AGILE SOFTWARE DEVELOPMENT IN EVOLVING BUSINESS ENVIRONMENTS: INTEGRATING MODERN TECHNIQUES INTO COMPUTER SCIENCE CURRICULA

Maher, Peter E.
Webster University

Kourik, Janet L.
Webster University

ABSTRACT
In today's fast-changing business environment software is playing an increasing important role in essential organizational operations. A great many challenges are arising throughout the software development lifecycle as budgets become tighter, applications expand, and technologies change rapidly. Such conditions often create difficulties including coding errors, customer dissatisfaction, changes to business requirements, personnel turnover, missed deadlines, and even cancellation of projects. Agile methods are a set of modern and successful concepts in software development that are increasingly being integrated in the global business environment; they are designed to address the common problems faced when developing software. As agile techniques become more prevalent in industry, it is becoming especially important to incorporate agile methods into traditional computer science curricula. In this paper we describe some of the key agile techniques, how they directly apply to today's business environment, and how they are taught in Webster University's agile development course. Further, we describe ways in which some of the concepts are presented and implemented in this course, and report on how these ideas influence the graduate program.

INTRODUCTION

The information technology industry is being transformed by the advent of innovative technologies, faster hardware, and an ever-expanding need for communication among global business entities. Within this context software is playing an increasing significant role. Larger applications are being developed, budgets are becoming tighter, and technologies are changing rapidly. As a consequence, software companies in all regions the world are experiencing many difficulties throughout the software development lifecycle. These issues include missed deadlines, cancellation of projects, application errors, failure to meet customer requirements, shifting business needs, and personnel turnover.

All of these issues must be anticipated and a software development methodology must be identified to successfully respond to potential obstacles. Traditional approaches to software development are centered on the assumption that business requirements are stable. With the assumption of stable requirements, a detailed, optimal design can be completed early in the project and a fully functional system can be delivered months later at the end of the project. There is a common belief that if the customer does not like the final product then there must have been errors in the implementation of the requirements. Such approaches are falling out of favor in today's dynamic business environment. Volatile customer requirements and swiftly-changing technologies may necessitate significant changes during a software project. As a result, a set of methodologies labeled "agile" has emerged to specifically address such current industry demands. The agile techniques are aimed at developing a

software development strategy that is sufficiently flexible to handle all of the potential problems described above.

The changing landscape of the software industry necessitates enhancing computer science programs to ensure that graduates are well prepared to work effectively in the international job market. Agile development can be viewed from a management standpoint in addition to an implementation perspective. Both views are necessary to developing a sound understanding of how problems faced by industry are to be addressed. Traditional computer science curricula, in general, have yet to embrace full-fledged courses on agile development. A more common approach is to embed some aspects of agile development in existing software development courses. As agile techniques continue to enter the mainstream, it will become increasingly important for students to learn agile methods. It is therefore critical that academics actively consider innovative ways to incorporate these techniques into computer science programs in order to best prepare students for the skills used in industry.

AGILE TECHNOLOGIES

In 2001 a group of highly experienced software methodologists initiated a process to address the major problems of software development. Their goal was to identify common characteristics of successful processes, not to describe a new methodology - indeed they made it clear that using one methodology for every type of application is not possible. The consensus of the group was that each application has its own needs, and methodologies need to be tailored to meet those needs (Cockburn, 2001). The group identified a set of principles that should be considered when determining an appropriate methodology. These principles were documented in the "Agile Manifesto". The key values that summarize the agile manifesto are shown in Table 1 below.

TABLE I
Agile Manifesto Values

Favored	Not Favored
Individuals and Interactions	Processes and Tools
Working Software	Comprehensive Documentation
Customer Collaboration	Contract Negotiation
Responding to Change	Following a Plan

The first value is to emphasize using the skills of the developers rather than relying on a specific, formal process or software tool to ensure the success of a project. Generally, the only real measure of progress in application development is *working software*. Placing an emphasis on producing documentation, particularly at an early stage of a project, is considered a mistake; the major reason being that the customer will not value documentation, however detailed, until the project is complete. Moreover, requirements will change, causing much of the documentation to become obsolete and needing to be re-produced.

An emphasis on ensuring effective and regular dialogue with customers is considered vital. Without close *collaboration* with the customer, the development team will have little insight into whether their effort is being well spent. Pre-determining a rigid project plan, at the outset of a new project, is

considered difficult to honor. There are so many variables that may change, thus following this plan may eventually become a hindrance rather than a benefit. Requirements will *change* – these must be accommodated within any methodology in a quick and effective manner. In order to recognize and incorporate change in a project, communication between developers and clients must be integrated into the working process.

With the above requirements in mind, the group identified four major characteristics that are fundamental to all agile methodologies:

a) Iterative
The *iterative* approach enables feedback to be received from the customer at regular intervals. This timely feedback gives the opportunity to make quick and effective changes to the application as issues arise, therefore avoiding the waste of resources in pursuing an inappropriate development path. A tenet for any agile process is that each iteration within the process must result in some measurable value, thus enabling meaningful feedback to be provided. Further, if the project were cancelled at the end of the iteration, the software as developed through this iteration could be released and deployed.

b) Communication
Efficient and effective *communication* is considered absolutely vital. To enable communication, it is always preferable to have a customer representative present at the developer's worksite. It is generally accepted that if the application is really worth building then it must be worth the outlay to place a customer onsite. Additionally, all channels of communication must always be open; for example, regular, brief team meetings, and an open workspace for ad hoc inquiry and discussion. Furthermore, developers should be encouraged to communicate with the customer directly when questions, particularly about business rules and practices, arise. More responsibility is therefore placed on the development team in an agile environment and more individual initiative is required.

c) Straightforward
Methodologies should be *straightforward*, and hence very easy to follow. Some traditional design and development methodologies require a lot of training and very close supervision. Such a learning curve, at least initially, has developers spending more time and energy on the methodology than on the software project itself. Agile values advocate for the opposite; that more time should be spent on the development project than on the methodology. The process of developing an application should not place any unnecessary demands on the development team members.

d) Adaptive
All agile methodologies should be *adaptive*; that is, the methodology must be able to react to changes as and when they occur. The problems when a development team does not understand the customer's requirements, as well as the issues that arise if an iterative process is not used, were described above. The agile approach is therefore to create a methodology that will gracefully adapt to misunderstandings and business changes. The earlier misunderstandings or changes are identified in a project the lower the cost to implement revisions. The advantage of agile methods is that developers must confer with the customer on a frequent basis as the project proceeds, offering more opportunities to align the work with the customer's needs.

As the software industry changes rapidly, computer science curricula must evolve to keep abreast of these developments. To be an effective member of an agile software development team in today's

business world requires skills that are not always common among typical computer science students. Graduate students are generally well-versed in technical skills, however, developing the students' abilities to communicate effectively with customers, adapt to fast-changing requirements, and provide accurate estimates for remaining tasks are all critical to preparing students to be agile team members. It is vital that students learn the necessary practical skills that will allow them to be productive as soon as possible upon graduation. Businesses want new employees to be productive and able to work independently as demanded by ever-increasing financial constraints. If current, state-of-the-art practical skills are actively being taught in the classroom then students will have at least some exposure to the skills that are often bundled under the phrase experience-is-necessary. Agile methods succeed in part because they respond to the ambiguities often found in industrial development projects. Such ambiguities are less common in classroom assignments. Introducing more realism, and its related ambiguity, requires that students be more engaged in the course material to be successful. Most students will greatly benefit from being exposed to sound examples of how theory is applied to practice in the real world.

AGILE DEVELOPMENT IN THE CLASSROOM

Expanding commonly-used agile skills in into a full-fledged course was highly challenging. Webster University developed a graduate course, Agile Software Development, in which the main components of agile methodologies are discussed, and implemented via student projects.

Webster's graduate level course, Agile Software Development, follows the Object-Oriented Programming course in the curriculum. Students, therefore, are assumed to have a solid understanding of object oriented programming techniques. Within the agile course, students are given the option of using either C++ or Java as their implementation language. An object-oriented language is useful to permit the appropriate agile techniques to be incorporated effectively. Further, students will gain experience with a standard unit testing framework. Several open source frameworks are freely available, such as CppUnit and JUnit. Students select a framework to match their chosen programming language, and begin to use this framework in their projects one-third of the way through the course.

The course is organized in a typical manner, with class lectures and a major project that runs through the semester. The project uses the skills discussed in lectures, giving students practical experience in the techniques. We adopt the principles of the extreme programming methodology (Beck, 2000) for the project. In the following description we focus on the practical components of the course.

Graduate classes at Webster University run in a nine-week format and are limited to a maximum of 18 students. Courses meet one night per week. Evening meetings are necessary as most of the students have fulltime employment, often in an information-technology area, outside of class. In many ways the accelerated course schedule enhances the agile experience by requiring a semester-long project while students are working fulltime. In essence the course format and student employment embody resource and time limitations common in industry. Yet, through the use of agile methods, students deliver working applications for each of eight iterations. The result is a sizeable application that is successfully developed under considerable resource and time constraints.

The instructor plays the role of the customer, as well ensuring that student implementations are progressing in a technically correct manner. In particular, student implementations must adhere to industry-standard *coding guidelines* specified by the instructor. Technical correctness also implies that designs must always be as *simple* as possible per current requirements.

One of the goals of the agile course is to learn how to work within a *team* environment. A relatively small class size enables teams of only two or three students to be formed during the first week; note that the instructor is considered to be an additional member of every team, playing the role of the customer. The instructor describes some typical types of applications that would be suitable for the agile course. Examples of applications might include: Theatre Booking System, Auto Sales System, and Library Reservation System.

Student teams are then responsible for devising a project outline. The chosen project may be based on one of the examples, but alternatively can be something entirely different. It is vitally important for the instructor to ensure that the chosen application has suitable characteristics for the course; in general it should have a clear set of features, should not be overly computationally intensive, and be of an appropriate scale to be completed within nine weeks. Within the students' outline they must identify approximately 12 features to implement. Each team must maintain a document containing a *metaphor* of the project along with a description of the features of the intended application. As the project progresses, changes to these features must be accurately recorded in the document.

Each week will be deemed equivalent to an iteration, as described previously. In every subsequent week, the customer ('instructor') will meet with each team and identify a set of features to implement, or modifications to previously-implemented features, before the subsequent week's class. The customer is encouraged to change requirements in a radical way in some iterations. Such changes may result in students having to discard certain aspects of their code, a new experience compared to static problems assigned in most courses. Discarding code is a reality in agile applications and it is critical to ensure that the students appreciate the notion that a project cannot be fully and accurately specified at its outset. The team then makes a commitment to complete the stated features or modifications within the next iteration. Further, all new and modified code must be fully *integrated* into the existing application within the timeframe. At the end of the week, the teams must give a live demonstration of the features that were promised.

It should be noted that the team has to learn how to estimate the amount of work that they can successfully complete within the week – the promised features absolutely must be implemented completely; if not it is considered to be a failed task. Teams should be encouraged to make estimates based on what they can reasonably achieve within a week, and be able to *sustain* this pace throughout the course. Evaluation is based on delivering a working application that meets requirements in the allowed timeframe. Students often aspire to build more features than time may permit. In some cases students assume that Herculean size commitments will garner a higher score. More often, such attempts to impress, lead to failure in an early iteration. Although the work to which a team commits within an iteration may be small, there should be some clear value to the customer. The notion of the *planning game* in an agile project is therefore experienced. In this manner students experience the delicate balance required for estimation and scope control. Students are expected to update their project document with a clear description of when a particular new or modified feature was promised, and when it was successfully completed.

Unit Testing is introduced in the third week of the course (Meszaros, 2007). This type of testing is designed to verify the correctness of each piece of code, at a granular level, within a software application. Such testing is performed immediately upon the completion of each component – generally a function or small code segment. It is intended to ensure that the component itself works as required, but also that all existing elements of the application are unaffected by the integration of this new component. Students should be given a testing framework, CppUnit or JUnit depending on their choice of implementation language, and asked to utilize it in all future code development. An a*cceptance test* is generally based on teams successfully demonstrating the intended feature functions

correctly. However, the customer can design additional acceptance tests that must be implemented by the team.

Refactoring is discussed, via lecture material, in week four (Fowler, 1999). Refactoring code involves revising past work without altering the function of the application. Typical revisions focus on non-functional characteristics such as visual formatting and improved documentation. The resulting code is easier to read and maintain, saving time throughout the life of the application. Other revisions may include reducing complexity or adapting modules to a specific information architecture. Subsequently, students are required to refactor existing code on a weekly basis. This activity must again be recorded in the project documentation.

During some sessions students are asked to participate in *pair programming* exercises. Each member of a team should take a turn to write code while other team members review what they type, and provide some appropriate suggestions. It is particularly useful if, in this exercise, one team member is responsible for modifying another team member's code. This added constraint introduces the notion of *collective code ownership*, and perhaps highlights deficiencies in other team member's design. It also introduces students to the reality of maintenance programming in contrast to new development in typical programming assignments.

Grading for the practical component of the course is largely based on the completion of the weekly agreed-upon tasks, not on the overall magnitude of the project at the end of the semester. As described previously, weekly "tasks" can include simplification of designs and refactoring, in addition to the implementation of new features. Feedback is therefore given to each team on a weekly basis and includes a tally of the number of tasks successfully completed, given from a customer viewpoint, as well as a technical evaluation from an instructor perspective. Although grades are generally assigned on a team basis, the instructor strives to identify individuals who are not contributing in an adequate way.

In our experience, the major challenges faced by the students are:

- Inability to make accurate estimates of workload
- Resisting the temptation to expand a design beyond the immediate requirements
- Effectively using a standard testing framework
- Having the courage to discard code in appropriate situations.

With careful guidance, we have observed significant improvement in all of these areas throughout the semester. In particular, the students' ability to estimate the likely duration of a task improves dramatically after the first two or three weeks of the course. They quickly develop an appreciation for the need to keep designs simple and the importance of restricting development to what is deemed necessary by the customer. Furthermore, according to the instructors for courses that follow, students appear to fully grasp the benefits of agile techniques and make every attempt to incorporate them in future projects.

Many of the techniques covered in this graduate course can certainly be introduced, to some degree, in much lower level courses. Testing, writing straightforward (well-refactored) code, and following strict guidelines are all able to be taught in early programming courses. Emphasizing these aspects of software development at an early stage will provide a good foundation for students' future study.

CONCLUSION

As agile software development become more widely accepted, it is becoming increasingly important to incorporate the core techniques into university courses in a Computer Science curriculum. At the present time, Webster University's Agile Software Development course is offered at the graduate level. Based on our experience, we feel that an undergraduate course in which some of the important agile principles are introduced, would be greatly beneficial. The nature of the subject inherently implies some difficulties will exist in conveying the ideas in traditional software development courses. However, as described in this paper, with some careful planning, these difficulties can be overcome and a very successful course can be implemented. The course at Webster exposes students to very important software development principles that can be applied to a variety of application domains. The feedback has been extremely positive, and students continually inform us of real-world situations in which they have successfully applied the agile techniques learned.

REFERENCES

Adzic, G. (2009). "Bridging the Communication Gap: Specification by Example and Agile Acceptance Testing", *London, Neuri Limited*.

Beck, K. (2000). "eXtreme Programming eXplained: Embrace Change", *Boston, Mass., Addison-Wesley.*

Cockburn, A. (2001). "Agile Software Development", *Reading, Mass., Addison-Wesley*.

Fowler, M. (1999) "Refactoring: Improving Existing Code", *Reading, Mass., Addison-Wesley.*

Kelly, A. (2008). "Changing Software development: Learning to be Agile", *New Jersey, Wiley*.

Meszaros, G. (2007). "xUnit Test Patterns: Refactoring Test Code", *Boston, Mass., Addison-Wesley*.