eXtreme Programming at Universities – An Educational Perspective

Jean-Guy Schneider, Lorraine Johnston
Swinburne University of Technology
School of Information Technology
P.O. Box 218, Hawthorn, Victoria 3122, Australia
{jschneider,ljohnston}@swin.edu.au

Abstract

To address the problems of traditional software development, recent years have shown the introduction of more light-weight or "agile" development processes (eXtreme Programming being the most prominent one). These processes are intended to support early and quick production of working code by structuring the development into small release cycles and focus on continual interaction between developers and customers. As such software development processes become more popular, there is a growing demand from industry to introduce agile development practices in tertiary education.
This is not a straightforward task as the corresponding practices may run counter to educational goals or may not be adjusted easily to a learning environment. In this paper, we discuss some of these issues and reflect on the problems of teaching agile processes in tertiary education.

1. Introduction

Imagine translating a book from Chinese into English, and the book changes overnight, the paper and pencil change while you are writing. The accepted practices involved in Software Engineering (SE) seem to change almost as fast as this. Therefore we have tended to place too much reliance on traditional process models to "explain" to students the best way to engineer software. However, experience has shown that education in "traditional" software development practices, such as the water-fall approach, will not always endow students with the appropriate knowledge and understanding for the workplace.

We need therefore to question whether it might be appropriate to teach our students about more light-weight processes. From an academic point of view, we have to ask:

- Should we introduce the practices of agile development processes in tertiary education?
- If so, how should they be introduced to avoid contradicting our educational goals?

This paper considers issues with using eXtreme Programming to educate students in Software Engineering.

The rest of this paper is organized as follows: in Section 2, we briefly introduce the main principles and practices of eXtreme Programming. In Section 3, we further outline the background of our study and define educational objectives for Software Engineering courses at a tertiary institution. In Section 4, we evaluate the practices of eXtreme Programming with regards to the previously defined educational objectives. Based on these observations, we come up with a few recommendations for software engineering curricula in Section 5. We conclude this paper in Section 6 with a summary of the main observations and suggest issues for further investigation.

2. eXtreme Programming in a Nutshell

eXtreme Programming is a so-called *agile* software development methodology [4], conceived and developed to address the specific needs of software development conducted by small teams in the face of vague and changing requirements [1, 3, 8]. It is a *light-weight* methodology combining a set of existing software development practices in such a way that developers are freed from "unnecessary work" (e.g. extensive documentation) and can concentrate on the main purpose of software development: writing quality code. This is mainly achieved by a highly iterative and incremental process starting with a simple design that meets an initial set of requirements and is constantly evolved to add needed flexibility, removing unneeded complexity. In essence, it attempts to produce the "simplest possible solution" fulfilling current requirements. Furthermore, eXtreme Programming assumes there is always a customer on-site, enforces *pair programming* [1], and promotes automated testing as well as continuous integration.

0-7695-1877-X/03 $17.00 © 2003 IEEE

3. Educational Objectives

Software Engineering aims to support the building of software on time and within budget. While definitions of "Software Engineering" abound, there is a general consensus that it deals with the application of engineering processes to the development of large-scale systems. In such a development, typically no one person can carry all the details of the system in their heads, teamwork becomes a hallmark, and process issues need careful consideration.

In building an aircraft, one can specify the properties of the required product, design a solution, and test a small scale model in a wind tunnel to establish the suitability of some of the properties of the final system. In principle, this is simple for a tangible product like an aircraft. The very fact that software is intangible means that the only evidence of development is in the artifacts that are produced—code, specifications, reports, etc.

For software systems, the complexity grows with size, and alternative development techniques become necessary. The concept of scalability is often one that escapes young aficionados of software development. Similarly, it is a somewhat foreign idea that a complex system does not lend itself to a haphazard approach for development, and requires good management. Additionally, time and resource constraints usually limit the scope of the systems on which students can work. Therefore, the Software Engineering curricula must be carefully crafted so that students can learn about large-scale problems and solutions during the course of their study, in a situation where there is little opportunity to expose them to sufficiently large and complex systems.

3.1. Focusing the educational aims

The IEEE-CS and the ACM have jointly proposed a body of software engineering knowledge (SEEK) [13], as an appropriate basis for the education of software engineers. Each of the defined topic areas in SEEK is addressed briefly here, in terms of its relevance for young software engineers.

Fundamentals. The underpinnings of Software Engineering lie in Mathematics and in the foundations of Computing and Engineering. This includes the ability to deal with abstraction and to understand the concept of models. Most students would have little difficulty comprehending the reason for including these topics in an SE program.

Professional Practice. While one could debate exactly what "Professional Practice" should encompass, it is fair to say that students need to learn about the ethical and social aspects of their profession, as well as more business-oriented aspects such as contracting models.

Software Requirements. Empirically, we know that an inadequate consideration of the perspectives of all stakeholders leads to rework later on. Hence, it is important to consider all relevant viewpoints up front, including the perspectives of both the end-user and the client. Students frequently need reminding of the importance of these issues.

Software Construction. Students tend to regard this topic as the key part of software engineering endeavours.

Software Design. Small systems can usually be modified more easily than complex systems. Students need to understand at least the need for design documentation to communicate the design, and the need for capturing design rationale. Beyond that, there is an urgent need for more instruction in architecture and related issues such as component-based systems and the use of patterns. An area often not discussed is that of interaction design, and the resulting user interface. Omitting this from curricula promises to produce a generation of software engineers who do not understand that functionality is not everything in a software product. Given basic functionality, the marketplace differentiates on quality attributes such as task efficiency, learnability, ease of use, aesthetics, and engagement.

Software Verification and Validation. Without extensive exposure to larger systems and to the theory of testing, the perception of students is that verification and validation equate to testing, or even debugging. However, students need to learn about traditional approaches such as white- and black-box testing, test planning, and testing to see if the desired non-functional properties have been met. It is also important that new engineers develop skills with and understanding of, not only testing, but also other verification and validation techniques such as inspections.

Software Evolution. Most newly-graduated software engineers are likely to find themselves employed, not on developing new systems, but on maintaining existing systems. Therefore, they need analytical skills enabling them to identify where and what changes are needed. For this, they need a good understanding of the architecture of the system, as well as clear ideas on the role of regression testing.

Software Engineering Process. The concept of "Process" goes hand-in-hand with that of management. Without a strong framework of required activities, students tend to espouse, and indeed use, *ad hoc* work practices.

Software Quality. Quality is said to be noted in its absence, rather than in its presence. Unfortunately, it is often then too late to fix the problem. Therefore, students need a solid understanding of the techniques and methods that can help produce a quality product. However, there is the problem that the techniques for ensuring quality in a large-scale product often seem to be an overkill for the size of products students confront. It may sometimes be necessary to impose processes which are not entirely suitable, purely from the perspective of exposing students to large-scale techniques.

Software Engineering Management. Many undergraduate students have little experience of managing anything. Some are of the age and mind-set to rebel against authority, and it is difficult for them to suddenly have to consider how a project is managed. Additionally, they often have little experience in a workplace and are unaware of expected behavioural protocols. This can have implications for how they interact and organise a team-based project. Richardson [12] observed that teaching SE to undergraduates was a difficult proposition owing to (i) their lack of computing experience, (ii) their lack of academic maturity to deal with ambiguity and conflicts between reality and theory, and (iii) their underdeveloped interpersonal skills. Young people usually have very good memories, and trust in their own ability to recall information. In this context, it is often difficult to convince students about the value, let alone necessity, of software configuration management. In class, students tend to work on "toy" examples, and configuration management seems simply an unnecessary overhead.

Like any other engineering discipline, to be efficient in a production environment, tools as well as specific techniques and methods are needed. Students need motivation to learn to use the tools, but also need to understand both the rationale for their use and the principles on which they are built. It is good for them to be exposed to specific tools, but it is also important that they realise the range available. This becomes an issue of education for understanding and application of knowledge versus training to use a specific product, the latter being a non-extendable position.

3.2. Putting it together in a project

Isolated topics in software engineering only provide the basis on which to build an understanding of the development of large-scale systems. The real learning takes place when the topics are not seen as isolated entities, but are put together in a real project. That is, when students are required to interact with a real user to discover the client's real problems, the pieces start to fall into place. For example, why should one trace where the requirement originated? Or, later, why should a design decision be documented? As any parent knows, it is in the experience itself, that the best learning takes place. Most undergraduate software engineering programs have a final project to provide the capstone of the studies. In the discussion that follows, the context is that of a team-based, final year project.

4. eXtreme Programming and Education

As mentioned previously, eXtreme Programming (XP) has been conceived for software development conducted by small teams of a maximum of 12 to 15 team members [1]. As this is also the maximum number of students we would consider as useful for final year software engineering projects, using XP as the development process would certainly be a feasible approach.

Recent work has shown that despite its success, XP does not address some important issues of software development [17] or should not be used for larger-scale and/or safety-critical systems [4]. Although further investigation of the (non-)applicability of XP for real-world software projects is certainly an important issue, this is beyond the scope of this paper. In this section, the focus is on evaluating some of the practices used in XP with regards to the previously defined educational objectives (see Section 3), with a special focus on two-semester software projects.

eXtreme Metaphor. In [1] Beck says that XP is principally nothing new, but "takes a set of common sense principles and practices to extreme levels." Most of these principles and practices are fairly well-known, were part of other software development methodologies, and have been taught in the context of a variety of courses before.

Unfortunately, some of these principles have previously not reached the required level of acceptance with students, e.g. the importance of testing (i.e. writing test cases *before* writing code), the focus on producing *quality* software in the first place, and the importance of coding standards and configuration management. We noticed there is definitely a better acceptance and understanding of these practices when they are introduced in an XP-like context. The *eXtreme* metaphor is the main contributing factor here as practices previously considered as being tedious become trendy.

Pair Programming. Probably the most extensively investigated practice of XP is pair programming. Several encouraging experience reports have been published in both industrial and academic settings (e.g. see [1, 5, 7, 11, 18]). In particular, there is empirical evidence that a pair of developers working on one computer can produce better quality software faster than the pair working separately.

However, we argue that this observation only holds if pair programming is done in the "right" environment with appropriately trained people. XP assumes all developers have reasonable programming skills, are capable of working in pairs, and are willing to do so.

One of the authors adopted pair programming some years ago in a large class of first year computing students (app. 500 students). The goal was to improve programming skills as it was thought students would help each other learn by discussion. Inevitably, the stronger one of the pair did most of the work, the weaker one usually failed the later examination, and only where there were well-matched pairs was the goal reached. We also noted different personality types and/or genders had a significant impact on pair programming performance. Unfortunately, there is little evidence that researchers have considered this yet.

Educational systems seem to be primarily based on a scheme where good marks are most highly rewarded, and many students are uncomfortable leaving this framework. In an XP environment, however, an ego-less attitude is required as people have to favour team success over personal rewards and assist other team members improve their working skills. These issues require a level of maturity to accept.

Process and Discipline. Agile methodologies are considered to be light-weight and impose less process burden upon the developers, which are some of the reasons why they have become so popular. However, the practices of any agile methodology must be applied in a disciplined way for projects to succeed. The relevance of discipline in XP is captured by Smith [15]: "XP is not a free form, anything goes discipline—it focuses [...] on a particular aspect of software development and a way of delivering value, and is quite prescriptive about the way this is to be achieved."

It is in the nature of young university students that they are less likely to enjoy prescriptive development methods and prefer approaches which leave them more freedom to explore. Furthermore, experience has shown that they have the tendency to struggle if they have to work in a disciplined way over a longer period. The combination of these two observations leads to a situation quite accurately described by one of our colleagues as "students dislike process, but they like discipline even less." Hence there is a big risk that a light-weight development methodology such as XP cannot be successfully applied in an educational environment.

We therefore argue that a less flexible development process with more guidance is better suited to the educational environment than a process based on "less process, more discipline." The latter should only be applied if all participants have the necessary experience and maturity, and fully understand the consequences of choosing a light-weight development process. This was best summarized by another colleague: "XP is for professionals, not students."

Distribution and Availability. One of the main practices of XP (and most other agile development processes) is that the development team is co-located, preferably even in a single office. The physical arrangement of the office needs to facilitate, or even encourage, brainstorming, joint discussions, pair programming, etc. [1, pp. 78]. Furthermore, it is assumed all developers are on-site most of their work time.

Unfortunately, such an environment is fanciful in most universities. More often than not, laboratory space is a limited resource, and as many machines as possible are put into one room, making them very crowded and noisy. Joint working is therefore awkward and, unsurprisingly, students prefer working at home. Furthermore, as most students have part-time jobs and their timetables are relatively disjoint, it is often difficult to find suitable times for scheduling joint working sessions outside timetabled classes at all.

Hence, student software projects are probably more aligned with distributed projects than co-located ones. Thus, one of the main prerequisites for applying XP is not fulfilled and the benefits of a joint working environment are lost, in particular the *warm communication paths* [4]. As a result, stricter rules for documenting project activities have to be put in place and the whole process immediately becomes more heavy-weight. There are efforts to adapt XP practices to distributed work environments (e.g. a recent article by Kircher et al. [9]), but the necessary changes and the resulting consequences are not yet fully understood.

At our institution, we happily acquired a laboratory with a small number of workstations especially set up for the needs of two project teams. Interestingly, one team decided to do most of their design and coding there and scheduled their joint working sessions based on their respective timetables (mostly late afternoons or evenings). The other team's members, however, used the room infrequently, preferred to work at home, and for collaboration relied on phone calls or internet technology (instant messaging, email, online chat, etc.). They occasionally organized coding sessions at one of the students' homes. Encouragement of the second team to use the laboratory more seems to have failed because of ignorance of the benefits of joint working sessions. They reverted to "the way we have been working throughout the previous years of our studies."

On-site Client. A similar problem arises with regards to having an on-site customer. Considering that this is often wishful thinking even in industry projects, with students' timetables it is probably a fair assumption that being able to contact the client by email or phone during working hours is as "ideal" a situation as we can hope for.

We have noted on several projects that clients do not fully understand the benefits of regular developer-client interactions. They do not want to be "bothered" by giving feedback to all team members, and they often ask for a liaison person for interaction. Also, students are loathe to be continually contacting the client: "we do not want to bother our clients with small or trivial issues all the time."

As a result, students often spend quite some time in getting certain aspects of the system "perfect" and understandably get frustrated when they receive negative feedback on what they thought was completed work. The issue of not having feedback when needed is magnified by the problem of students working at home, alone and/or at night. Being able to talk to the client earlier and more freely would certainly help to rectify these kinds of problems.

Development Cycles. XP is a highly iterative and incremental development process with iterations taking one to two weeks and a new release coming out every month or two. In fact, short development cycles are crucial for successful projects [1, 3]. Each iteration is supposed to add

functionality of a certain value to the customer. To achieve these kinds of development, XP assumes that developers can work exclusively on the project for 40 hours a week.

In an educational setting, we do not have the luxury of having students exclusively available for 40 hours project work each week; an average of 10 to 15 hours per week is much more likely. To gain the same amount of value added per iteration or release as in an ideal setting, we end up with iterations of four to six weeks and release cycles of six to eight months! Considering that student projects generally run over a period of one to two semesters, this is neither a practical nor an acceptable option. To keep the cycles short(er), one must reduce the scope of each cycle. However, it is doubtful whether any significant value can be added to the software with these kinds of cycles, and we end up with a situation which is not really acceptable, either.

Project Deliverables. The authors of the Agile Software Development Manifesto [2] claim that "running software is considered to be the most important deliverable" of a software project and "working software is the primary measure of progress." There are very good reasons why this applies to certain types of projects (and possibly not to others), but we must question the relevance in an educational context.

There is no doubt that having running software at some stage in a project is desirable and reflects progress. Additionally, it provides both satisfaction that meaningful work has been done and the necessary motivation to keep working. However, the lessons learned during a software project are at least as important as running software and must be considered as a kind of deliverable, too. Unfortunately, the desired learning can often only be achieved by making mistakes, analyzing the resulting problems, and rectifying them. This takes time, and during this period, there is no progress which can be measured in terms of working code.

Furthermore, we believe that code being the main deliverable sends the wrong message to students. If we consider the cooperative game principle of Cockburn [4, pp. 31], then code only addresses the first goal of the software development game (i.e. deliver useful, working software), but neglects the second goal (i.e. prepare for the next game). There are many kinds of (larger-scale) projects where more than just running code is required, e.g., re-engineering projects where extracting and documenting the design of an existing legacy system is a necessary step for any forward-engineering activity [6].

Finally, one cannot equitably assess a team of developers (either individually or as a group) on running software alone, as issues such as personal initiative, dedication to quality work etc. are not judged to the required level of accuracy. Other "products" of the development process are needed. The reader should note this does not only apply to educational contexts, but also to situations in industry where promotions are based on any such assessment schemes.

Perception. Students without the required level of maturity to fully understand what is important in software development and the consequences of certain practices, can be mislead by agile development practices as they only tend to see "what can be omitted." As an example, we organized a complementary lecture on agile software development (given by an expert in the field) to the final year software engineering students halfway through semester one of their two-semester project. In the following team meeting with one of the teams, a student commented that "we should have chosen XP as our development process; then it would not have been necessary to write a requirements specification." As it turned out, misconceptions in the requirements resulted in that team running into trouble in semester two.

It can be said that development processes such as XP certainly have their benefits and can have a positive impact in an educational context. However, there are important educational goals which are not met and, therefore, changes to the process have to be made in order to address these goals.

5. Recommendations

SE education aims to produce software engineers with the knowledge and skills to adapt their practices in an ever-changing environment. This is distinct from training them as experts in the techniques and processes which are currently most popular and being touted as "the way to go." While XP has many positive aspects, many of its tenets are not compatible with the real environment of most students.

Some of the difficulties lie with the prior educational experiences of students. There are problems when the student mindset prioritizes outcomes in the order (i) a pass in a course, (ii) a good grade in a course, (iii) the learning experience, and (iv) helping another student. A re-engineering of this mindset is necessary before the practices of XP can work well. The question of how to refocus the student experience is being considered by many educationalists, but clearly an early focus on teamwork and cooperation is important for SE studies. Instilling the concept of "ego-less working" at an early stage is a key element, the introduction of learning communities [16] another.

Location and scheduling problems are difficult to overcome in an academic environment. Leverage may possibly be found with additional research into distributed lightweight methods, and their application to educational situations. Currently the co-location issues are difficult to solve.

One way to present a large-scale project to students is to offer a team-based maintenance project, where extensions are needed. This has some overlap with the idea of the Real World Laboratory at Georgia Institute of Technology [14]. If the base system is sufficiently large and complex, many of the large-scale objectives can be met. Students must view

the system at various levels of abstraction, and identify the location for change from the existing system artifacts. They must design and implement the changes, performing all the relevant levels of testing, and "plan" for future changes. Where there is little or no existing documentation, apart from in the code itself, students become very aware of how much planning for change went into the original product. They will more likely agree that running software alone should not be the criterion of a well-engineered product.

In academia, it is appropriate to place the primary emphasis on process, rather than product, which does not always delight the management-averse students. If the goal is to have students learn techniques suitable for large-scale development, then we must simulate at least the development of a complex system. It may be necessary to impose on their project work a regime which otherwise would not be the development process of choice for that system.

Rather than learn to apply only XP techniques, it is preferable that students are exposed to a set of techniques that help them manage a variety of projects (from small to large-scale), introduced the ideas in the context of the positive and negative issues associated with any process model, and given guidelines how to select a process model for the particular situation. The Crystal methodologies [4] is a process-kit which could be used for this purpose, the Rational Unified Process [10] another.

Education includes giving students a set of tools to craft a solution. Part of the collection of knowledge and skills needed by new engineers is the ability to tailor a process to suit their needs. Teaching students how to tailor their processes is likely to have a more beneficial effect than seeing a light-weight development process such as XP as the answer.

6. Conclusions

Agile development processes like XP are increasingly popular in industry. However, in tertiary institutions, there is little teaching experience of agile processes beyond pair programming. Further evaluation is needed to gain necessary insights. As discussed, some practices of XP can be used in a learning environment, others negate educational goals. Curriculum changes are needed so that students embrace a less self-centred philosophy than is currently the case. Consequently, we believe that eXtreme Programming as it stands does not lend itself for use in tertiary education.

References

[1] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[2] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas. Manifesto for Agile Software Development. Available at http://agilemanifesto.org/.

[3] K. Beck, M. Fowler, and J. Kohnke. *Planning Extreme Programming*. The XP Series. Addison-Wesley, 2000.

[4] A. Cockburn. *Agile Software Development*. Addison-Wesley, 2001.

[5] A. Cockburn and L. Williams. The Cost and Benefits of Pair Programming. In G. Succi and M. Marchesi, editors, *Extreme Programming Examined*, The XP Series. Addison-Wesley, May 2001.

[6] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann/dPunkt, 2002.

[7] M. Holcombe, M. Gheorghe, and F. Macias. Teaching XP for Real: some initial observations and plans. In *Proceedings of the Second International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001)*, pages 14–17, Cagliari, Italy, May 2001.

[8] R. Jeffries, A. Anderson, and C. Hendrickson. *Extreme Programming Installed*. The XP Series. Addison-Wesley, 2000.

[9] M. Kircher, P. Jain, A. Corsaro, and D. Levine. Distributed eXtreme Programming. In *Proceedings of the Second International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001)*, pages 66–71, Cagliari, Italy, May 2001.

[10] P. Kruchten. *The Rational Unified Process: An Introduction*. Object Technology Series. Addison-Wesley, Second edition, 2000.

[11] M. M. Müller and W. F. Tichy. Case study: Extreme programming in a university environment. In H. A. Müller, editor, *Proceedings ICSE 2001*, pages 537–544, Toronto, Canada, May 2001. IEEE Computer Society.

[12] W. E. Richardson. Undergraduate Software Engineering Education. In G. A. Ford, editor, *Proceedings of the Second SEI Conference on Software Engineering Education*, LNCS 327, pages 121–144, Fairfax, Virginia, Apr. 1988. Springer.

[13] Software Engineering Education Knowledge (SEEK). Second Draft, available at http://sites.computer.org/ccse/, Dec. 2002.

[14] D. M. Smith. Real World Laboratory. 3 semester project at Georgia Institute of Technology, refer to http://www.cc.gatech.edu/classes/RWL/Web.

[15] J. Smith. A Comparison of RUP and XP. White Paper, Rational Software Corporation, 2001.

[16] V. Tinto. Colleges as Communities: Taking Research on Student Persistence Seriously. *The Review of Higher Education*, 21(2):167–177, 1998.

[17] D. Turk, R. France, and B. Rumpe. Limitations of Agile Software Processes. In *Proceedings of the Third International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2002)*, pages 43–46, Alghero, Italy, May 2002.

[18] L. A. Williams and R. R. Kessler. The Effects of "Pair-Pressure" and "Pair-Learning" on Software Engineering Education. In S. A. Mengel and P. J. Knoke, editors, *Proceedings of the Thirteenth Conference on Software Engineering Education & Training*, pages 59–65. IEEE Computer Society, Mar. 2000.