A Comparison of the Value Systems of Adaptive Software Development and Extreme Programming: How Methodologies May Learn from Each Other

Dirk Riehle

SKYVA International
www.skyva.com, www.skyva.de

dirk@riehle.org, www.riehle.org

Abstract

Today, we see an increasing interest in new software development methodologies that put humans at the center of the development process. Adaptive Software Development, Extreme Programming, and others are exemplars of this new breed of development methodologies. They are all based on the assumption that for coping with high speed and high change, traditional management techniques are inadequate. Effectively, the new methodologies are based on a different value system than the old ones. A value system is a system of beliefs about what constitutes the fundamental aspects of software development: developers, customers, markets, products, requirements, etc. This paper presents a simple model of value systems and compares the value systems of two exemplary new development methodologies, Adaptive Software Development (ASD) and Extreme Programming (XP). The purpose of this comparison is to more easily determine whether techniques of one methodology can be adapted and used by another methodology, thereby helping authors of methodologies to better learn from other methodologies.

Keywords: Adaptive Software Development, Extreme Programming, value system, value system model, comparing methodologies.

1 Introduction

Traditional management practice of software development sees the development process as something that must be planned and controlled in order to reliably achieve the planned result. The underlying assumption is that the process can in fact be controlled and that this is beneficial

Published in *Proceedings of the First International Conference on Extreme Programming and Flexible Processes in Software Engineering* (XP 2000). Republished in Giancarlo Succi, Michele Marchesi (eds). *eXtreme Programming Examined.* Addison-Wesley, 2001. Page 35-50.

to the outcome of the process. In its extreme form, this belief has been formulated by Osterweil in his famous ICSE 9 keynote speech on software development processes: software processes are software too [Osterweil1987]. His keynote suggests that developers execute processes much like computers execute software applications. This approach has given rise to research in software process modeling and enactment that is still going on today. The underlying assumption that humans execute processes much like machines has found its way into current terminology and thinking. As a recent example, Pohl et al. write of developers as being guided by tools and as performing processes [Pohl+1999].

The underlying assumption of equating humans with computers and that processes can be planned on a fine-grain level is not shared by all. Osterweil's assurance has been rejected immediately by Lehman in his response to Osterweil's presentation [Lehman1987]. Evolutionary prototyping, for example, views software development as a shared learning experience of both customers and developers [Floyd+1992, Budde+1992]. Naur views computing as a fundamentally human activity [Naur1992]. Catering for human needs in software development processes is in stark contrast to viewing developers as resources that can be utilized at will.

This distinction has its consequences: what is called traditional management above uses different techniques for carrying out development than does evolutionary prototyping. For example, traditional approaches use formal textual requirement specifications to determine and communicate requirements to developers, while evolutionary prototyping prefers using prototypes to discuss requirements and ensure timely feedback from customers. This difference between the traditional approach and evolutionary prototyping is just one difference (albeit one of key importance). Other differences are how a development methodology views customers, how it fosters creativity and innovation, how it views changes in the market and in requirements, etc.

The overall set of assumptions underlying a development methodology is its value system, a system of beliefs about the world in general and software development in particular. The value system is reflected in the techniques the methodology provides to its users. A technique makes only sense in the context of a development methodology if it is compatible with the methodology's value system.

A technique is like a conceptual tool: it has been designed to do certain tasks well and hence can be used effectively to carry out these tasks. Beyond this context of application, the technique breaks down much like a tool breaks down. While human ingenuity in the use of a technique allows for some adaptation, a technique can not be used effectively, if it is at odds with the value system of the underlying methodology in the context of which it is being applied.

Similarly, users of the methodology must share the value system to make effective use of the methodology and its techniques. Only if these three pieces (a coherent value system, techniques that are compatible with the value system of a methodology, and users that share the values) come together harmoniously can a methodology be effective.

Currently, we see the emergence of several new development methodologies with similar underlying value systems, for example Adaptive Software Development (ASD) [Highsmith2000], Extreme Programming (XP) [Beck2000], SCRUM [Beedle+2000], Crystal [Cockburn2000], and others. They view themselves as lightweight methodologies, that is as methodologies that come without much overhead so that they can be applied easily. In my reading, "lightweight" means that these methodologies do away with much of the administrative overhead of traditional methods (like extensive paper-based documentation or elaborate process handbooks).

This understanding of being "lightweight" has its consequences: the new methodologies tend not to be dogmatic about when and how to apply the techniques they provide. The metaphor of a technique as a "conceptual tool" captures this spirit well: a technique is used when it seems appropriate, and it is dropped when it does not seem to help anymore in a specific situation. The methodology becomes a meta-framework that determines what is considered important about software development and how techniques have their place in it, but it does not specify when and how to use a specific technique.[1] Again, in contrast to this, traditional methodologies tend to be more prescriptive in when and how to use a technique.

For survival in the market and for providing continuing value to users, it is essential that the new methodologies are continually refined and possibly extended with new and enhanced techniques.

One possible way for a methodology to achieve this is to learn from other methodologies and to adapt from them what works for itself. How precisely can we define whether a certain technique from another methodology works for the methodology we are currently using? Obviously we could just try, but this may turn out to be too expensive. A better way is to analyze the technique in question for its compatibility with the methodology's own value system and the existing techniques. The result of this analysis may not provide the final answer, but the analysis may save time and money if it rules out incompatible techniques upfront.

This article carries out a comparison of the value systems of two recent and promising new development methodologies (ASD and XP) with respect to their value systems (Section 2, 3, and 4). The comparison is based on a model of value systems for development methodologies (Section 3). Based on the comparison, conclusions about compatibility and potential learning of techniques from the respective other methodology are drawn. The paper closes with a discussion of where the approach of value system analysis may take us (Section 5).

2 Review of ASD and XP

Adaptive Software Development (ASD) is a new software development methodology that addresses "the Internet economy", that is an economy of high speed and high change [Highsmith2000]. Highsmith uses Complex Adaptive Systems theory [Holland1995] to explain the fundamental assumptions he makes about software development and its markets.

Extreme Programming (XP) is another new development methodology that was specifically conceived to work in the face of vague and changing requirements, so it targets a similar environ-

[1] Extreme Programming (XP), one of the two methodologies discussed in this article, considers it important that all of its techniques are used together (to make up for each other's weaknesses). Hence it appears to be closed towards adopting new techniques and changing existing ones. I believe that XP will evolve further and that this aversion is only a temporary state of a development methodology it its early stages.

ment as ASD [Beck2000]. Beck, currently the only author of a book on XP, is explicit in what he views as the underlying values of XP (Communication, Simplicity, Feedback, and Courage).

This section reviews both methodologies (with a strong focus on what they view as their underlying value system). The next section presents a model of value systems that lets us reinterpret the value systems of both ASD and XP to better support their comparison.

2.1 Review of Adaptive Software Development

ASD addresses the economy of increasing returns [Arthur1996]. High speed and high change characterize this economy, which underlies the Internet and the market of today's dot-com companies. High speed and high change induce a complexity that can not be handled by traditional approaches. High speed and high change make a market unpredictable and the development process unplannable in the traditional sense of controlling the process.

Arthur, and then Highsmith, uses Complex Adaptive Systems (CAS) theory [Holland1995] to describe the complexity of this market. Both conclude that in the economy of increasing returns, being able to adapt is a significantly more important success factor than being able to optimize.

CAS theory provides three main concepts to explain the world: agents, environments, and emergence. Agents compete and cooperate to get work done, but the final result is not the outcome of the work of any particular agent or process. Effectively, the result emerges from the overall competition and cooperation of the agents. System behavior can not be predicted from the individual behavior of agents, because simple cause-and-effect reasoning has broken down.

Highsmith transfers the CAS model to software development, viewing the development organization as the environment, its members as agents, and the product as the emergent result of competition and cooperation. This has profound consequences. Accepting and living with unpredictability and uncertainty asks for a new approach to software development.

The first goal of any development organization is to be able to respond quickly to changes, that is to be adaptive. Adaptiveness can not be commanded, it must be nurtured. This nurturing is realized through a management model that Highsmith calls the Adaptive Leadership-Collaboration model. It leads to an environment in which adaptation and collaboration thrive so that local order can emerge. Local order scales over several levels from the individual to work groups and to the whole development organization. By nurturing adaptive behavior in every cell, the overall system becomes adaptive.

Highsmith recommends two key strategies for creating an adaptive and collaborative environment. The first strategy asks managers to focus less on process but on products, that is the results of collaboration. Managers must apply rigor to the results of the process rather than prescribing the process. The second strategy asks managers to provide tools and techniques for fostering self-organization across virtual teams. Virtual teams are teams that are distributed around the world. This second strategy is only needed if ASD is applied to large-scale software development. (The early RAD approaches have often been criticized as not scaling up. It is an explicit goal of Highsmith to overcome this criticism with ASD.)

The focus on products asks for an iterative approach, because the result of each iteration becomes the main input to steering the process. Each iteration consists of the three phases speculation, collaboration, and learning. Speculation on the product means the discussion and subsequent definition of what is to be achieved in an iteration. Highsmith calls this activity speculation to make explicit that what others may call planning is truly speculation about the future. It follows a collaboration phase in which team members collaborate towards a product that incorporates the features as suggested from the speculation phase and from the on-going external input. In the final learning phase of an iteration, the result is reviewed in light of the speculation and the next iteration is being prepared.

Each iteration, called adaptive cycle, has the following properties:

- it is mission-driven based on the project vision;
- it is component rather than task-based (result-driven);
- it is limited in time;
- each time-box is only one iteration in a larger set of iterations;
- it is risk-driven;
- it is change-tolerant.

Change is viewed as the opportunity to learn and to gain a competitive advantage rather than as a detriment to the process and its results.

Executing the speculation/collaboration/learning activity in each cycle is supported by different techniques. Highsmith presents a host of such techniques that support the development process.

It is important to note which role Highsmith assigns to techniques as part of the overall process. No single activity is important and has to be applied, not even specific combinations are a must. Highsmith expects no activity to be a silver bullet for anything, even though he expects specific learning and collaboration technique to be used in most ASD projects. The main reason for this is that every technique has a certain context of applicability. Beyond its context, a technique starts to work unreliably or even breaks down. Because Highsmith has a wide variety of projects in mind for ASD he does not enforce specific technique, as there will certainly situations in which it breaks down.

2.2 Review of XP

Beck's description of Extreme Programming separates it into several parts:

- *Values.* A value in the XP sense is a description of "how software development should feel." (This is the best definition I have found in [Beck2000]). XP is based on the following four values: Communication, Simplicity, Feedback, and Courage.
- *Principles.* A principle in XP is something that we use to determine whether a practice (see below) is something that can be used successfully in an XP context. Beck presents the fol-

lowing five principles, as derived from the values: Rapid Feedback, Assume Simplicity, Incremental Change, Embrace Change, and Quality Work.

- *Activities.* In a separate dimension, Beck views the following four activities as the cornerstones of software development, and hence to be supported by matching practices: Coding, Testing, Listening, and Designing.
- *Practices.* A practice in XP is a technique that project members use to successfully carry out any of the aforementioned activities. In [Beck2000], Beck presents 12 practices, ranging from The Planning Game as a technique to carry out schedule and feature negotiation to 40-Hour Week, a warning not to do overtime over extended periods of time.
- *Strategies.* Finally, to successfully execute practices in the real world, Beck presents several strategies that describe experiences and heuristics of how to achieve this.

Of primary interest here are the values and principles and how they interact with the so-called practices, because the practices are the techniques that can possibly transferred to other methodologies.

The *value of communication* represents the XP belief that communication between project members is key to a successful project (and hence needs to be supported by practices). It is not stated why communication is so important, but it is safe to assume that communication is viewed as the main enabler of coordination and collaboration in a project. It is important to note that in [Beck2000] communication always means verbal communication.

The *value of simplicity* represents the XP belief that you should not invest into the future but only into your immediate needs. If future needs materialize as immediate needs at some point in the future, the proper application of XP practices will have put developers into a situation that lets them successfully cope with the new need. Underlying this value is the assumption that the future can not be reliably predicted and that taking care of it today is economically unwise.

The *value of feedback* represents the XP belief that it is important to have a running system at any time that gives developers reliable information about its functioning. (Here feedback is not feedback between humans but rather feedback about the development state.) Effectively, the system and its code base serve as the incorruptible oracle to report about the progress and state of development. Feedback serves as a means for orientation and deciding where to go.

The *value of courage* represents the XP belief that humans are the single most significant aspect of software development. It is human courage that solves problematic situations and lets a team leave local optima behind to reach greater goals. The value of courage represents XP's fundamental trust in humans to make a project succeed.

The values are elaborated and made more precise through the aforementioned principles, which are basically an explanation of the values.

The 12 XP practices presented in [Beck2000] are closely knit. They tie in with each other well. As Beck explains, this was an explicit goal, because each of the practices is not only an old and well-known practice, but also a practice that has been shown *not* to work in many circumstances. Only by integrating them are the weaknesses of any of the practices rendered irrelevant, because the other practices make up for them.

The XP practices are in flux. On the web, we can find a more elaborate (and evolving) description of the XP practices [Wiki2000, XPorg2000]. These web pages, as well as the discussions on the extreme programming mailing list [XPmlist2000] suggest that XP is open to adopt new techniques.

3 Value Systems

From the outset, both ASD and XP appear to be very similar. Both address software development processes that face uncertainty and continuous change. A closer look, however, shows many differences, for example:

- *Motivation.* ASD is motivated by the inappropriateness of traditional management in the new economy of increasing returns. Making software development a humane experience again is a key motivation of XP.
- *Attitude towards techniques.* ASD believes that techniques are important, but no silver bullets. Developers use techniques judiciously to solve problems at hand. XP strongly relies on a specific combination of pre-defined techniques.
- *Levels of scale.* ASD addresses both small, medium and large development teams that are potentially distributed. XP addresses only small (up to 10 developer) teams that must be co-located.

Still, ASD and XP seem to have a common core of themes and beliefs, which we will identify as a value system below. If we want any of the methodologies to learn from the other, we must provide a basis for comparing value systems to determine compatibility of techniques. No such basis exists to my knowledge.[2]

3.1 A model of value systems

We need to step back from the specific descriptions of ASD and XP given above. As I mentioned, my review of these two methodologies uses the structure of the main books on this subject. This structure may not necessarily represent the weight each of the methodologist assigns to an aspect of his methodology. It may simply be a teaching device to better communicate certain ideas while deferring the discussion of further issues to later expositions.

[2] It is interesting to note that the Capability Maturity Model (CMM) plays this role for traditional software development methodologies. The CMM is a set of metrics for evaluating instances of development methodologies as they are practiced by a specific organization [Paulk1995]. While this may not have been its original intent, the CMM represents quite a clean description of the value system underlying traditional development methodologies. Every methodologist that submits his or her methodology to the CMM acknowledges the value system of the CMM to be the value system of his methodology. This is helpful, because most methodologies have no explicit value system.

This section presents a simple model of value systems for software development methodologies. This model is not exhaustive; I view it more as a beginning of further research into value systems for development methodologies. However, the concept of value system has grown out of earlier research into development methodologies carried out by Floyd et al. [Floyd+1992] and Züllighoven et al. [Budde+1992]. In their critique of the dominant management paradigm, they were very much aware of the fundamental differences in assumptions about software development.

The model has the following fundamental dimensions.

- *Role of humans in software development.* What is the role and contribution of humans in software development? This includes the role and responsibilities of customers, developers, and managers, as well as their recognition as being humans beyond playing roles.
- *Relationship between humans in software development.* What is the role and importance of human relationships in software development? How important is communication, collaboration, and competition? Should it be hindered or fostered, structured or free-floating?
- *Relationship between humans and technology.* What is the role of technology (techniques, tools, and media) in software development? Does technology dominate humans or do humans control technology? How is it applied, how is technology worked with?
- *Purpose of software development.* Why do we carry out a specific software development process (next to satisfying customers)? To "rush and cash" by delivering a good enough product as fast as possible, to do satisfying work, or to advance humanity[3]?

Each dimension needs elaboration, and each aspect of a dimension needs to be weighted against the other aspects. Quantitative weighting is possible, but difficult. I use dimension-specific weighting. For example, I qualify each aspect of the Purpose of Development dimension as either being key, support, or unimportant (not applicable is also possible, which is to say that the aspect is not mentioned in the methodology description).

3.2 Comparison of ASD and XP

Table 1 compares ASD with XP using the model of value systems just described.

[3] I think it is not naive to list human advancement here. For example, the IEEE code of ethics confirms that every IEEE member agrees on fostering human advancement through better understanding and use of technology. Similarly, most academic projects have this goal.

	ASD	XP
Role of humans	Central undisputed agents.	Central undisputed agents.
• Role of developers	Steer product, adapt it to changing requirements.	Steer process/product, use pre-defined techniques to do so.
• Role of managers	Steer process/product, adapt it to changing requirements.	Steer process/product, use pre-defined techniques to do so.
• Role of customers	Provide input to steer process; are involved.	Provide input to steer process; are involved.
Role of human relationships	Main enabler of innovation, ability to adapt.	Main enabler of innovation, work satisfaction.
• Role of communication	Is key for emergence.	Is key for working towards results.
• Role of cooperation	Is key for emergence.	Is key for working towards results.
• Role of competition	Is key for emergence.	N/A.
Relationship between humans and technology	Humans control technology; technology is a tool.	Humans control technology; technology is a tool.
• Application of technology	Humans utilize technology at will.	Humans utilize technology within predefined framework.
• Type of technology	Lightweight technology is preferred.	Lightweight technology is preferred.
Purpose of development	Survival and thriving of organization.	Product delivery while doing satisfying work.
• Product delivery	Is key.	Is key (“playing to win”).
• Do satisfying work	Is support.	Is key (“playing to win”).
• Advance humanity	N/A.	N/A.

Table 1: High-level comparison of the value systems of ASD and XP.

The comparison of Table 1 is rather coarse-grained. However, it provides a framework to zoom in on more fine-grained issues.

4 Compatibility of ASD and XP

ASD and XP are similar in many aspects (which should not come as a surprise by now). Of more interest are the differences between the value systems of these two methodologies. From Table 1, we gather the following differences:

- In ASD, people use technology as they see proper. In XP, they use technology only in so far as it fits the framework of predefined techniques.
- In ASD, human relationships are a key enabler of emergent results. In XP, communication and cooperation are important to work towards a result, competition is not.
- In ASD, doing satisfying work supports the main goal of product delivery, but is not of key importance. In XP, doing satisfying work is equally important as delivering a product.

A more fine-grained analysis shows further differences. It is not carried out here, though. For the purposes of this paper, these three key differences and their consequences are sufficient material for discussion.

4.1 Use of techniques/practices

ASD has a laissez-faire approach towards specific techniques. If they are useful and solve a problem at hand, project members use them, otherwise they ignore them. From this perspective, every technique may be put to use, in particular if we keep in mind that adaptive behavior of agents is key in ASD. Therefore, techniques need not be compatible with each other, but adaptation through humans will in the end make them compatible.

XP has strong feelings about its techniques. They must be practiced, and they must be practiced together, otherwise a developer would not be doing XP. As mentioned above, XP's set of techniques is not fixed. However, it will be slow to adapt new techniques if the requirement of closely-knit integration is maintained.

As a consequence, ASD will be fast to adopt new techniques, while XP will be slower to do so. As another consequence, XP's techniques will continue to be more effective than ASD's techniques (even if these are the same techniques), because of XP's strong focus on keeping the techniques closely integrated.

On the other hand, XP is tied to its techniques, while ASD can easily let go of techniques that appear inadequate for a specific project situation. Hence, XP's applicability is strongly constrained to a context in which its predefined techniques work, while ASD is open for any new situation. In some sense, ASD has a stronger and better defined meta-framework than XP.

4.2 Competition among project members

ASD's belief in emergent results is one of the strongest differences to other methodologies, including XP. This is in-line with ASD's departure from a traditional management approach. As a consequence, ASD techniques are unlikely to fully block competition among project members but rather try to set appropriate goals in the form of project results.

XP does not mention the issue of competition among project members. Being unclear in this respect, XP managers are left to their own devices when facing a situation of competition. This may lead to problematic decisions. An XP manager should not adopt an ASD technique that fosters competition, if he or she does not also believe into positive and constructive competition between project members.

It is clearly advisable for XP to review its position towards competition to avoid potential clashes if it wants to adopt techniques from ASD.

4.3 Good enough work vs. quality work

ASD's focus on adaptation rather than optimization supports the concept of "good enough" work and results. Highsmith takes the derogative connotation out of "good enough" by calling it the total best solution in a given situation.

XP on the one hand strives for simplicity and not investing into the future unless immediately needed. On the other hand, the importance of doing satisfying work and equating it with quality work suggests that the quality and hence the investment into the product may go beyond what is actually needed. Obviously, this strongly depends on the individual developer and what he views as quality work that satisfies him or her.

Therefore, some of the ASD techniques that strongly focus on developing good enough software do not work for XP. For example, for ASD, redundant source code is not a problem, it may even be a necessary byproduct of a competitive environment. XP insists on stating everything "once and only once" and thereby tries to fully eliminate redundant code.

5 Conclusions

In the world of software development methodologies, no specific methodology stands out as the one dominant player. Also, as of today, no clear winner is on the horizon. To survive, thrive, and grow, a methodology must be able to learn from other methodologies and successfully adapt to changing market requirements.

Learning and adaptation can be a painful experience, in particular if the underlying value systems of two methodologies are incompatible. But even with compatible value systems, learning may still be difficult. This paper compares two new development methodologies, Adaptive Software Development (ASD) and Extreme Programming (XP). Both have compatible value systems so that one might expect that they could easily exchange techniques and learn from each other. However, an analysis of the value systems of both methodologies shows that despite all similarities, several incompatibilities remain that hinder immediate adoption of techniques from the other methodology.

On a more general level, the model of value system presented in this paper can serve as a basis for comparing further methodologies with ASD and XP and with each other. The model needs to be extended and refined, but it already serves a useful purpose, as this paper shows. As the emergence of the new breed of methodologies shows, there is a clear need for distinguishing and comparing them. I expect models of value systems to be at the heart of methods for comparing these new software development methodologies.

Acknowledgements

I would like to thank Jim Highsmith, Hans Wegener, and Heinz Züllighoven, as well as John Arrizza, Malte Kröger, Dave Thomas, and the anonymous reviewers for their helpful feedback on this article.

References

[Arthur1996] W. Brian Arthur. "Increasing Returns and the New World of Business." *Harvard Business Review* 74, 4 (July 1996). Page 100-110.

[Beck2000] Kent Beck. *Extreme Programming Explained: Embrace Change.* Addison Wesley, 2000.

[Beedle+2000] Mike Beedle, Martine Devos, Yonat Sharon, Ken Schwaber, Jeff Sutherland. "SCRUM: A Pattern Language for Hyperproductive Software Development." In *Pattern Languages of Program Design 4.* Edited by Neil Harrison, Brian Foote, and Hans Rohnert. Addison-Wesley, 2000. Page 637-652.

[Budde+1992] R. Budde, K. Kautz, K. Kuhlenkamp, H. Züllighoven. *Prototyping.* Springer Verlag, 1992.

[Cockburn2000] Alistair Cockburn. Crystal (Clear) *http://members.aol.com/acockburn.* AOL, 2000.

[Floyd+1992] C. Floyd, H. Züllighoven, R. Budde, R. Keil-Slawik. *Software Development and Reality Construction.* Springer Verlag, 1992.

[Highsmith2000] James A. Highsmith III. *Adaptive Software Development.* Dorset House, 2000.

[Holland1995] John H. Holland. *Hidden Order: How Adaptation Builds Complexity.* Addison-Wesley, 1995.

[Lehman1987] M. M. Lehmann. "Process models, process programs, programming support." In *ICSE 9 Conference Proceedings.* IEEE Press, 1987.

[Lichter+1997] Horst Lichter, Matthias Schneider-Hufschmidt, and Heinz Züllighoven, "Prototyping in Industrial Software Projects: Bridging the Gap Between Theory and Practice." In

Software Project Management: Readings and Cases. Chicago, Boston: Irwin Verlag: 1997. Page 306-317.

[Naur1992] Peter Naur. *Computing: A Human Activity.* Addison-Wesley, 1992.

[Osterweil1987] L. J. Osterweil. "Software processes are software too." In *ICSE 9 Conference Proceedings.* IEEE Press, 1987.

[Paulk1995] Mark C. Paulk. "How ISO 9001 Compares with the CMM." *IEEE Software* 12, 1 (January 1995). Page 74-83.

[Pohl+1999] K. Pohl, K. Weidenhaupt, R. Domges, P. Haumer, M. Jarke, R. Klamma. "PRIME--Toward Process-Integrated Modeling Environments." *ACM TOSEM* 8, 4. Page 343-410.

[Wiki2000] Various authors. *http://c2.com/cgi/wiki?ExtremeProgramming.* C2, 2000.

[XPorg2000] Don Wells. *http://www.extremeprogramming.org.* Unknown ISP, 2000.

[XPmlist2000] Various authors. *http://www.egroups.com/group/extremeprogramming.* Egroups.com, 2000.

About the author: Dirk Riehle is a software developer and resident metamodeler at SKYVA International in Boston, MA. In his work, he focusses on the architecture and implementation of metamodels for model-driven business systems. Dirk holds a Ph.D. from ETH Zürich and is a frequent author on the subjects of object orientation, patterns and frameworks, and software architecture. He welcomes feedback at dirk@riehle.org or through www.riehle.org.