Less Extreme Programming

James Noble   Stuart Marshall   Stephen Marshall   Robert Biddle

Informatics Group
Victoria University of Wellington,
PO Box 600, Wellington, New Zealand
Email: kjx@mcs.vuw.ac.nz

Abstract

Industrial practice in software engineering has developed in recent years from rigid heavyweight document-based development techniques, such as the Rational Unified Process, to incorporate more agile, iterative, communication-centric approaches such as Extreme Programming. This shift has created a need for a similar shift in software engineering education. We report our experience of incorporating an Extreme Programming option into an existing document-centric software project course. While students taking the option were generally positive about Extreme Programming, the projects' external clients had a more mixed experience.

Keywords: Extreme Programming, Software Engineering, Project Work.

1 Introduction

Many Computer Science, Software Engineering, and Informatics programmes culminate in a large-scale project work course. These capstone courses are designed to cover much of the material students have learned in the three or four years of their degrees, and, by bringing together theoretical insights with best industrial practice, provide students with a taste of the world they will face when they graduate.

Such courses must by necessity involve more than just the skills explicitly required to complete such a project — such as programming, testing, design and analysis. The management and coordination activities necessary to complete a large-scale project (along with pedagogical imperatives to evaluate) mean that such projects must implicitly involve some software methodology, especially if carried out in groups. In the past ten years or so, such process methodologies have remained quite familiar in outline, even as fashions change from structured to object-oriented design, and standards evolve from MIL-STD-2168 (US Department of Defense 1988) to OPEN (Henderson-Sellers, Simons & Younessi 1998) or the Rational Unified Process (Krutchen 1999). Such methodologies involve standard project management skills (aligned with the PMBOK (Project Management Institute 2000)), and adopt a traditional software life-cycle model, with separate analysis, design, and implementation phases. These phases are linked by significant documentation (analysis models, design models, etc) that can be assessed effectively and relatively

efficiently.

More recently, industrial practice in significant areas of software engineering is exploring so-called *agile* or iterative software methodologies, rather than rigid, heavyweight, document-based methodologies. Extreme Programming (XP) (Beck 1999) is the most well-known agile methodology, although there are several alternative brands, including The Crystal Methods (Cockburn 2001), Lean Development (Poppendieck & Poppendieck 2003), Scrum (Beedle, Schwaber & Martin 2001), and Evolutionary Development (Highsmith 2002).

This shift has created a need for a similar shift in software engineering education, and in particular, in large-scale project work courses, because document centric project methodologies do not align well with students' reasonable expectations of more agile working methods. Unfortunately, from a pedagogic perspective, agile projects are significantly more difficult to manage than documentation centric methodologies, as they do not mandate traditional phase-transition details, nor produce readily-assessable intermediate products.

In this paper, we report our experience of incorporating an Extreme Programming option into an existing (document-centric) capstone software project course. We begin in section 2 by describing the agile methodology we adopted (Extreme Programming) and the structure of the existing document-centric course, COMP389. Section 3 then presents our design for an Extreme Programming Option, which we offered alongside the traditional project stream. Section 4 then describes our students, staff, and project clients' experience with this option, and finally section 5 reflects on this experience and presents our conclusions.

2 Background

Extreme programming is oldest and most popular agile software development methodology. As presented in the Agile Manifesto, the agile development values individuals and interactions over processes and tools; working software over comprehensive documentation; customer collaboration over contract negotiation; and responding to change over following a plan (Beck, Beedle, van Bennekum, Cockburn, Cunningham, Fowler, Grenning, Highsmith, Hunt, Jeffries, Kern, Marick, Martin, Mellor, Schwaber, Sutherland & Thomas 2001).

Iteration and flexibility are the keys to the agile approach, and what distinguishes it most clearly from a classic life-cycle development. In an agile, iterative approach, the same activities are repeated constantly, within each iterative cycle; while in the classical life-cycle, a single activity is carried out until one phase in complete, then the development proceeds to the next phase (see Figure 1). From this figure, we can also see

that agile methods focus on particular activities, such as programming and testing, rather than more esoteric activities such as analysis and design (Extreme Programming in particular follows this model). One key consequences of this is that an agile programming team will consist of generalist programmers, rather than specialist analyst and designers.

Traditional	PPPAAAAAAADDDDDDCCCCCCTTT
Iterative	TCCPTCCPTCCPTCCPTCCP

Figure 1: Iterative development versus traditional development. (Phases shown: Planning, Analysis, Design, Coding, Testing)

Because of their flexibility rather than a initial fixed plan, agile approaches have been characterised as "*postmodern*" when compared with traditional, "modern" software development methodologies based on forward planning and documentation (Robinson, Hall, Hovenden & Rachel 1998, Cockburn 2001, Noble & Biddle 2002). Furthermore, the traditional methodologies are now incorporating some elements of agility, although still maintaining the classic lifecycle and development phases.

2.1 Extreme Programming Practices

Although fitting generally within the wider agile approach, Extreme Programming as an identifiable methodology is distinguished by twelve main *practices*, shown in Figure 2, along with a number of secondary practices. These practices are similar to the activities or techniques of conventional methodologies, in that they are particular things that programmers actually *do* to produce software (Henderson-Sellers et al. 1998).

These practices synergistically support the principles of Extreme Programming, including:

- **Do the Simplest Thing That Could Possibly Work**. Rather than designing complex systems because they are elegant or technically appealing, Extreme Programming requires programmers to write simple, quick programs that meet the most basic user requirements.

- **You Ain't Gonna Need It**. Extreme Programming discourages programming systems that can cope with potential future requirements. Extreme Programming requires developers to build only what is immediately required to meet a customer's requirements.

- **Coaching.** As Extreme Programming is still an emerging methodology, many developers may not be well acquainted with all the practices and principles. To address this, XP teams often include a *coach* as well as developers. The coach is not themselves a developer, designer, or manager, and does not write any code or make any design decisions. Rather, the role of the coach is to mentor developers to ensure that they understand the Extreme Programming practices, and more importantly, actually put them into practice.

Space does not permit a further description of the tenets and practices of Extreme Programming here: a wide number of overviews are generally available (Beck 1999, Beck & Fowler 2000, Jeffries, Anderson & Hendrickson 2001, Williams & Kessler 2002).

1. On-Site Customer
2. Planning Game
3. Pair Programming
4. Forty Hour Week
5. Small Releases
6. Test First
7. Continuous Integration
8. Collective Code Ownership
9. Coding Conventions
10. Simple Design
11. Refactor Mercilessly
12. System Metaphor

Figure 2: Extreme Programming Practices

2.2 Extreme Programming in Education

Although Extreme Programming is still quite new, there have been several documented attempts to integrate XP (or individual practices) into formal educational courses, and also to measure their effect. This work falls into two main categories: the adoption of coding-level practices (Pair Programming and Test First in particular) in existing, often introductory, programming courses, and attempts to adopt all of Extreme Programming, often in the context of full-time block courses.

Experience and empirical studies of the programming practices have generally been very positive. Laurie Williams (Williams & Kessler 2002) has done much work in this field. In an early study (Cockburn & Williams 2001), she found that pair programming increased many measures of code quality, although at a slight increase in programming time. A later study found that pair programming increased both grades and retention in a first programming course (Williams, Wiebe, Yang, Ferzlid & Miller 2002). Astrachan et al. have used pair programming between a lecturer and an entire class to teach programming, and also the Small Releases and Refactoring practices to teach software design (Astrachan, Duvall & Wallingford 2003). Pechau et al. describe a one-week project offered at the end of first year, based around the XP programming practices (Becker-Pechau, Breitling, Lippert & Schmolitzky 2003).

On the other hand, experience with integrating the "whole cloth" of Extreme Programming into courses has been much more mixed. Peter Lappo, for example, describes a twelve-week full time course taken by Masters' students in which only the Planning Game and 40-hour week practices were successfully established — the students failed even at Pair Programming (Lappo 2002). Ivan Tomek describes some more positive results, although for very small courses of only four or eight students (Tomek 2003). John Noll and Darren C. Atkinson conducted an experiment in a ten-week software engineering course, where traditional development teams managed to deliver significantly more functional (although more buggy) software than the Extreme Programming teams: again, the teams were unable to adopt many practices, with weak customer engagement, a lack of collective code ownership, and batch integration (Noll & Atkinson 2003), while Macais et al. describe a second experiment where both development styles produce

Week 1	Introduction	
Week 2	Project Plan Due (D1)	
Week 3		
Week 4	Requirements Specification Due (D2)	
Week 5		
Week 6	Architectural Design Due (D3)	
Two-weeks	Break.	
mid-semester	Break.	
Week 7		
Week 8	User Interface Design Due (D4)	
Week 9		
Week 10		
Week 11		
Week 12	Detailed Design Due (D5)	
	Deliverable Software Due	
Semester	Personal Essay Due	
End		

Figure 3: Timetable for Classic Project

Project Plan (D1)	10%
Requirements and specification (D2)	30%
Architecture design (D3)	20%
User Interface Design (D4)	10%
Detailed design (D5)	10%
Deliverable Software	10%
Team Process	10%

Figure 4: Classic Assessment Items

similar results (Macias, Holcombe & Gheorghe 2003). Mugridge et al. describe a series of more successful projects, although again several practices such as On-site Customer and 40-hour weeks could not be instituted (Mugridge, MacDonald, Roop & Tempero 2003); Johnson and Caristi report similar experience (Johnson & Caristi 2003). Finally, Dean Sanders surveyed students' attitudes to both Pair Programming and Extreme Programming more generally, finding that students thought Pair Programming particularly useful in introductory courses, but were much more circumspect about XP (Sanders 2003).

2.3 COMP389

Our Software Engineering Project course, COMP389, is in essence a traditional ("classic") capstone software engineering project course that has been offered in some form at Victoria University of Wellington over the last twenty years, and is taken by between thirty and forty students each year. Originally offered as assessed group work within a final year software engineering course (COMP301), workload concerns forced the project into a separate optional course about ten years ago (Brown & Dobbie 1998, Brown & Dobbie 1999, Brown 2000). While the details of methodology and technology have changed markedly over that time (from structured design in COBOL, through C, Pascal, C++ and Tcl, to web-based object-oriented design in Java, Perl, or VisualBasic), necessitating great changes in teaching equipment and technical support, the underlying structure of the course has remained much the same over all this time: every 2-3 weeks (of a 12 week course) students would have to complete a document to meet a design milestone, beginning with a project plan and finishing with deliverable software (see figure 3).

A key feature of the course is that students work together in groups of between four and seven students. Students are assigned into groups by course staff in the first week of the semester, with the assignment attempting to take account of students preferences for working style, project topic, and co-workers. Each group produces software and documentation for an actual, external client. These clients were generally staff members employed within the university but outside our school: either academic or general staff, often without any particular knowledge of software development. Clients were promised an exploration of a particular problem that may be amenable to a software solution, and were invited to comment

on all documents produced by the groups, including any software produced. Most project groups would produce a working prototype (with varying degrees of polish): several projects would eventually "go live" within the university, typically after clients arranged funding to allow team members to extend their projects after the conclusion of the course.

Each project group is assigned to a tutor, generally a graduate student who had done well in COMP389 in a previous year. The tutors, employed for around four hours per week, meet with their groups for an hour a week, to give guidance on development processes, architecture and designs and implementation technologies. The remainder of the tutors' time is spend proofreading, mentoring, and assessing the teams on the creation of the documents they must produce — each document is handed in twice, a draft one week prior to the due date, and then a revised version which contributes to the final grade.

To assess the course, a group project mark is calculated by weighing the marks for the final versions of each document and the final product (see figure 4). This group mark is then moderated by qualitative individual weightings for each group member, to produce a project mark for each student. This project mark contributes 60% of each student's overall mark for the course, with the remaining 40% being direct individual assessment: reflective diaries (10%), providing ratings on other team members (10%), and an essay in lieu of a formal examination (20%).

The overall flavour of the course is betrayed by figure 4: producing deliverable software is worth only 10% of the project grade (the same as keeping to the software process) while 80% of the project grade is for document production.

Although the separation from COMP301 reduced the amount of student's effort required (the resulting COMP389 has no taught material of its own), this course still demands a high workload, especially when document deadlines draw near, and some students traditionally sleep in the labs during the last week — and not at all the night before the project demonstrations — although we do our best to discourage this practice!

3 COMP389 Extreme Programming Option

In this section, we describe the changes we undertook to COMP389 to introduce an option for students to complete a project using a variant of Extreme Programming.

3.1 Project Iterations

The major problem we faced in introducing Extreme Programming into COMP389 was how to fit an open-ended, iterative, agile project into the confines of a twelve-week semester. In retrospect, the key decision was the decision to base the project around a two-week iteration cycle.

In commercial Extreme Programming projects, iteration cycles are typically between two and four

Week 1	Introduction
Week 2	Project Plan Due
Week 3	Iteration 1
Week 4	
Week 5	Iteration 2
Week 6	
Two-weeks	Break.
mid-semester	Break.
Week 7	Iteration 3
Week 8	
Week 9	Iteration 4
Week 10	
Week 11	Iteration 5
Week 12	Deliverable Software Due
	Architectural Design Due (D3)
Semester	Personal Essay Due
End	

Figure 5: Timetable for Extreme Programming Projects

weeks long (Beck 1999, Beck & Fowler 2000), with most projects opting for cycles of three or four weeks. Given only twelve weeks in a course, a four week cycle would only have allowed students to experience three iterations; a three week cycle, only four. The two week cycle time we imposed nominally allowed six iterations within the life of the course. For pragmatic administrative reasons, however, full iterative work could not begin during the first two weeks of the course, as this time is required for organising the course, assigning students into groups, allowing groups to contact their clients, and providing a window for students who did not wish to commit themselves to COMP389 to drop out. Beginning in the third week, groups typically spend the first iteration getting various aspects of the supporting infrastructure sorted out and making decisions about language and supporting software, before making active progress from the second iteration.

Figure 5 illustrates the overall timetable of the Extreme Programming version of COMP389; it is interesting to compare this with the "classic" timetable in Figure 3 above, as they take six two-week blocks but use them in very different ways. Both begin with the first two week block for group formation (week 1) and then each group has to write and submit a project plan for assessment (week 2). But, where the classic projects then proceed through two-week blocks of analysis, design, detailed design, user interface design, ultimately to implementation, the majority of the Extreme Programming timetable is taken up with five two-week iterations. The final iteration (weeks 11 and 12) is also somewhat special as it is the end of the course, so this iteration must include time for a demonstration of the software. To ensure even the XP students gained some experience with producing documentation we required XP students to produce software design documentation in this iteration.

3.2 Practices

Within the two-week iterations, we then had to determine how Extreme Programming's practices would fit into this structure in the context of an academic course. Some practices we found we could adopt directly, some needed modification, and some we omitted:

On-site Customer At the highest level, the practice of having a customer representative on site completes the iterative nature of XP: rather than writing plans, the team consults directly with the customer as part of the Planning Game practice (see below); and just as importantly, while they are actively programming, the customer is always directly available to the team. Unfortunately, mainly because of the status of the student projects, it was clearly unrealistic for the project clients (the XP customers) to spend even eight hours a week with the project teams (a similar effect has also been identified in full-scale XP projects (Martin, Noble & Biddle 2003)). External client participation has always been important in COMP389 — so, for example, we would not substitute a client with a tutor playing the role of a client, even if we had sufficient resources to provided that level of tutoring. The compromise we selected was that clients needed to be on-site, physically available for the teams at least two hours per week for the Planning Game practice, and then available via telephone or email to the teams for the remainder of the time.

Planning Game The Planning Game is a structured negotiation between the on-site customer and the development team to determine the priorities for each development iteration. By writing or updating short descriptions written on "story cards", the customer proposes or deletes stories to be included in the iteration. The team estimates the stories on the cards, and the process concludes when the sum of the estimates is less than the time available in the iteration. This practice is central to XP, subsuming the planning, analysis, and priorisation phases of traditional methodologies, and so we required each COMP389 team to complete this practice at the start of each iteration.

Pair Programming The practice of Pair Programming is one of the ways in which XP differs most from all other forms of programming – although pair programming is not itself particularly novel (Dijkstra 2001), insistence that all code must be written in pairs certainly is. We consequently mandated this practice: after playing the planning game, pairs of team members are allocated story cards for them to implement in the current iteration, and students must program all these stories simultaneously with their partner. This had two main consequences — first, that students taking the course needed to be able to arrange their timetables to be able to program in pairs, and second, that suitable laboratory space was available.

10-hour week In terms of actual programming, one of the main Extreme Programming practices is the "40-hour week" which states that programmers should not work more than 40 hours per week, and should not do overtime. Now, although in later versions of XP, this practice has been renamed to be "sustainable pace" (in Germany, the working week is restricted to 35 hours, for example), students in COMP389 could clearly not be expected to put in 40 hours per week in the course. According to the University's workload formula, the course was scheduled to involve only ten hours per week, so we adopted this time limit as the practice in COMP389 (even though the traditional version of COMP389 regularly exceeded this limit). In fact, time for programming and work in XP teams was restricted further, as students were required to allocate two hours per week for attending whole-class meetings, personal reading, and writing their individual essay, leaving eight hours per week to work on the project.

220

Small Releases To keep the team honest, Extreme Programming requires that the team make frequent small releases of working software that customer can evaluate. This could be easily incorporated into COMP389 by requiring the team to make a release at the end of every two-week iteration.

Test First Extreme Programming has a very strong test culture, requiring unit tests and acceptance tests to be written *before* the code to which those tests apply. This testing discipline is also very important to the flavour of actually programming with XP, so we were keen to maintain this in the course. We required students to write automated tests for all the code, before they wrote the actual code, and maintain these tests throughout the life of the project.

Continuous Integration XP requires the whole team to maintain a "continuously current" version of the software, by integrating the changes necessary to implement a user story as soon as that story is completed. Since we were using the planning game, and the whole team was required to work simultaneously, this practice could be easily incorporated into COMP389.

Collective Code Ownership XP forbids individual teams members to "own" parts of the project: all code must be modifiable by all team members. We attempted to incorporate this practice into COMP389 in the same way as full commercial projects: team members programming pairings and the way story cards are assigned to pairs are rotated throughout the project, so that every group member works on all parts of the code with all other group members.

Coding Conventions Reinforcing Collective Code Ownership, Extreme Programming encourages all programmers to adopt the same programming style, so that the whole program appears to have been the work of one programmer (Jeffries et al. 2001). We adopted this by requiring each team to choose a set of style guidelines, such as Doug Lea's guidelines for Java programming (Lea 2000).

Simple Design This practice reinforces the YAGNI (You Ain't Gonna Need It) and Do The Simplest Thing That Could Possible Work principles, emphasising the most basic designs that meet (albeit barely) the immediate requirements. Teams were encouraged to adopt this practice also. The general structure of the planning game and the short, two-week iterations keep teams focused on completing the customer's user stories rather than developing over complex systems.

Refactor Mercilessly This XP practice mitigates against possible side effects of the Simple Design practice, and supports the "Once and Only Once" principle (that a project should contain no repeated or redundant code). In Extreme Programming, once a pair of programmers has a working implementation of a user story, they are then required to rewrite both their own code and as much of the system as necessary to ensure a good design. This practice could be directly incorporated into COMP389.

System Metaphor The least popular (and least understood) Extreme Programming practice is that of System Metaphor: at the start of the project, the team brainstorms a metaphor that will guide the design of the system. Certainly when the first year of COMP389 ran, this practice was quite underdefined — how, for example, did such a metaphor differ from a high-level or architectural design, both of which are outside the principles of XP. As this practice was itself unclear, nor how it would fit into the iterations of the project, we did not include this practice in COMP389. To maintain duodecimal correctness, however, we replaced this with the Exploration practice.

Exploration Originally known as "Spike Solution", this is considered a "supporting" practice in many descriptions of Extreme Programming. Exploration requires teams to address technical concerns (such as the feasibility of a particular design, or the performance of some software infrastructure) by writing a small piece of code to demonstrate feasibility, rather than just guessing. The aim is to write this code as fast as possible, so programmers are permitted to ignore the other practices when producing exploratory code, on the condition that exploratory (expletory?) code must never be included into the system under development: it must be rewritten from scratch first. As mentioned above, having omitted the System Metaphor practice, we required teams to follow this practice in its place.

3.3 Coaching

One of the complications of applying the XP approach in practice with the COMP389 students was their lack of familiarity of how these practices actually worked when applied to programming. Similarly, clients of the teams were usually uncertain about how to get a programming team to work on their project within an XP context. As mentioned above, Extreme Programming teams may employ a coach to ensure that all of the XP practices are followed by demonstrating or enforcing their use.

To address this role in the COMP389, we used the project tutors to play the role of the Extreme Programming coach. While this role remained important during the course of projects, the coach/tutor also needed to ensure that the teams communicated effectively with the client and got on to the project quickly given the limited time available. One additional role of the coach that was particular to the teaching context was the need also to balance the requirements of the client with the need for the students to succeed within the constraints of the assessment process being used. This conception of the coach as tutor is certainly not inconsistent with the XP methodology: the coach's responsibility is to accompany the whole XP process — to teach, to support learning, to supervise the process of learning, and to ensure the balanced introduction of the XP methodology and practices (Beck 1999, Jeffries et al. 2001).

3.4 Facilities

The existing traditional software process for COMP389 is accommodated quite easily within our teaching facilities. All Computer Science and Software Engineering courses within our School share a series of common computer labs, comprising between 20 to 30 machines per room for a total of 150 machines in 6 rooms. All these machines run a common environment based on Open Source software. Commercial software is also available where necessary, typically for software engineering courses, including Rational Rose for diagram creation and software modelling, and Microsoft-brand operating

systems, productivity, and programming tools running on quarantined servers. Many students use either their own or a familial computer to complete work off-campus, however political considerations mean that the School is required to provide sufficient computing resources for all students who need them.

These laboratory spaces have proved quite adequate for the development, programming, and documentation tasks required by traditional software engineering processes. Because these labs are public, shared spaces, group meetings cannot be held there, however, the School and University provide a range of group study and meeting and small seminar rooms that may be booked by students engaged in group work. Because the major group meetings are quite short (tutors are present and students have commitments to other courses), the facilities are quite adequate for these meetings.

Extreme Programming's practices, however, require quite a different kind of space (Beck 1999, Jeffries et al. 2001, Williams & Kessler 2002). Pair Programming in particular requires space where two programmers can work at a single computer terminal, and, just as importantly, talk to each other while doing so. More generally, entire teams of up to three pairs (six students) need to be able to discuss what they are working on while they are actually doing so. To meet the 10-hour week requirements, these activities need to be supported in blocks of up to four hours at a time.

This manner of working is insupportable within the existing work spaces in the university. Students are constrained from working and talking in groups in the common computer labs, while the group meeting rooms do not provide computer facilities, and cannot be booked for sufficient blocks of time.

Luckily, we were able to commandeer sufficient dedicated laboratory space to support these Extreme Programming practices. Adjoining one of the main labs, there was a small, glass-fronted room, used primarily as a dumping-ground for junked equipment, and occasionally as a student lunch room. We moved two machines into this lab, and then students in the course arranged them with enough space for two people to work on each machine, and furthermore acquired a table and whiteboard from elsewhere in the building (see Figures 6 and 7). This "Glasshouse" lab then became the main room used by COMP389 Extreme Programming groups, timetabled so that each group could have the uninterrupted time they required.

3.5 Assessment

The classic COMP389 project assessment criteria, included in figure 4 above, were heavily based on the production of documents. As Extreme Programming does not produce documents, we had to modify the assessment to reflect those things that were of value to Extreme Programming. Figure 8 shows the project assessment we developed for the Extreme Programming option in COMP389. Marks for the analysis and design documents have been replaced by marks for the quality of the code and test cases produced by the teams. The marks for keeping to the software process and the quality of the final product (from the users', not the programmers' perspective) have also been increased. Finally, to provide students with some experience creating documents, we required them to produce the same project plan as the traditional methodology groups, and also a version of the architectural design document, although this is due at the end of the project to describe the program the team has built once coding has finished.

Figure 6: The Glasshouse lab: Note the abandoned Macintosh computers.

Figure 7: Pair Programming in the Glasshouse

As in the classic methodology project, project assessment is moderated by a qualitative score to reflect each student's participation in the project, and accounts for 60% of the student's grade. The remainder is made up of marks for a diary, an essay, and marks for rating the other members of the student's group.

3.6 Optionality

A consequence of many of these decisions was that we felt unable to simply replace the classic software development experience offered in COMP389 with Extreme Programming. Rather, we offered the ag-

Project Plan (D1)	10%
Tested, Refactored Code	30%
Test Cases	10%
Architectural Design (D3)	10%
Deliverable Software	20%
Team Process	20%

Figure 8: Extreme Assessment Items

ile methodology as an option to those students (or rather, to those groups of students) who wished and were able to take it up.

Although some students were excited by the prospect of practising Extreme Programming, other students were comfortable with the more traditional style (or, perhaps, as with many practitioners, were concerned that the emperor had no clothes (McBreen 2003)). More pragmatically, many students, particularly part-time students (and thus especially women, mature students, under-represented groups) were unable to commit to the blocks of time required at the university to carry out the pair programming and other group work required by XP. The traditional style, relying on much shorter meetings and individual work, would clearly suit these students better.

In the two years we have been offering Extreme Programming in COMP389, approximately half the students have chosen each option.

4 Experience

Overall, the Extreme Programming projects in COMP389 have been a success, with teams tending to produce more functional programs than the traditional methodology, and with end-user clients being at least as satisfied. Because of the small number of groups involved (seven over the last two years) we have not been able to complete any quantitative analysis, however during and after each offering of the course we have interviewed students, clients and the course staff. In this section we report on the experience of these students, clients, and the staff involved with the course.

4.1 Students

Students taking this option uniformly considered that they knew more about Extreme Programming than they had at the beginning of the course, and that such traditional lecture coverage they received in COMP301 and elsewhere in the curriculum was no substitute for actual project work using an agile method. Purely pedagogically, then, the option can be seen as meeting its major goal of giving students experience in this development style.

Productivity As an exercise in software development for "real" clients, too, the projects were successful: generally producing more useful and more functional programs that those produced by traditional teams. Also, whereas traditional teams will occasionally fail to produce any working code at all (due to interpersonal issues between students) every Extreme Programming team taking the course so far has at least managed to produce something! Note that this does not necessarily mean that the XP teams write more code than the traditional teams: they may even produce less code (especially if they have been refactoring assiduously) but the code they do produce is more aligned to the client's needs.

Workload Most interestingly, the Extreme Programming option had a positive effect on student's workload. Even though students produced more software, XP seemed like less work than the traditional methodology (thus the title of this paper): as the course's workload has always been a concern, this is another clearly positive development. The reasons for this reduction are not completely obvious: the course and room timetables involving large blocks of time dedicated to the course constrained students to work constantly throughout the semester, while the requirements to work in groups make it harder for students to do large amounts of overtime.

On the other hand, the block-of-time structure was clearly harder for some students — either officially part-time students, or those nominally full-time but with part-time jobs to pay off their student loans (often in the range of 20-50,000 dollars). Furthermore, XP students were often unhappy about the amount and structure of the relatively small amount of documentation they were required to produce, and didn't do as good a job as the traditional teams: raising the question as to whether they received enough practice in producing documentation.

Practices Regarding the twelve Extreme Programming practices we adopted, most of these were carried out well by students. The large-scale practices, including on-site customer and the planning game, worked as well as they could given the timetables of the external clients who, after all, had their own fulltime jobs to perform. The key programming practices, pair programming especially, were also generally well performed. On reflection, we believe that there are three main reasons for this success. First, ensuring that large blocks of time were available for the course, and then requiring all group members to be present whenever possible, greatly facilitated all these group practices: indeed, groups often took to socialising together at the university bar after each programming session (a practice encouraged by the London Extreme Programming groups, at least). Second, the Glasshouse lab physical facilities, and the fact that it had only two or three computers for groups of up to six members, strongly supported Pair Programming and other group meeting practices. The third reason is the skill of the tutors acting as coaches in working with the students: we discuss this further below.

Testing The main exception to this was the Test First practice, which was not as well observed as many of the others. There were a number of reasons for this. One sad reason is that our whole curriculum under-emphasises testing (up to and including the "traditional" COMP389 project). For many of the students the tests they wrote on this course were the first serious testing they would ever complete, however, the tests were greatly under the standard expected in commercial XP projects (around 1.5 lines of test code per line of product code (Williams 2002)). A second reason for the difficulty with testing is that XP emphasises automated, end-to-end tests: many of the projects undertaken in COMP389 are web-based projects, and students generally did not have access to good tools for regression testing of web pages, especially when the design of those pages rapidly evolves. That said, some teams did produce very effective automatic testing suites which produced useful feedback on their systems as they evolved, although they did not always use them routinely to check every change (as required by XP).

Preparation Student preparation was also an issue for some teams: certainly, some students could benefit from having more XP experience, but on the other hand, "learning on the job" is standard industrial practice for learning XP, and it is not clear how much more classroom experience would have helped. Students did benefit from access to Internet resources and standard textbooks on Extreme Programming, which they were able to use as they were needed.

Peer Rating   Finally, it is interesting that the students' consider each others participation in the Extreme Programming teams differently to the classic teams. Students in both streams are required to provide peer-ratings of the contributions of each other student in the group every two weeks. In the classic teams, these ratings often diverge greatly across the group, whereas XP teams often gave the same ratings for each group member.

4.2   Clients

The customers were pleased with the software products produced in the course: as mentioned above, the XP groups clearly produced better software than those using traditional methodologies. In effect, the XP approach generated a series of increasingly more useful prototypes which greatly assisted less technical clients with visualising how they could use the software and what additional features or other requirements they might have. The planning game process with its use of stories and index cards to document individual features and other tasks was easy for clients to follow and the negotiation over which features were worked on in which order was much more direct and simple to understand than more formal project documentation and project timetables.

Documentation   Our main unexpected interaction with clients was that some wanted more documentation than the Extreme Programming practices produced (even with the additional documentation we required, see figure 8). One client in particular worked on two projects, one with an XP team and one with a classic team, and afterwards strongly preferred the classic approach. We believe there may be a number of reasons for this. Some clients were working in a documentation-centric environment themselves; documentation is easier to read than code; some teams' choice of technology meant it would not be directly transferable into their clients' environments (but documentation would be); some clients may have trusted students to do documentation rather than write production-quality code. We hypothesize many clients participate in COMP389 as a way to develop their own ideas, and explore how they might solve a particular problem, and so need the documentation to drive their own reflections. These clients don't expect that they will get the final product in a form that will be useful, but, via the documents produced by the teams, they will better understand what a useful final product might look like.

Expectations   On the other hand, many clients' expectations were easily met, in some cases, almost too easily, as the program would be complete by the third or fourth iteration, leaving the teams with little work to do in the final weeks of the course. In these situations, the tutor, acting as coach, needed to encourage additional feature development to keep the team working! As a result, some tutors are concerned that the projects have become too easy, and that bigger projects would give the course a better, edgier, sense: teams would be forced to make harder trade-offs when deciding what to include in an iteration, for example.

While the tutor can encourage the client to request additional requirements, this introduces a somewhat artificial and arbitrary aspect to the work. This effect, is, of course, another symptom of the general success of the course, rather than a serious problem to be addressed! Reflecting on all the projects so far, however, it is clear that the keen participation of these "real" external clients was another critical factor for the success of the XP option.

4.3   Staff

The key staff in COMP389, both classic and XP COMP389 projects, are the tutors who work with the individual groups of students. These tutors may be fourth-year students, graduate students, or academic staff: and we have been fortunate over the years in attracting a team of highly experienced and skilled tutors to the course. Note that the academic staff coordinating the course and responsible for the (often quite qualitative) final marks do not participate as individual group tutors: this avoids the appearance of conflicts of interest, and provides a second level of moderation for the assessment.

Tutors as Coaches   It is clear that these tutors are even more important in the Extreme Programming option than in the traditional version of the course. To quote one of the tutors: *"thinking of yourself as a coach in the XP sense is crucial"*. Particularly early on, coaches need to be quite directive, talking students through the practices and modelling what is needed. It is also important for tutors to guide the students towards infrastructure (servers, languages, and so on) that are going to be relatively easy to get going in the available lab environment.

Assessment   All the staff are involved in the assessment of the course, and this leads to the main difference in activities with the adoption of XP. Essentially, coaching an Extreme Programming team involves more talking, and less documentation marking — especially as our practice is to mark each document twice, once as a draft and once as the final version. We generally found the Extreme Programming option projects more difficult to assess: although both documents and process must be assessed qualitatively and holistically, it is harder to judge intangibles such as process following or code quality. On reflection, we have evolved holistic qualitative standards for the documents produced by the traditional projects, and we expect this should happen over time for the assessment of the XP projects.

5   Conclusion

Extreme programming, and other agile methodologies, are becoming more popular in software engineering practice, and accordingly more common in software engineering education course work. This shift has implications for capstone project courses, which are generally constituted, structured and assessed in terms of heavyweight, milestone-based, documentation-centric software processes. We have described our experience in adapting such a project course to provide an option incorporating agile methodologies to those students who wish to pursue it. Overall, this option has been very successful, with students producing more substantial software than the traditional approach, often with less work. The key factors producing this outcome were: skilled tutors able to act as Extreme Programming coaches for the teams; timetabling and physical facilities that strongly support group working; and external clients who were willing and able to engage with the Extreme Programming processes. We plan to continue with the Extreme Programming option for COMP389 in the future, and are investigating ways in which Extreme Programming practices and agile methodologies can be incorporated more solidly into earlier courses in our curriculum.

References

Astrachan, O. L., Duvall, R. C. & Wallingford, E. (2003), Bringing extreme programming to the classroom, *in* 'Extreme Programming Perspectives', Addison-Wesley, chapter 21, pp. 237–250.

Beck, K. (1999), *Extreme Programming Explained: Embrace Change.*, Addison-Wesley.

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J. & Thomas, D. (2001), Manifesto for agile software development. Available from http://agilemanifesto.org/.

Beck, K. & Fowler, M. (2000), *Planning Extreme Programming*, Addison-Wesley.

Becker-Pechau, P., Breitling, H., Lippert, M. & Schmolitzky, A. (2003), Teaching team work: An extreme work for first-year programmers, *in* 'Extreme Programming and Agile Processes in Software Engineering: Proceedings of the 4th International Conference, XP 2003', Lecture Notes in Computer Science, Springer-Verlag, pp. 386–393.

Beedle, M., Schwaber, K. & Martin, R. (2001), *Agile Software Development with Scrum*, Addison-Wesley.

Brown, J. (2000), Bloodshot eyes: Workload issues in computer science project courses, *in* 'Proceedings of the Asia-Pacific Software Engineering Conference (APSEC)', IEEE Computer Society Press, pp. 46–52.

Brown, J. & Dobbie, G. (1998), Software engineers aren't born in teams: Supporting team processes in software engineering project courses, *in* 'Proceedings of Software Engineering Education and Practice (SEEP)', IEEE Computer Society Press, pp. 42–49.

Brown, J. & Dobbie, G. (1999), Supporting and evaluating team dynamics in group projects, *in* 'Proceedings of SIGCSE (ACM Special Interest Group in Computer Science Education)', ACM, pp. 281–285.

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

Cockburn, A. & Williams, L. (2001), The costs and benefits of pair programming, *in* 'Extreme Programming Examined', Addison-Wesley, chapter 14, pp. 223–244.

Dijkstra, E. W. (2001), What led to "Notes on Structured Programming". Available as http://www.cs.utexas.edu/users/EWD/ewd13xx/EWD1308.PDF.

Henderson-Sellers, B., Simons, A. & Younessi, H. (1998), *The OPEN Toolbox of Techniques*, Addison-Wesley.

Highsmith, J. (2002), *Agile Software Development Ecosystems*, Addison-Wesley.

Jeffries, R., Anderson, A. & Hendrickson, C. (2001), *Extreme Programming Installed*, Addison-Wesley.

Johnson, D. H. & Caristi, J. (2003), Extreme programming and the software design course, *in* 'Extreme Programming Perspectives', Addison-Wesley, chapter 24, pp. 273–285.

Krutchen, P. (1999), *The Rational Unified Process*, Addison-Wesley.

Lappo, P. (2002), No pain, no XP: Observations on teaching and mentoring extreme programming to university students, *in* 'Proceedings of the Third International Conference on eXtreme Programming and Agile Processes in Software Engineering', Universitá di Cagliari and Free University of Bolzano-Bozen, pp. 35–38. http://www.xp2002.org/prog_full.html.

Lea, D. (2000), Draft Java coding standard. Available as http://gee.cs.oswego.edu/dl/html/javaCodingStd.html.

Macias, F., Holcombe, M. & Gheorghe, M. (2003), Design-led & design-less: One experiment and two approaches, *in* 'Extreme Programming and Agile Processes in Software Engineering: Proceedings of the 4th International Conference, XP 2003', Lecture Notes in Computer Science, Springer-Verlag, pp. 394–401.

Martin, A., Noble, J. & Biddle, R. (2003), Being Jane Malkovich: A look into the world of an XP customer, *in* 'Extreme Programming and Agile Processes in Software Engineering: Proceedings of the 4th International Conference, XP 2003', Lecture Notes in Computer Science, Springer-Verlag.

McBreen, P. (2003), *Questioning Extreme Programming*, Addison-Wesley.

Mugridge, R., MacDonald, B., Roop, P. & Tempero, E. (2003), Five challenges in teaching XP, *in* 'Extreme Programming and Agile Processes in Software Engineering: Proceedings of the 4th International Conference, XP 2003', Lecture Notes in Computer Science, Springer-Verlag, pp. 406–409.

Noble, J. & Biddle, R. (2002), Notes on postmodern programming, *in* 'Proceedings of OOPSLA Onward! stream', Dreamsongs Press.

Noll, J. & Atkinson, D. C. (2003), Comparing extreme programming to traditional development for student projects: A case study, *in* 'Extreme Programming and Agile Processes in Software Engineering: Proceedings of the 4th International Conference, XP 2003', Lecture Notes in Computer Science, Springer-Verlag, pp. 372–374.

Poppendieck, M. & Poppendieck, T. (2003), *Lean Software Development: An Agile Toolkit for Software Development Managers*, Addison-Wesley.

Project Management Institute (2000), *A Guide to the Project Management Body of Knowledge (PM-BOK Guide)*, Project Management Institute.

Robinson, H., Hall, P., Hovenden, F. & Rachel, J. (1998), 'Postmodern software development', *The Computer Journal* **31**, 363–375.

Sanders, D. (2003), Student perceptions of the suitability of extreme and pair programming, *in* 'Extreme Programming Perspectives', Addison-Wesley, chapter 23, pp. 261–271.

Tomek, I. (2003), What I learned teaching XP, *in* 'Smalltalk Solutions 2003'.

US Department of Defense (1988), DOD-STD-2168 DEFENSE SYSTEM SOFTWARE QUALITY PROGRAM. Military Standard.

Williams, L. (2002), Summary of the third eWorkshop on agile methods. Available at `http://fc-md.umd.edu/projects/Agile/3rd-eWorkshop/topic4.html`.

Williams, L. & Kessler, R. (2002), *Pair Progamming Illuminated*, Pearson.

Williams, L., Wiebe, E., Yang, K., Ferzlid, M. & Miller, C. (2002), 'In support of pair programming in the introductory computer science course', *Computer Science Education* .