88 Journal of Database Management, 16(4), 88-100, October-December 2005

RESEARCH REVIEW

Agile Modeling, Agile Software Development, and Extreme Programming:

The State of Research

John Erickson, University of Nebraska at Omaha, USA
Kalle Lyytinen, Case Western Reserve University, USA
Keng Siau, University of Nebraska - Lincoln, USA

ABSTRACT

While there are many claims for the successful use of extreme programming (XP) and agile modeling (AM), and the proponents can often be vocal in the extreme regarding their supposed benefits, research evidence supporting proponents' claims is somewhat lacking. Currently, the only research appearing to investigate the phenomena consists of two prominent streams. A small number of case studies and experience reports that generally promote the success of XP in various development environments, and a well-established stream of research into pair programming has generated results that in part support the idea of XP. Research into AM appears to be even more sparse than that for XP. Case studies, comparative analyses, and experience reports comprise the majority of the research in the area, while very few empirical research efforts have been conducted. This article reviews the state of research in XP and AM, and recommends areas that could benefit from further study. Since nearly all empirical XP research relates to pair programming, a closer look into the unstudied XP core practices would be beneficial, although interaction between related core practice areas could confound such efforts. It might also be possible to group related core XP concepts and study the groups individually. Finally, there are those who claim that XP and AM, or even agility in general, are really nothing more than a repackaging of old concepts. This claim needs to be investigated.

Keywords: *agile software development; agility; agile modeling; AM; extreme programming; XP*

INTRODUCTION

In a world where system-development methodologies abound, and an entire area of research and practice, as evidenced in method engineering (Siau, 1999), has grown up focusing on the creation of software-development methodologies, it can often appear that there might be a different software-development methodology for every system. This means that the choice of system-development approach can be a daunting and an extremely difficult task.

Given the unacceptably high failure rates associated with systems-development efforts

(Hirsch, 2002; Siau, Wand, & Benbasat, 1997), and the fact that many traditional development methodologies are extremely complex and difficult to use, the choice of methodology becomes even more critical. Recently, the notion of theoretical and practical complexity was introduced (Erickson & Siau, 2004; Siau, Erickson, & Lee, 2002, 2005). In such a turbulent environment, where it seems obvious that one size does not fit all (Henderson-Sellers & Serour, in press; Merisalo-Rantanen, Tuunanen, & Rossi, in press), the agile software-development approaches would appear to be just what the doctor ordered. Extreme programming (XP) and agile modeling (AM), under the umbrella of the agile approaches to systems development, are two relatively recent emergent forces of the genre.

While there are many claims for the successful use of extreme programming and/or agile modeling (C3 Team, 1998; Grenning, 2001; Manhart & Schneider, 2004; Poole & Huisman, 2001; Schuh, 2001; Strigel, 2001), and the proponents can often be vocal in the extreme regarding the supposed benefits of both (Ambler, 2001b, 2001c, 2002a, 2002b; Beck, 1999), research evidence supporting the claimed benefits is somewhat lacking. At this point, the only exceptions seem to be research into the phenomena consisting of two prominent streams. First, researchers have conducted a scant few studies of extreme programming, consisting primarily of case studies and experience reports. While not detracting from the value of a well-conducted case study, additional research into the details of the purported benefits of the approaches would lend some much-needed weight to the existing body of work. Second, a well-established stream of research into pair programming has generated results that support, at least for one core practice, the idea of extreme programming.

The body of research into agile modeling appears to be even sparser than that for extreme programming. Case studies, comparative analyses, and experience reports comprise the majority of the scant research in the area, while very few empirical research efforts have been conducted. Other research efforts encompass the agile software-development approach as a whole.

This article reviews the state of research in extreme programming and agile modeling. In addition, research into agile software development will be examined. These goals will be accomplished by first briefly presenting the details of agility, XP, and AM. A literature review of the approaches follows. The article then identifies gaps in the literature, and proposes possible areas where future study would benefit both research and practice. Finally, we conclude the article.

AGILITY, XP, AND AM

Agility

Agility is often associated with such related concepts as nimbleness, suppleness, quickness, dexterity, liveliness, or alertness. At its core, agility means to strip away as much of the heaviness, commonly associated with traditional software-development methodologies, as possible to promote quick response to changing environments, changes in user requirements, accelerated project deadlines, and the like. The reasoning is that the traditional established methodologies are too set and often too full of inertia so that they cannot respond quickly enough to a changing environment to be viable in all cases, as they are often marketed to be.

The "Agile Manifesto" was composed by several XP leaders, promoters, and early adopters (e.g., Kent Beck, Martin Fowler, Robert Martin, Steve Mellor, etc.), and outlines the principles embodied in software and system agility (Lindstrom & Jeffries, 2004). Agile methodologies attempt to capture and use the dynamics of change inherent in software development in the development process itself rather than resisting the ever-present and quickly changing environment (Fowler & Highsmith, 2001). Among the agile approaches are XP, Crystal methodologies, SCRUM, adaptive software development, feature-driven development (FDD), dynamic systems development, and AM.

Information-systems development has generally followed a proscribed pattern or process over the past 40 years. Depending upon the specific methodology, the process has assumed many different names, each comprising unique steps. For example, systems developers have proposed the systems-development life cycle, the spiral method, the waterfall approach, rapid application development, the unified process, various object-oriented (OO) techniques, and prototyping, to name just a few (Booch, Rumbaugh, & Jacobson, 1999). Many of these time-tested design patterns have evolved into what are now termed "heavyweight" processes.

An influential trend impacting the systems-development landscape is the migration to encompass OO analysis and design methodologies. It seems likely that such a move has come about largely as a response not only to the emerging dominance of OO programming languages, but also due to their growing importance to a number of the more recent agile and lightweight development techniques (Fowler & Highsmith, 2001), such as AM and XP. In a very short time, agile software-development methodologies have created large waves in the software-development industry.

One of the most used and best-known goal-measurement approaches to assessing system complexity and success emerged from work done at Carnegie Mellon in the late 1980s, culminating in the capability maturity model (CMM; Paulk, 2001). The CMM takes a goal-measurement approach (Pfleeger & McGowan, 1990) and attempts to measure the maturity of the implementing organizations. Five levels of organizational capability and maturity as related to software development constitute CMM. An example of a truly large and encompassing process, the CMM guidelines for becoming a Level 5 organization consume more than 500 pages of requirements. Even stripped to the bare essentials, the CMM comprises 52 primary goals and 18 key process areas (Paulk, 2001).

The unified process (UP), while an OO analysis and design technique, is considered to be a heavy methodology as well. UP consists of four phases, nine disciplines, approximately 80 primary artifacts, 150 activities, and 40 roles (Hirsch, 2002). Even the most optimistic developer looking at UP for the first time would not call it a lightweight or agile methodology. However, Hirsch also provided an experience report consisting of two cases detailing how UP could be modified to be more agile.

The proliferation of development methodologies notwithstanding, it appears that the vast majority of these approaches can be condensed into (or at a minimum, contain) four critical steps: analysis, design, coding, and testing (ADCT). Essentially, AM and XP fit into the ADCT paradigm by breaking the process into very small steps, with each step including the critical analysis, design, coding, and testing elements.

Extreme Programming

XP encompasses four values — communications, simplicity, feedback, and courage — as well as four basic activities: coding, testing, listening, and debugging (Beck, 1999). According to Beck, these values and activities lead to the 12 core practices of XP: the planning game, small releases, metaphors, simple design, testing, refactoring, pair programming, collective ownership, continuous integration, 40-hour weeks, on-site customers, and coding standards (Beck; Jeffries, 2001; Wake, 1999).

Beck (1999) presented the primary details and advantages of the approach. According to Beck, XP essentially means to "embrace change." Beck began his exposition by proposing that the basic problem facing software development is risk. Jeffries (2001) proposed that:

Extreme Programming is a discipline of software development based on values of simplicity, communication, feedback, and courage. It works by bringing the whole team together in the presence of simple practices, with enough feedback to enable the team to see where they are and to tune the practices to their unique situation.

Simply put, XP is the coding of what the customer specifies, and the testing of that code to ensure that the prior steps in the development process have accomplished what the developers intended. No unforeseen or anticipated tools or features are engineered into the process because XP is oriented toward producing a product in a timely manner. The idea behind XP is that if features are needed later in the development process and the customer will notify the development team of them, the developers do not worry about those features for the present. Needless to say, this represents a vast departure from the normal software-development process, in which all requirements (and we naturally suppose these requirements to include features) must be specified up front. This can easily turn into a nightmare since the user requirements can often be seen as dynamic and changing rather than static and set.

According to Turk, France, and Rumpe (in press), XP's values, activities, and practices are quite interrelated, with a relationship structure as follows: Underlying the principles and practices are the basic assumptions that support the XP process. They go on to conduct a more thorough examination of the XP process and connect it to the core beliefs expressed in the "Agile Manifesto."

Many professionals have proposed different types of modeling or development processes to use with XP, and also to inform developers regarding new concepts in program development (Ambler, 2001a, 2001-2002; Fowler, 2001; Lindstrom & Jeffries, 2004; Palmer, 2000; Willis, 2001). For example, AM, the unified modeling language (UML), UP, and FDD have been used with XP. Since processes used to develop code require modeling, AM as related to XP has been developed. Ideally, modeling techniques help to communicate to the entire development team the specifics of a particular design. It appears that the modeling techniques used for AM are as diversified as there are software-development scenarios or ideas on the use of XP.

Agile Modeling

Ambler (2001a) described agile modeling as "...a practice-based software process whose scope is to describe how to model and document in an effective and agile manner." Naturally, a question then arises: Does AM apply to project development executed in an XP environment (and if so, how?)? Ambler (2001a) went on to develop and explain AM's core and supplementary principles: simplicity, iterative development, robustness, incremental releases, staying on task, producing a quality product, creating models and the accompanying documentation only as necessary, multiple models, fast and clear feedback on the latest changes to the model(s), and discarding models and documentation that go back more than just a few iterations (Ambler, 2001-2002).

What can be gleaned from the XP approach and applied to AM is the perception that the XP core practices, rather than consisting of isolated ideas about how to create better systems, are quite closely interrelated and interdependent. Essentially, to take the XP approach means to abandon many of the practices that many developers have come to hold dear as critical necessities to systems development. However, since XP merely develops systems, the analysis and design of those systems must also be considered. To do that, developers must model, and to analyze and design effectively for an XP development environment, they should therefore model with an eye toward XP. In other words, Ambler (2001a) was in essence proposing that in order to best exploit the benefits of XP, developers should use agile modeling as a lead-in to XP.

XP developers have taken two diametrically opposed perspectives to XP-based systems development (Ambler, 2001-2002). One group proposes that the use of an up-front modeling tool such as UML is necessary to successfully capture and communicate critical system architectures (Armano & Marchesi, 2000). Opposed to the more traditional UML modelers are those who promote the use of UML or other modeling tools only occasionally or simply for graphical representations of the system under development (Willis, 2001). Those developers propose that UML is too complex and heavy to be truly useful in an agile environment. There is some evidence to indicate

that UML is indeed complex (Erickson & Siau, 2003, 2004; Siau & Cao, 2001; Siau, et al., 2002, 2005; Siau & Lee, 2004). Furthermore, UML is likely to become even more of a heavy tool with the much-anticipated release of UML 2.0.

AM basically creates some common ground between the two camps by proposing that developers communicate system architectures by applying AM's core practices to the modeling process (Willis, 2001). This seemingly incompatible marriage of XP practices to UML-like modeling techniques represents the basis of AM. This melding requires two things. First, if a modeling approach is to successfully approximate XP in terms of core practices, then the model must be executable in that it can easily be converted into code and represent to a large extent the functions and features required in the final system. Second, in the context of XP, any models developed must be testable. Recall that developers test more or less continuously in the XP paradigm. This means that, contrary to the common use of UML as merely a tool to draw diagrams, UML in the AM paradigm must be utilized to its fullest extent and even extended so that the models are executable and testable (Ambler, 2001-2002). Two different tools extending the capabilities of UML into the AM arena have been developed or are currently under development: Argo/UML from the University of Hamburg, and a Petri Net creation named Renew (Ambler, 2001-2002). Ambler insists that as these tools move into more mainstream use, the potential advantages of the agile modeling approach combined with extreme programming should become clear.

EXTREME PROGRAMMING AND AGILE MODELING AND METHODOLOGY LITERATURE AND RESEARCH

XP Research

The literature for XP, as previously noted, can generally be split into two separate streams. First, there is a good number of case studies or experience reports that cover the XP approach as a whole. Second, there are research efforts related to one or more of the core practices associated with XP. The experience reports tend to claim success for adopting one or more of the XP practices for specific projects, but offer little in the way of success measures. Since the case studies and experience reports generally involve XP in its entirety, they will be discussed first, and the research related to the core practices second.

XP Cases and Experience Reports

The C3 Team (1998) at Chrysler adopted XP's simplicity value for its compensation-system development effort. The team insisted that the project could not have been done in the required time by using the traditionally applied waterfall method. The team found itself behind in implementing a difficult system and discovered that XP lent itself to what they were trying to do. However, the case description does not include much detail in terms of resistance to change that moving to XP might have caused among developers, or other problems encountered that could have been attributed to the XP methodology.

Iona Technologies found that code maintenance and software reengineering were best accomplished by implementing practices they later found to be part and parcel of XP (Poole & Huisman, 2001). They at least partially adopted 11 of the 12 core XP practices, failing only to go to a 40-hour week and noting that they lacked the courage to try at that point. They also were a bit hesitant about adopting pair programming, noting that many of their programmers were hesitant about trying it.

Schuh (2001) detailed another in-trouble project that was saved by implementing XP practices. The development team at ThoughtWorks was far behind schedule working on requirements collected by the previous consultant, while the customer had changed the specification and had a rigid delivery-date requirement. The project team in this case also partially implemented 11 of 12 XP practices, choosing also not to go to a 40-hour week. The team was also hesitant about adopting pair programming.

Another experience report indicates that they used a traditional "big design up front"

Table 1. Partial listing of XP core practices research

Author(s)	Type of study	Independent Variable(s)	Dependent Variable(s)	Results	Threats/Issues
Williams & Kessler (2001)	Experiment	Pair vs. nonpair	Code errors	Pair superior	Many, see p. 17
Müller & Padberg (2003)	NPV (net present value) model, simulation	Pair-programming and test-driven development	NPV	Lower NPV for XP practices	Simulation vs. real world
Kuppuswami, Vivekanandan, Ramaswamy, & Rodrigues (2003)	Simulation	All XP core practices (effort for each practice individually)	Total effort	Using XP decreases total effort	Simulation vs. real world
Alshayeb & Li (2005)	Field measurement of development project	Changes, growth in class names during project execution	SDI (system design instability)	Refactoring and error fix negatively correlate with SDI	Interaction effects and variables not measured in the study

methodology for software-development projects (Grenning, 2001). Of the core XP practices, only metaphors were not adopted. Again, while some detail of problems was provided, there is no clear way to discern whether the problems were related to XP or simply part of the process.

XP Core Practice Research

XP education has received its share of attention, meaning that as more industry-based development projects move toward adopting at least some of the XP practices, there has been increasing pressure on university computer-science, information-systems, and information-technology programs to adopt teaching pedagogies with XP embedded. Table 1 indicates some of the investigations and lists the metrics used in the research.

Williams and Kessler (2001) conducted an experiment in pair programming in which they found that traditional postsecondary programming education conditioned students to work alone, and that simply telling them to begin working together does not necessarily result in improved programs, that is, those that have fewer code errors (dependent variable) are relatively run-time efficient (dependent variable), and so forth. However, they also noted that once the solo-approach mold was broken, the improvements in finished code were measurable. This is supported by other classroom-based research (Erdogmus & Williams, 2003; Hedin, Bendix, & Magnusson, 2005; Williams & Upchurch, 2001).

In settings outside the classroom, research attempting to assess the benefits of XP and/or its core practices has also been conducted. Aiken (2004) provided support for XP's pair-programming practice, noting that although the commonly listed benefits proved attractive to potential adopters, implementing pair programming remained an extremely challenging task. Newkirk and Martin (2000) illustrated via their case study a common problem with software development that XP is suited to address. They noted that once the first iteration (of the product) was successfully developed, tested, demonstrated, and delivered to the customer, within a 50-hour window and according to XP practices, the customer then changed the requirements and added 11 stories to the project. In their view, this provided support for the XP approach since, they claimed, a heavy methodology would not have been able to easily incorporate the changes requested.

The research done by Müller and Padberg (2003) is one of the few empirical research efforts that dispute the claims made for XP. They created an economic model that output a net present value of software-development projects. The results indicate that, using the

XP core practices of pair programming and test-driven development, and comparing with a traditional heavier methodology, the end-product NPV was smaller for the XP-based project than for the traditional project.

Karlsson, Andersson, and Leion (2000) provided an account of their experiences regarding the implementation of XP practices at Ericsson, focusing on incremental releases, which they called "daily builds." According to the authors, the project benefited greatly from using the daily-build approach. However, since daily builds also imply daily testing and rigorous attention to coding standards, the implementation effort at Ericsson proved quite challenging.

Agile Modeling and Agile Methodology Research

Agile modeling research appears to be extremely scarce. Armano and Marchesi (2000) adapted UML to an XP-like software-development project using what appears to be a combination of the spiral model and the unified process. The project team committed to weekly builds and refactoring, and used UML to represent user stories via use cases. The team actually developed a tool they called UMLTalk (UML and SmallTalk) to aid in the project-development effort.

Other research efforts involve holistic approaches to agile systems. For example, Fujitsu, the Japanese technology company, appears to be one of the early adopters of agile methodologies. They developed an agile tool they named Agile Software Engineering Environment (ASEE) as early as 1993 (Aoyama, 1998). The company found itself attempting to complete a software-development project from multiple distributed locations and saw the need for an agile approach to solve their problems. The tool was Web based and enabled releases of software at six-month intervals for four years.

Manhart and Schneider (2004) found that Daimler-Chrysler's embedded software effort for buses and coaches was moving toward a "cautious extension of agile process improvement" after adopting a few (four) agile principles. The development methodology in use was CMM, and the culture appeared to be fairly resistant to change. However, the authors end with a call for more empirical evidence supporting the claims of agile methodologists.

As previously noted, Hirsch (2002) reported on the successful adaptation of UP for two small projects. Noteworthy of Hirsch's experience report is that the UP agile adaptation worked best for small projects of one- to four-years duration and small development teams of three to eight people. This appears to be a recurrent theme of agile methodologies in general and XP specifically.

Abrahamsson Warsta, Siponen, and Ronkainen (2003) compared nine different agile methodologies and found that most covered different portions of the common development sequence (ADCT) with little or no reasoning as to why they took their specific perspective. Abrahamsson et al. further noted that most agile methodologies did not "...offer adequate support for project management." They recommend a focus on quality over quantity, which interestingly enough is a mantra of many of the agile proponents, and end with a comment that empirical research is "quite limited."

POTENTIAL AREAS FOR RESEARCH RELATED TO XP, AM, AND AGILE METHODOLOGIES

The recurring themes in XP research seem to revolve around XP's pair programming practice. Evidence supporting the idea of pair programming is mounting, and while practitioners conditioned with the heavier approaches such as CMM or UP tend to resist embracing pair programming, it seems that educators are moving to incorporate pair programming into computer-science curricula.

In the case of pair programming research, the effects of programming in pairs are measured against programming individually. Generally, the number of code errors was one measure of differences between programming in pairs vs. programming individually, with some sort of regression testing suite used to assess the errors. Possible confounding variables in-

clude such elements as ambient noise (i.e., from other cubicles); using electronic collaboration systems, such as MS Net Meeting, to collaborate; the physical placement of the computer, keyboard, monitor, and so forth when two programmers collaborate; personal incompatibilities between the two programmers in a given pair; and confusion or ambiguity regarding the role of the person not physically coding. The size of the program to be written is also likely to play a role in the relative success or failure of pair programming as it naturally does with other approaches to coding. Programmer experience is also likely to affect the outcome of pair programming research.

The development and execution of the testing suite, which implies some interaction effects between pair programming and another core XP practice, will also complicate research in this area. At least a few of these moderators or confounders, and the threat(s) they pose to validity, can be controlled in experimentation, but pair programming in practice should also be examined as part of research in this area.

A relatively large number of experience reports regarding the adoption of some XP practices exist, but hard, empirically-based economic evidence is lacking. Many of the case studies and experience reports indicate that most, if not all, of the XP core practices were successfully adopted. The practice most commonly not adopted was the 40-hour week.

Most experience reports also mentioned that organizations were already practicing the planning game. XP's planning game can be compared to developing user requirements in a more traditional systems-analysis and design-development approach. Many of the older and heavier analysis methodologies have well-established evidence regarding the importance of this step. The time spent in analysis could be one analyzable metric, although there are many threats to validity from using time as a variable. In addition, the output of planning can also be measured if artifacts (e.g., UML diagrams) can be standardized and compared across groups.

Added to the problem just mentioned is another that continually plagues developers: that of the aptness of the system. In other words, the planning game might be wonderfully executed along with other core practices and the result might be exemplary, but due to changes in the requirements from the outset, the system developed might not be the correct system. Of course, that problem is endemic to systems development in general, but since XP proponents claim that the approach is superior, then fewer instances of "you built the wrong system" should be evidenced. As to research in this area, while planning is critical and essential for success, it remains to be seen as to whether the specifics of the XP approach are more beneficial than other more standard approaches to developing user requirements.

Companies or organizations using the heavier methodologies typically had trouble adopting incremental releases because of the implications that the core practice has for several other core practices: simple design, testing, refactoring, and continuous integration. These core practices appear to be closely related since, for example, a daily build means that the testing suite must also be ready daily, which in turn has implications for continuous integration and refactoring. Research into these core practices will nevertheless be necessary if the overall approach is to be accepted by the mainstream.

If pair dynamic programming is used, the coding standards of core practice means that developers must agree up front on the conventions used for naming classes, for example, as well as on a host of other coding practices as well. A coding standard in the end means that someone looking at a code segment cannot tell which team member wrote it. This should be something that programmers do for all projects, but sadly, it is not. Research should be implemented that compares practice with recommendation in both the traditional and XP areas. However, this instance also highlights once again the difficulties of examining XP's core practices individually: the likelihood that interaction or correlation between and among other core practices will be possible, and even probable. In this case, the coding-standards practice is related to and could be affected by pair

programming and development of the test suite, just to name two, and there are likely to be other interactions as well.

The efforts of Kuppuswami et al. (2003) represent a pioneering effort in XP research. They used a process-model simulation to vary the level (in labor) of XP's core practices one at a time to judge the effect upon total effort for the project. They found that increasing effort (independent variable) into XP core practices reduced the total effort (dependent variable) needed to create the system, although interactions and other moderating effects were not discussed at great length. While the research provides some support for XP practices, field verification of the simulation is definitely indicated and would be very beneficial.

Other empirical efforts to study XP, in total or its core practices, are quite limited as well. William's numerous and varied studies along with a few others (Alshayeb & Li, 2005; Müller & Padberg, 2003) are the primary exceptions in this area of research. Agile modeling is almost totally unstudied, and any research into the methodology would be an improvement over the current state of affairs. The models themselves could be used as the measures of the efficacy of the methodology, although assessing models as to their relative "goodness" or "badness" is at least somewhat subjective and a possible threat to the validity of research conducted in that manner. The study of agile methodologies appears to be unorganized and, for want of a better word, random.

CONCLUSION

From a research-based perspective, it appears the research community, practitioners, and educators might benefit from a more structured approach to the study of XP. The bulk of the existing research appears focused on validating the overall XP approach, which is probably, or perhaps arguably, satisfactory if one is only concerned with the macroperspective of XP as a whole. However, since the proponents, as noted previously, seem to universally accept the 12 core practices as integral and necessary parts of XP, then it would seem logical to empirically examine the efficacy of each of the 12 core XP practices separately if we want to examine what it is that makes XP successful (or not). In other words, do we want XP to remain an essentially "black box" and simply accept that it works? Other than pair programming, incremental releases, and at most a few of the other core practices, many of the others remain relatively unstudied, at least in an XP environment.

As to XP specifically or agility in general as approaches to systems development, there is anything but unanimous agreement that there is really anything new. Merisalo-Rantanen, Tuunanen, and Rossi (in press) conducted a case study and concluded that XP is really nothing new, but simply a repackaging of old (although arguably useful) techniques for developing systems. Turk, France, and Rumpe (in press) also indicated that the benefits to be gained from adopting agile methods are not realized if the underlying assumptions are not met.

Confounding factors could also cause problems with research in this direction. For example, what are the differences between the prescribed approaches (heavy or light) and practice, and what effect do these differences or gaps have on the success of the various development efforts? Also, if we begin looking at the efficacy of the 12 core XP practices separately, it opens up the possibility of interaction effects. In other words, it appears that a number of the core practices are obviously related to one another, like pair programming and collective ownership, for example. If that is the case for obvious and even for nonobvious relationships, then what effect upon the overall success of the project, and ultimately the methodology, does strict adherence to the rules of a prescribed approach for one core practice have if other practices are glossed over or even not used for whatever reason? Perhaps the 12 core practices of XP could even be grouped together into related areas, such as actors (participants in the development effort), technology, structure, and process, and studied from that perspective.

Another potentially critical issue facing software developers and researchers alike is that of software standards. In light of ISO and other standards imposed by governments, implementing organizations, or other regulatory

bodies, the quest to render development methodologies more agile by cutting away or eliminating some of the overhead could become difficult or even virtually impossible since the artifacts of development often become a large part of the documentation requirements.

Theunissen, Kourie, and Watson (2003) looked at the potential adaptability of agile software methodologies with regards to ISO/ISE 12207:1995, among other standards. They found that XP, in particular, could be used to satisfy many of the standard's requirements and developed a set of guidelines for potential users. However, research should be executed regarding whether the guidelines have been successfully adopted and used in practice.

There likely will never be an easy fix for these problems. There are no magic solutions (Germain & Robillard, 2005). In addition, from the extremely high failure rates commonly associated with system-development efforts — estimates range from 50% to 75% — it appears that there should be ample room for improvement in development efforts, whatever shape or form they take. Agile modeling and extreme programming represent a possible step in the right direction if developers have the courage to commit people and resources to the effort and pain involved in managing the changes that will inevitably occur as a result. However, organizations must also practice caveat emptor and clearly state that they cannot embrace a particular development methodology simply because a person or group of people says that it is good.

The proponents of AM and XP have expressed themselves quite clearly and forcefully on the subject of agile modeling and programming. And, judging from the current bleak and stony landscape of systems development, it appears that they are correct. Those that know (industry insiders and researchers) simply point to statistics that back up their claim that many of the processes we have used and are using to develop information systems are broken and likely unrepairable. When 60% of a typical system's O&M budget goes toward band-aiding the results of inadequate analysis and development, when two-thirds to three-quarters (depending upon whose statistics you wish to use) of information systems developed can be considered failures in that they do not provide the functionality required, when we have all been taught to build throw-away systems that can often be obsolete before projects are completed, when we systematically exclude from development those who we are building the system for, then it is indeed time to take a step back, look at the mirror, and say, "Just what is wrong with this picture?"

REFERENCES

Abrahamsson, P., Warsta, J., Siponen, M., & Ronkainen, J. (2003). New directions on agile methods: A comparative analysis. In *Proceedings of the 25th International Conference on Software Engineering*, (pp. 244-254).

Aiken, J. (2004). Technical and human perspectives on pair programming. *ACM SISOFT Software Engineering Notes, 29*(5), 1-14.

Alshayeb, M., & Li, W. (2005). An empirical study of system design instability metric and design evolution in an agile software process. *Journal of Systems and Software, 74*, 269-274.

Ambler, S. (2001a). Agile modeling and eXtreme programming (XP). *Agile Modeling.com*. Retrieved from *http://www.agilemodeling.com/essays/agileModelingXP.htm*

Ambler, S. (2001b). Debunking eXtreme programming myths. *Computing Canada, 27*(25), 13.

Ambler, S. (2001c). Values, principles and practices equal success. *Computing Canada, 27*(10), 11.

Ambler, S. (2001-2002). The principles of agile modeling (AM). *Agile Modeling.com*. Retrieved from *http://www.agilemodeling.com/principles.htm*

Ambler, S. (2002a). Agile development best dealt with in small groups. *Computing Canada, 28*(9), 9.

Ambler, S. (2002b). Know the user before implementing a system. *Computing Canada, 28*(3), 13.

Aoyama, M. (2000, November-December). Web-based agile software development. *IEEE*

Software, 56-65.

Armano, G., & Marchesi, M. (2000). A rapid development process with UML. *ACM SIAPP: Applied Computing Review, 18*(1), 4-11.

Aydin, M. N., Stegwee, R., Harmsen, F., & van Slooten, K. (2005). On the adaptation of an agile information systems development method. *Journal of Database Management, 16*(4), 24-40.

Beck, K. (1999). *Extreme programming explained: Embrace change*. Boston: Addison-Wesley.

Booch, G., Rumbaugh, J., & Jacobson, I. (1999). *The unified modeling language user guide*. Boston: Addison-Wesley.

C3 Team. (1998). Chrysler goes to "extremes." *Distributed Computing*, 24-28.

Cockburn, A., & Williams, L. (2000). The costs and benefits of pair programming. In *Proceedings of the XP 2000 Conference*, (pp. 1-11).

Erdogmus, H., & Williams, L. (2003). The economics of software development by pair programmers. *The Engineering Economist, 48*(4), 283-319.

Erickson, J., & Siau, K. (2003). UML complexity. In *Proceedings of the Systems Analysis and Design Symposium*, Miami, FL.

Erickson, J., & Siau, K. (2004). Theoretical and practical complexity of unified modeling language: A Delphi study and metrical analyses. In *Proceedings of the International Conference on Information Systems*, (pp. 183-194).

Extreme Modeling. (2005). Retrieved from *http://www.extrememodeling.org/*

Fowler, M. (2001). The new methodology. *MartinFowler.com*. Retrieved from *http://www.martinfowler.com/articles/newMethodology.html*

Fowler, M., & Highsmith, J. (2001). The agile manifesto. *Agilemanifesto.org*. Retrieved from *http://www.agilemanifesto.org/*

Germain, E., & Robillard, P. (2005). Engineering-based processes and agile methodologies for software development: A comparative case study. *Journal of Systems and Software, 75*, 17-27.

Grenning, J. (2001). Launching extreme programming at a process intensive company. *IEEE Software*, 27-33.

Hedin, G., Bendix, L., & Magnusson, B. (2005). Teaching extreme programming to large groups of students. *Journal of Systems and Software, 74*, 133-146.

Henderson-Sellers, B., & Serour, M. (2005). Creating a dual agility method: The value of method engineering. *Journal of Database Management, 16*(4), 1-23.

Herbsleb, J., & Goldenson, D. (1996). A systematic survey of CMM experience and results. In *Proceedings of ICSE-18*, (pp. 323-330).

Hirsch, M. (2002). Making RUP agile. In *Proceedings of SIG Programming Languages: OOPSLA*, (pp. 1-8).

Jeffries, R. (2001). What is extreme programming? *XP magazine*. Retrieved from *http://xprogramming.com/xpmag/whatisxp.htm*

Karlsson, E., Andersson, L., & Leion, P. (2000). Daily build and feature development in large distributed projects. In *Proceedings of the International Conference on Software Engineering*, (pp. 649-658).

Kuppuswami, S., Vivekanandan, K., Ramaswamy, P., & Rodrigues, P. (2003). The effects of individual XP practices on software development effort. *ACM SIG Software Engineering Notes, 28*(6), 1-6.

Lindstrom, L., & Jeffries, R. (2004). Extreme programming and agile software development methodologies. *Information Systems Management, 24*(3), 41-60.

Manhart, P., & Schneider, K. (2004). Breaking the ice for agile development of embedded software: An industry experience report. In *Proceedings of the 26th International Conference on Software Engineering*, (pp. 378-386).

Merisalo-Rantanen, H., Tuunanen, T., & Rossi, M. (2005). Is extreme programming just old wine in new bottles: A comparison of two cases. *Journal of Database Management, 16*(4), 41-61.

Müller, M., & Padberg, F. (2003). On the economic valuation of XP projects. In *ACM SIGSOFT Software Engineering Notes: Proceedings of the Ninth European Software Engineering Conference held jointly with 11th ACM SIGSOFT International*

Symposium on Foundations of Software Engineering, September (Vol. 28, no. 5, pp. 168-177).

Newkirk, J., & Martin, R. (2000). Extreme programming in practice. *OOPSLA*, 25-26.

Palmer, S. (2000). *Feature driven development and extreme programming.* Togethersoft Corporation.

Paulk, M. (2001). Extreme programming from a CMM perspective. *IEEE Software, 18*(6), 19-26.

Pfleeger, S., & McGowan, C. (1990). Software metrics in a process maturity framework. *Journal of Systems and Software, 12*(3), 255-261.

Poole, C., & Huisman, J. (2001, November-December). Using extreme programming in a maintenance environment. *IEEE Software,* 42-50.

Schuh, P. (2001). Recovery, redemption, and extreme programming. *IEEE Software*, 33-40.

Siau, K. (1999). Information modeling and method engineering: A psychological perspective. *Journal of Database Management, 10*(4), 44-50.

Siau, K., & Cao, Q. (2001). Unified modeling language (UML): A complexity analysis. *Journal of Database Management, 12*(1), 26-34.

Siau, K., Erickson, J., & Lee, L. (2002). Complexity of UML: Theoretical versus practical complexity. In *Proceedings of the 12th Workshop on Information Technology and Systems (WITS)*, (pp. 13-18).

Siau, K., Erickson, J., & Lee, L. (2005). Theoretical versus practical complexity: The case of UML. *Journal of Database Management, 16*(3), 40-57.

Siau, K., & Lee, L. (2004). Are use case and class diagrams complementary in requirements analysis? An experimental study on use case and class diagrams in UML. *Requirements Engineering, 9*(4), 229-237.

Siau, K., Wand, Y., & Benbasat, I. (1997). The relative importance of structural constraints and surface semantics in information modeling. *Information Systems, 22*(2/3), 155-170.

Strigel, W. (2001). Reports from the field: Using extreme programming and other experiences. *IEEE Software,* 17-18.

Theunissen, W., Kourie, D., & Watson, B. (2003). Standards and agile software development. In *Proceedings of SAICSIT*, (pp. 178-188).

Turk, D., France, R., & Rumpe, B. (2005). Assumptions underlying agile software development process. *Journal of Database Management, 16*(4), 62-87.

Wake, W. (1999). *Introduction to extreme programming (XP).* Retrieved from *http://xp123.com/xplor/xp9912/index.shtml*

What is refactoring? (2005). Retrieved from *http://c2.com/cgi/wiki?WhatIsRefactoring*

Williams, L., & Kessler, R. (2000a). All I really need to know about pair programming I learned in kindergarten. *Communications of the ACM, 43*(5), 109-114.

Williams, L., & Kessler, R. (2000b). The effects of "pair-pressure" and "pair-learning" on software engineering education. In *Proceedings of the Conference of Software Engineering Education and Training*, (pp. 1-10).

Williams, L., & Kessler, R. (2001). Experimenting with industry's "pair-programming" model in the computer science classroom. *Computer Science Education,* 7-20.

Williams, L., Kessler, R., Cunningham, W., & Jeffries, R. (2000, July-August). Strengthening the case for pair programming. *IEEE Software,* 19-25.

Williams, L., & Upchurch, R. (2001). Extreme programming for software engineering education. In *ASEE/IEEE Frontiers in Education Conference Proceedings*, Reno, NV, October 10-13 (pp. 1-6).

Wills, A. (2001). *UML meets XP.* Retrieved from *http://www.trireme.com/whitepapers/process/xp-uml/paper.htm*

John Erickson is an assistant professor at the University of Nebraska at Omaha. His current research interests include study of unified modeling language as an object-oriented systems development tool, software engineering, and the application of both to minimizing the user-designer communication gap, including research into problems with ERP implementations relating to the user-designer gap.

Kalle Lyytinen is the Iris S. Wolstein Professor at the Weatherhead School of Management at Case Western Reserve University and an adjunct professor at the University of Jyvaskyla. He is also the editor-in-chief of the Journal of AIS. *Kalle was educated at the University of Jyvaskyla, Finland where he studied computer science, accounting, statistics, economics, theoretical philosophy and political theory. He has a bachelor's in computer science and a master's and PhD in economics (computer science). He has published eight books, more than 50 journal articles, and more than 80 conference presentations and book chapters. He is well known for his research in computer-supported system design and modeling, system failures and risk assessment, computer-supported cooperative work, and the diffusion of complex technologies. He is currently researching the development and management of digital services and the evolution of virtual communities. Prior to joining Weatherhead, Kalle was the dean of the Faculty of Technology at the University of Jyvaskyla in Finland. He has held visiting positions at the Royal Technical Institute of Sweden, the London School of Economics, the Copenhagen Business School in Denmark, Hong Kong University of Science and Technology, Georgia State University, Aalborg University, The University of Pretoria, South Africa, and Erasmus University in the Netherlands.*

Keng Siau is a full professor of management information systems (MIS) at the University of Nebraska, Lincoln (UNL). He is currently serving as the editor-in-chief of the Journal of Database Management *and as the book series editor for* Advanced Topics in Database Research. *He received his PhD from the University of British Columbia (UBC), where he majored in management information systems and minored in cognitive psychology. His master' s and bachelor's degrees are in computer and information sciences from the National University of Singapore. Dr. Siau has more than 200 academic publications. He has published more than 75 refereed journal articles that have appeared (or are forthcoming) in journals such as* Management Information Systems Quarterly, Communications of the ACM, IEEE Computer, Information Systems, ACM SIGMIS's Data Base, IEEE Transactions on Systems, Man, and Cybernetics, IEEE Transactions on Professional Communication, IEEE Transactions on Information Technology in Biomedicine, IEEE Transactions on Education, IEICE Transactions on Information and Systems, Data and Knowledge Engineering, Decision Support Systems, Journal of Information Technology, International Journal of Human-Computer Studies, International Journal of Human-Computer Interaction, Behaviour and Information Technology, Quarterly Journal of Electronic Commerce, *and others. In addition, he has published more than 90 refereed conference papers, edited or co-edited 12 scholarly and research-oriented books, edited or co-edited nine proceedings, and written more than 15 scholarly book chapters. He served as the organizing and program chairs of the International Workshop on Evaluation of Modeling Methods in Systems Analysis and Design (EMMSAD) from 1996 to 2005 and on the organizing committees of AMCIS 2005 and AMCIS 2007.*