Case Study: Extreme Programming in a University Environment

Matthias M. Müller Walter F. Tichy
Computer Science Department
Universität Karlsruhe
Am Fasanengarten 5
76 128 Karlsruhe, Germany
{muellerm|tichy}@ira.uka.de

Abstract

Extreme Programming (XP) is a new and controversial software process for small teams. A practical training course at the university of Karlsruhe led to the following observations about the key practices of XP. First, it is unclear how to reap the potential benefits of pair programming, although pair programming produces high quality code. Second, designing in small increments appears problematic but ensures rapid feedback about the code. Third, while automated testing is helpful, writing test cases before coding is a challenge. And last, it is difficult to implement XP without coaching. This paper also provides some guidelines for those starting out with XP.

1. Introduction

Extreme Programming (XP) is a lightweight software development process for small teams dealing with vague or rapidly changing requirements. XP breaks with a number of traditional software engineering practices. First, documentation is almost entirely non-existent. The only "documentation" is a set of index cards on which the team members scribble planned features of the system. Other than that, the source code is the only documentation. The rationale is that writing documentation slows down developers and is mostly neglected anyway. Second, there is no software specification. Executable test cases, written before the code is developed, serve as a substitute. Third, there is no separate design or testing phase. Instead, design, implementation and test are done together, in small increments. Fourth, there is an explicit prohibition against design for change; only the simplest possible design satisfying the feature of the moment should be implemented. Fifth, there are no formal reviews or inspections. However, XP prescribes a combination of special practices instead. The major ones are pair programming (all code is written by two programmers, working together at one terminal), frequent integration of changes, automated regression testing, development in small increments, and occasional restructuring (called refactoring). Requirements and the order of feature development are determined incrementally (called Planning Game in XP). XP is designed for a team of software engineers to become a productive unit that embraces changes and incorporates them quickly into an evolving system.

Initial experience reports with XP are enthusiastic. For instance, Chet Hendrikson of DaimlerChrysler writes:

> When [following XP], we have been the best software development team on the face of the earth.[1]

Others view Extreme Programming as a fancy name for hacking [10]. This paper tries to move towards a fair evaluation of XP. It reports on the experiences made in an XP course held at the university of Karlsruhe in the summer term 2000. Subjects were graduate students. The main purpose of the course was to gather experience with XP in an unbiased fashion, without having to come up with a preordained result one way or the other. In general, the experience with XP was positive, although this paper presents some caveats, some suggestions for the XP-beginner, and proposals for thorough, evaluative research.

The study focused on

Pair Programming: All programming tasks are done in pairs at one display, keyboard, and mouse;

Iteration Planning: Designing and implementing only the functionality required for a small set of new features.

Testing: Test cases specify the functionality and are rerun continuously;

Refactoring: Restructuring the code while retaining its functionality.

Scalability: What is an appropriate team size for XP?

0-7695-1050-7/01 $10.00 © 2001 IEEE

During the course, the students faced several major problems following XP. First, it was difficult to design in small increments. The students nicknamed it "designing with blinders". Their designs were both large and good enough so they never needed to refactor. Another problem was caused by having to write test cases before coding. This approach was new to the students and they had problems accepting and following it, even though they found automatic regression testing useful. Students learnt from each other during pair programming, but the benefit leveled off over time. They also found interesting ways to share work in pairs, but it remains unclear how to structure pair programming without loss of productivity. The amount of communication involved in the planning game turned out to be prohibitive for larger teams. This part of XP definitely does not scale. Finally, without continuous supervision, encouragement and cajoling, students would have followed XP practices incompletely, if at all.

2. Related Work

To find out more about XP, the authors recommend an overview written by Kent Beck [1]. A detailed treatment is given in a book [2].

About the various practices of XP, only pair programming has been evaluated to a certain extent. Bisant and Lyle [3] investigated the effect of a two-person inspection method on programmer productivity. They used a pretest-posttest design with a control group, employing 29 undergraduate students. The students in the experiment group performed a design inspection, a code inspection, or both. During the inspections, two students paired off for 20 minutes and tried to find errors. The students in the control group developed the programs on their own. Bisant and Lyle reported a significant improvement in the experiment group as a result of using the two-person inspection method. The time saving was greater than the time lost in the pair inspection steps. This result may have more to do with the benefits of inspections than with pairing.

There are some claims that a pair is more than twice as productive as an individual. Nosek [7] conducted an experiment about continuous pair programming, but found no support for this claim. Five pairs and five individual professionals solved a challenging problem. The evaluation of the posttest questionnaire showed that pairs enjoyed the problem-solving process more and that the pairs were more confident in their solutions. On average, a single individual took 41% more time than a pair (though not statistically significant at the 0.05 level). Put another way, this means that two individuals, working independently, will be 30% more productive than a pair. Nosek argues that the loss of productivity is made up by better quality, but he has no strong data to support this claim. Williams also reports high confidence of pairs in the problem-solving process [4, 9]. She observed a reduction of the number of errors while pair programming with a significance of $p < 0.01$. Williams measured also a reduction in working time when comparing individuals to pairs, but she could not give estimates about the productivity effects of pair programming. Similar to Nosek, Williams argues that the productivity loss of pairs might be gained back when debugging. In sum, the real benefits of pair programming are unclear.

3. The XP Course

The course was held in the summer term of 2000. Participants were CS graduate students who needed to take a practical training course as part of their degree requirements. Most of the students had experience with team work, but only one with pair programming. Only one student had developed moving graphical displays of the sort needed for the project. Twelve students began the course; one dropped after the first three weeks because of lack of time; the rest completed.

In the first three weeks, students solved small programming exercises to familiarize themselves with the programming environment and to learn XP practices. The exercises introduced jUnit (the testing framework used throughout the course), pair programming, the test practices of XP (write test cases before coding, execute them automatically with jUnit), and refactoring. The remaining eight weeks were devoted to a project on visual traffic simulation. The course language was Java. All students had experience with Java from their early undergraduate courses. Table 1 summarizes the course details.

Table 1. Summary of course details.

Number of participants	12
Qualification	graduate students
Course language	Java
Testing tool	jUnit
Version control	CVS
First exercise	Matrix class
Duration	240 - 660 min
Second exercise	Visualization tool (VT)
Duration	240 - 630 min
Third exercise	Extension of VT with HTML or text output
Duration	120 - 420 min
Project	Traffic simulation
Duration	8 weeks, about 40h per team total

The first exercise covered the implementation of a matrix class. The second exercise was a small visualization

tool combined with a scanner to read data from a text file. The third exercise extended the visualization tool with output in HTML or a text format. The remaining eight weeks were spent with project work. The project was the implementation of a traffic simulation with cars, traffic lights, and trains.

The students met weekly for working together in six pairs. They paired voluntarily with different partners for each exercise and the project. The loose coupling of pairs within a team is a common practice in XP. After the first three programming problems, one student left the course and one of the instructors (Matthias) filled in, in order to have six full pairs. However, Matthias tried not give his team an unfair advantage by providing hints that others wouldn't get.

For the project, the students were divided up into two teams with three pairs each. Each team was tasked with the full project, including the graphical representation, the functionality for the moving cars, the right-of-way rules, and the traffic light control. Figure 1 shows a snapshot of the traffic simulation with seven cars, a crossroad, and a railway. The railway was meant as an extension, but never implemented due to lack of time.

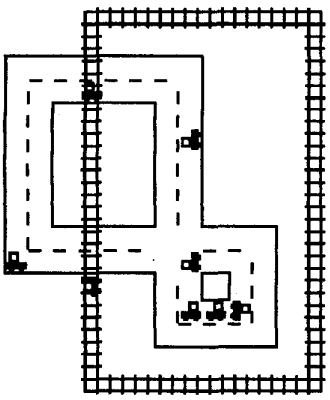

Figure 1. Snapshot of the traffic simulation.

The instructor plaid the role of the customer, telling the students which features to implemented next. In XP parlance, the instructor simulated the planning game.

Students filled out a total of five questionnaires. The first was handed out at the beginning and asked about education and practical experience. The others were filled out after each exercise and at the end of the project. These forms contained questions about XP practices, what the participants learnt, where they had difficulties, and suggestions for improvements. The tables 2-4 at the end of this study sum up the results.

4. Experiences

This section details the experience with XP in pair programming, iteration planning, testing, refactoring, and scalability.

4.1. Pair Programming

The students were asked to pair with different partners of their own choosing for each exercise and the project. Students had no problems adapting to pair work. All but one team enjoyed this style of collaboration and the resulting problem solving process. The exception was a team in which one member wanted to design, while the other wanted to get the task done. Neither student enjoyed the experience, which resulted in having an over-design on the one hand and a bad design on the other. These two students were later placed into different project teams.

Insights about the productivity of pairs vs. individuals cannot be expected from a case study such as this. However, the authors have some observations about the division of work in pairs. The social rules about pair programming by Williams [8] do not say enough about how work should be structured. Students noted that it is a waste of time to watch a partner during rote coding, such as writing a bunch of get and set methods. One student suggested that the partner not typing should perform a constant review of the code, while others suggested different kinds of work such as reading the Java API documentation. One student preferred to meet only for implementing a complicated algorithm while others liked to pair permanently.

Most of the pairs used the following two-display technique: On one display, they implemented while the other showed the relevant Java documentation. When a pair ran into difficulties, one team member consulted the documentation while the other studied the code. When asked if a single display would have been enough, 75% declined. Although this approach was used in a Java environment, it is an indication to soften the strict rule of pair programming at a *single* terminal.

Another aspect of pair programming is learning from each other. The students confirmed that they learned something from their partners. The topics ranged from matrix algebra to editor commands. 43% of the participants stated that they had learned something from pair programming, but this effect declined with the duration of the course.

In summary, pair programming still suffers from some waste of time and from an unclear division of work. It is also unclear whether the main benefit of pair programming, higher quality, could be achieved by a less personnel-intensive approach such as pair inspections.

4.2. Iteration Planning

Given a set of new features to implement, XP's guidelines say to develop the simplest possible solution. The rationale is that software changes are cheap and no time should be wasted developing unneeded generality.

This recipe proved problematic. All our students continuously planned for the future. In order to get students to focus on just the next set of features, the instructors had to publicly abandon one speculative feature after another. For example, at the beginning of the project, a street editor was mentioned that would simplify the construction of the traffic scenarios. The students also heard about trains and level crossings. The instructors made it clear that these were speculative extensions that would probably not be implemented. But once these ideas were out, our students would continuously think ahead to accommodate trains, level crossings, and the street editor. At the end of the project, it became clear that they had always planned for these features. Yet the features were never added, because time ran out. We wonder what would have happened if we had mentioned overtaking cars.

The authors do not know why thinking about minimal solutions is hard. Perhaps this is because our students have been trained to design for the future. They are told in the software engineering core course that there is one thing for certain about every useful software, and that is change. They should plan for likely changes with information hiding, extensible designs, etc. It is quite telling that students called the XP-approach "design with blinders". Thinking ahead may also be a sign of an ingrained (and commendable) thoroughness, a desire to a job well and to be dissatisfied with shoddy work.

In summary, the authors expect that a fair number of good software designers will have difficulty with ignoring knowledge about future extensions. At this point, it is unclear whether minimalistic design is merely a matter of training, or actually a bad idea. It is clear what must be done, though, if one wants to practice the XP approach with somewhat experienced personnel: withhold information about future extensions. This is probably the approach for the next iteration of the XP course. At a later stage, it may be possible to disclose lots of features and still ask developers to focus on only one at a time.

4.3. Testing

There are two aspects of testing in XP: first, to write the test cases before coding, and second, to make them execute automatically for regression testing.

Writing test cases before coding is a substitute for specification. What exactly do the methods do, what parameters do they take, and what are the (testable) results? This approach was new to most students. Only 25% applied it to their development prior to the course, see table 4. Most students adopted it naturally right from start, some needed our intervention in the early stage of the project, but one pair adopted it not until they needed the test cases for restructuring the code. This pair developed the Java class for the crossroads without having written the accompanying test class. When asked whether they forgot the test cases they answered:

> No, we didn't. Why should we implement test cases if we don't know exactly what we have to do? We are still figuring out the desired functionality.

The instructor urged them to write the tests. At the next meeting, they had the test cases written, but they had also changed both the underlying representation and the interface. They cleaned up the code while establishing the necessary test cases at the same time. The result was that there was no running program for two weeks. Had they had the test cases, they could have first concentrated on restructuring and then evolving the interface, while always having a running system for the rest of the team.

The pairs building the graphical display were unable to provide fully automated test cases. They wrote the test cases (traffic scenarios) and watched the display for errors. To automate these tests would have required storing bit maps and comparing them, which seemed too much effort under the circumstances.

At the end of the course, the students were convinced of the benefits of writing test cases prior to coding. It is the testing approach that the students considered the best practice in the final review of the course. 87% stated that the execution of the test cases strengthened their confidence in the code and all of them were planning to try out this practice after the course, see table 4. All students saw jUnit as a suitable test framework.

Writing tests forces software engineers to distinguish between the functionality to implement and the base conditions under which the implementation has to work. These base conditions are specified and written down with XP in form of test cases. The conditions are verified every time the test cases are executed.

Re-executing test cases was uniformally seen as positive. Seeing the benefits of automatic regression tests increased participants' motivation to write executable test cases early. Participants also reported an increase in confidence in their software, but one participant also noted that regression testing can produce a false sense of confidence.

In summary, there are situations where test cases are difficult to automate (graphics) or are wasted effort (during prototyping). But writing test cases early, especially when there are no specifications, and regression testing were seen as beneficial.

4.4. Refactoring

The students never got to a point where they needed to refactor. One team had a complete design that did not need to be improved, the other team had a situation (the crossroads example in the preceding section) where one team sort of refactored a prototype, but without the benefit of test cases. Lack of refactoring may be caused by a combination of several factors: the small size of project and doing full rather than minimal designs.

4.5. Scaling

Within iteration planning, team members break down the requirements from the Planning Game into small pieces. Later, these pieces are processed in pairs. The division of the requirements requires that the team members agree on a common terminology. Otherwise, team members lose a lot of time. The communication problem is one limitation to the team size of XP because larger teams face much more communication overhead than smaller ones. This limitation is a bit relaxed if team members have worked together before.

During development (after iteration planning), communication needs are also high. Students initially thought pairs could work independently but quickly moved to team sessions. XP requires an ongoing information exchange. This intercommunication aspect is more crucial in the first stages of a new project, because the software is small and the pairs are likely to work on the same components. The small and emerging software also forces the team members to exchange vague and changing information [5]. This dependence on informal communication diminishes with project age but never disappears. The main strategy at the beginning of an XP project is to develop as fast as possible a small piece of running software which contains many small classes, as mentioned in [2]. With this software layout, the whole team can start to work on the project as early as possible.

In summary, team size is a crucial factor for XP. Small teams of not more than eight engineers have less communication overhead and are therefore more efficient than larger ones. Lorge et al [6] point out that small groups are more efficient working on abstract problems than larger ones. The authors expect the optimal group size of XP to be in the range of six to eight individuals, which is not much larger than the actual team sizes in the course.

There are two conclusions to draw for the next course. First, instructors have to insist about team meetings, because the informal information exchange can not be replaced by e-mail or other means of communication. And second, as a student project in a university course always suffers from a tremendous lack of time, it is preferable to provide a software skeleton at the beginning from which they can start development.

5. Conclusions and Open Questions

This paper presented experiences about XP with Computer Science graduate students. The course included three simple tasks and a project. All development work was done in pairs. Project teams consisted of six students (three pairs). After some initial difficulties, both teams adopted the XP methods.

The authors made the following observations.

1. Pair programming is adopted easily and an enjoyable way to code. However, it is unclear what type of work *not* to do in pairs and how best to structure pair interaction. Additional research is needed to compare the effectiveness of pair programming with reviewing techniques.

2. Design in small increments ("design with blinders") is difficult. Holistic design behavior may be difficult to abandon and more research is needed to test whether this is actually a good idea. If one wants developers to design in small increments, at least one pair member should be trained in it.

3. Writing test cases before coding is not easily adopted and is sometimes impractical. Is it really essential to write the test cases first and then the code, or is it possible to do it the other way around?

4. Due to the communication overhead, XP as is does not scale well. It is definitely meant for small teams (6-8 members).

5. XP requires coaching until it is fully adopted.

Are these conclusions generalizable to professional software developers? The subjects are certainly comparable to young professionals with an undergraduate degree in Computer Science. Whether more experienced developers are as willing as students to adopt a new process is questionable.

Observations 4 and 5 will likely hold in general. The effects of pair programming, small increments, and XP's testing practices are subject to future research.

6. Acknowledgments

We thank the students of the XP course: Daniel Hahn, Daniel Lindner, Gerd Flaig, Hanna Hakala, Jens Lukowski, Marcus Denker, Olaf Kleine, Paul Schmidt, Thomas Holl, Tobias Küfner, and Ulf Krum.

References

[1] K. Beck. Embracing change with extreme programming. *IEEE Computer*, pages 70–77, Oct. 1999.

[2] K. Beck. *Extreme Programming Explained.* Addison Wesley, 1999.

[3] D. Bisant and J. Lyle. A two-person inspection method to improve programming productivity. *IEEE Transactions on Software Engineering*, 15(10):1294–1304, Oct. 1989.

[4] A. Cockburn and L. Williams. The costs and benefits of pair programming. In *eXtreme Programming and Flexible Processes in Software Engineering, XP2000*, Cagliari, Italy, June 2000.

[5] R. Kraut and L. Streeter. Coordination in software development. *Communications of the ACM*, 38(3):69–81, Mar. 1995.

[6] I. Lorge, D. Fox, J. Davitz, and M. Brenner. A survey of studies contrasting the quality of group performance and individual permformance. *Psychological Bulletin*, 55(6):337–372, Nov. 1958.

[7] J. Nosek. The case for collaborative programming. *Communications of the ACM*, 41(3):105–108, Mar. 1998.

[8] L. Williams and R. Kessler. All I really need to know about pair programming I learned in kindergarten. *Communications of the ACM*, 43(5):108–114, May 2000.

[9] L. Williams, R. Kessler, W. Cunningham, and R. Jeffries. Strengthening the case for pair-programming. *IEEE Software*, pages 19–25, July/Aug. 2000.

[10] N. Wirth, Septmber 1999. personal communication.

Table 2. Description of the students and their experiences prior to the course.

Topic	Answers	
Experience in team work	no:	17%
	yes:	83%
Experience in pair programming	no:	58%
	yes:	42%
Working alone while pair programming	no:	0%
	yes:	92%
	no answer:	8%
Duration of individual work phases during pair programming	0-10%:	0%
	10-20%:	8%
	20-30%:	59%
	30-40%:	8%
	40-50%:	8%
	no answer:	17%
How was quality assured (several answers allowed)	design review:	25%
	code review:	58%
	debugging:	67%
	assertions:	25%
	testing:	67%
Refactoring done prior to the course	no:	0%
	yes:	100%
When was refactored (several answers allowed)	complex:	92%
	extension:	92%
	duplicate code:	25%
	search for errors:	17%

Table 3. Evaluation of the questionnaire for the first three program assignments.

Topic	Answers	
Knowledge of problem domain	little:	4%
	average:	74%
	well:	22%
Who typed	partner:	13%
	both:	87%
Enjoyment of pair programming	bad:	17%
	average:	35%
	good:	48%
Support from partner in finding a solution	no:	4%
	yes:	96%
Longer discussions during pair programming	no:	9%
	yes:	91%

Table 4. Evaluation of the questionnaire after the course.

Topic	Answers	
Enjoyment of pair programming	bad: average: good:	13% 62% 25%
Accept a job in industry with pair programming	no: yes: don't know:	25% 62% 13%
Pair programming wastes time (several answers allowed)	design: solving difficult problem: new environment: code formatting: new programming techniques:	13% 25% 50% 63% 0%
Task assignment while pair programming (several answers allowed)	typing and review: typing and reading doc:	87% 62%
Two displays better than one?	no: yes:	25% 75%
Advantages of pair programming (several answers allowed)	fewer errors: better problem understanding: more confidence in solution: learning from partner:	37% 37% 75% 100%
Requirements for pair partners (several answers allowed)	friendly: merry: communicative: unhurried: others:	100% 25% 62% 0% competent same knowledge
Specification with test cases prior to the course	no: yes:	75% 25%
Did the implementation only meet the specifications of the test cases ...	no: yes:	37% 63%
... or verified the test cases only the most necessary parts	no: yes:	50% 50%
Strengthened test cases confidence in program	no: yes:	13% 87%
Trying out XP's testing practice after the course	no: yes:	0% 100%
jUnit well suited	no: yes:	0% 100%