eXtreme Programming—Helpful or Harmful in Educating Undergraduates?

Jean-Guy Schneider and Lorraine Johnston

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

Abstract

Criticism is sometimes leveled at the academic Software Engineering community on the basis that current educational practices are too document-centric. Both students and practitioners have suggested that one of the popular, lighter-weight, agile methods would be a better choice. This paper examines the educational goals for undergraduate Software Engineering education and considers how they might be met by the practices of eXtreme Programming. Our judgment is that education about some agile practices could be beneficial for small-scale development. However, as it stands now, eXtreme Programming as a package does not lend itself for use in educating about large-scale system development in tertiary education.

Key words: eXtreme Programming, Agile Methodologies, Tertiary Education, Software Engineering Education

1 Introduction

Despite the fact that Software Engineering is a discipline that is clearly maturing with great achievements on projects of significant complexity, the underlying practices still seem to change rapidly. The situation can be compared with translating a book from one language to another where the book changes overnight and the paper and pencil change while you are writing. Many tertiary institutions have tended to place overly 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 waterfall approach, will not always endow students with the appropriate knowledge and understanding for the workplace.
But what are the alternatives? Can we make Software Engineering education more practice-oriented and address the misconception that Software Engineering is mainly concerned with writing one document after another? What are the trends in industry which could guide us in making the right decisions?

In recent years, so-called agile development processes [6] have become increasingly popular. These processes are intended to support early and quick production of working code by structuring the development into small release cycles and focusing on continual interaction between developers and customers. By considering working code as the primary focus of any development activities, they have become very fashionable for those people who conceive Software Engineering as a very document centric activity. Therefore, there is an increasing groundswell from both students and industry for incorporating agile methodologies into the Software Engineering curriculum.

eXtreme Programming (XP) is probably seen as the most prominent member of the family of agile development methodologies [1,13]. Originally defined to address the specific needs of software development conducted by small teams in the face of vague and changing requirements, it has also attracted interest in academia as an alternative to teaching “traditional” software development practices [11,12,17,26]. Although these reports are mainly positive, they fail to address one important issue: what are the educational goals the participating institution wants to meet introducing XP? From our perspective, this is one of the key issues. For this work, we have therefore taken the approach of defining a list of educational objectives first before setting the practices of eXtreme Programming in relation to these goals.

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

2 Educational Objectives

The goal of Software Engineering is to support the building of quality software on time and within budget. While there are a multiplicity of definitions of “Software Engineering”, there is a general consensus that it deals with the
application of engineering processes to the development of large-scale systems. In such complex development projects it would be unusual for any one person to be able to carry all the details of the system in their heads. Therefore good teamwork is a necessity, and the issues of resources and risk must be considered carefully. That is, the actual process used becomes critical.

In other engineering disciplines it is possible to specify the properties of the required artefact and then prototype a scale model to see if the desired properties will be met. For example, models of aircraft are tested in wind tunnels and models of ships can be tested for buoyancy and balance against wave actions—in principle a straightforward activity. In the built environment, scale models can be constructed to check whether the design is appropriate, and scale models of furniture can be put into a model of a building to see if enough space has been provided. It would be convenient if software could be prototyped and tested similarly. However in software development prototyping is used to check the feasibility of achieving a particular product attribute such as performance, but not the overall properties with a view to scaling up.

Software is not tangible in the sense that other engineered products are, and modeling with the current formal software techniques does not lend itself to being useful for more than critical sections of code. In the more usual, less formal types of software development the only evidence of progress is in the artefacts—code, specifications, reports, etc. In software systems, the complexity grows with size, and alternative development techniques must be sought.

Scalability and the necessity for careful management in software development are concepts which undergraduate students commonly have difficulty grasping. Student experience is usually filled with small projects where they understand all the ramifications of design decisions, and where there is no need to formalise their process very much at all.

The problem is compounded by the fact that time and resource constraints usually limit the scope of the systems on which students can work. Therefore, Software Engineering curricula need to be carefully crafted so that students learn about large-scale problems and solutions during the course of their study, in a situation where there is limited opportunity to expose them to sufficiently large and complex systems.

2.1 Focusing the educational aims

The ACM and the IEEE-CS have jointly proposed a body of Software Engineering knowledge (SEEK)[19] to be an appropriate basis for the education of software engineers. Each of the defined topic areas in SEEK is addressed briefly in the following paragraphs, in terms of its relevance for educating
young software engineers.

**Fundamentals** The underpinnings of Software Engineering (SE) lie in mathematics and in the foundations of computing and engineering. An important part of this includes the ability to deal with abstraction and to understand the concept of models. Further, knowledge of computing theory provides a pathway for updating knowledge later as paradigms change. Most SE curricula provide a thorough grounding in these aspects, and few would question the value of the inclusion of such material as part of a SE curriculum. Without a knowledge of typical engineering practices such as requirements specification and risk management, software engineers run the risk of not being able to communicate “on the job” with other engineers. In large complex system development it is common for other types of engineers to be involved, and it is important that software engineers can communicate with them with understanding and a common vocabulary.

**Professional Practice** While one could debate exactly what “Professional Practice” should encompass, it is fair to say that students need to learn about the environment in which they will be working. Issues of relevance to them include the ethical and social aspects of their profession, and more business-oriented aspects such as contracting models.

The Code of Ethics drawn up by the joint ACM/IEEE committee very clearly enunciates the expected ethical behaviour of a software engineer [10]. The social aspects for any particular cohort of undergraduates are likely to vary a little, depending on the cultural environment. However it is important for new software engineers to understand how a workplace functions, so they can readily become contributing professionals. This could even include issues such as the protocol expected in sending email, or in requesting input from colleagues in an organisation. The authors observe that one particular cohort of overseas students tends to send email messages entirely devoid of punctuation and upper case letters, thus causing comprehension difficulty for the reader. Knowledge of contracting models allows the software engineer to understand some of the flow-on effects of poor estimation, and of how it is usually in no one’s interest to have fixed-price contracts.

**Software Requirements** In practice it is commonly found that an inadequate consideration of the perspectives of all stakeholders often leads either to rework later on or to outright failure of the product. Hence, it is important to engage all relevant stakeholders up front, including both the end-user and the client as well as representatives of any others likely to be affected by the software.
In an environment where a common aim is for students to produce code that reinforces the learning of an aspect of computing such as the efficiency of an algorithm, students are used to considering themselves the client and of being sure they know what is needed. However, in industrial usability observations it is commonly reported that end users have been mystified by user interfaces that the designer considered to be “obvious”. The key to producing a suitable solution is knowing what the real requirements are for any project—and this includes understanding the tasks the user needs to accomplish.

Being able to analyse and model these requirements are further issues. Complex systems present opportunities for different stakeholders to have very different perspectives on the required product. Negotiating a suitable set of requirements and developing a conceptual model are valuable experiences for students. Further, it is imperative that they also learn about the importance of managing both the changing requirements and the change process itself during development.

Software Design The topic of design often receives too little attention in the education of software engineers. Small-scale design approaches are usually picked up when students are taught to program, but these approaches seldom scale up. Most of our educational programs for software engineers would be improved by the inclusion of further material on software architectures. As in other engineering disciplines, software engineers need to be able to capitalise on previous successful solutions and reuse them. Therefore students need to have their horizons extended into the areas of patterns and frameworks, as well as into relevant tools and techniques, so that they are in a position to make informed decisions.

Software Construction From the student perspective, this is what software is all about—coding. Such an emphasis seems to be put on the generation of code to meet the requirements of assignments, that all other related activities appear pale in comparison. The usual reaction is to start coding immediately. Therefore it is necessary also to teach students how to be able to guarantee that their solution does, indeed, meet the requirements. Educators thus have the responsibility to teach students how to design a solution to the problem before coding, and to plan for the testing of that solution.

Software Verification and Validation In industry, it is common to hear discussion about “V&V” activities. However, in student circles, “V&V” usually equates to testing. While it is important to instruct students in such topics as basis paths and equivalence partitioning, if they think that testing is the answer to all software development problems, they do not understand the big picture. For large-scale systems there is no possibility that they can be tested
to completion and we need to resort to other techniques to guarantee the suitability of our products. Empirically, Basili and others [20] have shown that using inspections and reviews to detect defects in software can improve productivity considerably, as rework is diminished. Perspective-based reading can be very instructive for students to see how many problems can be identified if they merely change their viewpoint in reading a requirements specification.

**Software Evolution** In time past, there was a general acceptance that getting a requirements specification “right” was the passport to a straightforward development. No more would rework be required, as the requirements would be fully understood, and therefore fixed. Latterly we have realised the naivety of this, and come to understand that there will always be changes required for any large-scale system. On one hand, the environment of complex systems is highly likely to change. Newer technologies offer faster processing, legislation changes mean that information systems must be updated, or old hardware must be upgraded. On the other hand, Carroll et al. [5] identified the task-artifact cycle where the existence of an artifact, in this case some software system, provides users with an idea of using it for an activity that was never dreamt of in the original specification, thus requiring alterations to the system. As the old saying goes, “There is nothing as constant as change.” Changes therefore mean that no software engineer can avoid being involved at some stage in an activity related to maintenance, reengineering, or reuse.

**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. While such practices are adequate for relatively simple programming exercises, they are not sufficient to support development of complex systems where considerations of the size of the team impose constraints not otherwise needed, such as leaving trails of rationales to help later developers.

**Software Quality** Quality is often said to be noted in its absence, rather than in its presence. However, having the ability to note an absence of quality does not mean that the situation can be improved. It may be that by this stage, the most cost-effective solution is a work-around, with a subsequent impact on maintainability at the very least. Therefore, students need a solid understanding of the techniques and methods that can help produce a quality product. 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 projects students confront. It may sometimes be necessary, therefore, to impose processes which are not entirely suitable, purely from the perspective of exposing students to large-scale techniques.
Software Engineering Management  More often than not, undergraduate students come to their studies with little or no relevant experience of the workplace. Neither do they have a wide experience of the communication protocols expected there. This can have implications for how they interact and how they organise a team-based project. Richardson [18] observed that teaching SE to undergraduates was a difficult proposition owing to

• their lack of computing experience,
• their lack of academic maturity to deal with ambiguity and conflicts between reality and theory, and
• their underdeveloped interpersonal skills.

Further, 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. They simply have not had the experience of dealing with sufficiently complex information. Therefore a key part of their education resides in helping them learn about how to manage time, budget and resources to meet the appropriate quality goals for a project.

In many cases, academics regard the capstone project as an ideal opportunity for students to learn management skills for themselves. They can make their own decisions, within reason, about how they will schedule their activities, how they make design decisions, how they interact with each other, their clients and their instructors. The role of the instructor can become more limited—to one of making sure they do not stray too far from the path to an appropriate solution to the software problem.

3  eXtreme Programming – a brief Introduction

eXtreme Programming is a so-called agile software development methodology [6], conceived and developed to address the specific needs of software development conducted by small teams in the face of vague and changing requirements [1,3,13]. It is a light-weight methodology centred around 12 core development practices: Planning Game, Small Releases, Metaphor, Simple Design, Testing, Refactoring, Pair Programming, Collective Ownership, Continuous Integration, Fourty-hour Week, On-site Customer, and Coding Standards.

Most of these core development practices have been known for some time. The “novelty” of eXtreme Programming is that these practices are combined in such a way that developers can concentrate on writing quality code and are freed from “unnecessary work” (extensive documentation, heavy-weight quality assurance procedures etc.) [1]. It nominates programming as the key
activity throughout a software project and promotes activities not directly related to programming being conducted only on a “call-by-need” basis. This approach is motivated by empirical evidence that too many software projects fail; they do not deliver the required functionality or quality within the anticipated time and cost-frame.

The outcomes of eXtreme Programming are mainly achieved by a highly incremental and iterative development process which starts 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” by fulfilling current requirements and avoiding planning for future requirements as much as possible. Due to the highly incremental and iterative nature of the process and continuous refactoring of existing code, eXtreme Programming advocates that the cost of changing software does not rise exponentially as in traditional development processes. Hence, postponing the integration of requirements to a later stage becomes feasible [1].

One of the key requirements for an eXtreme Programming project to succeed is that there is a customer on-site who is not only in charge of formulating requirements or user stories, but can also assist the development team in testing activities and resolving ambiguities without unnecessarily delaying project progress. Other key features of eXtreme Programming are that it enforces the production of all software in pairs (i.e. Pair Programming), promotes writing tests before coding, and emphasizes continuous integration of all developed code.

Like many other light-weight development processes, eXtreme Programming can only be applied successfully if the working environment of a development team is set up in a way that facilitates and encourages collaborative work. Such an arrangement is needed to ensure an effective and efficient way of communicating amongst the members of the development team. To address this issue, Beck suggests co-locating the development team, preferably in a single project room, and having all developers on-site most of their work time [1, pp. 78]. The team must be allowed to rearrange the project room according to their needs if the previous arrangement did not facilitate their work practices. Hence, the working environment must be set up based on the needs of the development team, and not the other way round.

In essence, eXtreme Programming has to be considered as a package as the 12 practices are interlocked; it is not possible to omit any of the practices without affecting the effectiveness of the process. From a different perspective, the 12 practices can be seen as elements of a pattern language. A pattern on its own has only limited value; only in combination with other patterns is its true value revealed [4]. Therefore, other support mechanisms with the same
or similar effects need to be put in place if some of the practices of eXtreme Programming are not followed.

4  eXtreme Programming and Education

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 eXtreme Programming as the development process would certainly be a feasible approach. In this section, we will therefore set the practices of eXtreme Programming in relation to the educational objectives we defined in Section 2 and elaborate whether these objectives can be met. However, it is not our intention to investigate the (non-)applicability of XP for real-world software projects. Although this is certainly an important issue, it is beyond the scope of this paper (refer to [6] or [24] for further discussions about this topic).

As mentioned at the end of Section 3, the 12 practices of eXtreme Programming must be considered as a package. Therefore, we argue that it does not make sense to investigate individual practices in isolation, but instead analyse their impact from a “holistic” perspective.

4.1  eXtreme Metaphor

XP is principally nothing new, but “takes a set of common sense principles and practices to extreme levels” [1]. 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. There is anecdotal evidence that these practices are better understood and accepted when they are introduced in an XP-like context. We assume that the eXtreme metaphor is the main contributing factor here as practices previously considered as being tedious become “trendy.”

On a positive note, there are some concerns related to software development which have become more prominent in the context of eXtreme Programming rather than in traditional software development methodologies, and can be ex-
posed more easily to students. For example, XP promotes that out of the four development “variables” cost, time, quality and scope, only three can be chosen; the fourth variable is always a consequence of the others. The shorter the development cycles, the more important this issue becomes. Furthermore, extreme Programming promotes that software developers are valuable resources and should only work overtime on an occasional basis, something students tend to hear with delight.

4.2 Competence in Large-Scale System Development

In [6, pp. 162], Cockburn characterizes software projects by their level of criticality, their priorities and their respective communication load. Briefly summarized, he illustrates that:

- the more critical a project, the more care needs to be taken to get things right (a project that may cause the loss of life needs more careful checking than a project that is not safety-critical),
- projects which are prioritized with legal liability concerns require more care, in particular in relation to tracking of the work, and
- the more people you have on a project team, the bigger the communication overhead amongst the team members; hence coordination mechanisms suitable for small teams are no longer suitable for larger teams.

On the subject of eXtreme Programming, Cockburn argues that it is only applicable in a handful of project types: projects up to 15 people with at most medium criticality. From an educational perspective, these are precisely the types of projects we can expose our students to; it would probably be neither practical nor ethical to expose students to any projects which go beyond that range. Hence, at a first glance, using an XP approach seems to be quite a reasonable thing to do.

As mentioned in Section 2, being competent in a variety of development technologies, methods and programming languages suitable for large-scale systems is one of the primary concerns for any student graduating with a Software Engineering degree. Therefore, it is important that SE students are exposed to a number of development techniques during their studies and come to understand that

- there is no methodology which fits all projects; the methodology used in a project must match the characteristic of that project,
- some development techniques scale well within limits whereas others do not, and
- software development involves more than just programming; it has to be considered as an engineering activity.
As these three issues are so fundamental for Software Engineering, it is our belief that they should not only be taught in a class room situation, but also be emphasized in project situations. Applying only the practices of eXtreme Programming seems to concentrate on one aspect without the other.

4.3 Quality Assurance

Developing quality software is one of the major challenges for most software development projects. More often than not, a considerable amount of effort needs to be spent in putting appropriate quality assurance (QA) mechanisms in place to achieve the required quality goals. These mechanisms can range from simple program testing to extensive validation of requirements using formal approaches. From an educational perspective we ask two things: (i) how best to introduce QA mechanisms to student and (ii) whether eXtreme Programming can assist us in doing so.

Most of the 12 practices of eXtreme Programming (refer to Section 3) deal with quality assurance in one form or the other (coding standards, write-test-before-code and continuous integration are probably the more obvious ones). If we have a closer look, all quality assurance mechanisms of XP are very much code centric: they all focus on improving the quality of the source code. There is certainly nothing wrong with producing quality code, but XP does seem to care less about other issues, e.g., how to validate requirements (i.e. user stories) before they are implemented and how to trace requirements in both design and source code. It is mostly left up to the individual team to set up additional procedures to cover these aspects where necessary.

And that is probably where one of the main concerns lies: like many other light-weight process methodologies, eXtreme Programming works under the premise that a team has at least a few experienced developers who will “sense” when additional non-programming related activities (such as QA procedures) have to be introduced and, therefore, can take the corresponding issues into consideration before major problems eventuate. In the context of an educational environment, however, we rarely have the required experience within a student team; a team will always rely on “external” assistance, most probably in the form of a tutor or staff member who initiates additional procedures and will have to make sure that they are followed through. Hence, this runs counter to our educational goal that student teams manage their own projects, with limited supervision.

Refactoring is another practice where XP has a problem in an educational environment. Whereas an experienced developer notices when “code smells” [9], a developer with little experience (such as our students) simply has not
seen or written enough code to pick a code smell. Therefore, we cannot really expect students to be very proficient in refactoring code. This, however, is an issue of concern as an agile process like eXtreme Programming heavily relies on refactoring to keep the code base clean.

Having a user-centred process can help with achieving functionality and usability; others would argue that it should be a “usage”-centred design instead, as it is often the role of the user which is important, not the actual user. The software needs to support whoever happens to be performing the task in question. The XP “customer-on-site” approach could be considered to be a kind of degenerate user centred design, as there is little attempt to examine how the tasks are performed in the real world. Large-scale system development has a multiplicity of stakeholders, so one customer representative alone is hardly likely to provide all the views that need to be considered. It should also be noted that Pair Programming provides a form of review.

We would therefore argue that the core practices of eXtreme Programming do not cover the issues of quality assurance enough; other practices have to be introduced which are not immediately obvious in the context of an XP project, but would make more sense in a larger and/or safety-critical project. If these practices are omitted and the focus is solely on writing quality code, educators might run the risk of sending the wrong message to students that quality assurance is only a matter of writing good code.

4.4 Collaborative Working

One of the main lessons any software engineer must learn in his/her career is that software development is hardly ever a one-person activity, but is conducted in a team. Hence it is important that SE students are exposed to teamwork at various stages of their education, in both smaller and larger teams. During this time, they also need exposure to the various roles members of a larger team will perform, in particular organizing and coordinating the work to be performed.

There is no doubt that a development process like eXtreme Programming enforces teamwork. In fact, it should be considered as one of the core metaphors of the process model as XP cannot work properly without having a team whose members continuously collaborate during the development. Furthermore, XP heavily stresses that fact that there is one team at work and not a collection of loosely collaborating sub-teams. Hence, from a teamwork perspective, using an XP approach would certainly be very beneficial. But let us have a look at some issues in further detail.
**Pair Programming**  Probably the most extensively investigated practice of XP is Pair Programming. The basic premise of this practice is that a pair of developers working on one computer can produce better quality software faster than the pair working individually. Several experience reports have been published “confirming” the effectiveness of Pair Programming in both industrial [1,7] and academic settings [12,17,26].

eXtreme Programming makes the assumption that all members of the development team have at least reasonable programming skills, are capable of working in pairs, and are willing to do so. If considerable differences in programming skills exist within a team, XP promotes the idea of pairing an experienced programmer with a less skilled one and having the former act as a mentor and train the latter [1]. Furthermore, the team is encouraged to change the pairs frequently so that each member does not constantly pair up with the same partner. Given such a context, there is a high probability that Pair Programming can be effective and the level of imbalance within a team can be reduced. More studies need to be done with experienced practitioners to further justify this claim [7]. This raises the question of whether we can transpose such a context into a learning environment.

In a recent publication, McDowell et al. examined the effectiveness of Pair Programming in the context of a large introductory programming course [16]. They were particularly interested in assessing how Pair Programming affects student performance, confidence in produced solutions, and satisfaction with programming. Their results indicate that students working in pairs are more likely to succeed in the subject, have a greater confidence in their solutions, and take a greater enjoyment out of programming [16]. Their study also indicates that there are no statistically relevant gender differences.

In contrast, one of the authors of this paper was involved in the adoption of Pair Programming 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. Unfortunately, the stronger one of the pair did most of the work, the weaker one usually failed the following examination, and only where there were well-matched pairs was the goal reached. Similar observations were also made by the other author during laboratory sessions of an introductory programming course. It was also apparent that when two equally competent students paired up the student not using the keyboard sat back and thought about something other than the problem to be solved, defying the purpose of working in a pair. In both cases we further observed that different personality types seem to react differently to working in pairs. The resulting performance is likely to be impacted by this, but to date there is little evidence that researchers have considered this.
**Pair vs. Team** Most studies done in the area of Pair Programming have been conducted in the context of programming subjects with a particular focus on their respective programming skills [12,25,27]. However, software development is more than just programming and (at least to our knowledge) there are no experience reports about how students introduced to Pair Programming early on in their curriculum will perform in larger-scale development projects.

One of the main benefits of working in a pair is the fact that they are exposed to a collaborative working environment and are forced to communicate with a peer, generally resulting in better learning outcomes compared to individuals. But is it really the pair which makes the difference? Should we not compare situations where a number of students collaboratively work on the solution of a problem with individuals trying to do the same thing? In most situations, developing software will be an activity involving a team, and the team size will rarely be two. We therefore suggest that as educators, we should not restrict our students to work as pairs, but have them gain experience in various team sizes with changing team members. Only in this way will students experience what team work really involves and how problems in teamwork can be overcome.

**Student Attitude** Educational systems are 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 need to favour team success over personal rewards and assist other team members to improve their working skills. These issues require a level of maturity to accept, in particular as personal benefits from helping peers are not all that obvious to students. This issue is revisited in the following section.

### 4.5 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 (possibly not to others), but how does this compare with our educational goals?

There is no doubt that having running software at some stage in a project is desirable and reflects progress. In fact, the success or failure of most industrial projects is solely determined by the quality of the delivered software. Having developed running software provides both satisfaction that meaningful work has been done and the necessary motivation for a team to keep working. However, in an educational context, the lessons learned during a software project
are at least as important as running software and must therefore be considered as a kind of deliverable. Unfortunately, the desired learning can often only be achieved by making mistakes, analysing the resulting problems, and rectifying them. This takes time, and during this period, there is no progress which can be measured just in terms of working code.

We believe that code being the main deliverable sends the wrong message to students. If we consider the cooperative game principle of Cockburn [6, 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 various 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 [8].

Any software project “produces” various artifacts during its lifetime; not all of them are source code. As examples, consider project-specific guidelines on how to refactor a particular module or additional knowledge about a third-party library that is used. It is a generally accepted fact that such artifacts are very volatile if they are not documented in some form. Students must be made aware of this fact and, therefore, special attention needs to be paid to recording such artifacts and discussing their importance for the current or future projects.

On various occasions, we noticed that students do not have the required level of maturity to fully understand the consequences of certain practices and the purpose of some non-code deliverables; they can be mislead by agile development practices and only tend to see “what can be omitted”. For example, we organized a complementary lecture on agile software development (given by an experienced practitioner) 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.

In Section 4.4, we noted that in an XP-like project work must be done in an egoless fashion and the team objectives need to be the focus for all individuals. In such a situation, it is not always easy to identify individual achievements. However, the educational environment is one where it is the duty of staff at the institution to see that graduating students do indeed reach the required level of competency. Therefore, it is important that the contribution of individuals is identified and assessed. eXtreme Programming does not offer clear support for doing this. One cannot equitably assess a team of developers (either individually or as a group) on running software alone, as issues such as per-
sonal initiative, dedication to quality work etc. are not judged to the required level of accuracy. Other evidence from the development process is 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.

4.6 Client Interaction

It is a common educational goal to expose students to projects where they deal with real clients. Due to the fact that eXtreme Programming projects explicitly have a customer representative being a part of the development team, this goal can be met, probably more easily than in traditional “out-of-the-book” student projects. But how realistic is it to have an on-site customer for student projects? 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 student 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.” This problem is magnified in an XP context as the success of a project relies heavily on continual interaction between the developers and the client.

Without sufficient interaction with the client, 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 when students revert back to the all so familiar habit of 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.

From an educational point of view, it is probably only of secondary concern to have an on-site customer. Much more important is that students experience the value of feedback. As supervisors, we therefore need to be careful in selecting only projects whose clients clearly understand this and are able and willing to regularly provide the required feedback.
To conclude our discussion about eXtreme Programming in an educational environment, we address three issues which are not directly related to educational goals, but have to be dealt with if an agile methodology is considered for a Software Engineering project.

**40 Hour Week**  One of the main practices of XP 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 which becomes very crowded and noisy. Only under rare circumstances will students have the opportunity to make rearrangements based on their needs. 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* [6]. As a result, stricter rules for documenting project activities need 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. [14]), 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 in 2002. 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.”

**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 the success of such projects [1,3]. Each iteration is supposed to add functionality of a certain value to the customer. To achieve development cycles of this length, 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 or 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. This approach has been used by Hedin et al. in the context of a one semester team programming course [11]. Their project only ran over 7 weeks and only three versions could be released during this time. Interestingly, they restricted system development to one compulsory 8 hour lab session per team per week. To facilitate the progress in each project team, the subject conveners of the course developed the first release and, therefore, it is difficult to judge how appropriate their approach is from a practical perspective. Nevertheless, the resultant learning outcome seemed to be very positive.

Hence, we argue that there is a need to carefully analyse whether significant value can be added to software with cycles with considerably reduced scope and whether there is enough complexity in each cycle to emphasize the importance of adequate development strategies. At present, we do not have sufficient evidence to state any conclusions on these issues.

**Process and Discipline**  Agile development 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 [22]: “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 many undergraduate 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.” Given this context, it is questionable whether agile development processes can be applied successfully at all.

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.”

5 Recommendations

There is a temptation in some tertiary institutions to train students in current technologies so that on graduation they have a skill set which is immediately transferable to the needs of industry. However, having such a skill set does not necessarily equate to having the understanding and knowledge that will assist them in adapting to new technologies as they become available. This underlines the need for the category of “Fundamentals” in the defined body of knowledge. The authors believe that Software Engineering education should aim to produce software engineers with the knowledge, skills and capacity to adapt their practices in an ever-changing environment, not just engineers with the ability to be immediately productive. Therefore while XP has many positive aspects, many of its tenets are insufficient for the learning needs of most students.

Some of the difficulties lie with the prior educational experiences of students. There are problems when students prioritize their educational 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. It is necessary to re-engineer their attitudes before the practices of eXtreme Programming can work well. The question of how to refocus the student experience is being considered by many educationalists, but for SE studies it is clear that there should be an early focus on teamwork and cooperation. Instilling the concept of “ego-less working” at an early stage is a key element, the introduction of learning communities [23] another.

In most academic environments location and scheduling problems are difficult to overcome. In time, additional research into both distributed light-weight methods and their application to educational situations may show a potential
benefit for such practices in education. Currently a solution to the co-location problem is beyond the means of most institutions.

Most real-world student projects are still too small to really need processes suitable for large-scale software. One way to present some of the problems of large-scale project to students is to offer a team-based maintenance project, and require students to make extensions. This has some overlap with the idea of the Real World Laboratory at Georgia Institute of Technology [21]. With a sufficiently large and complex base system, 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. The process also requires them to consider the new requirements from all perspectives, i.e. designing, testing, and implementing the necessary, as well as leaving a trail for future maintainers. Where the code itself provides the sole key to the system and the documentation is merely the comments embedded in it, students become very aware of how useful documentation is. After such an experience, it is common for them to agree that running software alone should not be the criterion of a well-engineered product.

Students who are steeped in technological aspects sometimes belittle the importance of the less technical aspects of a project. From our experience these are the students who are most likely to be most ardent in their desire to use an agile process. Often, however, these are the very students who should be actively encouraged to learn how to communicate with others on a project. They often are unaware that others do not think in the same manner as they do, and need prompting to capture design rationale or to write up test reports, for example. Learning to communicate verbally with clients to establish their needs, giving presentations and writing documents such as specifications are necessary life-long skills for software engineers in industry. A “document-free” process like XP does not lend itself to communications which are more than very informal interactions. On the other hand, we could consider Metaphor to be a form of documentation or communication, even if it is verbal and informal.

Running software is not the only goal for our budding software engineers. We believe it appropriate to place the primary emphasis on process, rather than product, often to the chagrin of students. If they are to learn techniques suitable for large-scale development, then we must attempt to simulate the development of a complex system. Therefore 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.

Instead of exposing students to a popular package of practices, it is preferable that they are exposed to a set of techniques that help them manage a variety of projects, from small to large-scale. This needs to be done in the context of the
positive and negative issues associated with any process model, and students need to be given guidelines on how to select a process model for the particular situation. The Crystal methodologies [6] is a process-kit which could be used for this purpose, the Rational Unified Process [15] 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 eXtreme Programming as the answer.

6 Conclusions

Agile development processes such as eXtreme Programming are becoming increasingly popular in industry and are often touted as the way to go in software development. Hence, there is a growing demand from industry to introduce agile development practices in tertiary education. However, in such a context, there is little experience in teaching agile development processes which goes beyond Pair Programming.

In this paper, we have examined a number of objectives for Software Engineering education in a tertiary institution. In particular, we have argued that the structure of a Software Engineering curriculum must be defined in a way that students become competent in a variety of development technologies, methods and programming languages suitable for large-scale systems. Especially focusing on team-based, final year “capstone” projects, we have set the practices of eXtreme Programming in relation to these educational objectives.

Keeping this perspective in mind, it appears that eXtreme Programming has limited value for educating about large-scale system development. First of all, limited resources in many educational institutions makes it questionable whether the necessary infrastructure can be set up so that eXtreme Programming is applicable effectively. Second, some of the practices are well suited for a learning environment, others contradict educational goals. In particular the code centric view of eXtreme Programming makes it difficult to introduce practices which are needed for larger-scale systems. Furthermore, students generally do not have the necessary maturity to fully understand the practices and their respective consequences of agile methodologies. Curriculum changes are needed to ensure that students embrace a less self-centred philosophy than is currently the case. As a consequence, we conclude that eXtreme Programming as a package as it stands now does not lend itself for use in tertiary education.

On the other hand, selected practices of eXtreme Programming may be helpful
for educating about small scale development. The beneficial effects of Pair Programming in particular is worthy of further investigation. Further studies in this area would be useful to gain more conclusive insights. These studies could also include issues such as alternatives for creating suitable collaborative learning environments and how some of the practices of eXtreme Programming could be introduced in a distributed or online learning environment.

Acknowledgements

Many people have contributed in the process of writing up this work. We especially mention Martin Dick, Steve Hayes, Catherine Lang, Oscar Nierstrasz and Rajesh Vasa for their invaluable comments as well as the anonymous reviewers for feedback on earlier drafts.

References


