Adapting Extreme Programming For A Core Software Engineering Course

Anuja Shukla
Department of Computer Science
North Carolina State University
Raleigh, NC 27695-7534
ashukla@unity.ncsu.edu

Dr. Laurie Williams
Department of Computer Science
North Carolina State University
Raleigh, NC 27695-7534
williams@csc.ncsu.edu

Abstract

Over a decade ago, the manufacturing industry determined it needed to be more agile to thrive and prosper in a changing, nonlinear, uncertain and unpredictable business environment. Recently, the software engineering community has come to the same realization. A group of software methodologists has created a set of software development processes, termed agile methodologies that have been specifically designed to respond to the demands of the turbulent software industry. Each of the processes in the set of agile processes is comprised of a set of practices. As educators, we must assess the emerging agile practices, integrate them into our courses (carefully), and share our experiences and results from doing so. This paper discusses the use of Extreme Programming, a popular agile methodology, in a senior Software Engineering course at North Carolina State University. It then provides recommendations for integrating agile principles into a core Software Engineering course.

1. Introduction

In the earliest days of programming, people simply used computers to "find the answer" – to perform computation that took too long to do "by hand." For this type of development, code-and-fix or ad-hoc program development was perhaps acceptable. The vast majority of software engineering process research emphasized *repeatability, predictability, and quality control.* During this time, much of our software development was for large systems with relatively stable requirements; product development cycles were often measured in years.

More recently, our software development is less likely to be for large systems with stable requirements, and our product development cycles are now measured in small numbers of months. The changing environment of the software industry necessitates the need to quickly respond to change and to produce software assets quickly, while continuing to be repeatable, predictable and highly concerned with quality.

This paper will discuss a Software Engineering classroom experience with a popular agile methodology, XP. This discussion will include the description of the XP methodology, the classroom project, student experiences and reactions, and the authors' recommendations for implementation in a Software Engineering course.

2. Agility

2.1. Agility in manufacturing

The manufacturing industry has focused on being agile for a decade. In the summer of 1991 at the Iacocca Institute of Lehigh University, the “Agile Competition” was formed.

A group of industrial leaders with facilitators from the Institute convened in an intense series of meetings over three months They perceived the emergence of an industrial system which was changing the structure of companies and the roles of people. They identified the main characteristics of this system and called it "Agility". Since that time, the concept of Agility has gained tremendous recognition from industrial and political leaders not only in the United States, but in Europe, South America, and Asia [13].

These leaders were facing the challenge to develop and exploit capabilities to thrive and prosper in a changing, nonlinear, uncertain and unpredictable business environment. The notion of agility in manufacturing has prospered significantly.

3. Agility in software engineering education

As Software Engineering educators, we need to consider prudently how to incorporate these essential agile lessons into our curriculum. In this section, we will describe our classroom experiences with integrating XP into an undergraduate, senior-level Software Engineering course at North Carolina State University. We then make recommendations to other educators based on our experiences.

The course was a typical 16-week semester class. The students completed four Java programming projects during the course of the semester. Three of the projects were completed as the students were learning and using more traditional software development practices. These practices were based on the Collaborative Software ProcessSM (CSPSM) [14], developed by Williams. Using the CSP, students prepared Use Cases/Use Case Diagrams [14-16], held Collaboration-Responsibility-Class (CRC) card [17] sessions to brainstorm initial design; prepared class diagrams [18], performed inspections, and wrote black and white box test cases prior to coding. They documented these items along with their configuration management plan in a Software Process and Management Plan (SPMP). They estimated and recorded time and defect information, as done in the Personal Software ProcessSM [1], using a web-based application developed at North Carolina State University. They organized themselves into teams consisting of four members each, based on the roles outlined in the Team Software ProcessSM (TSPSM) [20].

Subsequently, the students had one four weeklong team project in which they used the twelve XP practices. The experiences of this last project will now be discussed.

3.1 XP case study

The XP project was a two-player card game called Pokéwomon, which was based on the Pokémon© trading card game by the Nintendo© Corporation.

3.2 XP practices

For each of the XP practice, we describe (1) the intended steps, (2) any special classroom instruction techniques for these practices, and (3) a verbatim comment from a student on using the practice.

Table 1, below, summarizes student acceptance of each of these practices. Among the more popular were Collective Code Ownership, Simple Design, Testing and Coding Standard. Among the least used were Metaphor and Refactoring.

Table 1. Acceptance of xp practices

XP Practices	Tried & Liked it and succeeded	Tried & Didn't like it	Tried & did not succeed	Did not try
Metaphor	36%	3 %	3%	58%
Collective code ownership	94%	-	6%	-
Simple design	97%	3%	-	-
Refactoring	36%	-	8%	56%
Release planning	73%	2%	17%	8%
Small releases	50%	6%	17%	27%
Continuous integration	82%	2%	8%	8%
On site customer	56%	8%	28%	8%
Testing	86%	-	12%	2%
Pair programming	78%	2%	6%	14%
Coding standard	94%	-	3%	3%
40 hour week	64%	6%	8%	22%

Metaphor. XP espouses that each application have conceptual integrity based on a simple metaphor, which explains the essence of how the system works. The idea of a system metaphor is very abstract and often mysterious to experienced practitioners and students alike. As can be seen in Table 1, it was the least used practice.

"We didn't really have a metaphor for our project. I'm not really sure that I see the point of the metaphor portion of XP."

Collective code ownership. On an XP development team, no single programmer 'owns' any part of the code. Once entered in the code base, every member of the team owns the code and can change the code without asking for 'permission' from anyone.

The students used Concurrent Versions System (CVS) as their source code control system. As Table 1 indicates, the students were successful, though admittedly, some found the transition to joint ownership difficult.

"All code was stored in an archive that was accessible to all team members. Team members were allowed to add or change code as they wished. Since most of the changes occurred during Pair Programming, it was the responsibility of all programmers to maintain 100%

functionality of the unit test cases. This keeps any one member from completely changing the project design and negating the work done by other team members."

Simple design . XP stresses that programmers should not try to predict future needs and to design accordingly. Another aspect of Simple Design is "The Source Code is the Design." No design documents were required, as opposed to the SPMP document, which was required for the project in the first phase of the class.

Students reported success in using Simple Design since, students often naturally do the simplest thing that could get them a satisfactory grade and prefer not to do documentation. However, as educators, this is likely an area of concern. Educators understand the long-term value of documentation; students cannot appreciate the value.

"I like having the code be the design. This works very well for object oriented programs since you can see your objects, what they know, and what they do without having to have written documents about all of it. We didn't do any CRC cards; we just jumped right in and started talking about how some class would look, then just started coding. I found myself sometimes saying we were going to need such and such class later, but then tried to follow the YouArentGonnaNeedit philosophy. It was hard to break out of old habits."

Refactoring. Refactoring is the process of improving the code's structure while preserving (not improving) its function [25]. XP advocates refactor code continuously and explicitly.

Students did not find this practice very valuable. Given that it was a 4-week project, they did not get to the point that refactoring was necessary. Some used the practice and appreciated it, though.

"Re-factoring was very useful (and necessary) in our project. We were able to integrate functionality so that both players in the game could easily use it. It also made the code easier to follow and removed redundant code. Thanks to the testing, this process did not cause problems with the existing code."

Release planning. First, customer requirements are written in natural language, informal "User Story" cards, similar to use cases [15, 16]. Software developers record time estimates on each card and customers assign priorities to each card. In the "Planning Game, "the customer chooses those User Stories that comprise the most important content for a short, incremental deliverable of about one month. Each short implementation increment is accepted and tried by the customer. Then, the remaining User Stories are re-examined for possible requirement and/or priority changes, and the Planning Game is re-played for the next imple mentation increment.

We simulated the Planning Game in the class, a challenging task to achieve with 150 students. Based on an average/consensus of the resources required for each user story and the priority of the user story, the students were assigned to complete a set of user stories, assuming that each student would work on the project 9 hours/week.

"This technique was beneficial to the team. When we started the project, we did not know a lot about it and this technique helped us established a rough draft as to how much work we thought had to get done and how much approximate effort it would take."

Small Releases. XP heightens the pace of spiral development by having short releases of 3-4 weeks. At the end of each release, the customer reviews the interim product, identifies defects, and adjusts future requirements.

In our class, the entire project lasted 4 weeks resulting in one small release. The students were encouraged to have shorter, one-week internal iterations. However, the teaching staff (as the customers) did not check their work for these interim iterations.

"When we wrote the code we wrote a section and then made sure that it did what it was expected to. This is useful so that if you run into a problem you have a good idea of when it was introduced, or at least what the code is that is making it obvious. It is also nice to have the newest thing to show to the customer."

Continuous integration. Coding assignments are broken up into small tasks, preferably of no more than one day. When each task is completed, it is integrated into the collective code base. As a result, there are many product builds each day.

The students were encouraged to continuously integrate their code when sections were completed. Most found this an easy, desirable change.

"Continuous integration makes the project grow together early, instead of the project being pieces and hoping that in the end they work together. I see no disadvantages with continuous integration."

On-site customer. The customers are always readily available and accessible to the developers for the purpose of clarifying and validating requirements; preferably, customers are on-site.

The teaching staff played the role of the customer. Unfortunately, this is a tough "on-site" role to play because we obviously cannot be in all labs at all times. We simulated being on-site by being responsive to emails and by instituting a project message board.

"Our on-site customer for this program would be the message board and e-mailing, and talking to Dr. Williams. The advantage of having an on-site customer is that if there is something the user does not need, or changes their mind about. The disadvantage is that this may encourage the customer to keep making changes, which in turn, will delay the developers also you can get off track when trying to get a question answered. Also, having an on-site customer may increase the pressure to have the product finished by the due date leaving room for errors."

Testing

Unit testing. Extensive, automated white box test cases are written before production code is produced and then added to the code base. Before a programmer can integrate their code into the code base, they must pass 100 % of their own test cases and 100% of every test that was ever written on the code base. Unfortunately, students reported that they often did not write the unit test cases before they wrote the code, which is contrary to XP. Many admitted to writing all the JUnit test cases after completing the whole program just to fulfill the testing requirement of the class.

"Test early and test often" was the philosophy that we used. The advantage of this was that we could see exactly how much work we had left because we already had the test cases written up. Another useful advantage was that we did not have to keep writing the same test case over and over because we had a common test base."

Acceptance test. XP promotes the use of customer-written acceptance test cases for tracking project completeness. When an acceptance test case is successfully passed, it can be considered that a specified functionality has been implemented properly. Project completeness is based on the percentage of acceptance test cases that have been passed.

Each student group
was assigned a user story. They had to propose to their customer the set of black box test cases they believed should comprise the acceptance tests for that user story. Ultimately, the students were provided with the full set of black box test cases that would be used by grading. This served to simulate the "certainty" of acceptance criteria that XP customers provide the development team.

Pair programming. At all times, two programmers work side-by-side at one computer, collaborating on the same design, algorithm, code or test.

Most students pair programmed. Students who did not pair program generally did not do so because they did not want to coordinate schedules with their peers or preferred to work in their dorm rather than in the university laboratory.

"I love pair programming. It works so well in our group. I don't feel alone and I have someone else to get the ideas out that are stuck in my head. Errors are picked up quicker and logic problems get solved quicker. I wish all my computer classes were like this because coding alone is fun but much better when another person is with you."

Coding standard. In order for developers to easily understand each other's code, an agreed upon coding standard is followed.

Students were instructed to adhere to the Sun Java Coding Standard i.

"Coding standards were important in developing the program. We followed naming conventions for classes, methods, and variables. This made it easier for one another to trace through code. This was necessary because of the fact that there was no documentation. The conventions served as an outline."

40-Hour week. XP advocates that programmers do not tire themselves out by overworking themselves. The team decides to work at a pace, which is comfortable with all the team members. Developers have found that during crunch periods when they worked overtime, the artifacts produced were poor.

"This was an advantage because this would ensure that when we wrote code, we would be fresh and energized and could do the best work we could."

Students wrote comparative analysis papers of XP vs. CSP. Overall, the students felt that XP was appropriate for small projects and small teams, such as the project required. Most preferred it to the more traditional practices of the CSP. By teaching both, students were provided with a rounded experience.

3. Suggestions for implementation

We found that one semester is not long enough to teach two very different methodologies nor for the students to perform meaningful assignments using two very different methodologies. Additionally, instructing students with agile methodologies raises a pedagogical dilemma, particularly considering the fact that currently many Computer Science curricula have one Software Engineering course. The emerging methodologies focus on being lightweight, agile, and minimalist and often do not specify/mandate the use of best practices, such as inspections and UML modeling. If students are only taught an agile method, they will not gain experiences with many best practices. As educators, we must continue to ensure Software Engineering students learn an array of skills and techniques, agile or otherwise so that they can prudently utilize the best practices for their specific project.

As a result, Williams has devised a hybrid methodology that consisted of a set of practices from both agile and traditional development methodologies. She used this hybrid methodology in subsequent course offerings. In addition to completing a course project with this hybrid methodology, students also learned and performed an exercise in inspections and discussed pair programming as an alternative to formal inspections. Additionally, they completed several Use Cases and discussed these as an alternative to User Stories. Students also learned about other UML diagrams, such as sequence and state diagrams.

Using this hybrid methodology, the course project begins by the teaching staff giving the students a problem statement for the project. Students translate this problem statement to a set of User Stories. Student teams then play the "Planning Game" to determine the order of completion of the User Stories and, ultimately, which student(s) will own each User Story.

Student teams begin a project by completing a SPMP. This documents (1) the team organizational structure (team leader, etc.); (2) the text of the User Stories, who owns them, and the projected completion date; (3) a configuration management plan; (4) an initial high-level design, documented in the form of UML class diagrams using tools such as TogetherSoft Control Centerii or Rational Roseiii, and (5) a comprehensive set of acceptance/black box test cases.

Students hold CRC Card brainstorming sessions to begin their high level design. During development, students work in pairs. They write unit test cases (preferably using an automated testing tool such as JUnit) before writing code. They must ensure all unit test cases run at all times during development. In the course of development, students practice continuous integration, collective code ownership, refactoring and follow a coding standard. The teaching staff plays the role of customer and tries to be responsive to student questions.

4. Summary

Agile software development processes are emerging and gaining popularity in industry. In 2001, a Software Engineering course was taught at North Carolina State University. In this course, students learned both traditional software development methodology and XP, a popular agile methodology. In general, students reacted well to the practices of XP, and learned a great deal about emerging agile methodologies. However, the authors feel that it is not advisable to teach and practice entire traditional and agile methodologies in one semester course. Additionally, there are time-tested skills and techniques, such as inspections, UML, and use cases that need be taught at some point in the curriculum. If students had only one Software Engineering course and were solely taught to use a methodology such as XP, students would not learn about such techniques. We suggest a hybrid process that includes both agile and traditional practices.

References:

[1] W. S. Humphrey, *A Discipline for Software Engineering*. Reading, Massachusetts: Addison Wesley Longman, Inc, 1995.

[2] I. Jacobson, G. Booch, and J. Rumbaugh, *The Unified Software Development Process*. Reading, Massachusetts: Addison-Wesley, 1999.

[3] A. MacCormack, "How Internet Companies Build Software," *MIT Sloan Management Review*, pp. 75-84, Winter 2001.

[4] A. MacCormack, R. Verganti, and M. Iansiti, "Developing Products on "Internet Time": The Anatomy of a Flexible Development Process," *Management Science*, vol. 47, pp. 133-150, January 2001.

[5] B. Boehm, A. Egyed, J. Kwan, D. Port, A. Shah, and R. Madachy, "Using the WinWin Spiral Model: A Case Study," *IEEE Computer*, vol. 31, pp. 33-44, July 1998.

[6] K. Beck, *Extreme Programming Explained: Embrace Change*. Reading, Massachusetts: Addison-Wesley, 2000.

[7] A. Cockburn, *Agile Software Development*. Reading, Massachusetts: Addison Wesley Longman, 2001.

[8] J. Highsmith, *Adaptive Software Development*: Dorset House, 1999.

[9] J. Stapleton, *DSDM: The Method in Practice*: Addison Wesley Longman, 1997.

[10] A. Cockburn, *Crystal "Clear": A human-powered software development methodology for small teams*: Addison Wesley, 2001 (in preperation).

[11] P. Coad, J. deLuca, and E. Lefebvre, *Java Modeling in Color with UML*: Prentice Hall, 1999.

[12] L. Rising and N. S. Janoff, "The Scrum Software Development Process for Small Teams," *IEEE Software*, vol. 17, 2000.

[13] P. Agile, "Agile Competition is Spreading to the World," *http://www.ie.lehigh.edu/*, 1996.

[14] L. A. Williams, “The Collaborative Software Process PhD Dissertation,” in *Department of Computer Science*. Salt Lake City, UT: University of Utah, 2000.
[15] A. Cockburn, *Writing Effective Use Cases*. Reading, Massachusetts: Addison-Wesley, 2000.
[16] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard, *Object-Oriented Software Engineering: A Use Case Driven Approach*. Wokingham, England: Addison-Wesley, 1992.
[17] D. Bellin and S. S. Simone, *The CRC Card Book*. Reading, Massachusetts: Addison-Wesley, 1997.
[18] M. Fowler, *UML Distilled*. Reading, Massachusetts: Addison Wesley, 2000.
[19] ObjectMentor, “Junit,” , 2001, http://junit.org.
[20] W. S. Humphrey, *Introduction to the Team Software Process*. Reading, Massachusetts: Addison Wesley, 2000.
[21] L. Williams and R. Upchurch, “In Support of Student Pair Programming,” presented at ACM SIGCSE Conference for Computer Science Educators, Charlotte, NC, 2001.
[22] L. A. Williams and R. R. Kessler, “The Effects of "Pair-Pressure" and "Pair-Learning" on Software Engineering Education,” presented at Conference on Software Engineering Education and Training, Austin, TX, 2000.
[23] L. A. Williams and R. R. Kessler, “All I Ever Needed to Know About Pair Programming I Learned in Kindergarten,” in *Communications of the ACM*, vol. 43, 2000.
[24] R. Jeffries, A. Anderson, and C. Hendrickson, *Extreme Programming Installed*. Upper Saddle River, NJ: Addison Wesley, 2001.
[25] M. Fowler, K. Beck, J. Brant, W. Opdyke, and d. Roberts, *Refactoring: Improving the Design of Existing Code*. Reading, Massachusetts: Addison Wesley, 1999.

---

[i] http://java.sun.com/docs/codeconv/
[ii] http://www.togethersoft.com
[iii] http://rational.com