Using Extreme Programming in a Capstone Project

Karen Keefe[(1)], Martin Dick[(2)]

Department of Computing & Information Technology[(1)]
Swinburne TAFE
144 High Street Prahran, Victoria 3181, Victoria
School of Computer Science and Software Engineering[(2)]
Monash University
900 Dandenong Road, Caulfield East 3156, Victoria

KKeefe@groupwise.swin.edu.au[(1)]

martin.dick@csse.monash.edu.au[(2)]

Abstract

This paper describes the experience at Swinburne TAFE of using the Extreme Programming software development methodology with a final year capstone project. It found that it was possible to use the methodology successfully for such a project, but that students need to be actively coached in the skills necessary to make XP. A positive result was that less skilled students made more progress than probably would have been the case using a traditional methodology.

Keywords: Capstone Projects, Extreme Programming, Software Development Methodologies

1 Introduction

In recent years, agile software development methodologies have received a great deal of attention in the software development world. Extreme Programming (XP) was one of the original agile software development methodologies to have emerged during this period. XP is not only one of the first agile methodologies, but one of the most widely recognised of this type of methodology (Juric 2000; Nawrocki et al. 2001; Newkirk 2002).

The use of XP in industry has been claimed to provide significant benefits (Beck 1999) and there seems to be potential in the use of the methodology for student projects. This paper looks at the experiences of using the XP methodology in a capstone project at Swinburne Technical and Further Education (TAFE).

2 Overview of Extreme Programming

The differences that distinguish XP from traditional software development methodologies are the emphasis on (Beck 2000) :

- continuous concrete feedback from short cycles of development;
- an incremental approach to planning designed to develop a high level overview of the system under development, which can then continuously evolve;
- an adaptive approach to the way businesses requirements change and hence an adaptive approach to the implementation of functionality required by the software system;
- the emphasis on an automated testing process that is designed to catch defects injected into the software earlier;
- using testing, source code and oral communication as a means to communicate the systems structure and purpose as opposed to system documentation; and
- a continuous and evolving design process that lasts as long as the system is in existence

At the heart of XP are the following four values (Beck 2000):

1. **Communication**: XP aims to keep communication flowing by employing many practices that cannot be done without direct communication. It stresses that most communication should be direct face-to-face communication.
2. **Simplicity**: XP stresses the importance of keeping designs simple According to Beck (2000) for software systems to be considered simple the following criteria should be met – the code will run all tests, communicates everything to the programmers that needs to be revealed, there is no code duplication and has the least number of classes and methods required.
3. **Feedback**: Developers and clients should receive feedback on the state of the system as frequently as possible.
4. **Courage**: This value is based on the fact that developers need to be able to see that the development process has gone in the wrong direction and that corrections are necessary. Correcting the problems might entail throwing away days of work and rewriting the code, even though the code had previously passed tests.

The values of XP are implemented and enforced by the use of twelve practices. These practices are core to the use of XP. They are:

Practice	Practice Overview
Planning Game	Determining scope of project and releases, by combining the priorities of the business with the technical estimates. An ever-changing plan.
Small Releases	Frequently releasing simple systems, and releasing a new version on a very short cycle (1 to 3 weeks).
System Metaphor	Simple shared story of how the system works, to give both developers and clients common ground.
Simple Design	Keeping the system design as simple as possible and finding and removing extra complexity as soon as possible.
Testing	Continuously writing and running required tests, each time new code is written or existing code is changed including unit and client written acceptance tests.
Refactoring	Improving design of a project without changing the functionality of the existing code. Removing duplication, improving communication, simplification and flexibility.
Pair Programming	Writing production code with two developers at one machine.
Collective ownership	All developers responsible for all code, therefore being able to make changes to any piece of code at any time when necessary.
Continuous integration	Integrating all changes once completed versus developing them in separate branches.
40-hour week	To keep developers interested, creative and fresh no one should work more than 40-hours maximum in one week. And no developer should do more than a week's overtime in a row.
On-site customer	A customer is part of the development team as a domain expert to aid the developers in the production of the system. The customer is located at the same site as the development team.
Coding Standards	Developers write all code in accordance with the standards agreed upon, by the development team, to ensure that communication is made through code.

Table 1 Overview to the twelve key practices (Beck 2000)

The twelve practices combine together to form a coherent whole, where the weakness of one practice is covered by the strength of another practice. For example, refactoring to remove complexity can be a very dangerous process, with the change causing significant problems in other sections of the code, this is handled by having very high levels of testing and continuous integration which will very quickly reveal any problems caused by the refactoring.

3 Related Work

XP has not been used extensively in educational settings. Three studies have been identified. The first of these studies was conducted at the University of Karlsruhe using graduate students participating in a practical training course as part of their degree requirement. (Muller and Tichey 2001)

The second study was held at the University of Calgary with fourth year students completing a design project requirement. (Kivi et al. 2000)

The final study was conducted at the North Carolina State University with 150 seniors completing a software engineering course. (Shukla and Williams 2002). The following sections detail the results of each of the studies in relation to the practices of XP.

3.1 Planning Game and Small Releases

The course at North Carolina State University was held over a four-week period and they had one release. The Planning Game was simulated in a classroom environment with all one hundred and fifty students being involved. The students were encouraged to break the release down into smaller iterations, but the customers did not participate in the iterations. (Shukla and Williams 2002) Student felt the process of the Planning Game, establishing the user stories was beneficial to their understanding of the project. The small releases and iterations helped the students identify where problems and defects were injected into the code base.

Kivi et al (2000) found that the process of incremental deliverables and small releases was beneficial to the team, as the team was able to better focus on the customer requirements and allowed for a feedback process to be established. The team from this case study used use cases, from the Unified Modelling Language (UML), to establish the client requirements. The team rather than the client wrote the use cases after establishing the general requirements for the whole system with the client. (Kivi et al. 2000) This differs from the XP Planning Game practice where the customer writes the user stories,

prioritises requirements to be completed in an iteration and for the overall release.

At the University of Karlsruhe the instructor was the customer and established the requirements for the iterations. Students in this study found it difficult to only plan and develop for the iteration. Students kept planning for possible extensions to the project that were never implemented in the end product. (Muller and Tichey 2001)

3.2 System Metaphor

The case study from the North Carolina State University was the only one to explicitly mention the system metaphor practice. Some of the students adopted the metaphor, but the overall findings were that students either did not try to use a metaphor, or they did not like using the metaphor. One comment from the students was that they did not see the point of using a metaphor to describe the concept of the system. (Shukla and Williams 2002)

3.3 Simple Design

Students from the University of Karlsruhe had problems with the practice of simple design, instructors involved in the course believe that this was due to students being taught throughout most of their course to think about reuse, extensibility and for future developments. (Muller and Tichey 2001) Students in this study called the process "design with blinders". The findings of this case study were that it was a difficult practice for the students to implement.

The students from the University of Calgary had difficulty with the lack of future planning. During their second increment they spent much of the time changing the architecture so that they could implement the requirements for that increment. (Kivi et al. 2000)

With the third of the case studies, the instructors found that the students reported success with the implementation of simple design, because students tended to do simplest thing possible anyway and not document. (Shukla and Williams 2002) The authors of this case study believe that there are problems, from an educational viewpoint, of the students not completing any documentation.

3.4 Testing

The students from the University of Calgary used jUnit for their unit testing. Though they said that they neglected to write their test cases upfront, they were writing them afterwards; this was due to the slow progress that was being made when writing the test cases upfront. (Kivi et al. 2000) They did recognise that the implementation of upfront unit testing would have been more beneficial if implemented from the start of the project.

The students from the University of Karlsruhe did, for the most part, complete upfront testing. The production of test cases upfront made the student more confident with their code. (Muller and Tichey 2001) The continuous running of the tests was also seen as beneficial as due to the automatic regression testing. One problem that the students from this case study faced was that those who were developing the graphical display had no tool for automated testing. They wrote manual tests and ran those, but they were still ensuring that the code that they wrote was tested. (Muller and Tichey 2001)

The third case study had mixed results in encouraging students to write their test cases upfront. Some students from this case study wrote their test cases after the production code was written and only wrote the test cases to complete the testing requirements of the project. (Shukla and Williams 2002) Students did report advantages to upfront testing, for example knowing how much work was left on a task and not having to rewrite tests as the test base was already in existence. (Shukla and Williams 2002)

All three case studies used jUnit as the testing framework for the test cases. jUnit is a Java testing framework that was developed by Kent Beck for XP.

Of the three case studies only one, North Carolina State University, dealt with acceptance testing. The students had to suggest to the client the test cases that would make up the acceptance tests. (Shukla and Williams 2002)

3.5 Refactoring

The students at North Carolina University found that refactoring was a necessary and valuable practice. (Shukla and Williams 2002) Refactoring improved their code, making it simple by removing unnecessary complexity and duplication. They also combined the practice of testing successfully to ensure that they did not break their code.

Refactoring was not an issue for the students of the University of Karlsruhe; they found that their designs were complete and simple enough without having to refactoring them. The authors believe that there are two possible reasons for this; one being the small scale of the project and the other being the students did a full design, rather than minimal designs. (Muller and Tichey 2001)

The students from the University of Calgary found that initially they did not need to refactor their code, there were few defects and they were producing working releases quickly. (Kivi et al. 2000) Once more requirements came into the development team they found that they spent more time refactoring than implementing new features. Team members did not see how refactoring was adding value to the project. The study found that if more time had been spent in the initial planning stages of the project, then much of the refactoring might not have been necessary as there would have been a better understanding of the system by the developers. (Kivi et al. 2000)

3.6 Pair Programming

Two of the case studies reported that the students had positive experiences with pair programming. Students generally found that pair programming helped them solve problems and pick up defects quicker. (Shukla and Williams 2002) Due to the students' schedules it was not

always possible for them to work together. Students also agreed that they learnt from the partners when pair programming. (Muller and Tichey 2001) There was also the transfer of knowledge between the teams involved within the project teams, with the pairs sharing their knowledge of the code base with their team members. (Kivi et al. 2000)

Students involved in one of the case studies, however, discontinued the pair programming practice due to scheduling problems and because they felt that coding known or accepted code together was wasting time. (Kivi, et al. 2000) The students from the University of Karlsruhe felt that pair programming was also wasting time when developers were coding "get and set methods" that is a low level of programming. (Muller and Tichey 2001)

3.7 Collective Code Ownership

The North Carolina State University case study was the only one that specifically mentioned the implementation of collective code ownership within the project that their students undertook. The students involved used Concurrent Versions System (CVS) as their source code control system. (Shukla and Williams 2002) Students found the transition to collective code ownership hard initially but a successful practice by the end of the project. (Shukla and Williams 2002)

3.8 Continuous Integration

The North Carolina State University case study was the only study to discuss continuous integration in terms of the number of builds of the code base per day. Students developed their coding tasks and integrated it back into the code base once the task was competed. Students saw this as a benefit as they could see the project growing and working as a whole. (Shukla and Williams 2002)

The Muller and Tichey (2001) only mention continuous integration or configuration management in relation to the tool CVS that was used to manage the code base. Kivi et al (2000) discuss the manual version control system that the students initially used. It was error prone and difficult to manage. By the end of their project the students were using Java Concurrent Version System (JCVS) that enabled the students to be more consistent with the management of their code base. Neither of these articles discusses how often students went through the process of building and integrating the code base once tasks were completed.

3.9 40 Hour Week

The article by Shukla and Williams (2002) discussed the forty-hour week practice. This practice was implemented by basing the ideal programming week on nine hours per week. They found that when students were pushed into working overtime their artefacts were of poorer quality.

3.10 On-site Customer

In all three of the case studies teaching staff played the role of the client. There were limitations to the staff simulating on-site customers, but tools such as emails and message boards substituted for the clients being with the development team at all times. The three case studies do not discuss the role of the client in more depth.

4 Research Background

Capstone projects are incorporated into the final year of most computing courses at tertiary learning institutions and aim to bring together the skills that a student has learnt throughout the duration of the course. (Clear et al. 2001) The common formats of capstone projects are development projects or research based projects. With both these styles of the capstone course there are common elements, a project, a team, a sponsor, an instructor and a coordinator. (Clear et al. 2001)

Swinburne TAFE is made up of six campuses, Croydon, Wantirna, Prahran, Hawthorn, Lilydale and Healesville. The Diploma of Information Technology – Software Development is conducted at Croydon, Wantirna and Prahran. Capstone projects have been incorporated into the final semester of the two-year Diploma of Information Technology – Software Development since 1997.

The capstone course gives the students a holistic experience of working on a larger scale project and the opportunity of putting into perspective the different roles of software development teams. Students also experience working with real-world clients, gaining invaluable skills in dealing with clients, which are not developed in other parts of the diploma course. Students work within teams of four to six, experiencing a team dynamic for an extended period of time. This improves upon their team skills, skills that are valued by employers.

The capstone course at Swinburne TAFE is product-oriented. The emphasis is for students to produce robust, quality software that the clients can continue to use long after the software delivery. The standard structure for the capstone project is an incremental object-oriented lifecycle. The incremental model combines a linear sequential model with the iterative nature of a prototyping model. (Pressman 2001) Student projects consist of two increments, the first increment delivered at the end of the fourteenth week of the project and the second increment delivered in week eighteen, which is the final week of the capstone project.

The time that the students are required to spend in official classes is 360 hours for the eighteen-week semester. These 360 hours are divided over four subjects. Part of the 360 class hours will be spent teaching students necessary skills to complete their projects, while the remainder of the hours will be spent working on the projects. In addition to the 360 hours class time, students will be expected to work on the projects in their own time. Each student is expected to spend approximately one additional hour outside of the classroom for every hour that is spent in the classroom. This will give the students a total of 720 hours for the semester or 40 hours per week, though this time is dependant on the projects that are undertaken, the skill level and dedication of the students involved.

4.1 XP and the Capstone Project

The capstone project was structured around a system that was to be developed to provide a simulation of the lifecycle of a software development project where the player takes the role of the project manager and has to guide their project through to completion (simProject). The simulation is designed to be used in the Bachelor of Computing at Monash University to assist in the teaching of IT project management. The author based at Swinburne TAFE acted as the XP mentor for the team, while the author from Monash University acted as the client for the project.

The project was proposed to the four groups as a possible project and the use of XP was made explicit in the proposal along with an explanation of the XP methodology. One of the teams was quite enthusiastic about the methodology and volunteered. The team was then given a one week introduction to the tools used in the XP process and then commenced the project.

4.2 Research Method

With only one team to work with and the exploratory nature of the research, it was decided to use an action research method. Action research encourages researches to take action and to effect positive change based on findings, rather than simply report findings. (Mills 2000) The action researcher has a routine that involves looking, thinking and acting in a continuous cycle throughout the duration of the research period. (Stringer 1996). This was felt to suit the circumstances of the research well.

Data was gathered through interviews, observations self-reflection on the part of the XP mentor and a post-mortem report written by each of the students. Interviews were conducted with the four students in weeks 6, 12 and 18. The first and second sets of interviews took between twenty and thirty minutes, with the third set taking between forty-five and sixty minutes.

The interviews were transcribed and then analysed using the NVIVO tool (Richards 1999) and the following high-level categories were used:

- *Students* – category created to hold all information contained within a transcript based on the interviewee.
- *Experiences* – detailed students' experiences with the other students in the XP team.
- *Difficulties* – detailed difficulties relating to the project, but not explicitly linked to XP. For example difficulties relating to estimating or the technologies that were implemented to enhance the XP practices.
- *Extreme Programming* – data explicitly related to the XP practices and values.
- *Future* – foreseeable future problems within the project either directly related to XP or not.
- *Project* – discussions relating directly to the project under development, that is simProject.

Ongoing results were fed back into the project to improve the implementation of the XP practices as the capstone project progressed.

4.3 Teaching Framework

A teaching framework for the capstone project was established prior to the students commencing. A detailed description of the framework can be seen in Appendix 1.

4.4 Ethical Considerations

Given the experimental nature of the application of XP to a student capstone project at Swinburne, it was decided that the use of a single team and project using XP would allow the authors to concentrate their efforts and give the students in the team the best chance of success. Ethically, this was an important consideration as we did not want to disadvantage the team in comparison to the other project teams in the Diploma.

5 Results

The results are presented in terms of the twelve practices of Extreme Programming and a number of other lessons that were learned.

5.1 Planning Game and Small Releases

Students found that the Planning Game was mostly a positive experience. One of the positive aspects was that the students were able to concentrate on small parts of the system at a time, which allowed more flexibility in the requirements gathering process. Students found that this aspect of the Planning Game was a double-edged sword, as they did not know the details of all stories that were in the future. Students at Swinburne TAFE are used to working to more traditional lifecycles where all the requirements are gathered during an analysis phase and they have a whole picture of what is expected in the development of the project.

"The positives [of the Planning Game] are if you brainstorm about something you can add it in and you are not set to a whole document. But in the same way this is a negative, because you do not know what's coming ahead."

The constant client involvement in the Planning Game proved to be the most effective way for the team members to get a full understanding of the requirements of the project. Students received constant and direct feedback from the client ensuring that they had a clear understanding of what they were required to do.

Issues that were raised during the Planning Game were brought about by the students' poor estimating skills. Students involved in simProject were inexperienced with non-trivial development tasks and the estimation of these tasks. Initially when estimating tasks at the beginning of development students were overestimating and tasks were completed ahead of the estimated schedule. This had a twofold effect of giving the students the impression that they were not achieving as much as they should have been, while having a secondary effect of making the

students complacent with the progress of the project. As the project progressed to more complex tasks the team underestimated how long it would take them to complete their programming tasks. There were two reasons for the inaccurate estimations; a) the complexities of the task at hand and b) the lack of experience the students had in creating estimates for themselves.

Another problem that the students had was the breaking down of the stories into the task level, stories themselves were eventually found to be too small and therefore too hard to be broken down. Or the when broken down they were so small that they were too dependant on other tasks for them to be completed as discrete tasks. This may be linked to the size of student capstone projects. Capstone projects must be able to be completed within a set time frame, which may then impact upon the level of complexity and the size of the project under consideration.

5.2 System Metaphor

This practice was the least understood and least liked of the practices. The XP team did work with an external System Metaphor in the initial stages of the project development. Game simulators such SimCity and King of Dragon Pass were used initially to give the students an understanding of the concept behind simProject, but within only a few weeks simProject became the metaphor for itself. The students found that the concept of simProject was not difficult for them to understand and therefore there was no need for them or the client to use an external source for them to get a common language.

5.3 Simple Design

Students initially developed the system essentially so that it would work, they did not explicitly keep in mind the need to keep the design as simple as possible. Throughout the project with refactoring students have been able to keep the design as simple as they could, but with time constraints they were not able to refactor the code enough to simplify the design as much as they could have done.

Students felt that they would have benefited from more light upfront design, which as well as helping keep their system design simple, would have assisted them with their testing and with the continuous integration practice. At different stages of the project, when tackling complex tasks, some light upfront design was completed, to the benefit of the students. This design gave the students a "roadmap" to the task at hand and an understanding of how the components of the system fitted together. Students appreciated the design sessions, but were not self-disciplined enough to continue the design sessions on their own. Instead they would simply work together on the task until they achieved a workable design. Observations from the researcher indicate that tasks performed after a design session were completed quicker and with less associated problems, for example the test cases were written or updated without problems and integration of the new code with the existing code base was completed with less problems.

5.4 Testing

XP uses the testing process to drive the design of the system at hand, but the students who had little design experience, whether in a traditional or in the XP manner, found that writing the test cases this way was problematic. The main concern for the students was that they did not know what they were going to test prior to the code existing. Students wrote test cases once a class was either partially or completely written. Student C said of the testing process:

"You don't know what all the methods are, so it's hard to write upfront tests."

Another issue with the testing process was that the students had to learn how to use jUnit, and initially, before jUnit was installed on the development server, students were writing their own test harnesses to test their classes.

"We did some testing of the servlet that we created, we wrote a little test harness for MonkeyReader class, a straight forward harness, doing it pretty much what jUnit was doing, but doing it manually, through a standard test harness."

Once jUnit was installed and running correctly the students then had the prospect of learning the new tool, and trying to assess how jUnit would work. Initially students were not comfortable using the jUnit framework. Student B said of the jUnit framework and why they persisted with manual testing:

"I had a play with jUnit, because I wasn't sure about all of the commands that were available, I wasn't quite comfortable within jUnit so that's why I stuck to the manual testing."

The attitude to jUnit changed as the project progressed and the students became more comfortable with the tool, and they used it more frequently this attitude changed. Student C said:

"jUnit was one of the new technologies that I actually picked up quite well, and I am happy to learn something new. It helped me a little bit with my logic and understanding of methods in the actual Java classes."

5.5 Refactoring

Of all the XP practices, refactoring was one of the most successful. Students found that refactoring was a natural process and implemented it throughout the project. The design of the system was greatly simplified by careful refactoring at the commencement of each new task, or during tasks. The students understanding of the code improved as they refactored the code, and their coding abilities improved as they found better ways to write pieces of the system whilst refactoring. Student B said:

"Refactoring helps make the code work better and it helps you understand the code better; expand your horizons of the code. There are lots of different methods for doing one particular feature, different ways of doing it."

5.6 Pair Programming

When the students implemented pair programming it was a successful practice. Over the last month of the project, due to external forces, where one student was not able to work within the tertiary learning institution for much of the time, pair programming was essentially abandoned. The researcher observed that the pair programming experience of the XP students is consistent with that of students in other studies: improved quality in the code, more efficient coding, communication between students in the group improved, better problem solving and a more enjoyable experience for the students pairing. When not pair programming the converse was true. Within a tertiary learning environment students are not always able to pair program, due to scheduling conflicts of students, available resources, that is computer availability, and influences external to the control of the tertiary learning institute. The students were more likely to pair with students of similar capabilities and enthusiasm. Another benefit of pair programming was that the technically weaker students were more confident in coding with a partner, especially if their partner was of similar abilities, and they also believed that they coded more within the XP team than they would have in a non-XP team.

5.7 Collective Code Ownership

Students were slow to adapt the practice of collective code ownership because there is an emphasis on producing your own work and not sharing work with classmates. The capstone project had an individual coding portfolio component, which had an initial negative impact on sharing code throughout the team. Once students were assured that this individual component would not cause a reduction in their overall results, collective code ownership was embraced and students were actively sharing code between the team members. The technically stronger students were initially reluctant for the weaker students to make changes to their code. This attitude gradually changed throughout the project, especially when pairing with the weaker students, though the stronger students did tend to pair with each other. The technically weaker students were not concerned with who changed or improved their code.

5.8 Continuous Integration

This practice was one of the least successful implemented by the students. There were several reasons for the lack of success with continuous integration: a) Ant, the integration tool that was being used for the project, was not running correctly for the first half of the project; b) students did not fully understand how to use CVS, when and what code to check into CVS; and c) lack of co-ordination between the development pairs.

5.9 40-Hour Week

The technically weaker students found that initially they were struggling to complete a full forty hours of work per week. In other capstone groups, of current and past years, students who were not technically strong spent much of their hours creating or working on the various pieces of documentation that is required to be completed. Within the XP group, because of the reduction in documentation, the technically less capable students, whose coding tasks were not as intensive, averaged thirty-five hours per week

The benefit of this practice was that the XP team worked a consistent number of hours over the eighteen weeks of the project, rather than leave the bulk of the work to the final few weeks in the project, a practice that happens with many groups at Swinburne TAFE undertaking capstone projects.

5.10 On-site Client

Having a client on-site available to students is impractical within the Swinburne TAFE environment. The client involved with the project was external to the tertiary learning institution and as such the client was not available to be located on-site. Students made frequent use of emails to contact the client when he was on-site. The development team and the client met on a weekly basis to discuss the project progress, issues that may have arisen with the requirements and to continue the Planning Game.

Though the project didn't implement an on-site client, the constant contact and communication with the client proved to be a very successful practice. The weekly meetings were most useful, regardless of whether the students felt that they required the meetings or not. The students were able to clarify issues that were raised quickly and were always able to inform the client of the progress of the project. Students were more honest in their dealings with the client because of the relationship that was formed during the project, and there was no hesitation when having to tell the client when the project was slipping behind schedule.

5.11 Coding Standards

The XP team chose the Java Coding Conventions, defined by the Sun Microsystems, for the teams coding standards. The standards that the team used in practise were a combination of Java Coding Conventions and those taught during the programming subjects of the previous two semesters. Though students did not adhere strictly to the Java coding conventions, they did adhere to their own set of standards. The adoption of standardised coding conventions was a gradual process, and once completed communication was made through the code, but it would only be clear to other members of the team. Adopting coding standards such as the Java Coding Conventions would have been preferable as the development of simProject is to be continued by another group of students.

5.12 Development Environment Issues

A variety of problems occurred with establishing the development environment for the project, partly due to the remote location of the server that the team used and partly due to teething problems with the environment. The environment used was:

- the jUnit Testing framework;
- Ant continuous integration tool;
- Tomcat web application server; and
- CVS concurrent version system.

The lessons learned from this exercise:

1. the development server should be located where the XP team, mentors and associated staff are located and they should have control over the administration of the server; and
2. the development environment needs to be set up well in advance of the tools being required and it needs to be fully tested.

Students also were required to learn the development tools during the course of the project, which caused some problems. Ideally, the students should have been introduced to the tools prior to the commencement of the project.

5.13 Process Coaching

One issue that became apparent during the course of the capstone project was that when students were faced with difficult situations, the students quickly reverted back to non-XP practices. Pair programming, continuous integration and upfront testing were the practices that students generally abandoned when time was running short or there were other pressures. This was true later in the project after students had experienced the benefits of pair programming on their programming. Acknowledging that they worked more efficiently and improved the quality of their code whilst pair programming, students still reverted back to programming as individuals in an attempt to complete more tasks. With continued coaching, students went back to implementing the XP practices.

When the use of process coaching was at its greatest students were implementing the practices continuously and not reverting back to their own habits. It was the researchers decision to gradually reduce the amount of coaching in the final four weeks of the project after the students had been implementing the practices for fourteen weeks. At this point in the project there were external influences on the project and the students participating that were beyond the control of the researcher and affected the way in which students were able to work. One student was unable to travel to Swinburne TAFE to work, but was still able to work from home. The students reverted back to their old habits of developing software, and many of the XP practices were discontinued.

5.14 Student Learning

It was observed throughout the XP project that the weaker team members improved their coding skills. In many software development capstone projects that involve group work, team members may be pigeonholed into positions within the team. Team members whose technical skills are not as strong as their colleagues are put into a position where creating and maintaining documentation is their primary task. Within the XP team, where there was not an emphasis on documentation, the technically weaker students were required to be more involved in the coding aspects of the project and there was an improvement in their technical skills.

6 Conclusions

This research found that is feasible to implement the Extreme Programming software development methodology for a capstone project within a tertiary learning environment. The students gained many useful skills throughout the course of the capstone project, not only in XP but also in software development and they successfully completed a system that met the client's requirements gathered throughout the Planning Game process.

It has highlighted several important points that need to be kept in mind when implementing XP in capstone projects. These important points are:

- the need for XP coaching;
- scheduling for pair programming;
- set up and testing of the development environment;
- early introduction to XP tools; and
- encouragement of light upfront design sessions.

We are currently rerunning the simProject development with a second team using the XP methodology and appear to have overcome some of the problems that were faced with the first team. In addition, we feel that several XP practices (pair programming; automated testing; continuous integration; coding standards; and refactoring) can be of value in non-XP projects and we are currently working on integrating these into the Diploma of Information Technology.

While the initial use of XP has had positive results, the small scale of the research and the results indicate that we have only just begun the process of integrating XP into the capstone project curriculum and more research is needed to determine the benefits and disadvantages of using XP and also the best methods of introducing XP to students.

7 Appendix 1 – Teaching Framework

Week No	Teaching Hours	Planned Activities
1	6	Introduction to XP *Teaching:* XP practices, philosophies. *Practices Taught:* Planning Game and Planning Game exercises; Coding Standards; On-site client; Small Releases; 40-hour week
2	6	Commencement of Release 1, Iteration 1. Introduction to the client. First round of the Planning Game. *Teaching:* Pair programming; Planning Game, breaking down stories into tasks, signing up for tasks; introduction to CVS, Ant; System Metaphor; Simple Design; Collective Ownership. *Practices Taught:* Pair programming; Planning Game; System Metaphor; Simple Design; Collective Code Ownership *Implementation:* Planning Game, Pair Programming, Collective Code Ownership, Coding Standards, Simple Design, On-site client, System Metaphor
3	4	Continuation of Release 1, Iteration 1 *Teaching:* upfront testing, jUnit, Ant *Practices Taught:* Testing; Continuous Integration *Implementation:* Continuous Testing, Continuous integration
4	2	Completion of Release 1, Iteration 1 *Teaching:* refactoring *Practices Taught:* Refactoring *Implementation:* Refactoring *Interviews:* first set of research interviews
5		Release 1 Iteration 2
6		Release 1 Iteration 2
7		Release 1 Iteration 3
8		Client acceptance testing, release, rollout of production code
		Release 2, Iteration 1
10		Release 2, Iteration 1
Mid-semester break		
		Interviews: second set of research interviews held during mid-semester break.
11		Release 2, Iteration 2
12		Release 2, Iteration 2
13		Release 2, Iteration 3
14		Release 2, Iteration 3 Deliverables due for Project Programming, System Testing and Project Management 2 Training and installation manuals Individual coding portfolios Individual testing portfolios
15		Release 3, Iteration 1
16		Release 3, Iteration 1
17		Release 3, Iteration 2
18		Release 3, Iteration 2 Final delivery and rollout of system. Project Programming, Project Management 2 and System Design 2 deliverables due: Individual Lessons learned reports User documentation System documentation Source code Version control logs Project Management Report and Budgets *Interviews:* third set of research interviews.

Table 2 Teaching Plan for Extreme Programming Software Development Methodology

8 References

Beck, K. (1999). "Embracing Change with Extreme Programming." IEEE.

Beck, K. (2000). Extreme Programming Explained: Embrace Change. Upper Saddle River, NJ, Addison-Wesley Longman, Inc.

Clear, T., F. H. Young, et al. (2001). Resources for Instructors of Capstone Courses in Computing. 6th Annual SIGCSE/SIGCUE ITiCSE Conference, Canterbury, UK.

Juric, R. (2000). Extreme Programming and its Development Practices. 22nd International Conference Information Technology Interfaces ITI 2000, Croatia.

Kivi, J., D. Haydon, et al. (2000). "Extreme Programming: A University Team Design Experience." IEEE.

Mills, G. E. (2000). Action Research: A Guide For the Teacher Researcher. New Jersey, Prentice-Hall, Inc.

Muller, M. M. and W. F. Tichey (2001). Case Study: Extreme Programming in a University Environment. 23rd International Conference on Software Engineering, Toronto, Canada, IEEE Computer Society.

Newkirk, J. (2002). Introduction to agile processes and Extreme Programming. International Conference on Software Engineering.

Nawrocki, J., B. Walter, et al. (2001). Toward a maturity model for Extreme Programming. 27th Euromicro Conference, Warsaw, Poland.

Pressman, R. S. (2001). Software Engineering - A Practitioner's Approach. New York, NY, McGraw-Hill.

Richards, L. (1999). Using NVivo in Qualitative Research. Melbourne, Qualitative Solutions and Research.

Shukla, A. and L. D. Williams (2002). Adapting Extreme Programming For A Core Software Engineering Course. Conference on Software Engineering and Training (CSEE 2002), Covington KY USA.

Stringer, E. T. (1996). Action Research: A Handbook for Practitioners. Thousand Oaks, California, SAGE Publications, Inc.