Integrating Agile Practices into Software Engineering Courses

Gregory W. Hislop, Drexel University
Michael J. Lutz, Rochester Institute of Technology
J. Fernando Naveda, Rochester Institute of Technology
W. Michael McCracken, Georgia Institute of Technology
Nancy R. Mead, Software Engineering Institute
Laurie A. Williams, North Carolina State University

ABSTRACT

Agile software development methodologies are gaining popularity in industry although they comprise a mix of accepted and controversial software engineering practices. It is quite likely that the software industry will find that specific project characteristics will determine the prudence of using an agile or a plan-driven methodology – or a hybrid of the two. Educators must assess the value and applicability of these emerging agile practices and decide what role they have in software engineering curricula. This paper provides a brief overview of several agile methodologies, including a discussion of evaluative research of agile practices in academia. The paper also considers instructional issues related to agile methods and the impact of agile methodologies on existing curricular references such as SWEBOK.

INTRODUCTION

> *Plan-driven methods work best when developers can determine the requirements in advance . . . and when the requirements remain relatively stable, with change rates on the order of one percent per month*.
>
> -- Barry Boehm (Boehm 2002)

Plan-driven methods are those in which work begins with the solicitation and documentation of a complete set of requirements. Some examples of plan-driven methods are various waterfall and iterative approaches, such as the Personal Software Process (PSP) (Humphrey 1995) and the Rational Unified Process (RUP) (Jacobson et al. 1999). Beginning in the mid-1990's, many found this initial requirements documentation step frustrating and, perhaps, impossible (Highsmith 2002). As Boehm suggests in the comment above, these plan-driven methods may well start to pose difficulties when change rates are still relatively low. The industry and the technology move very quickly. Customers have becoming increasingly unable to definitively state their needs up front. As a result, several consultants independently developed methodologies and practices to embrace, rather than reject, higher rates of change. Many of these methodologies and practices are based on iterative enhancement, a technique which was introduced in 1975 (Basili and Turner 1975).

The consultants developed these methodologies independently, though there were fundamental similarities in their practices and philosophies. In February 2001, these consultant joined forces (Agile 2001, Fowler and Highsmith 2001). They decided to

classify their similar methodologies as "agile" – a term with a decade of use in flexible manufacturing practices (Lehigh 1991). They formed the Agile Alliance and wrote a set of Agile Values and Agile Principles. The methodologies originally embraced by the Agile Alliance were Adaptive Software Development (ASD) (Highsmith 2000), Crystal (Cockburn 2002), Dynamic Systems Development Method (DSDM) (Stapleton 1997), Extreme Programming (XP) (Beck 2000, Auer and Miller 2001), Feature Driven Development (FDD) (Coad et al. 1997) and Scrum (Schwaber and Beedle 2002). Since then, other methodologies have striven to be classified as agile.

The agile methodologies share three overarching philosophies: process control theory, emergence, and self-organization (Schwaber 2001). Each is discussed in the sections that follow.

Process Control Theory

Defined processes are those in which repeating a set of pre-defined steps can predictably lead to a desired outcome. An example of a defined process is the assembly of an automobile. Engineers can design a process to assemble the car; they can specify an order of assembly and actions on the part of the assembly-line workers, machines and robots. With little variation, if these steps are followed, a high quality car will be produced.

The agile viewpoint is that software development is not a defined process; we cannot repeat a set of steps and expect a predictable outcome. (Schwaber and Beedle 2002) Instead, software development requires rapid "inspect and adapt" cycles and feedback loops. As a result, agile methodologies devote limited resources to up-front, exhaustive collection of customer requirements. Agile methodologies use very short (2-6 week) iterations focused on producing *working* software the customer can try out (or inspect). The customer inspection allows the development team to adapt their plans and priorities for the next iteration.

Agile methodologies also utilize the "inspect and adapt" cycles to improve product quality. Pair programming is a continuous feedback loop. Early and incessant testing techniques are also very effective.

Emergence

When faced with changing requirements and technologies, agile methodologists do not believe that a software application can be fully specified up-front. Instead, the true requirements leading towards the development of a system a customer *actually* wants (as opposed to what they initially thought they wanted) should be allowed to emerge over time. The agile methodologies welcome changing requirements, even late in the development cycle. The short iterations and customer inspection of working software from each iteration provide the mechanism to allow requirements to emerge. The goal of this flexibility is to deliver what the customer really wants even in the face of constant change and turbulence.

Self-organization

Agile methodologies give the entire (extended) development team the autonomy to self-organize in order to determine the best way to get the job done. Team members are not

constrained by pre-determined roles or required to execute obsolete task plans. Managers of agile teams place a great deal of trust and confidence in the entire team. In self-organization, the emphasis is on face-to-face conversations, rather than on communicating through formal (or informal) documents. Software developers talk with software developers, business people talk with software developers, customers talk directly with either business people or software developers.

Agile methodologies also advocate the use of post-mortem meetings in which team member reflect on how to become more effective. The team then tunes and adjusts its behavior accordingly.

Section 2 of this paper is a brief survey of selected agile methodologies. Section 3 examines the implications for Software Engineering education. In Section 4, we conclude our discussion.

SELECTED AGILE METHODOLOGIES

The sections that follow present brief overviews of three agile methodologies: XP, FDD, and DSDM. The intent is to provide a survey of the landscape including a summary of key features and any research results available for the methodology. Each section also identifies key references for additional reading.

Extreme Programming

Just the expression "Extreme Programming" (Beck, 2000), might evoke fear in traditional software engineering circles. But a closer look reveals Extreme Programming (XP) to be a sound structure based on a number of principles and practices to which any engineer can relate. After practicing XP, doubts might be replaced with a conclusion that XP offers a reasonable alternative to building software systems using traditional, prescriptive, document-heavy software development processes some organizations have adopted.

XP departs from the traditional software engineering methodologies by shifting the focus of attention to a discipline of engineering code. It advocates building simple, clean, and tested systems in small installments through the coordinated effort of a team of engineers.

Values and Principles - XP can be best explained in terms of its main structural components as shown in Figure 1. The four values, *communication*, *simplicity*, *feedback*, and *courage* can be thought of as the basis of XP's work ethic. On the other hand, the four basic activities, *coding*, *testing*, *listening*, and *designing* can be viewed as forming XP's technical backbone. As shown in Table 1, the four values yield 15 principles and the four basic activities form the basis of 12 practices.

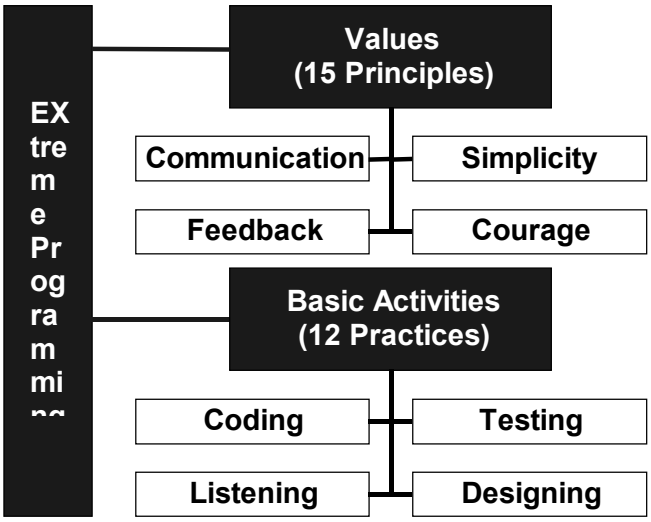

Figure 1. XP's structure.

Communication among team members and between the team and its clients is expected to be qualitatively high and frequent. This is so because in XP, documentation in the traditional sense is limited. Clients communicate functional requirements as "stories" written on small file cards that are used to guide the development.

Simplicity requires that the engineers build the "simplest system that will do the job." As functions are incorporated into the evolving system, the engineers modify it as much as needed without unnecessarily complicating the design. Modifying the system might include refactoring, an activity that seeks to improve the code's structure while preserving its functionality.

Feedback in XP works at the scale of minutes and days and it can take various forms. Constant system integration, for example, gives the engineers immediate feedback on the state of the system. When customers write new stories, the engineers estimate them right away to give customers immediate qualitative feedback on their work. Pair programming in which to programmers work together at one computer, is a continuous feedback loop. Early and incessant testing practices also provide rapid feedback to programmers.

Courage, in XP means that engineers are responsible for ensuring that the evolving design is the best and simplest possible. But arriving at simple designs may require courageous decisions. For instance, large chunks of code might be thrown out, or the system might need to be re-engineered to eliminate duplicated code.

XP has fifteen basic principles that support and uphold its values. There are outlined in Table 1.

Ultimately, there has to be a program. Thus *coding* is the one activity we cannot do without (Beck, 2000). Coding is seen in XP as a learning activity. It helps the engineer test his or her thinking: if the thinking is correct, the code will do what it is designed to do.

Principles	Practices
Rapid feedback	Planning Game
Assuming simplicity	Small releases
Incremental change	Metaphor
Embracing change	Simple design
Quality Work	Testing
Teaching learning	Refactoring
Small initial investments	Pair programming
Playing to win	Collective ownership
Concrete experimentation	Continuous integration
Open, honest communication	Coding standards
Working with people's instincts	On-site customer
Accepted responsibility	40-hour week
Local adaptation	
Traveling light	
Honest measurement	

Table 1. *XP's Principles and Practices*

Testing is woven through the entire development process as new functions are incorporated into the existing system. Engineers are required to maintain an automated test suite, which must run to 100% completion before integrating the next function.

Writing tests requires knowledge. Engineers can only write tests if they know *what* is to be tested. XP engineers *listen* to their clients as they develop their stories and help them refine them.

It may seem that the XP development cycle boils down to a continual iteration of "listening, then writing a test case, then making the system work for that test case." In actuality, "design as you go along" is a better descriptor. *Designing* is a constant activity in XP as the evolving system is refactored to incorporate new functions. Instead of an architectural design, XP relies on a metaphor to guide its design. A metaphor is a brief description that conveys the system's main functional attributes.

Of all of XP's practices, the one that seems to raise the most questions is *pair programming*, in which a pair of engineers write code together at the same computer. This is contrary to the common belief that two engineers working independently are more productive than two engineers working together. Yet there is some evidence that two engineers working together not only produce better code but also, in the long run, produce *more* code than two individuals working alone (Williams et al, 2000).

Feature Driven Development

Feature Driven Development (FDD) was conceived as a merger of two practitioner's ideas of managing development projects. The merger occurred when Jeff De Luca and Peter Coad were brought in as consultants on a project that was in trouble. Previously, Peter Coad had been applying feature-oriented development techniques on his projects and Jeff De Luca had been applying a streamlined, lightweight process framework. They merged their two concepts into a method that became known as FDD. They applied the

FDD process to the failing project and began delivering product to a pleasantly surprised customer within two months of their intervention. The previous developer had spent two years writing requirements specifications and had declared the project undoable.

FDD is conceptually similar to other agile methods, but has its own set of practices. The method relies on a basic architecture that is developed early in the project. The architecture is represented as UML class diagrams. With the architecture in hand, the lead designers decompose the business practices of the organization into features and plan, design, and code features. This orthogonal view of the classes is used to manage the development and specify the incremental deliveries of the project. Features are delivered to the customer as they are completed, in approximately two-week cycles. The features outlined in FDD are similar to the features of Rapid Development as described by (McConnell, 1996).

FDD contains five processes that can be adequately described in about a page. The processes are: Develop an overall model; Build a features list; Plan by feature; Design by feature; and Build by feature.

FDD also relies on Chief Programmers as originally described by Harlan Mills. The Chief Programmers are responsible for generating the features lists. The process of generating features is a functional decomposition from Subject Area to Business Activity to Business Step (a feature). A Business Activity is realized as a composition of 10-20 features. Another aspect of features and their associated compositions is that they are described in business rather than technical terms. The project is organized with Chief Programmers responsible for features and Business Activities. There are also class owners who are developers responsible for a class and feature teams that are temporal groups that develop the feature. The Chief Programmers manage the complexities of features as business entities and work packages that are sets of features that are technically related by using matrix management techniques to temporally manage the developers responsible for specific features.

Like most agile processes FDD emphasizes early and frequent delivery of working code to clients with a minimum set of deliverable documents. Unlike XP, FDD does require preliminary architectural design in the form of class diagrams and analysis of features via sequence diagrams. Similar to XP and other agile processes, FDD emphasizes early and continuous client involvement in the project and group development practices. XP's test case development prior to coding is supplanted with inspections.

The only documented evaluation of FDD is the creator's reports of the success of applying FDD on two projects. The first was the project mentioned in an earlier section for a large Singapore bank's lending application. The second was a bank re-engineering its technology platform.

Dynamic Systems Development Method

The Dynamic Systems Development Method (DSDM) was defined in 1994 by a consortium of IT organizations, primarily commercial firms in Europe. Their goal was to build on the proven successes of Rapid Application Development (RAD), while avoiding the proliferation of processes that hindered creation of support tools and environments. The resulting DSDM concepts have proven a stable base for the refinement and evolution

of the method's framework, now in its fourth major revision. Information on DSDM can be found in (Stapleton, 1997) and at the consortium web site (http://www.dsdm.com).

As with other agile processes, DSDM focuses on flexibility with respect to requirements while holding resources and time fixed. In this way, systems delivering value can be brought on line quickly, and these systems serve as the basis for further evolution. The model is based explicitly on the Pareto principle, where 80% of the functionality can be delivered with 20% of the effort - and then the resulting product is assessed to decide how to continue.

For all its emphasis on flexibility, DSDM is definitely at the conservative end of the agile method spectrum. Unlike XP, DSDM identifies five distinct phases in the development lifecycle: feasibility study, business study, functional model iteration, design and build iteration, and implementation.

The feasibility and business studies are done sequentially, and, as they provide the context for the project, must precede the other phases. However, both of these studies are time boxed - the feasibility study takes a few weeks at most, and the business study typically takes about a month. The primary goal of the feasibility study is to determine whether or not DSDM is appropriate for the proposed project. If so, the business study scopes the overall activity and sets a framework for both business and technical activities. At the end of these phases the high-level requirements are base lined, functional and information models are produced, and the system architecture is outlined.

The functional model iterations produce a sequence of prototypes that converge to cover the high-level functional and information requirements. The following design and build iterations focus on evolving these prototypes to create a system of sufficient internal and external quality to be safely released to users. The final product of the two core iteration stages is a working system. As with other agile approaches, testing is not a distinct phase, but is woven throughout all development activities.

In the last phase, the system is implemented within the user organization and responsibility for operation is transferred to users. At this point the effort can take declared complete or additional phases including complete new cycles may be undertaken.

As noted above, DSDM offers a more completely defined development process than some agile methodologies. However, underpinning the method are nine principles that reflect the common core values of all agile processes:

1. Users must be actively involved throughout the development process.
2. Teams (including both users and developers) must be empowered to make decisions without explicit approval from higher management.
3. Frequent delivery of products has highest priority.
4. Deliverables are evaluated primarily with respect to their fitness for business purposes.

5. Rapid iterations and incremental delivery are key to converging on acceptable business solutions.
6. No changes are irreversible - backtracking to or reconstructing previous versions must be possible.
7. High-level requirements are frozen early to allow for detailed investigation of their consequences.
8. Testing is integrated throughout the development process.
9. Collaboration among all stakeholders is the key to success.

The DSDM consortium's membership - approximately 400 full and associate members from academia, government, business, and industry - attests to the method's broad applicability. DSDM also provides a base for organizations seeking ISO 9000 certification as the method has been certified. However, as most of the information on DSDM is available only to members, it is difficult to get verifiable data reflecting the properties of the method. Though DSDM has been applied longer than most other agile processes, like them, little in the way of rigorous evaluation has been published to date. Given the growing influence of DSDM, however, and many anecdotal success stories, one can hope that such evidence will be forthcoming soon.

IMPLICATIONS FOR SOFTWARE ENGINEERING COURSES

The constant effort required to keep software engineering courses current is complicated by the uncertain value of new technologies. The following sections address this issue by considering agile methodologies from several perspectives. These include questioning whether agile methods have value at all, suggesting one possible framework for including them with existing material, discussing how they relate to existing curricular references, outlining issues for research to establish their value in education, and commenting on implication of including agile methods in courses.

Deja vu all over again?

There have been many books, presentations, courses, and discussions about agile methodologies and practices. As with many new methodologies, it seems that people are either violently in favor or violently opposed to them. It seems likely however that the true value lies somewhere in the middle. It is not surprising that a close look at agile methodologies reveals many familiar elements of software development.

The agile concepts include: stories, pair programming, refactoring, standup meetings, iteration, and a focus on teamwork. To experienced practitioners, some of the terms seem familiar. Others are less so. The real question is: “What is new?” Stories are used to help elicit and capture requirements. Pair programming refers to the concept of two programmers working together on a coding project. Refactoring is, in effect, revisiting the architecture and design, after some amount of implementation is complete, to see if it needs to be revised. Standup meetings are just that – brief meetings where no one is seated. Iteration is the familiar concept of iterating on a product. Another tenet of agile methods is a focus on teamwork.

These agile concepts each have connections to familiar counterparts. Stories correspond to requirements elicitation. Pair programming has built into it the concept of peer review. Refactoring implies redesign or reengineering. Standup meetings are similar to daily status reviews. Iteration is just another term for incremental development. Chief programmer teams are well represented in FDD. So, for each of these 'revolutionary' concepts, there is a traditional counterpart, many grounded with supportive empirical results.

Nevertheless, the hype leads one to ask a lot of questions about agile methodologies. For example, although it is clear what the advantages might be for a commercial organization for whom time to market is paramount, what about other kinds of software products? Does it make sense to use agile methods for safety-critical or mission-critical software? Similarly, the documented successes often seem to involve a highly paid 'agile' consultant. Is this needed for success? Does it require a brilliant team of programmers? What if your team consists of the lower 50% of the programming pool? Finally, suppose the client is not willing to select or prioritize requirements in order to get early capability? We don't hear much about risk mitigation, when agile methods are discussed. Are there risk mitigation strategies that are useful?

Agile concepts have been roundly criticized as an excuse for 'cowboy' programming (Rakitin, 2001). Agile methodologies can be perceived as allowing the programmers to ignore process, and to 'hack' code instead of doing boring documentation. It suggests that coding is the objective, and haggling over details is not important, or even worse, just a distraction. It promotes a focus on the solution without understanding the problem, and without planning.

It seems clear that much debate remains, and data to support one view vs. another still is hard to come by. At the same time, it appears that many of the agile practices are not the radical departure from established practice that they might first seem. Ultimately, it is quite likely that the industry will find that specific project characteristics will determine the prudence of using an agile or a plan-driven methodology – or a hybrid of the two, as suggested in (Boehm, 2002).

Connecting to Established Concepts

One possible framework for addressing various process approaches is the Win-Win Spiral Model. The Win-Win Spiral Model is a widely known model of software development that has its roots in the late 1980's in two streams of work. One of these (Boehm, 1988) defined the original spiral model in an attempt to address problems in prior models such as the waterfall and evolutionary development model. The second source was an effort to apply Theory W management to requirements definition (Boehm et al, 1998). This approach emphasizes negotiation to ensure outcomes that allow all stakeholders to be "winners." The Win-Win Spiral Model brings these ideas together by adding Theory W negotiated win concepts to the start of each iteration of the spiral model.

The Win-Win Spiral is a higher-level model and not an agile methodology like those described in the prior sections. However, it provides a useful point of reference for some of the key ideas in the agile methodologies, suggests an approach to linking agile

methods to existing work, and provides interesting references for discussion of evaluation.

The basic structure of the Win-Win Spiral is one of iterative development with the same general steps in each iteration. These steps begin with negotiation of win conditions and definition of objectives to satisfy these conditions. This is followed by a risk analysis that can inform selection of the development process. Development follows and then the iteration ends with planning for the next iteration.

There are a variety of interesting connections between concepts from the agile methodologies and the Win-Win Spiral. In particular, a variety of the agile concepts e.g., as embodied in the Agile Manifesto (Fowler and Highsmith, 2001), can be seen as compatible or extensions of concepts in the Win-Win Spiral and other models. For example:

Role of documentation - The original work on the Spiral Model included recognition of problems caused by "emphasis on fully elaborated documents as completion criteria for early requirements and design phases" (Boehm, 1988). In this sense the Spiral model matches agile principles, although the Spiral model also allows for document-driven processes where appropriate.

Interaction and customer collaboration - The Spiral model does not have the intense focus on individuals, face-to-face communication, or user participation called for in the agile principles. However, the negotiated Win and process of risk analysis and reduction insure a substantial level of interaction and both customer and user involvement.

Iteration - Both the Spiral model and agile approaches use a form of iterative development, but the agile approaches place more emphasis on short iterations that produce code. It is interesting to note that the original Spiral model while recognizing that evolutionary development "gives users a rapid initial operational capability", also points out two potential problems: (a) that evolutionary development can be "difficult to distinguish from the old code and fix model", and (b) that the approach may produce "a lot of hard to change code before addressing long range architectural and usage considerations." Both comments clearly anticipate criticisms currently mentioned in discussions of agile methods.

Process trade-off - The Spiral model clearly allows for selection of process approach based on particular needs of a project. Similarly, the basic agile tenets are stated as a series of trade-offs of the form "We value X over Y". The Spiral model can accommodate the agile preferences, but goes a step farther in suggesting that risk analysis for each project cycle is the correct basis for making these value judgments. In this regard, the Spiral model is less prescriptive than the agile methodologies. While not focusing specifically on the Spiral model, Boehm discusses this role by noting that "risk management offers another approach that can balance agility and discipline…" (Boehm, 2002).

Evaluation – The Win-Win Spiral model also provides an interesting point of reference for agile methodologies in considering the issue of validation and use in education. The

Win-Win Spiral has been applied in several operational settings and results published in a series of papers, including those already cited. In addition, the Win-Win Spiral has been applied in the classroom and this work is reported in papers such as (Boehm, et al, 1995; Boehm and Egyed, 1998; and Boehm et al, 1998).

These reports provide solid and ongoing information about how the model can be applied successfully in a classroom setting.

In summary, the Win-Win Spiral offers a possible framework that could be used to incorporate agile practices into a curriculum. This framework provides a context that relates agile approaches to a well-recognized and accepted model. In addition, the notions of risk reduction and process selection in the Spiral model would also provide the context to support teaching of plan and document oriented processes.

Relationship to Established Curricular References

Few of the agile process practices are directly addressed in primary curricular references for software engineering. The current CC2001 documents (ACM, IEEE 2001) focus on computer science, where process issues have traditionally received little treatment. The corresponding volume on software engineering is still under development, and here the prospects for addressing agile process principles are more promising. In particular, agile approaches will probably be incorporated into the general area of process and process improvement, where they can be compared and contrasted to more traditional, defined, and prescriptive models.

Currently, the Software Engineering Body of Knowledge (SWEBOK 2001) does not explicitly include agile processes as a topic. However, the topic areas of requirements, design, testing, etc., all contain items that address specific agile practices, though it would probably be better to expand these topics. In addition, the principles of the Agile Manifesto would fit within the general area of software processes. What is needed, however, is a cross-cutting index that shows how the various process approaches depend on specific topic entries. This would make navigation much easier for those trying to understand the relationships among processes, practices, and product.

Evaluative Research

As mentioned previously, there has been relatively little work done to evaluate the claims made for agile methodologies or their practices – both in industry and academia. Reports of a few initial studies have been published, such as the pair programming studies (Williams and Kessler, 2001; Bevan et al 2002; McDowell et al. 2002), but in general there is no way of knowing whether the agile practices have potential to improve student work or improve student learning.

In considering possible approaches, it seems more reasonable that educators will apply selected practices rather than entire methodologies. To begin with, teaching an entire methodology would require a significant commitment of scarce classroom time, as addressed in (Shukla and Williams, 2002). In addition, the agile methodologies typically have some aspects that do not have clear application in the classroom (e.g., the “40 hour week” rule in XP).

In this case, the question then becomes which practices seem most promising to try in teaching basic software engineering? Based on trials thus far, pair programming has emerged as the first candidate for careful examination. The studies mentioned above indicate positive results in introductory programming classes. Additional studies seem warranted in this area.

Discussions among participants at a recent workshop at the Conference on Software Engineering Education and Training identified testing as a second candidate for investigation in the classroom. The notions of "test first" and automated test seem reasonable to integrate into class work and may provide some improvement in student understanding and appreciation of testing. Refactoring was also identified as a candidate practice to try in class. In this case though, there was much more hesitation about probability of success in using the practice. One common concern was that refactoring was likely to be much more difficult to teach. In addition, it was much less clear how the result could be evaluated to demonstrate effectiveness.

Studies for these practices and others are clearly needed. Authors of this paper and others have started to work on these questions, but additional research effort will be needed to help the software engineering education community determine how to incorporate agile practices in software engineering courses.

CONCLUSION

For software engineering educators, the need for evaluative research on classroom use of agile practices is clear. For the moment however some approach must be chosen. This might begin with several particular thoughts.

First, a sequence must be chosen for dealing with agile processes and more traditional processes. One thought is that the agile processes should not be taught until after students have been taught more traditional development approaches. This would give students a needed grounding in the more formalized methods of developing software before being exposed to the lighter weight approaches to development. An alternative strategy might be to introduce agile practices as a stepping-stone to more formal processes. As another approach, agile and traditional practices can be taught in parallel and tradeoffs between and applicability of the two approaches considered.

Second, the need for significant client interaction throughout the project may be difficult to support in project courses. Disappearing client problems are common in class projects and this raises the concern that it might be difficult to have engaged and active clients during the project's life (which typically run for 2-3 months). Alternative approaches might include surrogate clients or developing products for other faculty members, but these suggestions may not be enough to mitigate the problems of needing client involvement throughout the development.

Third, agile processes might help engage the students in projects because of the short delivery cycles of product. The delivery of features in 1-2 week increments might be much more satisfying to students than traditional projects where a working product does not appear until the completion of the project and semester.

It is important to combine these specific issues with a broader perspective as educators. We should understand both sides of the debate, keep up with development of agile methodologies, and contribute to the evaluation of them from the educators' perspective. Finally, we will serve our students best if we teach a range of process approaches and also help students develop the ability to learn and critique new methods.

BIBLIOGRAPHY

ACM, IEEE (2001), *Computing Curricula 2001 - Final Draft*, http://www.computer.org/education/cc2001/final/index.htm.

Agile Alliance (2001), http://agilealliance.org.

Auer, K. and R. Miller (2001), *XP Applied*. Reading, Massachusetts: Addison Wesley.

Basili, V. R. and A. J. Turner (1975). "Iterative Enhancement: A Practical Technique for Software Development." IEEE Transactions on Software Engineering 1(4).

Beck, K. (2000), *Extreme Programming Explained: Embrace Change*, Reading, Massachusetts: Addison-Wesley.

Bevan, J., Werner, L., McDowell, C. (2002). Guidelines for the Use of Pair Programming in a Freshman Programming Class, *Fifteenth Conference on Software Engineering Education and Training (CSEE&T 2002).*

Boehm, B., (2002) *Get Ready for Agile Methods, with Care*, IEEE Computer, p. 64-69. January.

Boehm, B., Alexander Egyed (1998) *Improving the Life-Cycle Process in Software Engineering Education*, 24th Euromicro Conference.

Boehm, B., et al (1998) *Using the Win Win Spiral Model: A Case Study*, IEEE Computer, p. 33-44. July.

Boehm, B., et al (1995) *Software Requirements Negotiation and Renegotiation Aids: A Theory-W Based Spiral Approach*, Proc. International Conference on Software Engineering.

Boehm, B., (1988) *A Spiral Model of Software Development and Enhancement*, IEEE Computer, p. 61-72. May.

Coad, P., deLuca, J., and Lefebvre, E. (1997), *Java Modeling in Color with UML*, Prentice Hall.

Cockburn, A. (2002), *Agile Software Development*, Addison Wesley.

Fowler, M. and J. Highsmith (2001), *Agile Manifesto*, Software Development. August

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

Highsmith, J. (2000), *Adaptive Software Development: A Collaborative Approach to Managing Complex* Systems, Dorset House.

Humphrey, W. (1995), *A Discipline for Software Engineering*. Reading, Massachusetts: Addison Wesley Longman, Inc, 1995.

Jacobson, I., Booch, G., and Rumbaugh, J. (1999), *The Unified Software Development Process*. Reading, Massachusetts: Addison-Wesley.

Lehigh University (1991), "Agile Competition is Spreading to the World," *http://www.ie.lehigh.edu/*, 1991.

McConnell, Steve (1996), *Rapid Development*. Microsoft Press. Redmond, WA.

McDowell, C., Werner, L., Bullock, H., Fernald, J. (2002). The Effects of Pair Programming on Performance in an Introductory Programming Course, Proceedings of the Conference of the Special Interest Group of Computer Science Educators (SIGCSE 2002).

Rakitin, S. (2001), Letters to the Editor, *Manifesto Elicits Cynicism*, IEEE Computer, p. 4. December

Schwaber, K. and M. Beedle (2002), Agile Software Development with SCRUM, Pearson Technology Group.

Schwaber, K. (2001). "Will the Real Agile Process Please Stand Up?", *E-Project Management Advisory Service, Cutter Consortium*, Vol. 2, No. 8.

Shukla, A. and Williams, L. Adapting Extreme Programming For a Core Software Engineering Course, *Fifteenth Conference on Software Engineering Education and Training (CSEE&T 2002).*

Stapleton, J. (1997), *DSDM: The Method in Practice*, Addison Wesley Longman.

SWEBOK (2001), *Software Engineering Body of Knowledge - Stoneman Trial Version 0.95*, http://www.swebok.org.

Williams, L, Kessler R., Cunningham, W., Jeffries, R. (2000), *Strengthening the Case for Pair Programming*, IEEE Software, p. 19-25. July/August.

Williams, L. A., & Kessler, R. R. (2001). Experimenting with Industry's "Pair Programming" Model in the Computer Science Classroom. *Journal of Computer Science Education, pp. 1-20.* March.