von stefan roock und
henning wolf

mehr zum thema:
xpbuch.it-agile.de
www.agilealliance.org

FÜNF JAHRE XP: WIE ERWACHSEN SIND WIR GEWORDEN?

Fünf Jahre ist es bereits her, dass Kent Beck sein Buch über „eXtreme Programming" (XP) veröffentlicht und damit die Welt der Softwareentwicklung auf den Kopf gestellt hat. Obwohl die Einzelteile aus XP nicht neu waren, hat XP eine immer größer werdende Diskussion um agile Methoden losgetreten. Die hinter Agilität stehenden Werte und auch die damit verbundenen Erwartungen an erfolgreiche Entwicklungsprojekte sind heute so weit anerkannt, dass sich immer mehr Teams und Methoden für agil erklären. Nur ein Teil von ihnen beansprucht dieses Etikett zu Recht. Agil zu sein ist auch gar nicht so einfach. Wir beschreiben die zwölf häufigsten Fehler, die wir selber gemacht oder in anderen Projekten beobachtet haben.

die autoren

Stefan Roock
(E-Mail: stefan.roock@it-agile.de) und

Henning Wolf
(E-Mail: henning.wolf@it-agile.de) sind Senior IT-Berater bei der it-agile GmbH und beschäftigen sich seit sechs Jahren mit XP und anderen agilen Methoden. Zusammen mit Martin Lippert haben sie ein Buch über ihre Erfahrungen mit XP geschrieben, das demnächst in einer zweiten, erweiterten Auflage erscheint.

Von 1999...

Ende 1999 veröffentlicht Kent Beck das XP-Manifest („eXtreme Programming Explained", vgl. [Bec00]) und trifft damit das Establishment des Software-Engineerings vollkommen unvorbereitet. XP stellt große Teile von dem in Frage, was als Stand der Kunst anerkannt war. Softwareentwicklung wird radikal auf ihren Kern reduziert und die Nützlichkeit allen Drumherums wird in Frage gestellt. Die verbliebenen Techniken werden ins Extrem getrieben (z. B. bei *Test-First*). Und Beck polarisiert weiter: Wer 80% der XP-Techniken einsetzt, bekommt angeblich nur 20% des damit verbundenen Nutzens.

Ein kleiner Teil der Softwareentwicklungsgemeinde ist sofort hellauf begeistert, aber der Großteil hält XP für gefährlichen Blödsinn, der auf keinen Fall funktionieren kann.

1. **Agil = schwammig/ beliebig**
2. **Kommunikationsprobleme**
3. **Überanpassung**
4. **Kein Tracking / Controlling**
5. **Wir stellen uns dumm**
6. **Naives Design**
7. **Deployment wird unterschätzt**
8. **Teamgröße skaliert nicht**
9. **Falsche Retrospektiven (zu technisch)**
10. **Unechter Kunde**
11. **Grandioser Prozess**
12. **Kunde unterliegt Entwicklern**

Kasten 1: *Die 12 häufigsten Fehler beim XP-Einsatz*

...bis heute

Inzwischen hat sich die Welt der Softwareentwicklung deutlich gewandelt. Neben XP haben sich weitere agile Methoden verbreitet (z. B. *SCRUM*, vgl. [Sch04]) und XP selbst ist in einer zweiten Generation verfügbar. Beck hat mit der zweiten Auflage seines XP-Buchs ([Bec04]) XP so weit redefiniert, dass häufig auch von XP2 gesprochen wird. *Industrial XP* nimmt als XP-Ableger viele Erfahrungen aus den letzten fünf Jahren auf (vgl. [IXP04]).

Die agilen Methoden sind heute im Wesentlichen positiv besetzt. Ohne Agilität scheint gar nichts mehr zu gehen. Auf Konferenzen werden Tutorials angeboten, die zeigen, wie der *Unified Process* und *RUP* agil verwendet werden können. Das V-Modell unterstützt in der neuen Version *V-Modell XT* (XT = „eXtreme Tailoring", vgl. [BMI05]) nach eigenen Angaben agile Vorgehensweisen, und die Standish Group empfiehlt in ihrem aktuellen Chaos-Report ([Sta04]) sogar explizit die Verwendung agiler Methoden.

Selbst außerhalb der reinen Softwareentwicklung greift Agilität um sich. Das CIO Magazine kürte beispielsweise unlängst die 100 agilsten Großunternehmen (vgl. [Pre04]).

Die Gemeinde der „Agilisten" in der Softwareentwicklung hält sich vornehm zurück: Neue agile Methoden entstehen nur selten. Man hat sich im Agilen Manifest ([Man01]) darauf geeinigt, was Agilität ausmacht. Ein agiler Methodendschungel ist uns erspart geblieben.

Alles das sind Indizien dafür, dass die agilen Methoden erwachsen geworden sind. Diese These wird auch dadurch unterstützt, dass wir in unseren eigenen Softwareprojekten und in Beratungssituationen immer wieder auf dieselben Probleme stoßen: Es gibt kaum noch Überraschungen. Für diesen Artikel haben wir unsere persönlichen *Top 12 Fehler* (unser dreckiges Dutzend) gekürt (*siehe auch Kasten 1*).

Fehler 1: Agil = schwammig/beliebig

„Wir sind ja so agil! Und agil heißt doch, dass man schnell auf Änderungen reagiert. Also, da hätten wir hier schnell mal ein paar neue Anforderungen. Das schaffen Sie schon. Bis morgen, ach, übermorgen reicht auch noch."

Klar sind agile Methoden schneller als klassische Vorgehensweisen, aber man muss aufpassen, dass die Reaktionsgeschwindigkeit auf neue Anforderungen nicht dazu führt, dass man ständig nur noch Anpassungen des Plans vornimmt. Zumindest für eine Iteration und damit für mindestens eine bis sechs Wochen sollten die Anforderungen allenfalls konkretisiert, aber keinesfalls umgeplant werden. Ansonsten ergeben sich Probleme, denn die Entwickler orientieren sich bei ihren Entwürfen an den Iterationszielen. Ändern sich diese Ziele stark, bleiben meist auch Schmutzrückstände im Entwurf zurück.

Die häufigste Ursache für dieses Problem ist ein falsches Verständnis von Flexibilität. Das wird dann noch gepaart mit Kunden, die nicht mal für eine Iteration im Voraus planen können und sich ständig umentscheiden. Man kann sich vorstellen, wie verloren solche Kunden dann erst bei klassischen Herangehensweisen sind, die komplette Anforderungen in hoher Detaillierung zu Beginn des Projektes erfordern.

Als Gegenmaßnahme schlagen wir vor, dass die Entwicklung gestoppt wird (oder schon im Vorwege eine Exploration stattfindet) und Kunde und Entwickler gemeinsam eine grobe Vision des Systems erstellen. Außerdem sollte es für das Release und die aktuelle Iteration eine Feature-Liste geben. Die Vereinbarung zwischen Entwicklern und Kunde sollte sein, dass sich während einer Iteration die Anforderungen nicht ändern dürfen; gegebenenfalls kann man die Iterationen kleiner machen, um dem Kunden die Möglichkeit zu bieten, seine Wünsche häufiger berücksichtigt zu finden.

Unser Hinweis: *Agiles Vorgehen bedeutet, äußerst diszipliniert vorzugehen.*

Fehler 2: Kommunikationsprobleme

„Wir schaffen das schon. Ich will keine schlechten Nachrichten hören."

Wir gehen davon aus, dass der Großteil aller relevanten Probleme in Projekten auf Kommunikationsprobleme zurückgeht. Allerdings sind Kommunikationsprobleme meist kein quantitatives, sondern ein qualitatives Problem. Es geht nicht darum, einfach mehr zu kommunizieren. Vielmehr geht es darum, klare Strukturen in der Kommunikation zu etablieren und beizubehalten: Wer nimmt welche Aufgaben wahr? Wer ist wofür verantwortlich? Wer hat welche Entscheidungskompetenzen? Wer muss dann wann zu welchem Zweck befragt werden? Wer braucht deshalb welche Informationen zu welcher Zeit? Diese klaren Strukturen müssen die Projektbeteiligten dann natürlich auch klar ausfüllen.

Das Problem entsteht dadurch, dass Menschen dazu neigen, in der Kommunikation den einfachen Weg zu gehen und nicht den richtigen. Ein typisches Beispiel: Man weiß, dass der Chef keine schlechten Nachrichten hören will. Häufig sagt er das sogar („ich will nur Positives hören"). Daran orientieren sich dann die Projektbeteiligten und verheimlichen die schlechten Nachrichten. Bei den allermeisten Projektkatastrophen wussten die Entwickler lange vor dem Management von den Problemen. Tragischerweise hätte man häufig sogar noch gegensteuern können, wenn die Entwickler die Probleme gleich bei ihrem Auftreten kommuniziert hätten. Ehrlichkeit wird auch vom „IEEE Software Engineering Code of Ethics and Professional Practice" eingefordert (vgl. [IEEE99]).

Strukturelle Kommunikationsprobleme können eigentlich leicht aufgedeckt werden, z. B. durch Retrospektiven; hilfreich ist dabei ein externer Moderator.

Bitte beachten: *Kommunikation muss offen und ehrlich sein.*

Fehler 3: Überanpassung

„Wir haben ja eigentlich immer schon agil Software entwickelt." „Gerade diese eine Technik kann man bei uns nicht einsetzen." „Technik XYZ machen wir nicht. Wir machen ABC, das ist ja fast das Gleiche."

Genau so, wie es im Buch von Beck zu lesen ist (oder auch in [Wol05]), wird XP von kaum jemandem angewendet. Da steht dann auch irgendwo, dass man das ganze dann noch an projektspezifische Gegebenheiten anpassen muss. Prima, dann sind wir auch agil, ist ja auch „hype".

Sieht man sich an, was in manchen Projekten vom agilen Gedankengut oder von XP übrig geblieben ist, wird man häufig nicht viel finden. Jede einzelne Abweichung hat schon ihren Grund und solche Teams finden sich (zu schnell) mit diesen Gründen ab. Lustigerweise sind ja allerhand Teams angeblich schon immer agil gewesen (O-Ton: „Das haben wir schon immer so gemacht.").

Weinberg beschreibt in seiner „Fast-Food-Fallacy" (**siehe Kasten 2**), dass häufiges „Das macht eigentlich keinen Unterschied" in der Summe sehr wohl einen Unterschied macht.

XP zeigt einen Weg auf, wie man sich ständig verbessern kann. Das bedeutet Lernen, was nicht immer angenehm ist. Also könnte man sagen: „Wenn es nicht weh tut, ist es nicht XP". Wenn sich was ändern soll, muss man was verändern! Dabei ist es durchaus okay, wenn man nicht bei 100% XP anfängt, aber man sollte doch zumindest anstreben, dem Ziel immer näher zu kommen.

Es gilt: *Wenn sich was ändern soll, müssen wir was ändern.*

Fehler 4: Kein Tracking/Controlling

„Tracking bringt uns nichts. Das macht uns auch nicht schneller."

Tracking ist die Voraussetzung, um Planabweichungen zu erkennen und auf

> Früher waren Hamburger noch echte Mahlzeiten, zubereitet aus vielen frischen Zutaten mit Salat, Tomaten etc. Dann sind ein paar schlaue Geschäftsleute auf die Idee gekommen, Hamburger als Massenware unters Volk zu bringen. Und über die Jahre haben sie immer wieder unmerkliche Kostenoptimierungen vorgenommen: „Wenn wir eine Gurkenscheibe weniger auf den Hamburger legen, merkt das sowieso keiner und wir sparen ein Zehntel Cent je Hamburger". Und so sind aus einstmals saftigen Hamburgern Dinge geworden, die mit dem Original nur noch sehr wenig gemein haben.

Kasten 2: *Fast-Food-Fallacy (nach [Wei01])*

diese zu reagieren. Agile Methoden zeichnen sich gerade dadurch aus, dass sie eine Reaktion auf Planabweichungen erlauben. Wer kein Tracking einsetzt, verschenkt die Potenziale agiler Methoden.

Man kann immer Gründe finden, warum man im Moment kein Tracking durchführen kann („alles so unklar", „zu früh im Projekt"), aber dann sollte man es lieber sehr grob und ungenau machen als gar nicht. Nach unserer Erfahrung kostet Tracking nur sehr wenig zusätzlichen Aufwand.

Wir raten: *Wir sollten lieber wissen, wo wir stehen, als mit Höchstgeschwindigkeit in eine beliebige Richtung vorzupreschen.*

Fehler 5: Wir stellen uns dumm

„Wir brauchen uns nicht um XYZ kümmern, das wird der XP-Prozess schon für uns richten."

Ein expliziter Entwicklungsprozess ist eine feine Sache. Wenn er dann auch noch mit möglichst wenig Wörtern beschrieben wurde, ist es sogar noch besser. Wenn man glaubt, der Prozess würde dann schon in jedem Fall ausreichend und allumfassend sein, dann ist das zu kurz gegriffen. Nehmen wir z. B. das Thema Risikomanagement. In der Beschreibung von XP durch Beck steht davon erst mal explizit nichts. Das ist aber definitiv noch kein Grund, sich gar nicht erst mit Risikomanagement zu beschäftigen.

Ein anderer beliebter Streitpunkt: Wie viel Vorausschau brauchen/sollten die Entwickler haben? Sollen sie entweder nur die Storys der gerade aktuellen Iteration kennen oder alle Storys und Features des gesamten Releases? Es scheint klar, dass Letzteres eigentlich nicht schaden dürfte. Aber wir haben es auch schon erlebt, dass Leute der Meinung waren, zu viel Vorausschau führe zu *Upfront Design* und sei daher zu unterlassen. [1])

Wie auch bei anderen Methoden empfehlen wir: *Vorgehen nach XP bedeutet, im Brain-On-Modus zu arbeiten.*

[1]) *Als „Big Design Upfront" wird der nicht-agile Ansatz bezeichnet, den kompletten Systementwurf im Detail bereits zu Projektbeginn zu erstellen.*

Fehler 6: Naives Design

„Wir machen das, was im Moment den geringsten Aufwand verursacht."

Im Original-XP gab es die Technik *Simple Design*, die mit der Leitlinie gekoppelt war, das Einfachste zu machen, das gerade noch funktionieren könnte („do the simplest thing that could possibly work"). Dieses einfache Design wird häufig verwechselt mit *No-Design* oder simplizistischem Design und missinterpretiert in die Richtung: „Mach das, was jetzt am wenigsten Aufwand verursacht". Insbesondere versäumen viele XP-Entwickler das Aufräumen, nachdem sie eine durch eine Story geforderte Funktionalität implementiert haben. Fast immer bleiben kleine, manchmal auch große Unschönheiten übrig. Erst wenn auch diese beseitigt sind, ist die Story erledigt. Alles andere führt zwangsläufig zu Strukturverlust in der Software.

Mit einfachem Design ist gemeint, Entwurfsaufwände kontinuierlich auf die Entwicklungszeit zu verteilen (also kein langwieriges Detaildesign bei Projektstart) und beim Design einfache Strukturen anzustreben. In XP2 heißt die Technik *Incremental Design*, was den Kern der Sache besser trifft als *Simple Design*.

Ursache für Designschwächen ist mitunter auch fehlendes fachliches Verständnis der Entwickler. Das passiert z. B., wenn Entwickler in einem neuen Fachgebiet arbeiten sollen und direkt mit der ersten Release-Planung begonnen wird und nicht mit einer vorgeschalteten Explorationsphase (vgl. [Wol05]). Generell vernachlässigen viele Teams den fachlichen Entwurf zugunsten von Technologien (EJB etc.). Der fachliche Grundsatzentwurf stellt aber das Gerüst für die Storys dar. Ohne ihn sind Designprobleme vorprogrammiert. (Ein sehr empfehlenswertes Buch zum fachlichen Entwurf ist „Domain Driven Desing" ([Eva03]). In Industrial-XP stellt dies eine eigene Technik dar.)

Gar kein Design zu erstellen, ist verlockend, weil man dann Konflikte um den „richtigen" Entwurf nicht offen austragen muss. Er wird so implizit auf dem Rücken der Software ausgetragen – mit der Folge von Inkonsistenzen.

Bitte bedenken: *„Mache es so einfach wie möglich, aber nicht einfacher" (Albert Einstein). Einfaches Design bedeutet inkrementelles Design und nicht No-Design.*

Fehler 7: Deployment wird unterschätzt

„Wir brauchen vor Auslieferung des Systems an den Kunden keine Testphase mehr. Die Unit-Tests stellen sicher, dass wir keine Fehler mehr produzieren."

Die meisten Teams können das System nicht direkt nach der letzten Entwicklungsiteration an den Kunden ausliefern. Trotz Unit-Tests sind noch zu viele Fehler im System. Dann *muss* eine Stabilisierungsphase durchgeführt werden, in der keine Funktionen realisiert werden (*Feature-Freeze*).

In XP2 gibt es eine neue Technik, das *Daily Deployment*: Jeden Tag soll eine neue Systemversion an den Kunden ausgeliefert werden. Beck warnt davor, die Technik unbedacht einzusetzen. Voraussetzung sei, dass das Team nicht mehr als ein halbes Dutzend Fehler im Jahr produziert. Das werden die meisten Teams niemals hinbekommen. Trotzdem kann *Daily Deployment* als Zielvorstellung für Qualitätsverbesserungen geeignet sein. Es ermahnt uns die Qualität des Systems immer weiter zu verbessern.

Nur Mut: *Deployment direkt nach dem Release ohne weitere Testphase ist die hohe XP-Kunst. Man braucht Jahre, um sie zu erlernen. Viele werden es nie schaffen. Aber versuchen sollte man es weiterhin!*

Fehler 8: Teamgröße skaliert nicht

„In XP haben wir Collective-Ownership. Daher ändere ich überall alles. Das wird schon klappen." „Ich ändere hier gar nichts mehr, weil ich nicht weiß, was passieren wird …"

Collective-Ownership (in XP2 heißt die Technik *Shared Code*) skaliert nicht beliebig. Wächst das Team, entsteht schnell *No-Ownership* und es fühlt sich niemand mehr für die Qualität des Systems verantwortlich. Die Systemqualität und die Entwicklungsgeschwindigkeit sinken dann mit atemberaubender Geschwindigkeit. Die Folge ist sofortige Demotivation der Entwickler.

Collective-Ownership funktioniert nur innerhalb eines Teams, das einen starken Zusammenhalt hat. In vielen Teams funktioniert das bis zu einer Teamgröße von sechs bis acht Entwicklern, ganz wenige Teams können sich bis zu 12 Entwicklern hocharbeiten.

Daher müssen XP-Projekte in Teilprojekte zerlegt werden, wenn viele Entwickler teilnehmen sollen bzw. müssen. Jedes Teilprojekt bekommt sein eigenes Subsystem verantwortlich zugewiesen. Die Teams praktizieren *Collective-Ownership* nur innerhalb ihrer Komponente.

Wichtig ist es, sich darüber klar zu werden, wozu *Collective-Ownership* überhaupt praktiziert wird. Meistens ist das Ziel die Reduktion des „Truck-Faktors" (die Wahrscheinlichkeit, dass das Projekt scheitert, weil ein Mitglied von einem Truck überfahren wird). In den allermeisten Fällen reicht es für die Reduktion dieses Faktors vollkommen aus, wenn drei bis vier Entwickler eine Komponente beherrschen.

Zusammengefasst: *No-Ownership ist etwas anderes als Collective-Ownership. Komponenten beschränken Collective-Ownership auf ein sinnvolles Maß.*

Fehler 9: Falsche Retrospektiven (zu technisch)

„Wir haben nur Probleme mit der Struktur XYZ. Da müssen wir jetzt unbedingt mal ein Refactoring machen." „Wir müssen unbedingt Technologie ABC einsetzen."

Viele Projekt-Retrospektiven haben die Tendenz, zu technisch zu werden. Es fällt Entwicklern deutlich leichter, über versäumte Refactorings, unpassende und gewachsene Strukturen und vermeintlich Heil bringende neue Technologien zu reden als über Defizite im Prozess und in der Kommunikation. In gewisser Weise handelt es sich um einen Spezialfall des oben bereits aufgeführten Kommunikationsproblems: Entwickler scheuen vor notwendiger Kommunikation über den Prozess oder Kommunikationswege zurück und verstecken sich hinter technischen Diskussionen.

Wir empfehlen: *Retrospektiven moderieren lassen. Ein erfahrener Moderator wird die Diskussion wieder auf das wesentliche Thema bringen. Retrospektiven können auch mal unangenehm sein. Konflikte auf der Beziehungsebene lösen sich nicht von selbst.*

Fehler 10: Unechter Kunde

„Mein Kunde weiß gar nicht, was er will." „Warum sollte ich die Story des Kunden so implementieren, wie der das will, wenn der Boss das nachher sowieso ändert. Dann implementiere ich die Story lieber so, wie ich das meine ..."

Neben den vielen bemerkenswerten Änderungen für Entwickler enthält XP auch ein völlig anderes Rollenverständnis für die Kunden. Sehr leicht verlassen sich XP-Entwicklerteams darauf, dass der Kunde schon weiß, was er tut. Da es aber meist die Entwickler sind, die XP einführen (wollen), ist es wohl auch Teil ihrer Aufgabe, dem Kunden bei seiner Rollenfindung behilflich zu sein. Wenn es dem Kunden nämlich nicht gelingt, seine Rolle adäquat auszufüllen, dann kann er die Potenziale agiler Methoden für sein Unternehmen nicht ausnutzen. Auch wenn Kunden ihre Rollen passend ausfüllen, sprechen mehrere Kundenpersonen in größeren Projekten selten mit *einer* Stimme. Im Zweifelsfall redet dann auch schon mal ein Vorgesetzter einem Untergebenen in seine Kundenpriorisierung oder sogar fachlichen Vorstellungen hinein. In solchen Fällen kann es helfen, wenn man explizit die Rolle eines Produktmanagers einführt, der die Anforderungen der vielen Kunden bündelt und dann gegenüber dem Entwicklungsteam auftritt.

Immer beachten: *Die Kundenrolle ist sehr anspruchsvoll. Der Kunde ist verantwortlich für die fachliche Projektplanung.*

Fehler 11: Grandioser Prozess

„Wenn das jemand schafft, dann Sie."

Manche Teams können sehr viel produktiver sein als andere Teams. Dadurch kann der Eindruck entstehen, man könnte mit XP alles schaffen. Das stimmt natürlich nicht. Der Projekterfolg hängt nur zum Teil vom Entwicklungsprozess ab. Eine viel entscheidendere Rolle spielen die am Projekt beteiligten Personen.

Für jedes Team gibt es Probleme, die es nicht lösen kann. Seriöse Teams kennen ihre Grenzen und lehnen unlösbare Aufgaben ab.

Dran denken: *XP ist weder für Unmögliches noch für Wunder zuständig. Was nicht geht, geht nicht. Man muss lernen, „Nein" zu sagen.*

Fehler 12: Kunde unterliegt Entwicklern

„Der Kunde hat doch gar keine Ahnung. Wir Softwareentwickler wissen doch viel besser, was der Kunde braucht."

Besonders bei Produktentwicklung akkumulieren die Entwickler sehr viel Fachwissen. Die Produktmanager (Vertreter für Kunden) haben häufig weniger Fachwissen und daher mit Autoritätsproblemen im Planungsspiel zu kämpfen. Das Problem gibt es in verschiedenen Geschmacksrichtungen:

- Die Entwickler ziehen sich zurück und lassen ihr Fachwissen nicht in die Planung einfließen.
- Die Entwickler machen, was sie für richtig halten, und nicht das, was die Produktmanager entscheiden.
- Es gibt einen faulen Kompromiss zwischen Entwicklern und Produktmanagern (fauler Kompromiss: „Jeder bekommt, was keiner wollte").

Stattdessen braucht man einen gegenseitigen Vorteil, also eine gemeinsame Lösung, die nicht einfach nur akzeptiert wird, sondern auf die sich alle Beteiligten verpflichten. In diese gemeinsame Lösung muss die Fachkompetenz der Entwickler einfließen, ohne dass die Entscheidungskompetenz der Produktmanager in Frage gestellt wird.

Nicht vergessen: *Existierendes Fachwissen muss genutzt werden, egal, wer es besitzt. Die Entscheidungskompetenz bleibt auf jeden Fall bei dem, der die Verantwortung für die fachliche Projektplanung trägt: beim Kunden.*

Fazit

Die beschriebenen Probleme sind spezifisch für agile Methoden und XP und treten in dieser Form in anderen Methoden selten auf. Allerdings liegen die Ursachen für die Probleme außerhalb von XP. Sie führen auch in klassischen Methoden zu Problemen, aber zu anderen als bei XP.

Nach allem, was wir in den letzten fünf Jahren erlebt haben, behaupten wir: Die meisten Teams, die für sich in Anspruch nehmen, XP einzusetzen, erleben nur einen müden Abklatsch von dem, was XP sein kann.

Es geht nicht darum, XP zu praktizieren, um XP zu praktizieren. Die Projektteams müssen Wege finden, sich kontinuierlich

und beharrlich zu verbessern. XP kann auf diesem steinigen Weg sehr hilfreich sein, genauso wie andere agile Methoden und der Blick über den Tellerrand (z. B. zur „Lean-Production" und dem „Toyota-Production-System", nach [Pop03], [Ono95]). ■

Literatur & Links

[BecOO] K. Beck, eXtreme Programming Explained, Addison-Wesley, 2000

[BecO4] K. Beck, eXtreme Programming Explained (2. Auflage), Addison-Wesley, 2004

[BMIO5] Bundesministerium des Inneren, V-Modell XT Release 1.01, 2005 (siehe: www.kbst.bund.de/V-Modell/-,293/V-Modell-XT.htm)

[EvaO3] E. Evans, Domain Driven Design, Addison-Wesley, 2003

[IEEE99] IEEE, Software Engineering Code of Ethics and Professional Practice, 1999, siehe: www.computer.org/certification/ethics.htm

[IXPO4] Industrial-XP, siehe: www.industrialxp.com/

[ManO1] Manifesto for Agile Software Development, 2001, siehe: www.agilemanifesto.org

[OnoO5] T. Ono, Toyota Production System, Productivity Press, 1995

[PopO3] M. Poppendieck, T. Poppendieck, Lean Software Development, Addison-Wesley, 2003

[PreO4] E. Prewitt, The Agile 100, in: CIO-Magazine August 2004 (siehe: www.cio.com/archive/081504/overview.html)

[SchO4] K. Schwaber, Agile Project Management with SCRUM, Prentice Hall, 2004

[StaO4] The Standish Group, Chaos Research Results, 2004, siehe: www.standishgroup.com

[WeiO1] G.M. Weinberg, Secrets of Consulting, Dorset House Publishing, 2001

[WolO5] H. Wolf, S. Roock, M. Lippert, eXtreme Programming – Eine Einführung mit Empfehlungen und Erfahrungen aus der Praxis (2. Aufl.), dpunkt Verlag, 2005