Extreme Security Engineering: On Employing XP Practices to Achieve “Good Enough Security” without Defining It

Konstantin Beznosov
Electrical and Computer Engineering
University of British Columbia
2356 Main Mall, Vancouver, BC
Canada V6T 1Z4

beznosov@ece.ubc.ca

ABSTRACT
This paper examines practices of eXtreme Programming (XP) on the subject of their application to the development of security solutions. We introduce eXtreme Security Engineering (XSE), an application of XP practices to security engineering, and discuss its potential benefits and the scope of its applicability. We argue that XSE could help achieve “good enough security” while avoiding defining *a priori* what it is.

Categories and Subject Descriptors
K.6.1 [**Management of Computing and Information Systems**]: Project and People Management -- *Life cycle, Systems analysis and design, Systems development, Management techniques*; K.6.4 [**Management of Computing and Information Systems**]: Security and Protection.

General Terms
Security, Design, Economics, Human Factors.

Keywords
Security Engineering, Agile Software Development, Extreme Programming, eXtreme Security Engineering .

1. INTRODUCTION
One may think that as commercial customers slowly realize they want just “good enough”, instead of “absolute”, security, it becomes necessary for a security engineer to define what “good enough security” is. However, rather than attempting to define “good enough security” *a priori*, it could be more productive to let the customer of a security solution be in charge of the definition. Moreover, the customer should have the liberty of adjusting it almost as frequently as they want.

Avoiding complete specification and “freezing” of the requirements up-front in security engineering projects is critical due to insufficient understanding of requirements, their frequent changes, as well as budget changes and changes of other resources. Particularly, it is the author’s industry experience that at the beginning of most projects on enterprise infrastructures or software applications where security is a major, if not the only, objective, the customers usually have very vague idea about what and how much “security” (and other features) they want to get by the end of the project. At the same time, the developers, including security designers and architects, even if they were given complete and precise requirements, are unable to provide realistic estimates about the amount of effort necessary to support all the requirements due to the scale and uniqueness of each project. On top of these restrictions, the projects tend to span multiple fiscal periods and multiple political domains within the customer’s organization, objectively limiting the power of forecasts about the total budget available.

A possible way to provide control of “good enough security” to the customer is through applying the principles of agile software development (ASD) [8, 18] to security development/integration projects, also referred to as “security engineering” in this paper. We also consider other benefits an application of ASD to security engineering could enjoy.

ASD principles deserve particular consideration because of the popularity, even an orthodoxy in a way, they are gaining among practitioners and researchers. Due to its wide acknowledgement among software developers [3, 6, 7, 9, 15], for example, iterative and incremental development (IID) [10], a more conservative version of ASD, even became the recommended form of software development for US DoD contractors in 1994 [11].

Specifically, we consider an application of eXtreme Programming (XP) [2, 14, 17], arguably the most publicized and most documented ASD representative with a catchy name, to security engineering. It practices a specific set of techniques for implementing both ASD’s and its own principles. The ability of the customer in XP projects to adjust the definition of “good enough” software almost at any moment in the development process is especially important since both the budget priorities, driven by the market performance, and the requirements, driven by the business processes, tend to change frequently. On the other hand, the developers do not have to lock themselves into unrealistic long-term promises because they have a chance to estimate the cost of incremental changes to the system each time the customer asks for support of new requirements. Neither do the have to “blow out” the customer’s budget and thus jeopardize the existence of the project. Since XP has been shown [2] to be a successful approach for some commercial projects in development of "good enough software" with frequently changing or even unclear requirements and budgets, one wonders if the XP principles could be successfully applied to the development of "good enough security" solutions.

In this paper, we examine ways to apply XP principles to create "good enough security" solutions without defining what "good enough security" is. We refer to this application of XP to security engineering as eXtreme Security Engineering (XSE). The main contribution of this paper is the introduction of XSE and a discussion of its perceived advantages and disadvantages.

We make a case that the XSE approach could be successfully applied to security engineering projects to achieve "good enough security" as well as to improve project success rates and overall customer satisfaction. The range of XSE applicability to different kinds of projects is expected to be similar to the one of XP and other ASD approaches.

The rest of the paper is organized as follows. XP overview is provided in Section 2. We describe XSE approach in Section 3. Section 4 contains discussion of the proposed approach. Summary and conclusions are drawn in Section 5.

2. XP OVERVIEW

Along with other ASD approaches, XP addresses the following important problems faced by software development projects:

- The resulting system solves the wrong problem (requirements not met).
- The resulting system is out of date before it is in use (requirements change).
- Software quality is so poor that the system cannot be used.

To deal with the empirical nature of software development, all ASD methods rest on two cornerstone principles: the short "inspect-and-adapt" development cycles and the short feedback loop.

2.1 PRINCIPLES

In addition, here is a selection of other key principles that ASD methodologies, including XP, follow (further details are available at www.agilemanifesto.org):

- Customer satisfaction is achieved through early and continuous delivery of valuable software.
- Changes in requirements or business environment are embraced instead of being ignored or mitigated.
- Working software is delivered frequently.
- Customers and developers work together daily throughout the project.
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
- Working software is the primary measure of progress.
- Participating customers and developers should be able to maintain a constant pace indefinitely.
- Simplicity -- the art of maximizing the amount of work not done -- is essential.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

2.2 PRACTICES

XP carries out the above principles through its own practices, the most important of which are summarized below (adopted from [2]):

Small Releases. The system is put into production in a few months, before solving the whole problem. New releases are made often — anywhere from daily to monthly. The customer picks the next release by choosing the most valuable features (called "user stories" in XP) from among all the possible stories, as informed by the costs of the stories and the measured speed of the team in implementing stories.

Planning game. Customers decide on the scope and timing of releases based on estimates provided by programmers. Developers implement only the functionality demanded by the user stories in this iteration.

Metaphor. The shape of the system is defined by a metaphor or set of metaphors shared between the customer and the developers.

Simple Design. The system is designed in such a way that the design communicates everything the developers want to communicate, contains no duplicate code, and has the fewest possible classes and methods. The correctness of the design is insured by frequent testing.

Tests. The developers write unit tests minute by minute. Organized in test suites, these tests must all run successfully to maintain confidence in the correctness of the system design. Customers write functional tests for the stories in an iteration. These tests should also all run, although practically speaking, sometimes a business decision must be made by comparing the cost of shipping a known defect and the cost of a delay.

Refactoring. The design of the system evolves through transformations of the existing design that keep all the tests running.

Pair Programming. All production code is written by two people at one screen/keyboard/mouse.

Continuous Integration. The new code is integrated into the current system after no more than a few hours. When integrating, the system is built from scratch and all tests must pass or the changes are discarded.

Collective Ownership. Every programmer improves any code anywhere in the system at any time if they see the opportunity.

On-site Customer. The customer is present onsite and works with the team full-time.

40-hour Weeks. No one can work a second consecutive week of overtime. Even isolated overtime, when used too frequently, is a sign of deeper problems that must be addressed.

Open Workspace. The team works in a large room with small cubicles around the periphery. Paired programmers work on computers set up in the center.

Just Rules. By being part of an eXtreme team, you sign up to follow the rules. But these rules are not set in stone. The team can change the rules at any time as long as they agree on how they will assess the effects of the change.

Although user stories (or just "stories") are not identified as a separate principle, they deserve special attention because they are the results of requirements analysis in a short form of use cases. A user story is so short that it can fit on an index card. Each story is business-oriented, testable, and estimable. Although not mandatory, the use of index cards makes it easy for the customer to prioritize the stories, to select those that should be implemented in the next release or iteration, and to get a nice warm feeling when the cards are moved from the pile of desired stories into the pile of implemented.

2.3 HOW IT WORKS

By using small releases and even shorter iterations within releases, XP leaves it up to the customer of a software system to make some important decisions; they include particular (i.e., usually the most important) functional elements and non-functional properties that should be added to the system during the next iteration. Based on their current understanding of the budget and the requirements as well as the experience of using the previous release of the system, the customer decides what feature(s) will make the system good enough when its new release will become available in next several weeks or months. The customer describes the required features in user stories written on index cards.

The definition of these new features is achieved through the development of functional tests that check for presence of the asked features. The customers and developers work together on defining the tests thus insuring common understanding of the new features. Then the developers determine the tasks necessary to implement the stories for the current iteration and estimate amount of effort for performing each task as well as all the stories in the current iteration.

After the customer and the developers are happy with the total effort estimated for the iteration, and the stories to be supported after the iteration, the developers choose pairs and start creating unit tests that, once passed successfully, would indicate the completion of the corresponding tasks. While working on the tasks, the developers constantly (re)design, a.k.a., refactor, the necessary parts of the system. Before each task is completed, the system is rebuilt and re-integrated into the customer's environment. Since the development cycles are short, by the time a new system release is ready, the requirements and the budget could not have changed dramatically. After a release, which consists of several iterations, is completed, the updated system is put into use and the customers can decide if the updated requirements and budget warrant the next release, and, if so, what new most important feature(s) should be added to the system.

The stability of the system in the face of frequent iterations and small releases, and confidence in the measurable quality of the system are achieved via XP flavor of testing. A set of automated tests can clearly show if the new/old features have been added/deleted to/from the system. Test automation allows the creation of integrated test suites that check for the presence of all the features requested by the customers so far, leading to the stability of the system in the presence of frequent changes. Functional tests developed by the customer as well as unit tests implemented by the developers are the XP keys to the implementation of a short feedback loop.

During the process of developing a software system using XP methodology, each development increment is locally optimized, similarly to the Greedy Algorithms [1] approach, in the hope that some globally optimal solution will be achieved at the end. However, it is well known that in worst case scenarios, further addition of new (non) functional features could require dramatic changes to the system architecture. For example, belated introduction of a requirement for the support of "undo" operation could require a costly replacement of the transaction sub-system in a distributed application. This example also illustrates a point of view that ASD/XP are not universal solutions applicable for all software development projects. In some cases, however impractical it may sound, agreeing on all requirements at the beginning of a project could increase its chances of success.

3. EXTREME SECURITY ENGINEERING

Like software development, security engineering cannot be considered a *defined* [12] process because the budget, requirements, and technologies undergo too much change while a security solution is being developed. Instead of attempting to reduce the amount of change in a project, security engineers could benefit, as in ASD, from embracing frequent changes by employing short "inspect-and-adapt" cycles and frequent, short feedback loops, which are the necessities of *empirical* engineering processes [15] and the foundations of XP and other ASD methods.

Extreme Security Engineering (XSE) is an adoption of ASD principles (Section 2.1) in general and XP practices (Section 2.2) in particular to security engineering. The question is how XP practices can be applied to security engineering? In this section we discuss one probable way of adopting them while acknowledging that there could be other possibilities.

Only when applied together, all XP practices, are responsible for the cumulative effects seen in the software development projects where XP is applied. Researchers recently started attempts to measure the effects of each individual practice (or a subset of them) applied in isolation (e.g., pair programming by Williams and Cockburn [19]). Nonetheless, to simplify the discussion in this section, we consider the adoption of each of the major XP practices and other elements to security engineering separately. The reader, however, needs to keep in mind that the intent is to apply them in XSE all together.

Although we support such practices as Collective Ownership, 40-hour Week, Open Workspace, and Just Rules, we do not discuss here their applicability to security engineering due to their general role in the productivity of any workplace and the space constraints for this paper.

3.1 PLANNING GAME

Similarly to XP, the planning game's objective in XSE is to schedule small releases and short iterations in such a way that the project can continue with a sustainable rate while delivering "good enough security" in the form of most valuable tested units of functionality that make business sense to the customer.

To achieve the objective, the planning game has a set of rules that allows technical people to make the technical decisions and business people to make the business decisions. The development team estimates each user story in terms of ideal development weeks. The customer then decides which subset of stories is most important and, when implemented, would make the security solution usable and testable. Measured in the previous iterations, the project velocity is used to estimate either how many stories can be implemented before a given date or how long a set of stories will take to finish. Stories included in the upcoming release/iteration and completion dates are negotiated until the developers, customers, and managers can all agree upon the release/iteration plan. Each release/iteration planning is performed just before it begins and not in advance.

3.2 USER STORIES

It is the author's experience that failure usually occurs when conventional, plan-driven, ways are used to engineer all security requirements upfront. It is not a surprise. Due to their negative nature and highly technical level of security functionality in systems, the corresponding requirements tend to be more vague and confusing than requirements for other aspects of a system or infrastructure. Both security developers and customers regard the engineering of security requirements as a painful and non-productive process. It often results in a huge collection of outdated items by the time they are engineered. In addition, these items are in relationships and have priorities that both sides have a hard time to comprehend.

Applying XP user stories to security engineering, on the other hand, could allow customers to use familiar business-like language and capture, in the form of simple stories, what they want to see when the security solution is implemented. The opportunity for customers to put on a table 50-100 cards with user stories and then decide which stories have the highest priorities (and therefore should be implemented in the next iteration) could be a key enabler of "good enough security" without security engineers having to define what it is exactly. Using cards with desirable scenarios written in a plain language could only make an incremental improvement in the overall success rate of security projects, though.

What should make the difference is the combination of user stories with other techniques adopted from XP. Most important of them are small releases with short iterations and testing. Small releases provide frequent opportunities for customers to update and re-prioritize the user stories. Thus the definition of "good enough security," specified by the customer through required user stories, could fluctuate with every iteration reflecting shifts in both the technology and the business environment. Despite regular changes, due to frequent testing, both the security engineers and the customer can be confident that the security solution supports all implemented stories.

3.3 SMALL RELEASES

Small releases with short iterations provide the foundation that supports other XP practices. They are so short that neither the budget nor the requirements can drastically change in between, enabling the production of a working system that meets its requirements at the end of each iteration and release. Early critical feedback from the customer is another benefit of small releases.

Security engineering projects, which often combine custom software development with integration of COTS products and hardware systems, as well as procedural changes in the customer's organization, could be difficult to "slice" into 1-2 week iterations and even into 1-2 month releases. Yet, the benefits from short iterations, not necessarily of same length, could make it worth the extra effort.

Like in software development, the use of small releases with short iterations, combined with other practices, is expected to provide the following benefits:

- The delivery of even a partially working solution (or parts of the solution) at the end of each iteration allows the customer to have a clear understanding of what is actually being created and make better choices of high-priority user stories to be implemented in next iteration.
- Early and frequent feedback from the customer significantly helps security engineers "drive" the development efforts always in the direction that seems to be optimal for the customer.
- Security engineers could use measurements from previous iterations to make more precise estimates regarding the effort for further changes and additions to the solution being developed.
- Changes in the business environment, budget priorities, the customer's political landscape, and technology are easier to accommodate.

"Slicing" a security solution in small releases and short iterations, although more difficult, is similar to dividing a pure software project. In most security engineering projects, it is possible to identify small units of functionality that make good business sense and can be released into the customer's environment early in the project. For example, a directory infrastructure, the backbone of identity management and access control services in information enterprises, could be released first, just for using it as an electronic phone book by the employees, before it is employed in the security solution.

One of the obvious complications with small releases and short iterations applied to the feedback-driven development is the difficulty of maintaining traditional plan-based contractual relationships between customers and developers. Since neither side can claim the knowledge of the project's exact result, it is hard to negotiate a contract with the precise amount of work done and money paid. Alternatively, "body shop"-like relationships, where the developers are paid for the amount of time they spend on the project, although they seem to be more suitable for XSE, have well known drawbacks, where time-oriented, instead of result-oriented, nature of work is the biggest one. However, this issue should probably be left to the specialists in business management.

Another perceived shortcoming of small iterations combined with XP testing and continuous integration is the amount of effort spent on making incremental changes to the system. Like with XP, those who try to deliver the working system in several week iterations find too much overhead due to the frequent runs of test suites and the integration of the changed parts. True, short iterations require a meticulously organized development, testing, and integration environment including custom automation scripts and redundant resources, in order to avoid prohibitively painful delivery of each release. However, the payoffs could be significant: the opportunity for the customer to see alive and running what has been developed so far, the discovery of unexpected integration and deployment problems at the beginning of the project instead of the end, and the confidence that the developers are delivering a solution that will work in the customer's environment.

3.4 TESTING

Tests define what "good enough" security solution is and help to gain confidence in its quality as well as functionality. Written in plain business-like language, user stories cannot be used directly by security engineers as requirements to be implemented. Instead, each story translates into one or more functional test cases, developed by the customers themselves. Reaching the point when all the tests run smoothly is the indication for security engineers that they are done with the functionality. If particular features cannot be implemented as planned, failed tests help identify the missing parts.

In addition to functional tests, security engineers develop unit tests to control the quality of the developed parts (e.g., LDAP access to the directory service, smart card authentication devices, the authorization server for Web Services) and catch regressions. Furthermore, unit tests communicate the intent of the design, which is independent of the implementation details.

Due to the diversity of the technology and products commonly utilized in security solutions, unit test frameworks would require more effort to implement then in pure software development projects. Nonetheless, the significant payoff from XP testing practice reported by Beck in, although anecdotic, success stories [2] gives hope that similar testing adopted by XSE could also result in good quality and stability of security solutions in the face of frequent changes.

Another difficulty with testing in XSE could be due to the negative nature of security properties (e.g., lack of means to bypass the enforcement function in an access control mechanism), which makes them difficult to test. However, an application of XP practices to security engineering neither alleviates nor exacerbates this concern.

3.5 CONTINUOUS INTEGRATION

In some security engineering projects, integration is the dominating portion of the effort and the main activity. Still, following the XP strategy of evolutionary increase in the system functionality and continuous integration even in those projects could create some valuable payoff. Integrating the solution with the customer's environment earlier allows for avoiding the dangerous anomaly in the life of any security development project – the period before a system first goes into production.

What's more, continuous integration, combined with XP testing and short iterations, aids to discovering the differences between the customer's environment and the security engineers' understanding of it.

The main challenge in adopting this XP practice to security engineering is due to the difficulty (in some projects) of creating staging environment where early versions of the solution are deployed. Such an environment could be prohibitively costly because of expensive hardware and other non-software elements, or just hard to recreate (e.g., due to the need to process confidential data or to provide physical security).

3.6 SIMPLE DESIGN AND REFACTORING

Practicing simple initial design that evolves through frequent refactoring is essential for balancing the principles of early, continuous delivery and embracing unexpected, frequent changes in security engineering projects. The avoidance of collecting and fixing all requirements upfront is another driving force.

As with software projects, security engineers need to adhere to Einstein's principle of making everything "as simple as possible but not simpler" to stay away from the danger of simplistic design.

We expect frequent refactoring to be difficult to realize in those security projects, which include inflexible and costly non-software or COTS components.

3.7 PAIR DEVELOPMENT

If security engineering is similar enough to software development, pair development could have similar impact on security development projects. Recent research on the costs and benefits of pair programming [19] shows that software products can be produced in less time and with higher quality. Furthermore, the majority of programmers involved in the studies or surveyed in industry seem to enjoy the development process and feel more confident about the results of their work, when they work with a partner. Although not based on any evidence, our expectation is that security engineers, like software developers, working in pairs could produce higher quality results, possibly at the price of slightly lower performance.

3.8 ON-SITE CUSTOMER

The customer's time is distributed differently in XSE projects than in traditional plan-driven ones. It is spared initially by not requiring a detailed requirements specification and saved later by not delivering an uncooperative solution. Instead, the customer representative is actively involved in the development process on site and is responsible for:

- writing user stories,
- negotiating user stories to be included in each scheduled release,
- clarifying and possibly refining user stories for the developers while they are working on implementing the stories,
- providing additional details for the security engineers to complete development tasks,
- developing (or helping with) functional tests as well as defining the input and output data for the tests.

4. DISCUSSION

An adoption of XP practices to security development projects, XSE is meant to aid the projects developed for business customers with achieving "good enough security" without defining *a priori* what it is. Other important benefits from practicing XP techniques in XSE are expected to be increased customer satisfaction, lower defect rates, faster development times, and a way to handle rapidly changing requirements.

4.1 Can XSE Succeed?

The idea of applying XP practices to security engineering, outlined in this paper, has not been tried explicitly as of time of writing. Therefore, there is no direct evidence in support of our expectations of the overall benefit of XSE. However, the author's personal industrial experience from participation in commercial software development and security development projects, and the evidence collected in other fields raise a certain hope.

First, the way commercial software and security engineering are practiced today makes these two disciplines similar. For example, consider the following paraphrased list of reasons why waterfall, a.k.a., plan-driven, approach with complete specification and "freezing" of system requirements is impractical:

- A system's users seldom know exactly what they want and cannot articulate all they know.
- Even if we could state all requirements, there are many details that we can only discover once we are well into implementation.
- Even if we knew all these details, as humans, we can master only so much complexity.
- Even if we could master all this complexity, external forces lead to changes in requirements, some of which may invalidate earlier decisions.

The above points could very well be made about security engineering, although they were stated by Parnas and Clements in a discussion regarding software development only [13].

Another reason to believe that the two disciplines are similar enough is based on the personal experience of the author in both software development and security engineering projects. The experience indicates that:

- Both software and security development are *empirical* engineering processes [12] necessitating short "inspect-and-adapt" cycles and frequent, short feedback loops [15].
- The notion of "good enough" varies significantly from project to project in both disciplines.
- Waterfall-like plan-driven approaches to the development of both software and security solutions fail repeatedly.
- Commercial customers are not any more in a position to specify and, most importantly, "freeze" all requirements upfront.

The similarity could be exploited to apply those approaches that are successful in one discipline to the other.

Second, the experience of other engineering fields shows that some of the interactive and incremental development principles are successfully applied to non-software manufacturing. Examples of companies that used IID approaches for non-software products in the 1980s are Honda, Canon, and Fujitsu [16].

The author's personal experience from industry, the success of IID approaches in manufacturing, as well as anecdotic and scientific evidence of ASD benefits in software development provide the base for the belief in XSE as a better methodology for some security engineering projects.

4.2 What Kinds of Projects Could XSE Be Suitable For?

Obviously, not every security engineering project should be expected to benefit from XSE. Due to the novelty of XSE, one can only extrapolate the scope of XP or, more broadly, ASD applicability to XSE. By 2001 enough experience with ASD approaches has been collected to identify their scope as "nonsafety-critical projects with volatile requirements, built by relatively small and skilled collocated teams" [18].

Boehm and Turner [4, 5] suggest a finer demarcation between agile and plan-driven approaches in a five-dimensional space: size, criticality, dynamism, personnel, culture. Even more, they present a risk-based method for structuring projects to incorporate both agile and plan-driven approaches according to a project's needs. If security and software engineering are sufficiently similar disciplines, the method of Boehm and Turner could be applicable to security engineering projects as well.

4.3 What Needs to be Done for XSE to Succeed?

To complete the adoption of XP to the domain of security engineering, a number of adjustments and changes are necessary. They are all related to the incremental nature of XSE.

We see a need for developing techniques for the incremental risk analysis, including vulnerabilities analysis, as well as the incremental testing of security properties. The techniques are necessary in order to reduce the cost of short iterations and small releases. The XP process is supported by unit test suites implemented for different languages in the form of third-party libraries. They enable simple and easy ways to develop and add new test cases with each new feature in an incremental fashion. Ideally, similar solutions are necessary for the risk and vulnerability analyses.

Unlike most other properties, security properties are negative (e.g., protection from unauthorized access), which makes their testing hard. It is even harder is to perform incremental testing of these properties. On the other hand, the ability to do such testing in an incremental and inexpensive fashion is critical when XSE is employed.

5. SUMMARY AND CONCLUSIONS

In this paper, we attempted to bring to the attention of the security engineering community the fact that such a related discipline as software engineering is experiencing a turn from plan-driven to iterative and incremental development, a.k.a., ASD, approaches. Furthermore, we proposed eXtreme Security Engineering (XSE), an adoption of eXtreme Programming (XP) practices to security engineering projects.

In addition to the prospect of achieving "good enough security," XSE could improve the project success rate and overall customer satisfaction. It remains to be seen whether the XSE approach could be successfully applied to security engineering projects. However, the similarity between software and security engineering disciplines and the history of the positive application of ASD methods to software and non-software manufacturing raise certain hope. The range of XSE applicability to different kinds of projects is expected to be similar to the one of XP and other ASD approaches.

The next steps are to try XSE out and validate it through experimental as well as real-world projects.

6. ACKNOLEDGEMENTS

Participants of Software Engineering Exploration and Discussion Seminar (SEEDS) at the University of British Columbia, especially Paul Davies, Lee Iverson, Jeff Joyce, Philippe Kruchten, and Guy Lemieux, provided valuable feedback on this paper ideas from the perspective of software engineering. It is the feedback from Philippe Kruchten that influenced the author's decision to change the name of the approach from Agile Security Engineering (the way it was referred to in BizSec '03 workshop pre-proceedings) to eXtreme Security Engineering. The change reflects the fact that the majority of the XP practices have been adopted by the approach. Copyediting comments from Tatiana Teslenko made this paper much more readable.

7. REFERENCES

[1] Aho, A. V. and Hopcroft, J. E., The Design and Analysis of Computer Algorithms, 1st ed., Addison-Wesley, 1974.

[2] Beck, K., "Embracing Change with Extreme Programming", IEEE Computer, vol. 32, no. 10, October 1999, pp. 70-77.

[3] Boehm, B., "Get Ready for Agile Methods, with Care", IEEE Computer, vol. 35, no. 12, January 2002, pp. 64-69.

[4] Boehm, B. and Turner, R., Balancing Agility and Discipline: A Guide for the Perplexed, Addison-Wesley, 2003.

[5] Boehm, B. and Turner, R., "Using Risk to Balance Agile and Plan-Driven Methods", IEEE Computer, 2003, pp. 57-66.

[6] Cockburn, A. and Highsmith, J., "Agile Software Development: The People Factor", IEEE Computer, vol. 34, no. 11, November 2001, pp. 131-133.

[7] Crocker, R., Large-Scale Agile Software Development, Addison-Wesley, 2003.

[8] Highsmith, J., Agile Software Development Ecosystems, Addison-Wesley Professional, 2002.

[9] Highsmith, J. and Cockburn, A., "Agile Software Development: The Business of Innovation", IEEE Computer, vol. 34, no. 9, September 2001, pp. 120-122.

[10] Larman, C. and Basili, V. R., "Iterative and Incremental Development:A Brief History", IEEE Computer, vol. 36, no. 6, June 2003, pp. 47-56.

[11] Newberry, G. A., "Changes from DOD-STD-2167A to MIL-STD-498", Crosstalk: The Journal of Defense Software Engineering, April 1995.

[12] Ogunnaike, B. A. and Ray, W. H., Process Dynamics, Modeling, and Control, Oxford University Press, 1994.

[13] Parnas, D. and Clements, P., "A Rational Design Process: How and Why to Fake It", IEEE Transactions on Software Engineering, vol. 19, no. 2, February 1986, pp. 251-257.

[14] Paulk, M., "Extreme Programming from a CMM Perspective", IEEE Software, vol. 18, no. 6, November/December 2001, pp. 19-26.

[15] Schwaber, K. and Beedle, M., Agile Software Development with SCRUM, Prentice Hall, 2002.

[16] Takeuchi, H. and Nonaka, I., "The New New Product Development Game", Harvard Business Review, January 1986, pp. 137-146.

[17] Williams, L., "The XP Programmer: The Few-Minutes Programmer", IEEE Software, vol. 20, no. 3, May/June 2003, pp. 16-20.

[18] Williams, L. and Cockburn, A., "Agile Software Development: It's about Feedback and Change", IEEE Computer, vol. 36, no. 6, June 2003, pp. 39-43.

[19] Williams, L., Kessler, R. R., Cunningham, W., and Jeffries, R., "Strengthening the Case for Pair-Programming", IEEE Software, vol. 17, no. 4, July/August 2000, pp. 19-25.