Towards Understanding Quality Assurance in Agile Software Development

Article · January 2005

CITATIONS
14

READS
2,848

3 authors:

Juha Itkonen
Aalto University
39 PUBLICATIONS **1,615** CITATIONS
SEE PROFILE

Kristian Rautiainen
Aalto University
28 PUBLICATIONS **543** CITATIONS
SEE PROFILE

Casper Lassenius
Aalto University
137 PUBLICATIONS **6,202** CITATIONS
SEE PROFILE

ICAM 2005 International Conference on Agility
Helsinki July 27–28, 2005

Towards Understanding Quality Assurance in Agile Software Development

Juha Itkonen, Kristian Rautiainen, and Casper Lassenius
Software Business and Engineering Institute, Helsinki University of Technology,
P. O. Box 9210, 02015 HUT, Finland
firstname.lastname@hut.fi

Agile software development stresses individuals and interaction, customer collaboration, short development cycles and frequent deliveries of valuable software. From the testing and quality assurance viewpoint these principles are challenging, and agile methods seem to lack aspects that traditionally are considered important and fundamental to successful quality assurance. In this paper we identify these theoretical challenges and shortcomings in agile methods. We describe the quality assurance practices of four agile methods and show that agile methods emphasise constructive quality building practices. Quality evaluating practices, based on a destructive attitude, are few, if any. We think that agile development processes could benefit from the introduction of additional testing practices, and as examples of such practices we propose the role of an independent tester and the session-based exploratory testing approach.

Keywords: quality assurance, time pacing, time horizon, agile software development, software testing

1. INTRODUCTION

Agile software development methods are based on iterative and incremental development (IID) using short, often time-boxed, development cycles. The highest priority of agile methods is to satisfy the customer through early and continuous delivery of valuable software. The fundamental values of agile software development emphasise individuals and interactions, customer collaboration, and responding to change. Less emphasis is placed on processes, tools, documentation, and following a plan, which traditionally have been among the cornerstones of rigorous quality assurance and testing practices. [13, 15]

Quality assurance (QA) — all activities and practices used to ensure software product quality as a part of the development process — is a crucial component of most software development efforts. Testing, which is one part of QA, is the process of analysing a software item to detect the differences between existing and required conditions, and of evaluating the features of the items [17]. Testing approaches have traditionally included a set of activities concentrated to an integration and testing phase at the end of the development project [6, 22, 28]. This testing phase often has unpredictable length and effort requirements. Clearly, time-boxed agile projects require a different approach to quality assurance and testing. However, most agile methods do not say much about testing. Several of the methods include a set of good practices for developers, including automated unit testing. Still, only a few methods give any guidance for higher test levels than unit and integration testing [2].

This raises the question: is quality assurance sufficiently covered in agile methods? Some authors have identified problems in the quality assurance practices of extreme programming [11, 21, 30, 31]. This paper aims at increasing the understanding of QA in agile software development by identifying challenges and shortcomings in agile methods considering the traditional ideas and principles of QA.

The paper is structured in the following way. First, we describe the research methodology. Second, we contrast testing in agile methods with traditional ideas and principles of testing, revealing theoretical challenges and shortcomings. Then we present the Cycles of Control framework [27] and use it to identify quality assurance practices in existing agile methods. Based on our findings, we discuss the possibility to enhance quality assurance in agile software development. The paper ends with conclusions and suggestions for further work.

2. RESEARCH METHODOLOGY

The main inspiration for this research stems from the work we have performed in close cooperation with 11 small product-oriented software companies for four years. In these companies we have used action research to process improvement, successfully applying a temporal pacing framework [25, 26, 32]. During this work we have observed quality assurance rise as a critical issue that is currently poorly understood in the companies. All the companies have adopted practices from agile methods in their software development processes. These practices provided solid low-level developer practices, but testing the software products remained challenging. In an attempt to better understand quality

assurance in agile methods, we performed a literature study to contrast agile quality assurance with traditional quality assurance, with an emphasis on testing. This revealed theoretical challenges and shortcomings in quality assurance in agile methods. To further understand these challenges and shortcomings, we used the temporal pacing framework to identify the QA practices in existing agile methods on the heartbeat, iteration, and release time horizons.

3. CONTRASTING AGILE AND TRADITIONAL TESTING

Agile methods follow an iterative and incremental development process. Short, often time-boxed, iterations provide a mechanism for rapid feedback during development and enable frequent control points in which the development plans can be revised. This rapid pace creates challenges for quality assurance, especially testing. In the following subsections we contrast agile and traditional testing by first looking at how the agile principles cause challenges from a traditional testing viewpoint and then looking at how traditional testing principles reveal shortcomings in agile testing. We use the term *agile testing* as short for testing in agile methods. The term *traditional testing* refers to traditional ideas and principles of QA and testing.

3.1. Challenges in agile testing

The main principles of agile software development are described in the agile manifesto [13]. These principles describe the ideas that are common for all agile development methods. From the traditional testing viewpoint these principles bring forth several challenges. Table 1 summarises the main challenges that the agile principles place on testing.

First, the highest priority of agile development is to deliver valuable software to customers early and continuously with a rapid release cycle. This is a challenge for testing because the rapid release cycle puts fixed deadlines for testing activities and does not allow extending the testing period if more defects are found than estimated.

Table 1 Challenges that agile principles place on testing

Agile Principle	Challenge
Frequent deliveries of valuable software	- Short time for testing in each cycle - Testing cannot exceed the deadline
Responding to change even late in the development	- Testing cannot be based on completed specifications
Relying on face-to-face communication	- Getting developers and business people actively involved in testing
Working software is the primary measure of progress	- Quality information is required early and frequently throughout development
Simplicity is essential	- Testing practices easily get dropped for simplicity's sake

Second, agility demands that we should welcome changing requirements even in late stages of development. Testing and static QA methods have traditionally been based on specifications that are completed in a certain phase of development and after that point can be used as a basis for test design and other QA activities. If these documents are allowed to change even in late phases of the development cycle, it clearly challenges the traditional way of doing QA.

Third, agile development relies on conveying information through face-to-face conversation, and business people and developers must work together daily throughout the project. This means that the documentation that traditional testing is based on, does not necessarily exist. The detailed information about the expected results of the tests is in the heads of the developers and the business people.

Fourth, working software is the primary measure of progress. This means testing cannot be left as a phase in the end of an iteration since it must provide information on achieved quality early and promptly in order to enable evaluating if the produced code actually is working software or not.

Fifth, simplicity — the art of maximising the amount of work not done — is essential. This principle makes it challenging to keep necessary QA practices included in the organisation's development process, because the QA activities are easily seen as unnecessary and unproductive because they do not directly add value in terms of code and features.

3.2. Shortcomings in agile testing

Traditional quality assurance in software development has been based on generally accepted principles that have not gained much focus in the agile software development community. In this section we describe these fundamental principles focusing on testing principles. Table 2 summarises the traditional testing principles and the contradictions we have identified in agile testing with respect to those principles. These contradictions can be seen as potential shortcomings.

Table 2 Traditional testing principles and contradicting practices in agile methods

Testing principle	Contradicting practices in agile methods
Independency of testing	- Developers write tests for their own code - The tester is one of the developers or a rotating role in the development team
Testing requires specific skills	- Developers do the testing as part of the development - The customer has a very important and collaborative role and a lot of responsibility for the resulting quality
Oracle problem	- Relying on automated tests to reveal defects
Destructive attitude	- Developers concentrate on constructive QA practices, i.e., building quality into the product and showing that features work
Evaluating achieved quality	- Confidence in quality through tracking conformance to a set of good practices

One of the fundamental principles of testing is its *independency*. Myers states that programmers should avoid testing their own programs and that a programming organisation should not test its own programs [22]. In agile methods, the emphasis is mostly on developer-level practices, and unit and integration level testing by automated tests written by the developer. This is problematic, because it is hard to see problems in one's own code, and, more importantly, developers own tests do not reveal possible misunderstandings of the specifications or requirements. A separate tester role exists in extreme programming (XP) [5, 9], but the tester is still part of the development team. Crystal Clear defines the tester as a role rotating among developers, whose main task is reporting bugs [8].

Software testing is a creative and intellectually challenging task that requires a lot of specific skills and experience. It is a profession and requires a professional tester in order to be performed effectively and efficiently [19, 20, 22]. In agile methods, testing is usually seen as a task that developers do as part of the development tasks or as a task of the customer. In many agile methods, the customer is very tightly involved in everyday development and holds the responsibility for acceptance testing. This has been identified as a problem in [11, 21, 31]. Having testing as a customer task can work if the customer is capable of acting in a tester's role and has the skills and expertise to take care of testing. The DSDM method recognises the need for testing skills and Stapleton recommends that at least one lead developer or tester in each team has received training in testing [29].

The so-called oracle problem is one of the basic problems of software testing. The term *oracle* refers to the principles that are used for finding the correct result of the test. This is not a trivial problem, and professional testers find it very important to thoroughly inspect the results of each test in order to notice the defects that the test reveals [7, 22]. Also, in the test automation literature this is recognised as one of the hardest problems when automating tests [12]. In many agile methods, a lot of, if not all responsibility for testing is placed on the automated tests that are written by developers. From the viewpoint of a professional tester it is not obvious that these automated tests would be very effective in revealing defects in the code.

A destructive attitude to software testing can make testing more effective. Traditionally, the purpose of testing a program is to find problems in it: "Testing is the process of executing a program with the intent of finding errors" [22]. Test cases must be written for invalid and unexpected, as well as for valid and expected input conditions. In agile methods, where developers test their own code, this tester's attitude is very hard to achieve. Agile methods focus on constructive QA practices, i.e., building quality into the product. The agile literature describes the practices more as a means of showing that features work and demonstrating their benefits, rather than revealing the defects and problems in the code. This is problematic, because even though all the unit tests pass, the system may still be broken [11]. If you want and expect a program to work, you are more likely to see a working program, making you miss failures [19, 20].

Software testing includes more than finding defects. The purpose of testing is to provide information and identify defects and problems in a product to evaluate and improve the achieved quality [16]. Evaluating product quality requires metrics, e.g., the number of found faults, fault types, fault classification, and test coverage. In agile methods, testing activities are mostly based on conforming to certain good practices and tracking that these practices are followed. This does not provide direct information on product quality and does not enable evaluating the achieved quality. For example, automated unit tests that developers write for all functionality and always keep running and passing is a good development practice that can be tracked by the amount of written test code and test cases or the amount of covered methods. However, this practice does not give us information about the achieved quality. In order to be able to evaluate the achieved quality we would need some direct metrics on, e.g., faults, test coverage, and quality of the tests.

3.3. Need to enhance testing in agile methods

Based on the challenges and shortcomings that were discussed in the previous two subsections, we think that agile development processes could benefit from the introduction of additional testing practices. Most of the agile methods provide few instructions or little guidance on how, for example, system testing or testing different quality attributes should be handled [1, 2]. In addition, even though XP defines a complete set of very rigorous developer practices that supposedly work well together and lead to good quality, many other methods leave the testing part of development open ended, and up to the development organisation to decide upon. In these cases, the organisation has to find ways to combine, perhaps more traditional, testing approaches with the selected agile method or set of agile practices. For example, the Feature Driven Development (FDD) method does not include much guidance on testing, but instead recommends using the organisation's established QA practices with the agile development method [23].

Combining testing practices with agile processes is challenging [24]. Applying, e.g., the V-model is hard. It is based on sequential phases where activities in each phase use the completed work products (documents) of the previous phase. Thus, it cannot help us understand how the QA practices of agile methods work or how testing in agile development could be improved. In the existing literature, the problem of the sequential approach to QA has been identified, and a different, continuous approach is proposed. For example, XP, DSDM, and FDD emphasise the importance of building quality into the system with low-level developer QA practices and testing early and often throughout the development life cycle. However, it is not clear how the

testing activities are related and synchronised with the other development tasks.

We have used a temporal pacing framework, the Cycles of Control (CoC) [27], in order to better understand the dynamic nature of agile software development and the QA practices as part of an agile development process. In the next section we introduce the CoC framework and use it to describe iterative and incremental QA and to identify QA practices in existing agile methods.

4. AGILE DEVELOPMENT THROUGH TIME HORIZONS

The CoC framework, which is a general framework for describing iterative and incremental time-paced software development, can be used to understand agile software development methods through time horizons. The framework is based on the concept of time pacing, which refers to the idea of dividing a fixed time period allotted to the achievement of a goal into fixed-length segments [10, 14]. At the end of each segment, there is a control point, at which progress is evaluated and possible adjustments to the plans are made. Changes can only be made at such a control point. This both accomplishes persistence and at the same time establishes flexibility to change plans and adapt to changes in the environment at the specific time intervals. These time intervals, or time horizons, set the rhythm for product development. In accordance with the time pacing idea, the schedule (end date) of a time box is fixed, whereas the scope (developed functionality) is not.

Figure 1 shows an overview and example of the basic building blocks of the CoC framework. Each cycle represents a specific time horizon, and starts and ends with a control point in which decisions are made. The cycles and time horizons are hierarchical, meaning that the longer time horizons set the direction and constraints for the shorter ones.

The longest time horizon in Figure 1, strategic release management, deals with the long-term plans for the product and project portfolios of the company and provides an interface between business management and product development. Each individual product release is managed as a time-boxed project and dealt with in the release project cycle. Each project is split into time-boxed iterations, in which partial functionality of the final product release is developed. Daily work is managed and synchronised in heartbeats that represent the shortest time horizon. Using time horizons instead of traditional phased models makes it easier to understand the dynamic behaviour of agile development methods and the true nature of agile software development, which has also been noticed by Cockburn, who identified seven cycles of different time horizons in play on most projects [8].

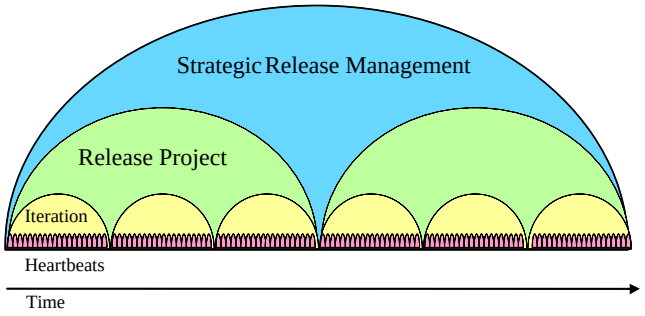

Figure 1 Cycles of Control framework [27]

In the next subsections we describe what the different time horizons mean from the quality assurance viewpoint. We also identify the QA practices in four existing agile development methods on the heartbeat, iteration and release time horizons. We selected four agile methods, namely eXtreme Programming [5], Feature Driven development [23], Dynamic Systems Development Method [29], and Crystal Clear [8] for our analysis. These cover best the QA practices of agile methods [1]. A summary of the QA practices of each of these four methods is presented in Table 3. In this table, the QA practices are divided on the three time horizons to illustrate the emphasis of quality assurance in the methods. Comparing the QA practices of the methods is somewhat challenging because the methods are described very differently. For example, Crystal Clear does not list all specific practices. Instead, it describes the essential properties and strategies of the method.

4.1. Heartbeat quality assurance

Heartbeat quality assurance includes the practices that are used to build quality into a piece of functionality during its implementation and to evaluate the achieved quality. QA activities at the heartbeat time horizon apply to all implemented features, and the implementation tasks are not complete before these activities are performed. A good example of a heartbeat QA practice is automated unit testing. Developers must write unit tests for each piece of code that they create and progress is taken into account only after the tests are completed and passed. Note, however, that heartbeat tasks are not time-boxed — a task can take several heartbeats to complete, but tracking and controlling is performed with a constant rhythm by, e.g., heartbeat meetings and daily builds. Thus, these practices tightly follow the daily rhythm of the development activities.

Table 3 Summary of agile methods' QA practices on time horizons

	eXtreme Programming	**Feature Driven Development**	**Crystal Clear**	**DSDM**
Release	-	-	-	- Contractual acceptance testing (if required)
Iteration	- Evaluating the acceptance test results	- Separate system testing	-	- Integration, system and acceptance testing inside each timebox - User testing - Evolutionary prototyping - Document reviews
Heartbeat	- Test-driven development - Continuous integration (daily-builds) - Pair programming - Acceptance tests - Collective code ownership - Coding standards - Simple design & continuous refactoring - On-site customer	- Unit testing - Regular builds - Design inspection - Code inspection - Individual code ownership	- Automated tests and frequent integration - Side-by-side programming - Osmotic communication - Easy access to expert users	- Unit testing - Reversible changes - Active user involvement

Heartbeat QA activities are not delayed to any later testing phase. Instead, these activities are performed during implementation, as part of the design and coding tasks, regardless of whom, e.g., a developer or tester performs these tasks. These activities give instant feedback to developers and therefore help drive development in the right direction.

In agile methods, quality assurance on the heartbeat time horizon is very strong as can be seen in Table 3, in which most of the quality assurance practices lie on the heartbeat time horizon. Developers' practices are particularly comprehensive and rigorously defined in XP, but the other methods also put a strong emphasis on unit testing, code and design inspections, regular builds and short integration cycles, which all are practices for the developers' everyday design and implementation work.

These practices work as a strong basis for the agile development process and strive for ensuring that good enough quality is produced in every development task.

Heartbeat QA is not restricted to the testing tasks of the developers. Heartbeat QA tasks can include, for example, designing and executing system level functional tests, executing regression tests, reporting test results and bugs, verifying bug fixes, and so on. Rhythm is the key: heartbeat activities are managed and tracked according to the heartbeat rhythm and the different roles must communicate and synchronise their work in every heartbeat.

4.2. Iteration quality assurance

Quality assurance on the iteration time horizon is concerned with activities and tasks that are not performed for each individual implemented feature at the heartbeat rhythm. Instead, these activities are controlled and tracked on the iteration time horizon and focus on fulfilling the iteration goal(s). This includes all implementation, testing and review activities that are needed to ensure that the quality of the end product of the iteration is good enough.

In practice, all needed QA activities for new functionality and code cannot be done at the heartbeat rhythm, and it is not even desirable. Many QA activities are not directly connected to individual features or enhancement tasks that developers perform. This kind of QA activities can be tracked on the iteration time horizon by including them in iteration goals and tasks. Many of the typical tasks of professional testers belong to the iteration time horizon. Specialised testers do a lot of testing (e.g. testing performance, reliability, and other qualities on the system test level that cover the system broader than functional testing of single functions or function groups), test case design, test environment setup, etc., which is not directly connected to the development tasks at hand. These specialists write and execute tests during the whole iteration, and they must synchronise their work with the developers. The synchronisation can be done using code handoffs at the daily or weekly build rhythm, and the testers can, e.g., participate in development heartbeat meetings. In addition, testing tasks that require specific

expertise or long periods of setup or execution time and thus cannot be easily performed as part of the implementation of individual features or components may best be managed on the iteration or release time horizon. Testing in different operating environments and in different combinations of environments are other examples of tasks that usually belong to the iteration time horizon.

Iteration tasks are time-boxed, because the length of an iteration is fixed. During the iteration, testers have to prioritise their work. This means that it is crucial to track the progress of the work and communicate the quality information constantly, e.g., in heartbeat meetings. Without up-to-date information it is hard to make scoping decisions in order to get the iteration QA activities performed so that the required product quality is reached by the end of the iteration.

In agile methods, the QA practices on the iteration time horizon are much fewer than the practices on the heartbeat time horizon. In Table 3, we notice that only one or two practices are defined for ensuring and evaluating the quality of the produced software increment on the iteration time horizon. In addition, the practices on the iteration time horizon are rather superficially defined compared to the heartbeat time horizon practices. Some methods, e.g. XP, rely almost purely on strong heartbeat practices and leaves only progress tracking to the iteration time horizon [4, 5, 18]. The other methods that have less rigorous heartbeat practices recognise the need for evaluating the achieved quality on the iteration time horizon by system testing, but do not give concrete guidance on how to perform it as a part of the process. For example, in FDD the only advice given in accomplishing this is to decide which builds, and how often, are handed over to separate system testing [23].

The DSDM method has a stronger approach to quality assurance on the iteration time horizon and not so detailed guidelines for the heartbeat time horizon. The approach is like a small waterfall process inside each timebox [29].

4.3. Release quality assurance

The goal of release quality assurance is to ensure the quality of the product on the release time horizon. This includes evaluating the test results and other quality information from the individual iterations, and practices to steer the development project based on that information (e.g. planning forthcoming iterations). Tasks on the release time horizon include testing that cannot be completed in the iteration schedule, tasks of a separate testing group, and testing in multiple environments. A common way of including release QA is to have a separate stabilisation iteration at the end of the release project. This is not iteration QA, because the stabilisation iteration evaluates the quality of the work done in the previous iterations. This also means that certain quality risks are not revealed until the last iteration.

On the release time horizon, it is very hard to find any QA practices in agile methods. For example, in the sample methods in Table 3, we could not identify any QA practices that would belong to the release time horizon. DSDM describes that in very large projects, or by contractual constraints, there might be cases where separate (acceptance) testing activities are needed outside the iterations, i.e., on the release time horizon [29]. However, also in DSDM these cases are considered exceptional.

5. TOWARDS ENHANCING AGILE TESTING

Agile methods rely strongly on customer or user collaboration and do not include many destructive testing practices. Some of the methods, e.g. XP, provide a very rigorous set of constructive developer practices that aim to produce good enough quality without other testing than user acceptance tests that are the responsibility of the customer. However, other agile methods do not provide such a complete set of practices and recognise the need for specific testing practices also on the integration, system and acceptance test levels. The most notable example is DSDM, which requires testing to be performed on several levels inside each iteration. To our knowledge, the current literature does not provide good guidance on how to enhance agile development processes with destructive and independent testing practices.

We showed that describing the development process and the used practices with the help of time horizons makes it easier to recognise spots where the process can be enhanced with testing practices. Understanding the heartbeat time horizon and its practices gives the synchronisation points for developers' and testers' activities. Heartbeat QA practices could be enhanced, e.g., by introducing the role of an independent tester who tests each completed feature in collaboration with the developer. This provides instant feedback on the achieved quality, based on the independent destructive tests and not only the developer's own constructive practices.

An example of agile testing on the iteration time horizon is session-based exploratory testing [3]. Exploratory testing is a testing approach that is not based on pre-specified test cases and seems to embody the agile philosophy. Session-based exploratory testing makes it possible to manage testing in short time-boxes, which makes it suitable to use in conjunction with short iterations. The exploratory approach allows testing the system in its entirety or, for example, interactions of several individual features. Our initial experiences indicate that exploratory testing is effective for testing applications from the viewpoint of the end-user and finding relevant defects with reasonable effort. The nature of exploratory testing helps reach the destructive attitude required in effective testing. Exploratory testing could also be used as a means to get business people or people with strong domain knowledge to participate in testing.

In some circumstances, testing tasks are needed on the release time horizon. So far we have not found agile practices for the release time horizon. In many cases it may be better to include as many QA practices as possible on the heartbeat and

iteration time horizons to provide quality information earlier and help mitigate the quality risks.

6. SUMMARY AND CONCLUSIONS

In this paper we identified challenges and shortcomings in agile software development methods from the viewpoint of traditional QA and testing principles. We described the QA practices of four agile methods on the time horizons of the CoC framework and showed how these emphasise the developers' quality building practices on the heartbeat time horizon.

Based on the challenges and shortcomings it seems that it would be beneficial to enhance agile development processes by introducing additional testing practices. As examples of such practices, we proposed an independent tester role for the heartbeat time horizon and session-based exploratory testing for the iteration time horizon.

This paper brings forth two challenges. First, evidence of the sufficiency of the constructive quality assurance practices of existing agile methods is required to show if enhancements are actually needed. Second, more empirical research is needed to find and try out testing practices that work in agile development.

In the future we intend to continue our research on quality assurance and testing in agile and time-paced software development. We are currently studying the exploratory testing approach and its suitability to varying contexts.

REFERENCES

1. Abrahamsson, P., et al. 2002. *Agile Software Development Methods: Review and Analysis.* VTT Publications 478, VTT, Finland.

2. Abrahamsson, P., et al. 2003. "New Directions on Agile Methods: A Comparative Analysis," *Proceedings of 25th International Conference on Software Engineering,* pp. 244-254.

3. Bach, J. 2000. "Session-Based Test Management," *STQE,* 2 (6).

4. Beck, K. 1999. "Embracing Change With Extreme Programming," *Computer,* 32 (10): 70-77.

5. Beck, K. 2000. *Extreme Programming Explained.* Canada: Addison-Wesley.

6. Boehm, B.W. 1979. "Guidelines for verifying and validating software requirements and design specifications," *Proceedings of EURO IFIP 79,* pp. 711-719.

7. Burnstein, I. 2003. *Practical Software Testing.* New York: Springer-Verlag.

8. Cockburn, A. 2004. *Crystal Clear: A Human-Powered Methodology for Small Teams.* Boston: Addison-Wesley.

9. Crispin, L., T. House 2003. *Testing Extreme Programming.* Boston: Addison-Wesley.

10. Eisenhardt, K.M., S.L. Brown 1998. "Time Pacing: Competing in Markets that Won't Stand Still," *HBR,* 76 (3): 59-69.

11. Elssamadisy, A., G. Schalliol 2002. "Recognizing and Responding to "Bad Smells" in Extreme Programming," *Proceedings of the 24th International Conference on Software Engineering,* pp. 617-622.

12. Fewster, M., D. Graham 1999. *Software Test Automation.* Harlow, England: Addison-Wesley.

13. Fowler, M., J. Highsmith 2001. "The Agile Manifesto," *Software Development,* 9 (8): 28-32.

14. Gersick, C.J.G. 1994. "Pacing Strategic Change: The Case of a New Venture," *Academy of Management Journal,* 37 (1): 9-45.

15. Highsmith, J. 2002. *Agile Software Development Ecosystems.* Boston; Cockburn,A.; Highsmith,J.: Addison-Wesley.

16. IEEE 2004. *Guide to the Software Engineering Body of Knowledge.* New York: The Institute of Electrical and Electronics Engineers, Inc.

17. IEEE 1990. *IEEE Standard Glossary of Software Engineering Terminology.* New York: The Institute of Electrical and Electronics Engineers, Inc.

18. Jeffries, R., A. Anderson, C. Hendrickson 2001. *Extreme Programming Installed.* Boston: Addison-Wesley.

19. Kaner, C., J. Bach, B. Pettichord 2002. *Lessons Learned in Software Testing.* New York: John Wiley & Sons, Inc.

20. Kaner, C., J. Falk, H.Q. Nguyen 1999. *Testing Computer Software.* New York: John Wiley & Sons, Inc.

21. Lippert, M., et al. 2003. "Developing Complex Projects Using XP with Extensions," *Computer,* 36 (6): 67-73.

22. Myers, G.J. 1979. *The Art of Software Testing.* New York: John Wiley & Sons.

23. Palmer, S.R., J.M. Felsing 2002. *A Practical Guide to Feature-Driven Development.* Upper Saddle River: Prentice-Hall.

24. Pyhäjärvi, M., K. Rautiainen 2004. "Integrating Testing and Implementation into Development," *Engineering Management Journal,* 16 (1): 33-39.

25. Rautiainen, K., C. Lassenius, R. Sulonen 2002. "4CC: A Framework for Managing Software Product Development," *Engineering Management Journal,* 14 (2): 27-32.

26. Rautiainen, K., L. Vuornos, C. Lassenius 2003. "An Experience in Combining Flexibility and Control in a Small Company's Software Product Development Process," *International Symposium on Empirical Software Engineering,* pp. 28-37.

27. Rautiainen, K. 2004. "Cycles of Control: A Temporal Pacing Framework for Software Product Development Management," pp. 1-54.

28. Royce, W.W. 1970. "Managing the Development of Large Software Systems: Concepts and Techniques," *Proceedings of Wescon,* pp. 1-9.

29. Stapleton, J. 1997. *Dynamic Systems Development Method.* Harlow, England: Addison-Wesley.

30. Stephens, M., D. Rosenberg 2003. *Extreme Programming Refactored: The Case Against XP.* Berkeley: Apress.

31. Theunissen, W.H.M., D.G. Kourie, B.W. Watson 2003. "Standards and Agile Software Development," *Proceedings of SAICSIT,* pp. 178-188.

32. Vanhanen, J., J. Itkonen, P. Sulonen 2003. "Improving the Interface Between Business and Product Development Using Agile Practices and the Cycles of Control Framework," *Proceedings of Agile Software Development Conference,* pp. 71-80.