, stated in 2003 that TDD encourages simple designs and inspires confidence.

Test-driven development is related to the test-first programming concepts of extreme programming, begun in 1999, but more recently has created more general interest in its own right.

Programmers often apply the concept to improving and debugging legacy code developed with older techniques.

TDD is being quickly adopted by agile software developers for development of application source code and is even being adopted by Agile DBAs for database development. TDD should be seen as complementary to Agile Model Driven Development (AMDD) approaches and the two can and should be used together. TDD does not replace traditional testing, instead it defines a proven way to ensure effective unit testing. [Ambler, 2003]

Agile testing

Agile is an iterative development methodology, where requirements evolve through collaboration between the customer and self-organizing teams and agile aligns development with customer needs. A software testing practice that follows the principles of agile software development is called Agile Testing.

Agile development recognizes that testing is not a separate phase, but an integral part of software development, along with coding.

Agile teams use a "whole-team" approach to "baking quality in" the software product. Testers on agile teams lend their expertise in eliciting examples of desired behavior from customers, collaborating with the development team to turn those into executable specifications that guide coding.

Testing and coding are done incrementally and interactively, building up each feature until it provides enough value to release to production. Agile testing covers all types of testing. The Agile Testing Quadrants provide a helpful taxonomy to help teams identify and plan the testing needed.

In contrast with other methodologies, agile testing focuses on repairing faults immediately, rather than waiting for the end of the project and by doing so, it is expected to significantly reduce cost. [Crispin, 2003]

2.03 Criticism of Agile Methodology

“For all its proponents,” writes Mike Brown of uTest, “Agile has its fair share of skeptics and detractors. These are people who have a much different Agile experience—one characterized by chaotic processes, lower quality, miscommunication and numerous other problems.”

In 2012, Max Smolaks of TechWeek Europe,reporting on research conducted by Voke Media to determine what 200 different software companies thought of their attempt to embrace Agile, wrote:

“Out of over 200 participants, 64 percent said that switching to Agile Development was harder than it initially seemed. Forty percent of respondents did not identify an obvious benefit to the practice. Out of those who did, 14 percent thought it resulted in faster releases, and 13 percent—that it created more feedback. Seven percent of participants noted that Agile developers were happier due to reduced future planning and documentation.”

These results strongly suggest that for organizations entrenched in years of non-Agile working practices, making the switch can be difficult, if not counterproductive.

Scott Barber explains “I believe that the trend to go Agile is misguided. If a company is developing good software, the people involved in developing that software are happy working there, the software development is sustainable, and the business is being adequately served by that software, there’s really no need for them to try to be more or less Agile. Agile has challenges like any other culture, but the single biggest challenge I find is companies trying to solve development, process, management, and/or schedule problems by going Agile. Teams who have grown up in a culture that is fundamentally different from Agile simply will not find it easy to go Agile.”

In other words, companies expecting Agile to be a magic bullet to fix whatever ails their production efforts may be seriously disappointed, as it demands more of a systemic culture-shift than merely embracing a new set of tools and procedures.

Moreover, it isn’t a one-size-fits-all approach. What works for a small San Francisco startup may be completely inappropriate for an enterprise with 5,000 employees distributed in cities around the world. Agile might not also be the right practice for software companies and IT firms that are subject to regulatory compliance mandates, such as government agencies and their contractors, where extensive documentation and procedures rife with checks and balances are essential.

Naturally, proponents of Agile suggest that virtually all of the problems, complaints, and negative experiences that companies attempting to use Agile have reported are due not to a problem with Agile itself, but with a failure to understand the approach and its limitations. Others, such as technology strategist Lajos Moczar, claim to understand Agile fully and yet contend that its guiding principles are flawed. In an article on CIO.com published in June 2013 and titled “Why Agile Isn’t Working,” Moczar ignited a firestorm, with pro-Agile and anti-Agile commenters on his post engaging in mild verbal warfare. “The comments in this thread have taken a strangely negative religious tone,” noted one participant, “like watching one sectarian argue doctrine with a member of another sect.” As with all ideological paradigms, some people embrace—or denounce—Agile with an almost spiritual fervor.

However, there are many signs suggesting that despite its potential drawbacks, the rapid adoption and popularization of Agile is largely based on the pragmatic demands placed on software production and information technology in the early 21st century. When everything is changing so fast, some agility is simply required.

Agile methodology was supposed to be a solution to solve all of our problems but some argue that is not. As issues appear when companies start to implement Agile in their organizations, a research has been done on seventeen companies using Agile methodology. [Coyle & Conboy, 2010]

The authors of the research considered the four below as the most important:

Developer fear caused by transparency of skill deficiencies

The Progress of each team member’s work is usually reported on daily basis. Therefore, every member knows how long it takes each person to do it. If for some reasons, a task took you more time than it should, you can have a feeling that everybody is judging you. Furthermore, design discussion with a whiteboard can highlight technical skill deficiencies or lacks in communications skills. Research has shown that many developers have a low self-esteem because of that. To prevent such problems team members should feel safe to expose their weaknesses.

A good idea would be a second meeting in a small group after the main one. Second thing is that developers should know that they can get help to improve their skills. Junior developers should have a mentor who would help them with their daily issues. Pairing is also an excellent practice. More experienced team members could share their knowledge with those less experienced. [Coyle & Conboy, 2010]

The need for developers to be a ‘master of all trades’

It is believed that to be a successful agile developer you need to be a coder, a tester, an architect, a customer, a quality assurance expert and a multitude of other things software-related. So how can a tester be an architect and database expert at the same time? According to the research some companies were sending their employees to all sorts of trainings. But it was expensive and not as effective as they thought it would be. Some claim that balance between “master of all” and “master of none” should be obtained here. [Coyle & Conboy, 2010]

A team leader should choose team members carefully. Developers should have a broad knowledge of all aspects of the software development as well as business knowledge, but at the same they should be specialists in certain areas. This could be easy within a small team but in case of larger ones it can be extremely difficult. [Coyle & Conboy, 2010]

Increased reliance on social skills

Because of the constant communication in Agile, team members should have good communication skills. Quite often we can see a great developer with poor social skills. This is a great issue when a team member can’t express their thoughts to the rest of the team. Developers can often talk to each other but they can’t get along with customers. The Agile methodology assumes that developers should contact their clients directly and talk about specific features.

The most intuitive solution is to send employees to social trainings and, as the research showed, it is the most effective solution. Also recording stand-up meetings and analyzing them seems to be a good practice. Instead of an individual contact with clients, a group of people can be chosen to talk with them. It has been suggested that when creating an agile team, each member should have good communication skills right on the start to prevent problems in future work. [Coyle & Conboy, 2010]

A lack of business knowledge among developers

It is a very common problem in case of larger teams and more complex software. Because each team member can get knowledge from the client directly, the knowledge is later spread throughout the team. After a while it appears that every member of the team is a specialist in a narrow area of software which is being created. It often happens that not every member has the same knowledge about the domain as the client. This can cause misunderstandings when a different from usual team member has to speak to a specific client. It may even end up in losing customer's trust in the team.

It is extremely important that each and every member of the team has some basic business knowledge, in order to speak with clients on equal basis. Differences in the knowledge between team members can be aligned by pair programming and short trainings. While pairing there is a knowledge flow between the developers. Inviting a domain expert, as research has shown, it is a great solution to lift team members' knowledge to an upper level of understanding. [Coyle & Conboy, 2010]

2.04 Teaching Software Engineering at MIT

A report by Hal Abelson and Philip Greenspun presents the teaching methods they employed at teaching software engineering at MIT. They discuss methods for involving alumni as teaching assistants and coaches and argue for the method of helping students achieve fluency by assigning five complete applications for construction in a semester rather than the traditional single problem in a software engineering semester.

The report argues that starting in the early 1990s, demand from computer engineering courses shifted toward server-based Internet applications. With 1000 users potentially attempting the same action at the same instant, the technical challenge shifts to managing concurrency and transactions. Given stateless protocols such as HTTP, software engineers must learn to develop stateful user experiences. Given the ubiquitous network and evolving standards for remote procedure calls, students can learn practical ways of implementing distributed computing.

To contribute to the information systems of the next 20 years, in addition to the material in the core computer science curriculum, the students need to be familiar with the principles of distributed computing, currency and transactions and how to build a stateful user experience on top of stateless protocols

“Scientists measure their results against nature. Engineers measure their results against human needs. Programmers ... don't measure their results. As a final overarching deep principle, we need to teach students to constantly measure their results against the end-user experience. Anyone can build a Web service. The services that are successful and have impact are those whose data model and page flow permit the users to accomplish their tasks with a minimum of time and confusion.” [Abelson & Greenspun, 2001]

In regard to teaching skills, Abelson and Greenspun argue that at MIT they concentrate on teaching principles rather than skills. For example, there is no course in the computer science department that teaches a computer language. Students learn Lisp as a notation for the computer science concepts in their first course. Students write code in Java in the first software engineering course. But they don't go through one language feature per lecture as some schools might.

They argue that they didn’t have to teach computer science students any skills because they'd “graduate into a job at Hewlett Packard or IBM” and they would be “sitting next to an experienced engineer, the graduate would learn his or her craft over a six-year period and emerge, at age 28, to lead a project or become the chief technologist at a small company”. [Abelson & Greenspun, 2001]

With the Web explosion, however, came an explosion in the number of organizations engaging in software development. Teams are smaller, deadlines are shorter, and there aren't enough qualified project leaders to go around. In the superheated job markets of today there is a surprising number of MIT graduates starting off as CTOs of startup companies or in lead engineering roles on Web projects for larger organizations. Thus focus is to teach some of the more important skills of an experienced engineer, notably rapid application development and dealing with extreme requirements.

MIT graduates should be able to take vague and ambitious specifications and turn them into a system design that can be built and launched within a few months, with the most important-to-users and easy-to-develop features built first and the difficult bells and whistles deferred to a second version. Students should learn how to test prototypes with end-users and refine their application design once or twice within even a three-month project.

As business decision-makers are no longer shy about presenting software engineers with extreme requirements they teach two methods of dealing with extreme requirements. The first is via automatic code generation. If the system requirements can be represented in a machine-readable form then a portion of the software can be generated by a computer program. If the requirements change mid-stream, it is only needed to run the code generator again. The second method of dealing with extreme requirements is via use of a toolkit. For example, in the case of the three-month accounting system project, starting with a toolkit such as SAP or Oracle Applications is probably a much better idea than trying to write all the code from scratch.

Students in a traditional computer science curriculum will often spend a term on each of these: learning the syntax of a language, how to implement lists, stacks, hash tables, how to interpret a high-level language, how to build a time-sharing operating system, learn about the underpinnings of several different kinds of database management systems and learn about AI algorithms. Students in MIT learn all of the above in one semester, albeit not very thoroughly.

In regard to projects, they insist on having a client for every project undertaken. This is a person who can describe their desired capabilities for an information system but offers no hint as to how to build it. They have identified that the best clients are people who are in fact passionate about some sort of Internet service and completely clueless about all matters technical such as CEOs, MBA students, non-profit organization directors, and university administrators. Students have the best opportunity for success when the client has a clear idea of what features are essential and when the client responds quickly to email announcing the availability of a revised prototype. Clients who change their minds about the application when seeing a first prototype are instructive for the entire class as it resembles the real world.

For making sure that projects stay on track, one valuable technique was to have each student group present privately to the lecturers once per week during the evening lab hours.

The best projects were ones with clients who had the means to extend and maintain the service after the course is over, possibly by hiring the students who built it. For example, there were students that build a volunteer matching and event coordination system for a group within MIT. The group was already up and running doing a dozen or more events every year and managing thousands of volunteer-days. So they had a real interest in a better computer support for their work and the ability to launch the system within the MIT community.

At the end of the semester the focus is to drill into the students' heads the cold hard facts of the world: nobody owes them attention. Each student group prepares an overview page that is a single HTML document, with a few screen shots, that demonstrates the major functions of the application that they've built.

However, according to the report [Abelson & Greenspun, 2001], learning on projects is not very uniform. Some students end up with projects that expose them to a big range of challenges but others get to do something trivial. Also because each project has different goals that are client-dependent, the students are sometimes not able to have very meaningful exchanges concerning their projects amongst themselves.

Abelson and Greenspun identify the ability to bring alumni back to campus to share their industrial software engineering expertise as opposed to a more structured tutoring as one of the most important success factors of their course. They conclude by offering the view that a significant improvement in teaching students software engineering skills can be achieved by challenging students to build multiple applications over a semester, bringing professional software engineers onto the campus to coach students, a terminal room where students can work together on a scheduled basis, projects with real clients and an emphasis on oral and written presentation of results.

2.05 Teaching agile software development

Several recent surveys show that agile methodologies like Scrum and Extreme Programming have been successfully adopted by many companies for software development. However, the same surveys show that only few of the agile practices are applied consequently and thoroughly. This is to a great extent due to the lack of skilled personnel. [Kropp & Meier, 2013] In their work, Kropp and Meier, proposed a more holistic approach for teaching agile software development, in which the required agile practices and values are not only integrated theoretically into the courses but also practically applied. The proposed concept was realized in a new a course at Zurich University of Applied Sciences during 2012. The researchers claim that the evaluation showed very encouraging results.

Kropp and Meier argue that “the early adopters of agile approaches were all highly mature and technically skilled experts in their fields. They had internalized the agile philosophy, were very productive and produced high quality results. Today’s agile teams, however, are “normal” software teams, with architects, seniors and juniors in one team, and many of them are not yet familiar with the agile philosophy. Even though those teams have improved in software development to some extent, they are far less productive than the early adopter expert teams. Survey results show that quality has partially even gone down and overall costs increased. One reason for this may be that many of the important agile practices are not applied as thoroughly as the agile pioneers proposed”.

They also argue that although agile software development has been around for more than a decade, teaching agile software development has only drawn some attention in educational and research conferences in the last few years. A reason for this might be that agile development is not based on a green-field theory but has been developed from practice. It is extensively discussed why software engineering programs should teach agile software development. They emphasize that software engineers not only need technical skills but also social and ethical ones, which are both corner stones of agile development. [Kropp & Meier, 2013]

In the recent study, in which 140 Swiss IT companies and almost 200 IT professionals participated it clearly showed that IT companies and IT professionals following the agile methods are much more satisfied with their methodologies than their plan-driven counterparts. The study also showed very clearly, that major goals of introducing agile development have been reached: A significant improvement in the ability to manage changing priorities, improvement of the development process in general and a much faster time-to-market.

However, although the survey shows very promising results at first view, there are also quite astonishing findings. It is reported that development cost, software quality and software maintainability have not improved as much as expected. It is remarked “this clearly contradicts the intention of the authors of the agile manifesto, who want to deliver high quality code that is easily maintainable”. [Kropp & Meier, 2013]

Lastly the research shows, that there are too few software engineers with the skills for agile development. This suggests that teachers do not yet educate the students with the required skills. This assumption is backed by answers to the survey where almost 70% percent of the participating companies think that undergraduates have too little knowledge of agile methods. [Kropp & Meier, 2013]

The researchers argue that before developing a new agile software engineering course, it is important to analyze the needed skills and competences for agile software development. The required competences can be divided into three major categories:

Mastering the technical skills or engineering practices, builds the foundation for being able to develop high quality software. These engineering practices are especially defined by eXtreme Programming and include best practices like unit testing, clean coding, test-driven development, collective code ownership and the like. Engineering practices are mostly competences that refer to the single individual.

On the second level come the agile management practices. They define how agile projects are organized and run. Agile management practices include iterative planning, short release cycles, small releases, strong customer involvement and highly interactive teams. Management practices are typically team aspects, which require the appropriate social competences.

On top of these competences come the agile values, which are articulated in the agile manifesto and are based on characteristics like mutual respect, openness, and courage.

Engineering practices can be taught very well in the classroom through lecturers and be learned by the individuals at their own pace.

Management competences are best taught through student projects in teams, as the research confirms.

These different competence levels have to be considered in an agile software engineering course and have guided the authors in the design of the new course.

Lastly it is argued that Agile values are difficult to teach. The approach for Kropp and Meier was through many discussions during the lectures and workshops to transport the message that these values are not just something the creators of the Agile Manifesto intended to give lip service to and then forget. They are working values. The concepts of agile values were introduced in the first part. Usage of the values was propagated in the second iteration through means like retrospectives, common code ownership or pair programming.

It is the authors' opinion that agile software development cannot be taught in isolated Software Engineering courses. A challenge will be the integration of agile development in "other courses like programming, object-oriented analysis and design, algorithms and data structures, etc." [Kropp & Meier, 2013] Special attention needs to be paid to the fact, that agile software development does not work well together with big-design up front approaches. This could mean a shift from BDUF to emergent design as advocates of Scrum propose it. That said, further work is necessary on how agile development can successfully be integrated into the computer science curriculum. [Kropp & Meier, 2013]

2.06 Bringing Agile into the Capstone Course

In his 2014 report [Kuhl, 2014] John Kuhl from the University of Iowa, identified the issues that arise when switching the capstone project course to agile development. In particular he noticed that it was challenging to quickly, but effectively, train student teams to utilize Scrum in order to get teams functioning as early as possible in the semester.

Other problems he faced were related to how to fill the Product Owner and Scrum Master roles on student teams, how to track team progress, how to identify problems in teams and provide formative feedback, how to insure that teams are employing rigorous testing practices throughout the development cycle and how to evaluate individual student performance for grading purposes.

The problem of getting students up to speed with agile/Scrum was addressed by revising a prerequisite software engineering methodology course to include coverage of agile development. A short (six week) project in the prerequisite course was also modified to use Scrum in order to provide students with some practical experience prior to the capstone course. At the beginning of the capstone course, four lectures were devoted to reviewing agile/Scrum development and other important concepts including testing and continuous integration. These lectures overlapped with the formation of student teams and development of product visions and initial product backlogs by teams so they did not substantively delay the start of project activity. Teams were able to begin their first development sprint at the end of the second week in the semester, allowing for the completion of six two-week sprints prior to the final week of the 14-week semester. The last week of the semester was reserved for class presentations.

Filling the Product Owner role on the student teams posed perhaps the biggest challenge. Since the Product Owner is responsible for development user stories, prioritizing features, and selecting the set of features to be implemented in each sprint, the role is critical to team success. Teams were strongly encouraged to identify a real customer for their project, and approximately half of the teams were able to do so. However, it was not practical for any of the customers to commit the time or effort required for the teams' product owner roles. Other teams chose self-defined project ideas and thus did not have an external customer. In both cases, teams were directed by the instructor to select one team member to serve as the Product Owner. In cases where the team had a real, external customer, this team member was expected to work with that customer as closely as possible to develop and prioritize user stories, select sprint backlogs, and make final feature acceptance decisions at the conclusion of each sprint. Most of the teams that lacked a real customer found it most productive to do product management activities consortially, with the designated Product Owner playing a limited role. In one case, the team delegated full product management responsibility to the designated Product Owner. Both scenarios seemed to work satisfactorily. However, the instructor found it necessary to watch the teams closely to make sure that product management responsibilities--particularly acceptance of completed features--were being adequately addressed.

The Scrum Master role was found to be less critical. The primary responsibility of the Scrum Master is to insulate the team from outside distractions and interface with management and other stakeholders on behalf of the team. In an academic setting, external influences are minimal and most teams found that they could function fine without a designated Scrum Master. Teams were given the option of selecting a Scrum Master and a few found the role useful for facilitating logistical issues such as arranging team meetings.

Tracking of team progress was facilitated by the use of a project management tool called Pivotal Tracker. Tracker is widely used by agile teams to manage product and sprint backlog and monitor product development progress. The tool provides a web-based dashboard that makes it easy to monitor the creation and management of user stories and assess team progress in implementing new features. Although the primary motivation for mandating the use of tracker by teams was to facilitate project management, the tool provided an excellent window for the instructor to view how effectively teams were functioning. Via this tool, the instructor was able to identify several emerging team problems during the initial sprint and provide formative feedback to the affected teams to address the identified issues.

An additional mechanism used to assess team performance and progress was in-class review/retrospective sessions. At the conclusion of each sprint, teams were required to provide a brief summary of their post-sprint review and retrospective meetings to the class and instructor. To insure that all aspects of team performance were addressed, the instructor identified a particular area of emphasis for each in-class review--e.g. testing, feature acceptance, configuration management, etc. Following each of these reviews the instructor provided detailed formative feedback to the teams, as needed, to address any identified problems or concerns. Following the third sprint, teams were required to give an in-class demonstration of their product.

Since continuous testing is central to agile development, the capstone course provided a heavy emphasis on all aspects of testing: unit testing, integration testing, regression testing, and acceptance testing. By the end of the third week of the semester, teams were required to submit a Test Plan addressing all of the above-listed testing areas. Teams were also required to utilize automated testing tools, of their choosing to expedite the testing process. In addition to submission of the written Test Plan, and in-class review was conducted. Following the fourth sprint, teams were required to conduct an in-class review and demonstration of their testing procedures and processes.

Lastly, the difficulty of Individual evaluation of student achievement in any team-based project course is noted by Kuhl. In this course the difficulty was exacerbated by the democratized nature of the teams and the lack of specifically defined roles for team members. [Kuhl, 2014]

2.07 Teaching Using Simulation Games

As with MIT's example, it is often argued that a typical software engineering course fails to teach its students many of the skills needed in software development organizations. Because lectures and class projects alone cannot adequately teach about the software process, researchers at the University of California, Irvine, have developed a pair of games in which the process is simulated, giving students an opportunity to practice it firsthand. [Navarro et all, 2004] It is an efficient way of simulating real projects and is feasible with larger class sizes as opposed to MITs way of having each student group work on a real project for clients.

In their report they maintain that there is a large difference between the software engineering skills taught at a typical university and the skills that are required of a software engineer by a software development organization. This problem seems to stem from the way software engineering is typically introduced to students: general theory is presented in lectures and put into practice in an associated class project.

They argue that although both lectures and projects are essential, they lack a practical, in-depth treatment of the overall process of software engineering. In particular, lectures allow only passive learning, and the size and scope of class projects are too constrained by the academic setting to exhibit many of the fundamental characteristics of real-world software engineering processes.

To address this problem, they have been in the process of researching, designing, building, and experimenting with two game-based simulation tools for teaching software engineering: Problems and Programmers, a physical card game that simulates a software engineering process; and SimSE, a computer-based environment that allows the creation and simulation of software engineering processes. Both allow students to "virtually" participate in a realistic software engineering process that involves real-world components not present in class projects, such as teams of people, large-sized projects, critical decision-making, personnel issues, multiple stakeholders, budgets, planning, and random, unexpected events. Moreover, the rapid and flexible nature of simulation allows experiences to be repeated, different situations to be introduced and practiced, and promotes a general freedom of experimentation in the training exercise.

Their research has shown that on average, students found the game quite enjoyable to play and relatively easy to play. They also felt that it was moderately successful in reinforcing software engineering process issues taught in the introductory software engineering course they had taken and equally successful in teaching software engineering process issues in general.

Perhaps most indicative is the positive answer from the students on the question as to whether the game should be incorporated as a standard part of a software engineering course, clearly a vote of confidence by the students who participated in the experiment.

Students were also asked to answer some open-ended questions about the game and the researchers found that their responses to these questions also reflected a positive attitude about the simulation. [Navarro et al, 2004]

2.08 Teaching Methods

Based on the software engineering methods, many instructional design methodologies have been developed and many others are emerging to face the problem of the course design, each of them carrying the same advantages and disadvantages, limits, conditions of applications and problems such as the originating methods.

Today, the most common methodology in European schools is the Dick and Carey one [Dick & Carey, 1990], or its several variants. Some of the typical problems with the application of this method are the marginal role of students in the instructional design process and the related problems of satisfaction; the often unbalanced student workload for each subject and term; the compelling choice of materials and technological tools at the beginning of the instructional process; the scheduling and the revision of the plan in case of failure. These problems are very similar to those arising from the application of traditional software engineering methods like, for example, the waterfall model.

In the book Agile Instructional Design, Peter Rawsthorne maintains that the software engineering methodologies have had influence over Instructional Design methodologies and, now agile methodologies are spreading and then they can provide new techniques to instructional design methodologies. According the author, from the point of view of learning theories, Agile Instructional Design methods enhance constructivism as they involve the learner in the curriculum development process.

In her 2007 book, De Vincentis [De Vincentis, 2007] argued that "students need to develop not only excellent numeracy and literacy skills, but problem solving skills, creative solution skills, strategy skills, relationship skills, think-on-your-feet skills" to be able to create new jobs and also be able to switch jobs easier in a fast pace environment.

The author observes that curricula are essentially based on values and the essential learning and that life skills are considered through the blending of discipline based content with values and life skills they. Moreover, equity and standardization are opposite and teaching is often for testing. She proposed a student-centric approach based on the following Agile Education Manifesto based on the Agile Manifesto by placing individuals and interactions over processes and tools, working education over comprehensive documentation, customer collaboration over contract negotiation, and responding to change over following a plan. [De Vincentis, 2007]

Current instructional design methodologies have been found to have various shortcomings when it comes to projects.

The following are highlighted in a technical report by Domenico Lembo and Mario Vacca:

The ineffectiveness of documentation and plan revision

The project plan related to each subject is made at the beginning of the course; each time that some problem happens imposing the plan revision, the new project plan should be rewritten in order to make this document effective.

Rewriting documents requires time and efforts to make the document coherent with the others related to the other subjects.

The time scheduling of the project

It is well known that projects are often delayed; it can happen that, in order to meet deadlines, the realization of some activities could be accelerated, yielding problems in student understanding skills and increasing the workload.

The marginal role of students in the design of course

Students have little space in the course design: they participate a few times a year to teachers' team meetings.

From an engineering point of view, the students role is essentially limited to the validation of the instructional contract.

The unbalanced student workload

Because of each discipline produces its own workplan applying the more appropriate method and without a detailed verification of the relations among the modules or activities of the different disciplines, it could be possible that in some period the student workloads results unbalanced or unbearable, producing negative results on the quality of learning. [Lembo & Vacca, 2011]

2.09 Virtual methods

Several approaches have been developed to teach Agile Methods in software engineering courses through capstone projects. A capstone project is a cooperative assignment that aims to provide students with an opportunity to integrate the concepts learned previously, deepen their understanding of those concepts, extend their area of knowledge; students are expected to apply their knowledge and skills in a real world experience.

Alfonso and Botia [Alfonso & Botia, 2005] compared the results of teaching Scrum practices in an undergraduate software course with a previous experience using a waterfall−like rigid process, resulting in a decrease in risks and process overhead. It is reported on the use of capstone projects to initiate future software professionals in the importance of agility, flexibility, and adaptability in professional contexts. By carrying out a project-work, the students followed an iterative and incremental teaching approach based on XP, Scrum and FDD. Mahnic [Mahnic, 2012] taught Scrum through a capstone project and described the course details, students' perceptions and teachers' observations after the course.

However, these approaches fail to address the teaching constraints in a university course such as large classes, multiple groups working at a time, limited space, and limited number of tutors. Using a room for multiple teams may jeopardize the effective implementation of the Scrum process since each team may require customized configurations of the room, whereas physical space and teaching materials for a personalized class may prove unviable.

In order to prepare students for their professional experiences in commercial software development, where Agile Methods are widely used, some researchers have introduced Scrum in software engineering courses by means of a capstone project. However, teaching through a capstone project is a challenging task, due to drawbacks such as complexity of projects, lack of physical resources, numerous groups of students and packed schedules, which affect the normal running of the course.

Another increasing trend in teaching Scrum practices is the use of simulation games, which may act as a bridge from academic knowledge to the industry . Along this line, the research [Rodriguez et al, 2015] found a number of agile-based games commonly used in industries and coaching, designed to strengthen and fix concepts and practices. These games require physical presence and are useful to learn how to adapt to changes in software requirements, customer management strategies and self-organizing dynamics in agile teams, among others; since they are cornerstone of Scrum. In the academic context, they found PlayScrum, a card game to allow university-level students to learn Scrum; just as in Virtual Scrum, students play different Scrum roles in a software development project. Unlike PlayScrum, Virtual Scrum uses a virtual world to simulate a real work environment handling 3D displays of the Scrum artifacts.

Games such as Agile Hour and XP Game teach XP practices through a web site; these practices complement the Scrum ones, by covering programming practices, and their incorporation in Virtual Scrum would maximize students’ programming skills. However it is noted that the use of 2D games results in unsuitable strategies to provide an immersion experience and realistic display of a Scrum environment.

In this context, virtual reality arises as an innovative technology that enhances students’ motivation, understanding and creative learning. Certainly, virtual worlds have great potential to make the learning experience both challenging and appealing by providing visuals that are easier to retain. Moreover, the user representation through avatars allows more active participation by manipulating the 3D artifacts.

The use of virtual reality has been addressed in several approaches to teach in various fields of science and engineering in university courses. In, the authors explored a 3D learning tool to easily observe and handle internal structures of a generating unit of energy, reducing the gap between the real experience and theoretical concepts in the area of systems of power.

In, a virtual−lab relied on an HTML Web page for control education is presented to illustrate the dynamic behavior of an industrial boiler in a user−friendly way. Diedro−3D is an application that addresses the challenges that architecture students face when using descriptive geometry, giving students greater autonomy to study geometry. In, a project to develop, build up, and evaluate solutions for virtual mobility and e−learning in biomedical engineering is presented. The authors included different educational materials to provide students with the possibility to revise course contents at any time, place, or pace. In, the authors studied the feasibility of introducing SimuSurvey in regular surveyor training courses to support students' learning experience. Along this line, the use of virtual reality simulations has also been utilized in to teach earth science concepts, improving students' spatial abilities. In addition, a 3D virtual hydraulic model to facilitate the teaching of hydraulic engineering is presented in, resulting in a reduction of time and effort in comparison with experiments in a traditional laboratory.

All the above-mentioned approaches support the notion that virtual worlds and simulations cater for teachers' needs in traditional courses, such as acquisition and maintenance of equipment for each group of students, the influence of location and time of the class on the effectiveness of the lesson, and the difficulty of clearly demonstrating every step to each student in the class at a suitable pace. What is more, virtual worlds provide students with hands-on experimentation without hazardous and costly laboratories.

In the context of the research, Virtual Scrum uses a virtual world to harness the 3D interfaces for training students in their performance in a simulated Scrum environment. As far as software teaching with virtual worlds is concerned, some approaches have been introduced. For instance, Ye et al. used a virtual world to enhance software engineering education by exploiting communication and collaboration tools to teach practices such as problem solving, plan formulation, interpretative analysis, and adaptation to rapid change, in a virtual office provided by Second Life. In addition, in order to evaluate the effects of Second Life when playing games, the authors developed the Second Life version of two so-called games to introduce software engineering practices: Groupthink and SimSE. The former focuses on teaching software specification practices, and the latter aims to train in project management skills. Likewise Ye et al., Parsons et al. used a 3D virtual world based on Open Wonderland (originally Project Wonderland) to support a workshop activity based on agile software development processes. This workshop enables students to take part in this activity despite their geographical distribution.

The approach tackles problem solving by defining and specifying user stories, and employing a shared board, in which users may sketch prototypes of designs. Improving the aforementioned approaches, Virtual Scrum supports a task board for planning and tracing user stories; a daily meeting artifact for solving problems, adapting rapidly to changes and removing impediments; and a burn−down chart for reflecting on the past sprint and making continuous process improvements during retrospective meetings. Furthermore, unlike Parsons et al., they concentrate on using Virtual Scrum to teach Scrum assisting professors and students in the development of a capstone project.

According to the data collected from the 45 students surveyed, we found that using Virtual Scrum, as a teaching aid, was helpful in improving students’ comprehension of the fundamentals of agile practices and principles of developing software with Scrum. It is worth noting that the tool outperformed students’ expectations with regard to the Virtual Scrum support for planning meetings, which increased students’ commitment; and follow−up metrics, which allowed students to self−reflect on their performance in the Sprint Retrospective meetings.

The students also provided valuable feedback on user interactions and traceability of the user stories through Virtual Scrum. Based on this feedback, we will intend to improve user interactions by upgrading the support of media aids, specially the avatar integration with current social networks. As for traceability of user stories, we found that students preferred using 2D tools for dealing with configuration management rather than a 3D representation of the artifacts; for this reason, we will complement Virtual Scrum with conventional and open−source development tools.

As a further stage in this research, we are planning to incorporate an intelligent agent to offer students personalized assistance during the use of Scrum. To tackle this issue, we will equip Virtual Scrum with performance indicators to obtain more information about students' interactions with the tool. This information will be useful to proactively assist the student by suggesting him/her personalized courses of action that will help them during the Scrum process. [Rodriguez et al, 2015]

2.10 Agile Teaching / Learning Methodology

Researchers at the City University of Hong Kong have developed the Agile Teaching / Learning Methodology (ATLM) [Chun, 2004] that is a teaching/learning methodology designed for higher-education based on the best practices and ideas from the field of software engineering and leveraging upon concepts and ideas from the field of software engineering and leveraging upon concepts from agile software methodologies.

Although ATLM was designed using concepts borrowed from software engineering, the methodology itself can easily be applied to a wide variety of courses that might require agility in teaching and learning.

An e-learning platform has been developed that makes use of a number of modern collaboration and knowledge sharing technologies such as blogging, commenting, instant messaging, wiki and XML RSS.

ATLM has been applied to the teaching of several Computer Science courses at the City University of Hong Kong for a number of years. Although used for teaching technology related courses, it is believed that the methodology itself is general enough to be applied to others disciplines as well. ATLM encourages communication, knowledge sharing and the learning process to nurture self-learning individuals. ATLM has been developed to support both teaching and learning.

It has been found [Chun, 2004] that the teaching process itself is very similar to the software development process in many ways. It involves multiple parties with different objectives, a very tight schedule to get things done, a fixed deadline, limited resources and a lot of changes along the way. Both the teaching and software development processes require detailed planning, tracking and management with continuous assessment and feedback from all parties. Making sure a course is taught properly and on schedule can also be challenging similarly to getting a software project done correctly and on time.

There are numerous teaching/learning best practices that can be practiced with ATLM. However, ATLM particularly promotes and emphasizes the following as part of the methodology: learn by sharing, teach how to learn and feedback is good.

ATLM makes use of the fact that students learn over an order of magnitude better if they also participate in the teaching process. The methodology facilitates this through knowledge sharing exercises.

It also emphasizes that in addition to the course content, it is important to also teach the learning process. ATLM does this through guided and targeted independent study tasks with knowledge sharing and collaboration as motivation.

Lastly, feedback is what makes ATLM agile. Without feedback, the teacher will not be able to improve the course delivery and teaching. Without feedback the students will not know if their work is on track and inline with expectations. ATLM facilitates feedback through informal weekly quizzes and feedback forms, surveys and online comments. [Chun, 2004]

2.11 Anti-practices while teaching Agile

Researchers at the University of Sao Paolo [Freire et al, 2007] have identified three organizational anti-patterns that are common and recurrent both in industrial and academic environments. They have presented “Bootstrap”, “Split Personality” and “Abandon Complex” in the form of a small anti-pattern language and offered different solutions to these anti-patterns.

Bootstrap addresses the issue of a team that starts to learn XP in a project with little or no code base. The team is new to XP and needs to be quickly productive. When starting a project from scratch, they have shown that allowing a small subset of the team to bootstrap the code base with the business model classes and then focusing on new business rules or variation storyotypes, is a simple solution to bootstrap. Using free and open source software as an initial code base is also a solution that can help teams strengthen other practices and techniques, such as testing. When the team is learning XP and other techniques, using storyotypes to split up the bootstrap story, and allowing code to be committed without complete test coverage, only during a short period of time, is a solution to bootstrap.

Split Personality addresses the issue of a person on the team being overloaded with the roles of Coach and Customer. When there is the possibility, one should use one or more Customer Proxies or a real Customer. When everything else fails, rely on the simple solution of using a hat or gadgets to distinguish clearly when one is acting as Coach or as Customer.

Abandon Complex describes troubles a team might face when the Coach has to leave. Instead of electing the most experienced developer to act as Coach, it is suggested by the research that the best solution is to have Champion of the Court accept responsibility, or practice Coach of the Week. [Freire et al, 2007]

2.12 Using Agile to teach Agile at IBM - Agile@IBM

IBM applied an Agile approach when its current framework by itself was not enabling the rapid response it needed. IBM's learning design team developed an instructional program to teach Agile software development at a time that there was no time for drawn-out analysis. What they decided to do is to use Agile methods to teach Agile. [Hall, 2012]

They created Agile@IBM — a program of instruction designed for software developers, engineers, testers, and leaders. Although the learning design team had managed challenging assignments in the past, Agile@IBM was different. In this case, the stakeholder was challenging the team to apply a new methodology to their classic design approach.

The main constraints encountered in the Agile@IBM project were the ever-changing requirements and short delivery schedules. At IBM, the frequent focus on technology topics decreases the shelf life learning content. Then, as learning content changes, so does the roster of content experts.

Meanwhile, IBM's acquisitions strategy and geo-expansion strategy have created changes in the makeup of its workforce as well as in the culture-based expectations of its learners. It is also found that it is needed to deliver learning in countries with technology infrastructure obstacles; in many cases the usual learning delivery platforms will be unusable in those locations for the foreseeable future. Finally, IBM operates in a highly competitive environment in which its immediate business goals are in a state of constant change.

It is not only learning design and technology that are changing; the learning experience itself is changing as well. IBM's learners used to be called away from their desks to participate in formal training sessions.

Now they require work-embedded, social, and informal learning delivered at the point of need in a variety of flexible platforms to accommodate their travel schedules and time zone differences. The learners are being asked to be as flexible in their learning habits as they are in their work habits. Many of them are feeling the loss of classic face-to-face learning events.

Ultimately, the role of learning designers is changing. Many of those who used to work alongside content experts to create self-paced instruction now find themselves working with the learners to co-create learning experiences. It is also noticed that the stakeholders are becoming more involved in the design process and are expecting more flexibility from them.

According to the team at IBM's learning design, delivering module after module of e-learning is no longer sufficient. "It is as though we once built trains, and now we design access to trains by designing a track architecture. As we look at the trains speeding along the tracks and wonder at the growing infrastructure, we realize that we find it difficult to predict what will be asked of us in the future." [Hall, 2012]

The term Agile is often described as a systems engineering method. Yet IBM believes that "when we dig deeper we find Agile to be an overarching collection of practices influenced by many disciplines". [Hall, 2012]

For example, from software engineering, Agile inherits Extreme Programming and its practices enabling business leaders and software developers to work together to determine and attain shared, realistic goals.

From both systems and software engineering, Agile borrows the Rational Unified Process and its iterative development methodology, which delivers useful output every few weeks.

From manufacturing, Agile gains Lean's emphasis on the elimination of waste. From product development, Agile inherits short, daily "scrum" update meetings that facilitate collaboration and keep teams focused on their incremental deliverables.

According to IBM [Hall, 2012], Agile adapts well to additional domains, including that of corporate learning. Currently, Agile is used by IBM instructional design teams to enable them to adapt to changing requirements, reduce risk of projects, increase visibility of projects' progress, involve stakeholders and learners from the beginning of the project onward and accelerate the value it brings to the business.

IBM suggests that those interested in pursuing Agile Learning Design, will need to follow one or more of these Agile key practices:

Emphasize individuals and interactions over processes and tools

With Agile, the team is all important. It is self-directed and regularly examines its own performance and seeks opportunities to streamline. Short, daily scrum meetings enable team members to share status and assist each other in timely fashion. IBM has found that scrum meetings significantly decreased time to delivery for their leadership development programs.

Process is regarded with a healthy degree of suspicion because it is associated with overhead and project bloat. At IBM, project managers and team leaders are provided with an Agile learning curriculum [Hall, 2012] as part of their certification requirements so they can help accelerate adoption of Lean methodology rather than delay it.

Tools that are intuitive and quickly deployed are generally preferred

Rapid prototyping finds itself right at home with Agile. According to Catherine Rickelman [Hall, 2012], IBM found rapid prototyping to be indispensable because it enabled them to lead discussion within an unusually large and diverse group of stakeholders.

Emphasize usable deliverables over comprehensive documentation

In the past, every day spent designing and developing learning was a day that employees went without the benefit of that learning. With Agile, the emphasis is on enabling employees to learn immediately and leveraging their experiences to drive improvements into the continuously improving overall learning experience. Rather than developing module after module of formal self-paced instruction, we emphasize providing access to content and designing learning experiences that use wikis, blogs, forums, surveys, and dashboards.

Documents and other artifacts are kept small in number and in size

Most are either of a throwaway nature, as is the case with rapid prototyping, or are living documents, as is the case of backlogs and “burndown” charts.

Emphasize collaboration over negotiation

Agile brings stakeholders into the project as fully embedded team members, ensuring they have continuous input in the project as well as in-depth knowledge of its progress. The media specialists, programmers, and educational specialists comprising the learning design team meet regularly with stakeholders from the earliest design discussions and prototypes.

The learners themselves are viewed as stakeholders

The use cases enable a team to view the entire project from the point of view of a typical learner right from the start. Throughout the project, iterative releases combined with feedback avenues enable a trial-and-error approach. The result is a significant reduction in risk to the project.

Emphasize responding to change over adhering to a plan

Those new to Agile often are surprised that changing requirements are welcomed, even late in a project. Rather than rein in change, Agile projects harness it to competitive advantage. Short iterations; lightweight processes, tooling, and documentation; and early and continuous feedback from business leaders and learners all work together to ensure that learning teams don’t fall behind the change curve.

Problems that no longer exist or are no longer important are easily tossed aside

Learners become confident that returning to the sites will expose them to fresh content and an improved learning experience. Perhaps most surprisingly, learning designers and developers find that they are able to maintain a comfortable, constant pace because they are not tethered to long release cycles and unexpected demands.

When the Agile@IBM initiative began, the learning professionals at IBM were new to the idea of Agile Learning Design. Nevertheless, they were able to successfully manage iterative design and development cycles, integrate the stakeholders and users from the start, and use their feedback and involvement to refine not only their solutions but sometimes the work processes themselves.

By applying Agile practices, they managed to roll out an extensive set of successful learning solutions. They began with a workshop for software development teams, then added workshops for project managers and team leaders. To promote inter-team collaboration, they deployed town halls, learning “suites” that gathered blended learning, a community of practice, and shared stories about best practices and lessons learned from real software projects.

They also created video lectures for those who couldn’t attend the face-to-face workshops. Lastly, live virtual classroom meetings enabled some software developers who were globally dispersed to work together on particular areas for improvement. The team also implemented a performance monitoring dashboard to track progress in the adoption of Agile practices as they moved toward their intended end state.

From this process, it was found [Hall, 2012] that social learning techniques work hand in hand with the Agile approach. Social software is enabling users to learn from peers who are applying the learning themselves, as well as from experts who are eager to distribute their knowledge to as many people as possible. This close collaboration brings realistic, workable, and current solutions directly to the learners.

“Seeing immediate results from applying these solutions is what matters to the learners today”. [Hall, 2012]

3. METHODOLOGY

In this section the research methodology and the way data was collected will be discussed. For this research the population examined were past students of Greek Open University. In particular students of Computer Science courses were required to answer a questionnaire regarding their experience and attitudes towards agile teaching. The students have been taught agile by participating in a capstone project during their course and by participating in the research will offer valuable isight into the validity and effectiveness of the method.

The first step was to come up with the appropriate questions to support the aims of the research as the design of the questionnaire is fundamental to the scope of research. The scope of the research is to collect the data from the students and point to any differences between them. The questions were sourced from the literature review that has taken place before it in regards with agile teaching methods and were designed to specifically serve the aim of the research.

The aim of the research is to identify patterns between students that have been taught agile and the differences in attitudes related to their experience with it. It will highlight the students' perception towards the importance of agile practices teaching, which practices are considered most relevant, their potential difficulty to master as well as their importance to potential employers. Lastly, it will try to rebuff the criticism against agile teaching and also determine whether the focus of agile teaching in university is aligned with the needs of the market. If not, will provide suggestions towards it.

The research took place by uploading the questionnaire online using Google Forms and sending it out to the selected students. Before sending out the questionnaire, it was tested internally in a pilot research to ensure its validity as in its structure, language and questions asked. During the pilot research it was established that the questions were easily comprehendible, engaging and offered answers relative to the scope of the research.

The questionnaire was sent out to 250 current and formers students at the Hellenic Open University. All of the students have or are currently attending a computer science course at the university that included agile teaching. The students had 15 days to answer and of the total 250 students, 61 have replied to the questionnaire, a percentage of 24.4%.

The questionnaire that was handed out to the students had three sections:

The first, included questions regarding the level of familiarity of the students with the teaching of agile both in university courses as well as in their work environment and some personal information about them.

The second, compiled of questions regarding the students' general attitudes towards agile teaching, included specific questions about the perceived effects of agile teaching, its benefits as well as concerns and criticisms against it. The students were asked to evaluate views towards agile that are found in bibliography and assess their validity, in context of their own experience with agile teaching. Furthermore the students were asked to validate the usefulness and real world usability of agile practices taught during their course at the university.

The third section focused on the views students had regarding the quality and focus of the agile teaching they received, its effect on group work as well as relevance to industry practices. At this section the students were also asked whether agile teaching has effectively improved their employability, a key selling point for most undergraduate programs.

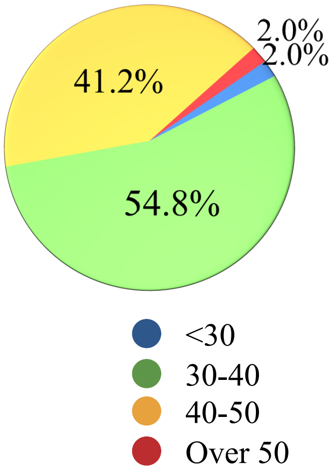
At the end of the questionnaire, the participants are asked to provide any other comments they might have regarding agile teaching by filling in an optional box.

The data collected by the questionnaire were examined and presented for statistical analysis to summarize and describe the data and graphs or tables were used to visualize the data and analyze variable frequencies. Descriptive statistics were used to summarize the data whereas inferential statistics were used to identify statistically significant differences between groups of data. T-test was used for comparing between student groups with different characteristics, such as levels of familiarity with agile or use in business. In cases where the groups are more than two, they will be examined by using analysis of variation (ANOVA). Both methods provided an insight in regard to the differences in views between the students as well as the alignment of agile teaching in the university with the use of agile methods in the workplace.

4. FINDINGS

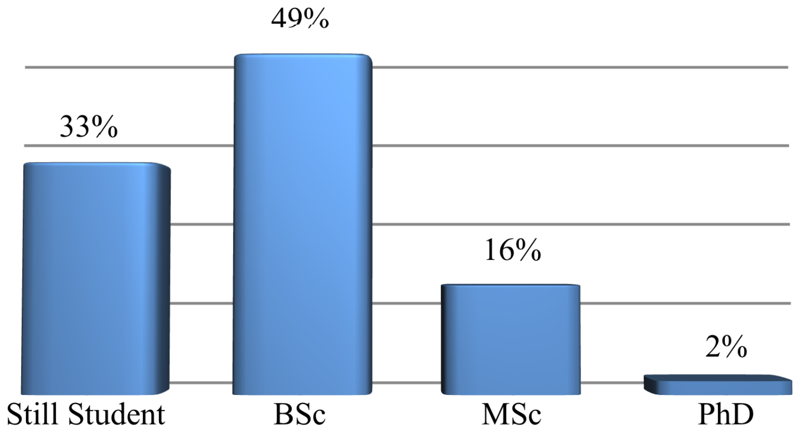
The age of the sample proved to be very homogenous with the majority (96%) of the students that participated in the survey being between 30 and 50 years of age. From the total number of participants 54.8% were between the ages of 30 and 40, 41.2% were between the ages of 40 and 50 while 2% were under 30 and another 2% were over 50 years of age.

Chart 1: Age



Apart from their age the participants were required to provide information regarding their education level. Almost half of them (49%) hold a bachelors degree, 16% hold a masters degree, 2% hold a Phd while 33% are still studying to obtain their first degree.

Chart 2: Education Level



The next set of the personal questions gathered information regarding the work experience. In the first question the participants were required to answer at which sector they work if they do as well as how many years they have been working on developing software. The results are highlighted in the graphs below:

Chart 3: Line of work

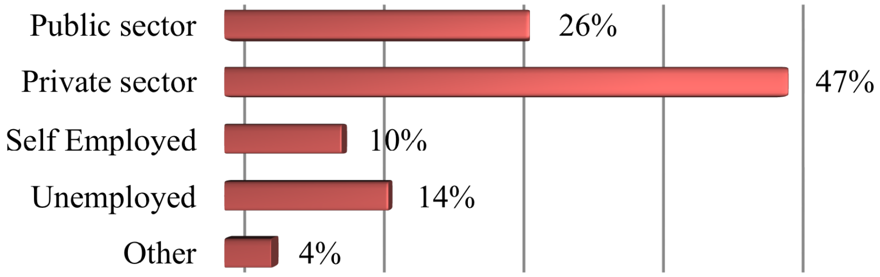
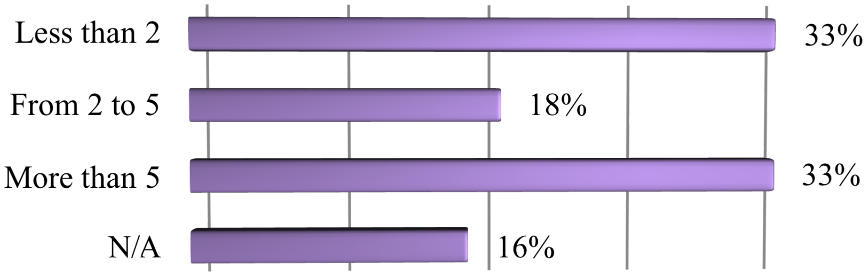


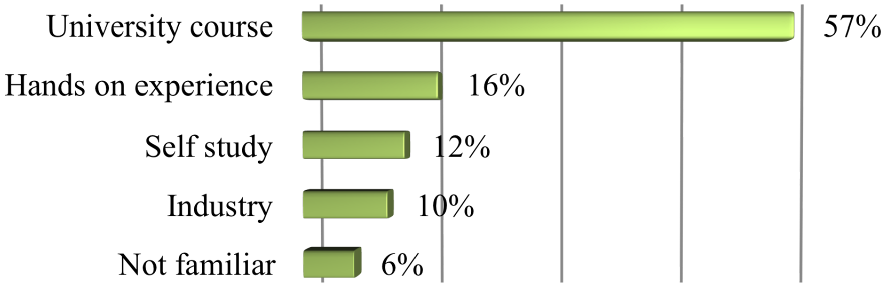
Chart 4: Work experience



The last of the personal questions was on where the participants had originally learnt agile.

A majority 56.9% learned agile methods through a university course signifying the importance of agile teaching at software engineering courses.

Chart 5: Way of learning agile



The other options were hands on experience (15.7%), self study (11.8%), industry (9.8&) and not familiar (5.9%)

The second part of the questionnaire included questions that highlight the students’ views regarding learning experience through the capstone project. The participants were required to answer the questions on a 1-5 scale, with 1 being strongly disagree and 5 being strongly agree. Their answers to these questions are summarized in the table below:

Table 1: Learning experience

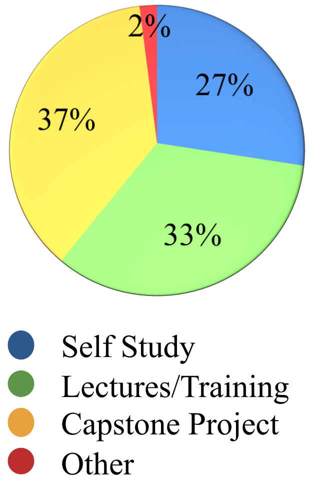
Item	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
The use of agile methods improve performance	2%	9.8%	35.3%	41.2%	11.8%
Working in a capstone project helped me accomplish my learning objectives more easily	2%	5.9%	39.2%	45.1%	7.8%
Learning agile methods through a capstone project suited my way of learning	2%	9.8%	43.1%	37.3%	7.8%
After finishing the project, I was able to identify and use many of the agile methods/ practices	2%	7.8%	37.3%	43.1%	9.8%
Working on the capstone project reduced my need for training at work	3.9%	17.6%	58.8%	19.6%	0%

Item	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
My overall experience with the capstone project was positive	2%	3.9%	27.5%	49%	17.6%
My experience on the project improved my professional performance	5.9%	7.8%	35.3%	41.2%	9.8%
The skills I learnt from the capstone project improved my employability	5.9%	9.8%	43.1%	35.3%	5.9%
The number of different skills required to be an agile developer made it hard to assemble groups	0%	17.6%	51%	27.5%	3.9%
Group members were often valued for their social skills instead of technical skills	2%	9.8%	54.9%	21.6%	11.8%

For the first 2 questions it should be noted that a majority of over 50% agrees that the use of agile methods improves performance and that their work in the capstone project assisted them in accomplishing their learning objectives. The majority of the participants felt confident identifying and using many of the agile methods they learnt through the capstone project after its completion.

However when asked whether learning agile methods through the capstone project suited their way of learning, only 45.1% agreed. 11.8% said it did not while 43.1% percent were indifferent. When asked which method they considered to be the most effective way of learning agile, opinion was divided between the students with 37% picking capstone projects, 33% picking lectures or training and 27% picking self study.

Chart 6: Preferred way of learning



Nevertheless, the majority of the participants (56.6%) agreed that their overall experience with the capstone project was a positive one while 51% believes that their experience on the project improved their professional performance. 41% of the respondents believe that the skills the learnt from the capstone project improved their overall employability as well.

However it should be noted that when asked about whether their experience with agile methods during the capstone projects reduced the need for training at work, the participants gave mixed answers.

Lastly, on the two questions regarding the criticism against the use of agile methods in group work the participants rebuffed the notions that the number of different skills required to be an agile developer made it hard to assemble groups and the in agile groups, members are often valued for their social skills instead of technical skills. In both question the vast majority of the participants fluctuated towards the centre of the scale.

The next question focused on the perceived advantages of using agile methods as part of their capstone project and its effect on the end result. Most students agreed that agile methods indeed improve software quality, increase flexibility, offer faster delivery and boost team morale as highlighted in the table below:

Table 2: Advantages of using agile methods

Item	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Improved software quality	0	3	14	31	3
Increased flexibility	1	0	19	22	9
Offered faster delivery	0	7	20	19	5
Improved team morale	1	4	16	24	6
Improved customer satisfaction	2	1	29	12	7

However the results where not clear in regard to agile methods offering improved customer satisfaction. More than half of the students that answered the question selected the option in the middle of the scale indicating mixed results in their experience.

When asked about the skills gained from their participation in the capstone project, the results were very positive (table 3). It has been highlighted in the bibliography that while these skills are considered very important for an agile developer, there is a lack of supply in the job market.

Table 3: Skills gained from capstone project

Item	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Familiarity with software development methods	4	0	18	25	4
Familiarity with software development agile methods	3	1	13	30	4

Item	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Knowledge of programming environments and programming skills	4	1	20	22	4
Analysis and design skills	3	2	21	20	5
Project management and planning skills	3	3	16	21	8
Effort estimation skills	3	3	19	22	4
Team working skills	2	2	14	23	10
Communication skills	2	3	18	18	10

The majority of the students agreed that participating in the capstone project helped them develop the skills in question.

The next part of the questionnaire focused on the individual agile practices, their difficulty to comprehend and their perceived usefulness in business. The fist question aimed to establish how often each practice was used during the capstone project. The answers to the question are show in the table below:

Table 4: Practice use frequency during capstone project

Item	Never	Rarely	Sometimes	Mostly	Systematically
Iteration Planning	4	4	27	13	3
Daily Standup	4	13	19	11	4
Unit Testing	6	4	16	16	9
Release Planning	7	5	21	16	2
Retrospectives	8	9	24	5	5

Item	Never	Rarely	Sometimes	Mostly	Systematically
Burn down	8	10	23	8	2
Continuous Integration	9	3	21	16	2
Automated Builds	12	8	16	15	0
Velocity	9	5	20	13	4
Refactoring	4	12	17	12	6
Coding Standards	5	8	15	22	1
Test Driven Development	6	9	11	21	4
Pair Programming	6	5	21	16	3
Continuous Deployment	8	4	16	18	5
Behavior Driven Development	9	7	18	15	2

The most used practices were TDD, Coding Standards, Unit Testing and Continuous Deployment. The least used were Automated Builds and Burn-down.

For measure we asked the students how hard to comprehend they considered each practice to be:

Table 5: Practice difficulty to comprehend

Item	Very Little	Little	Average	Fair	Significant
Iteration Planning	2	8	28	9	3
Daily Standup	6	7	21	13	3
Unit Testing	5	4	24	10	6
Release Planning	3	6	23	11	3
Retrospectives	4	10	27	4	4
Burn down	4	13	22	6	4

Item	Very Little	Little	Average	Fair	Significant
Continuous Integration	3	5	21	16	4
Automated Builds	8	7	18	11	4
Velocity	6	6	21	11	5
Refactoring	4	7	22	9	7
Coding Standards	4	8	17	13	6
Test Driven Development	5	7	16	13	7
Pair Programming	5	7	17	14	6
Continuous Deployment	5	3	24	13	4
Behavior Driven Development	4	6	24	13	5

The most difficult were Pair Programming, TDD and Continuous Integration with 20 students total claiming they required a fair or significant amount of effort to comprehend.

The students were also required to offer their input in regard to the usefulness of different agile practices in the workplace:

Table 6: Practice workplace usefulness

Item	Very Little	Little	Average	Fair	Significant
Iteration Planning	2	2	12	12	4
Daily Standup	1	3	15	6	7
Unit Testing	0	2	10	14	6
Release Planning	2	3	14	9	3
Retrospectives	2	3	16	6	3
Burn down	1	5	14	9	2

Item	Very Little	Little	Average	Fair	Significant
Continuous Integration	0	3	13	13	3
Automated Builds	1	4	16	7	4
Velocity	1	3	18	5	5
Refactoring	1	4	12	9	5
Coding Standards	1	2	10	12	7
Test Driven Development	1	4	14	7	5
Pair Programming	1	3	15	8	5
Continuous Deployment	2	3	12	11	4
Behavior Driven Development	1	5	18	6	2

The majority of students considered all practices to be fairly valuable with not a great variance in their answers. However, the most useful practices identified were Unit Testing and Coding Standards. This falls in line with their use during the capstone project where Unit Testing and Coding Standards were identified as the most used practices there as well.

The most useful practices with the least effort were Unit Testing and Iteration planning while according to the students who participated in the research Daily Standup and Release Planning required more effort to comprehend and were less useful.

A T-test was done to determine the statistical significance in regard to portions of the sample valuing the capstone course in relevance to their work experience on work projects. In Group A were people with less than

5 years experience on projects and in Group B people with more than 5 years experience.

T-test results:

P value and statistical significance:

The two-tailed P value equals 0.0965

By conventional criteria, this difference is considered to be not quite statistically significant.

Confidence interval:

The mean of Group One minus Group Two equals 0.43

95% confidence interval of this difference: From -0.08 to 0.95

Intermediate values used in calculations:

t = 1.7011

df = 41

standard error of difference = 0.254

Another T-test was done to determine whether the sample's age is significant in perceiving the skills learned during the capstone course as significant. The two groups were separated by age with people under 40 forming Group A and people over 40 forming Group B.

T-test results:

P value and statistical significance:

The two-tailed P value equals 0.8653

By conventional criteria, this difference is considered to be not statistically significant.

Confidence interval:

The mean of Group One minus Group Two equals -0.05

95% confidence interval of this difference: From -0.69 to 0.58

Intermediate values used in calculations:

t = 0.1708

df = 38

standard error of difference = 0.315

A third T-test was done to determine whether students with previous knowledge on agile methods benefited more from the capstone course. In Group A were people with none or little experience on agile methods before the project while in Group B were experienced users of agile.

T-test results:

P value and statistical significance:

The two-tailed P value equals 0.0076

By conventional criteria, this difference is considered to be very statistically significant.

Confidence interval:

The mean of Group One minus Group Two equals -0.67

95% confidence interval of this difference: From -1.15 to -0.19

Intermediate values used in calculations:

t = 2.7851

df = 49

standard error of difference = 0.239

5. CONCLUSION

As the knowledge of agile is crucial to most companies nowadays it becomes significant for computer science courses to teach agile methods and practices to students.

This emerging use of agile has opened a gap between the skills taught in classic academic contexts and the ones required by the software industry. As a result, different ways of teaching agile have been adopted by different institutions based on their experience and resources.

Despite industrial adoption of agile methodologies, their acceptance and incorporation into academic curricula has been limited. Most popular software engineering texts, have been updated to include some coverage of agile methodologies but are still organized around traditional waterfall methodology.

Previous researches have claimed that while engineering practices can be taught very well in the classroom through lectures, management competences are best taught through student projects in teams.

Various methods of teaching agile methods have been utilized with different success levels according the research. In this paper we evaluated the use of a capstone project as a means to teach agile.

In order to prepare students for their professional experiences in commercial software development, where agile methods are widely used, some researchers have introduced agile in software engineering courses by means of a capstone project. However, teaching through a capstone project is a challenging task, due to drawbacks such as complexity of projects, lack of physical resources, numerous groups of students and packed schedules, which affect the normal running of the course.

Nevertheless, the student survey conducted received positive feedback in regard to the teaching method and the perception of the students towards it. The majority of the participants in the survey valued the capstone project as an overall positive experience and highlighted the advanced understanding of agile methods they achieved after the completion of the course.

The skills taught during the project were found to be in line with what is required in the industry nowadays. The students agreed that teaching agile methods in software engineering courses is necessary and the majority of them pointed to the use of capstone projects as the one that better suits their learning patterns.

Lastly, it was found that the capstone project had similarly positive learning effects for both younger and older students. However, it should be noted that students with previous experience of agile methods were able to benefit significantly more than their counterparts that had less experience on agile methods.

Overall the capstone project was found to be a very efficient and effective way of teaching agile to students and most concerns about its use found in bibliography did not appear in the research.

6. REFERENCES

Abelson, H. and Greenspun, P., 2001. Teaching software engineering-lessons from MIT. In Proceedings 10th International World Wide Web Conference.

Abrahamsson, P., Salo, O., Ronkainen, J. and Warsta, J., 2002. Agile software development methods: Review and analysis.

Alfonso, M.I. and Botia, A., 2005, April. An iterative and agile process model for teaching software engineering. In 18th Conference on Software Engineering Education & Training (CSEET'05) (pp. 9-16). IEEE.

Ambler, S.W., 2003. Introduction to Test Driven Development. Agile Database Techniques: Effective Strategies for the Agile Software Developer, John Wiley & Sons.

Andrea, J., 2001. Managing the Bootstrap Story in an XP Project. *Proceedings of XP 2001*.

Beck, K., 2000. Extreme programming explained: embrace change. Addison-Wesley professional.

Berczuk, S., 2007, August. Back to basics: The role of agile principles in success with an distributed scrum team. In Agile Conference (AGILE), 2007 (pp. 382-388). IEEE.

Briggs, T. and Girard, C.D., 2005, October. Comparison of student experiences with plan-driven and agile methodologies. In Proceedings Frontiers in Education 35th Annual Conference (pp. T3G-18). IEEE.

Chun, A.H.W., 2004, August. The agile teaching/learning methodology and its e-learning platform. In International Conference on Web-Based Learning (pp. 11-18). Springer Berlin Heidelberg.

Cockburn, A. and Williams, L., 2000. The costs and benefits of pair programming. Extreme programming examined, pp.223-247.

Coplien, J.O. and Harrison, N.B., 2005. Organizational patterns of agile software development. Pearson Prentice Hall,.

Coyle, S. and Conboy, K., 2010. People over process: key people challenges in agile development.

Crispin, L., 2003. XP Testing Without XP: Taking Advantage of Agile Testing Practices. Methods and Tools.

Cubric, M., 2013. An agile method for teaching agile in business schools. *The International Journal of Management Education*, *11*(3), pp.119-131.

da Silva, A.F., Kon, F. and Torteli, C., 2005, June. Xp south of the equator: An experience implementing xp in brazil. In *International Conference on Extreme Programming and Agile Processes in Software Engineering* (pp. 10-18). Springer Berlin Heidelberg.

De Vincentis, S., 2007, October. Agile education: Student-driven knowledge production. In ACEL/ASCD conference, New Imagery for Schools and Schooling Sydney.

Deemer, P., Benefield, G., Larman, C. and Vodde, B., 2012. A lightweight guide to the theory and practice of scrum (version 2.0). Technical report, http://www.scrumprimer.org.

Dick, W., Carey, L. and Carey, J.O., 1990. The systems design of instruction.

Downs, G., 2011. Lean-agile acceptance test-driven development: better software through collaboration by Ken Pugh. ACM SIGSOFT Software Engineering Notes, 36(4), pp.34-34.

Dutson, A.J., Todd, R.H., Magleby, S.P. and Sorensen, C.D., 1997. A Review of Literature on Teaching Engineering Design Through Project‐Oriented Capstone Courses. Journal of Engineering Education, 86(1), pp. 17-28.

Dybå, T. and Dingsøyr, T., 2008. Empirical studies of agile software development: A systematic review. Information and software technology, 50(9), pp.833-859.

Evans, E., 2004. Domain-driven design: tackling complexity in the heart of software. Addison-Wesley Professional.

Felsing, J.M. and Palmer, S.R., 2002. A Practical Guide to Feature-Driven Development. IEEE Software, 7, pp.67-72.

Fowler, M. and Foemmel, M., 2006. Continuous integration. *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf*, p. 122.

Freire, A., Kon, F. and Goldman, A., Three AntiPractices while teaching Agile Methods. Technical report, IME-USP, 2007. 5.

Goldman, A., Kon, F., Silva, P.J. and Yoder, J.W., 2004. Being extreme in the classroom: Experiences teaching XP. *Journal of the Brazilian Computer Society*, *10*(2), pp.5-21.

Hall, M.J., 2012. Are you ready for agile learning design?. Human Resource Management International Digest, 20(6).

Hazzan, O. and Dubinsky, Y., 2007. Why software engineering programs should teach agile software development. *ACM SIGSOFT Software Engineering Notes*, *32*(2), pp.1-3.

Hedin, G., Bendix, L. and Magnusson, B., 2003, May. Coaching coaches. In International Conference on Extreme Programming and Agile Processes in Software Engineering (pp. 154-160). Springer Berlin Heidelberg.

Highsmith, J., 2013. Adaptive software development: a collaborative approach to managing complex systems. Addison-Wesley.

Kropp, M. and Meier, A., 2013. Teaching Agile Software Development at University Level. IMVS Fokus Report.

Kuhl, J.G., 2014. Incorporation of Agile Development Methodology into a Capstone Software Engineering Project Course. In Proceedings of the 2014 ASEE North Midwest Section Conference.

Kuranuki, Y. and Hiranabe, K., 2004, June. Antipractices: Antipatterns for xp practices. In *Agile Development Conference, 2004* (pp. 83-86). IEEE.

Lindvall, M., Basili, V., Boehm, B., Costa, P., Dangle, K., Shull, F., Tesoriero, R., Williams, L. and Zelkowitz, M., 2002, August. Empirical findings in agile methods. In Conference on Extreme Programming and Agile Methods (pp. 197-207). Springer Berlin Heidelberg.

Mahnic, V., 2012. A capstone course on agile software development using Scrum. *IEEE Transactions on Education*, *55*(1), pp.99-106.

Mary, P. and Tom, P., 2003. Lean software development: an agile toolkit.

Melnik, G. and Maurer, F., 2002, August. Perceptions of agile practices: a student survey. In *Conference on Extreme Programming and Agile Methods* (pp. 241-250). Springer Berlin Heidelberg.

Meszaros, G., 2004, August. Using Storyotypes to Split Bloated XP Stories. In *Conference on Extreme Programming and Agile Methods* (pp. 73-80). Springer Berlin Heidelberg.

Mugridge, R., MacDonald, B., Roop, P. and Tempero, E., 2003, May. Five challenges in teaching XP. In *International Conference on Extreme Programming and Agile Processes in Software Engineering* (pp. 406-409). Springer Berlin Heidelberg.

Navarro, E.O., Baker, A. and Van Der Hoek, A., 2004, January. Teaching software engineering using simulation games. In ICSIE'04: Proceedings of the 2004 International Conference on Simulation in Education.

Palmer, S.R. and Felsing, M., 2001. A practical guide to feature-driven development. Pearson Education.

Rico, D.F. and Sayani, H.H., 2009, August. Use of agile methods in software engineering education. In *Agile Conference, 2009. AGILE'09.* (pp. 174-179). IEEE.

Rodriguez, G., Soria, Á. and Campo, M., 2015. Virtual scrum: A teaching aid to introduce undergraduate software engineering students to scrum. Computer Applications in Engineering Education, 23(1), pp.147-156.

Solis, C. and Wang, X., 2011, August. A study of the characteristics of behaviour driven development. In 2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications (pp. 383-387). IEEE.

von Wangenheim, C.G., Savi, R. and Borgatto, A.F., 2013. SCRUMIA—An educational game for teaching SCRUM in computing courses. Journal of Systems and Software, 86(10), pp.2675-2687.

Wellington, C.A.

Wiley, J. and Sons, A.M., Effective Practices for Extreme Programming and the Unified Process. Scott W. Ambler, http://www.agilemodeling.com/essays/agileDocumentation.htm.

APPENDIX

Questionnaire

Personal Information

1. Age

○ <30
○ 30-40
○ 40-50
○ over 50

2. What is the highest education degree you received?

○ BSc
○ MSc
○ PhD
○ Still Student
○ None
○ Other: ....

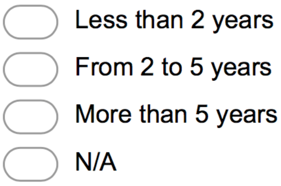
3. Currently you are working as?

○ Public sector employee
○ Private sector employee
○ Self employed
○ Unemployed
○ Student
○ Other: ....

4. How have you learnt agile methods?

○ I am not familiar with agile methods
○ From courses in industry
○ From courses at university
○ Self-study from books
○ Hands on experience
○ Other: ....

5. Please indicate the number of years you have been working developing software



- ○ Less than 2 years
- ○ From 2 to 5 years
- ○ More than 5 years
- ○ N/A

6. How many completed software development projects have you been involved with as an IT professional over the past five years?

0-40

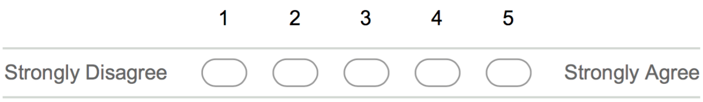
7. How many of the above projects made at least some use of agile methods?

0-40

Teaching agile methods

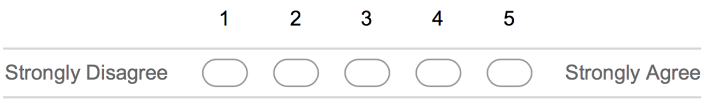
Please rate the statements below in line with your experience with agile methods

8. Agile methods are used at my workplace



	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

9. The use of agile methods improve performance



	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

10.As a student, which of the following education methods was the most useful?

○ Self study
○ Following lectures/training
○ Participating in a group capstone project
○ Other: ....

11. As a student, working in a capstone project helped me accomplish my learning objectives more easily

	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

12. As a student, learning agile methods through a capstone project suited my way of learning

	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

13. As a student, after finishing the project, I was able to identify and use many of the agile methods/practices

	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

14. The number of different skills required to be an agile developer made it hard to assemble groups

	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

15. Due to the nature of agile methods, group members were often valued for their social skills (eg. teamwork, leadership, communication) instead of technical skills (eg. programming, debugging, testing)

	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

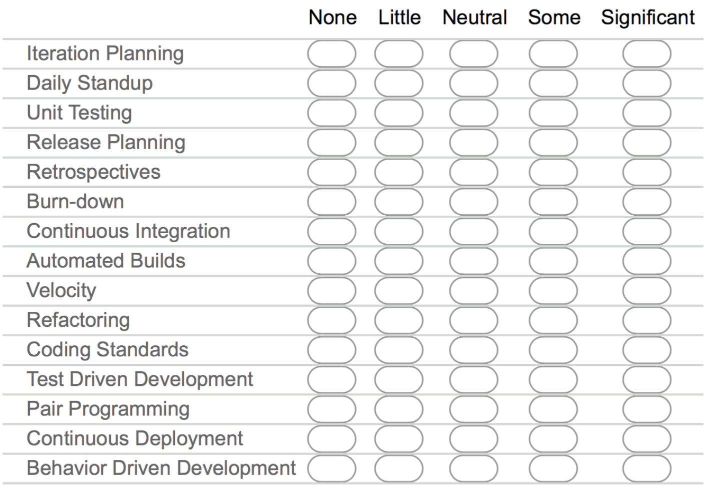
16. During your capstone project, how often did you use the following agile methods/practices? (Agile methods/practices can be found here: goo.gl/rBPhEq)

	Never	Rarely	Sometimes	Mostly	Systematically
Iteration Planning	○	○	○	○	○
Daily Standup	○	○	○	○	○
Unit Testing	○	○	○	○	○
Release Planning	○	○	○	○	○
Retrospectives	○	○	○	○	○
Burn-down	○	○	○	○	○
Continuous Integration	○	○	○	○	○
Automated Builds	○	○	○	○	○
Velocity	○	○	○	○	○
Refactoring	○	○	○	○	○
Coding Standards	○	○	○	○	○
Test Driven Development	○	○	○	○	○
Pair Programming	○	○	○	○	○
Continuous Deployment	○	○	○	○	○
Behavior Driven Development	○	○	○	○	○

17. Please rate the effort required to comprehend and use the agile methods/practices during your capstone project

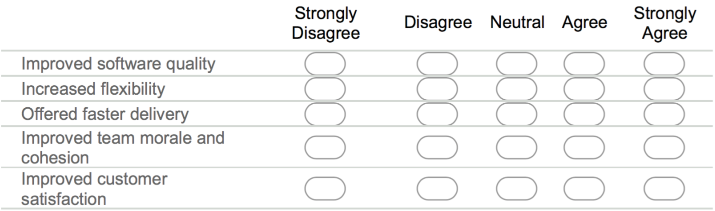
	Very Little	Little	Average	Fair	Significant
Iteration Planning	○	○	○	○	○
Daily Standup	○	○	○	○	○
Unit Testing	○	○	○	○	○
Release Planning	○	○	○	○	○
Retrospectives	○	○	○	○	○
Burn-down	○	○	○	○	○
Continuous Integration	○	○	○	○	○
Automated Builds	○	○	○	○	○
Velocity	○	○	○	○	○
Refactoring	○	○	○	○	○
Coding Standards	○	○	○	○	○
Test Driven Development	○	○	○	○	○
Pair Programming	○	○	○	○	○
Continuous Deployment	○	○	○	○	○
Behavior Driven Development	○	○	○	○	○

18. If you have used agile methods/practices at work, how do you rate their usefulness? (If you have not used them at work, please skip this question)



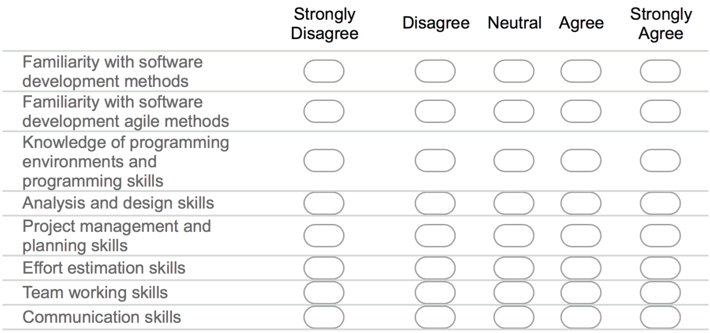
	None	Little	Neutral	Some	Significant
Iteration Planning	○	○	○	○	○
Daily Standup	○	○	○	○	○
Unit Testing	○	○	○	○	○
Release Planning	○	○	○	○	○
Retrospectives	○	○	○	○	○
Burn-down	○	○	○	○	○
Continuous Integration	○	○	○	○	○
Automated Builds	○	○	○	○	○
Velocity	○	○	○	○	○
Refactoring	○	○	○	○	○
Coding Standards	○	○	○	○	○
Test Driven Development	○	○	○	○	○
Pair Programming	○	○	○	○	○
Continuous Deployment	○	○	○	○	○
Behavior Driven Development	○	○	○	○	○

19. Implementing agile methods... (either as a capstone project or at a commercial project)



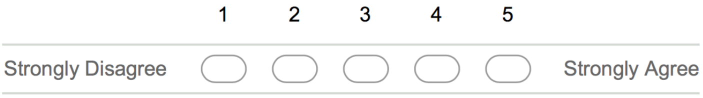
	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Improved software quality	○	○	○	○	○
Increased flexibility	○	○	○	○	○
Offered faster delivery	○	○	○	○	○
Improved team morale and cohesion	○	○	○	○	○
Improved customer satisfaction	○	○	○	○	○

20. Working on the capstone project has improved my…



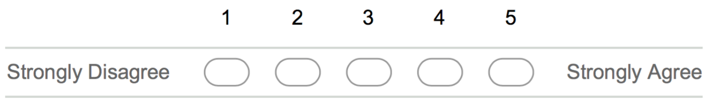
	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Familiarity with software development methods	○	○	○	○	○
Familiarity with software development agile methods	○	○	○	○	○
Knowledge of programming environments and programming skills	○	○	○	○	○
Analysis and design skills	○	○	○	○	○
Project management and planning skills	○	○	○	○	○
Effort estimation skills	○	○	○	○	○
Team working skills	○	○	○	○	○
Communication skills	○	○	○	○	○

21. Working on the capstone project has reduced my need for training on agile practices at work



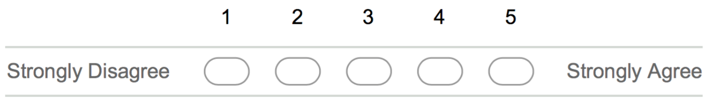
	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

22. My overall experience with the capstone project was positive



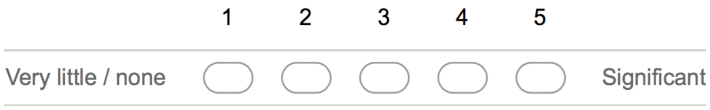
	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

23. My overall experience using agile methods/practices was positive



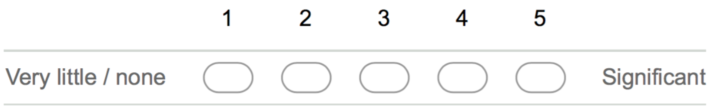
	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

24. Please rate your agile methods knowledge before finishing the project



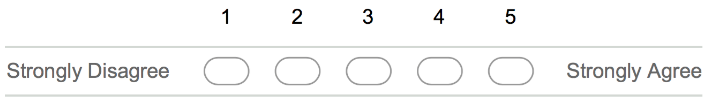
	1	2	3	4	5	
Very little / none	○	○	○	○	○	Significant

25. Please rate your agile methods knowledge after finishing the project



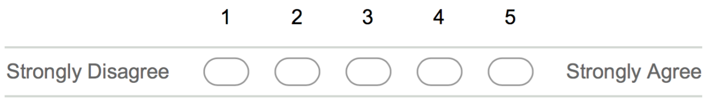
	1	2	3	4	5	
Very little / none	○	○	○	○	○	Significant

26. My overall experience on the project improved my professional performance



	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

27. The skills I learnt from the capstone project improved my employability



	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

28. Please write here any additional comments you have regarding teaching agile (optional)

………

………

………

………

………