Leichte Prozesse

Motivation und Überblick

Jens Coldewey
Coldewey Consulting
Curd-Jürgens-Str. 4
D-81739 München
Tel: 0700-COLDEWEY
(+49-89-74995702)
Fax: +49-89-74995703
jens_coldewey@acm.org
http://www.coldewey.com

Unsere Wirtschaft hat sich in den vergangenen Jahren radikal verändert. Dem Trend zur Globalisierung steht ein explodierender Markt[1] im Bereich wie e-commerce gegenüber, der von völlig neuen Geschäftsideen und –prozessen getrieben wird. Auch etablierte Unternehmen können sich den Veränderungen nicht mehr entziehen. Neue Marktsegmente tun sich auf Kosten alter Gebiete auf und wer nicht aktiv versucht diese zu besetzen, läuft Gefahr hart erkämpfte Marktanteile zu verlieren oder gar irgendwann ohne Kunden dazustehen. Betriebswirtschaftliche Ideen und technische Machbarkeit treiben sich gegenseitig an und die Änderungen sind längst nicht mehr auf einen Web-Server beschränkt, den man sich im Rechenzentrum unter einen ungenutzten Schreibtisch stellt. Längst sind die Kernsysteme der Unternehmen betroffen, deren über Jahrzehnte gewachsene Struktur den Trend hin zum Kunden nicht mehr unterstützen kann: Der Online-Zugriff auf ein Konto führt direkt zu der Forderung nach sekundengenauer Kontoführung – eine Forderung, der aus Batchsystemen entstandene Anwendungen nicht mehr gewachsen sind. Das Angebot an Versicherungskunden, Schäden über das Internet zu melden führt direkt zu Systemen, die Standardschäden automatisch bearbeiten können.

In dieser Welt der neuen Ideen bedeutet Flexibilität der Geschäftsprozesse einen strategischen Wettbewerbsvorteil, der sich in barer Münze auszahlt. Anders herum kann mangelnde Geschwindigkeit bei der Übernahme neuer Geschäftsideen zum Ruin führen. Neben der Fähigkeit zu immer schnellerer Innovation muß das Unternehmen aber auch in der Lage sein, Probleme im eigenen Geschäftsablauf schnell und flexibel beheben zu können. Die moderne „Lernende Orga-

[1] Angesichts der Börsenentwicklung im letzten halben Jahr ist „explodierend“ durchaus im doppelten Sinn des Wortes zu verstehen.

nisation“ lebt nicht mehr vom der detailliert durchgeplanten und mit der Präzision eines Uhrwerks ablaufenden Bürokratie, sondern von der Kundennähe und der Flexibilität [SKR+94]. Nur wenn die Datenverarbeitung diesen Herausforderungen gewachsen ist, kann sie zur Wertschöpfung beitragen und wird nicht zum störenden Wasserkopf des Unternehmens. Projekte, bei denen mehrere Jahre Entwicklungszeit zwischen Analyse und Produktion sind häufig nicht mehr zu verantworten, sie laufen Gefahr, nach drei Jahren auch die Lösung zu den Problemen vor drei Jahren auszuliefern – Probleme, die bei der derzeitigen Geschwindigkeit der Märkte kaum noch aktuell sein dürften. Um sich dieser Herausforderung zu stellen, sind „Leichte Prozesse“[2] neben komponentenorientierter Entwicklung eine der interessantesten Entwicklungen der letzten Jahre,.

Leichte Prozesse und Änderbarkeit

Was ist nun das Besondere an leichten Prozessen, was ist ihre Grundidee? Das klassische Software-Engineering geht davon aus, daß die Beseitigung von Fehlern um so teurer wird, je später sie entdeckt werden. Boehm unterstellte 1981 einen exponentiellen Kostenanstieg [Boe81], eine Annahme, die so plausibel war, daß sie über eineinhalb Jahrzehnte in keinem Buch über Software-Engineering fehlen durfte – und auch kaum ernsthaft in Frage gestellt wurde. Fast alle Vorgehensmodelle zogen die einzig mögliche Konsequenz aus diesem exponentiellen Anstieg: Man muß Fehler vermeiden, bevor sie sich im Code niederschlagen. Bei einem exponentiellen Anstieg lassen sich fast alle Kosten rechtfertigen, die Fehler frühzeitig aufdecken. „Make it right the first time“ war das erklärte Ziel. Aber gilt dieses Ziel noch heute?

In den letzten zwanzig Jahren haben sich die Werkzeuge der Software-Entwicklung fast ebenso radikal geändert, wie das betriebswirtschaftliche Umfeld. Wurde in den siebziger Jahren noch überwiegend in maschinennahen Sprachen programmiert (Assembler, C, COBOL, FORTRAN), so haben mittlerweile objektorientierte Sprachen wesentliche Marktanteile erworben. Sie bieten mit Datenkapselung eine der wichtigsten Voraussetzungen für änderbare Programme und mit den verschiedenen Formen der Vererbung die Abstraktionsfähigkeit, die notwendig ist, gute Datenkapseln zu finden. Der Durchbruch objektorientierter Entwurfsmuster in den frühen Neunzigern [GHJ+95] ermöglichte es, konsequent auf Entkopplung und Änderbarkeit ausgerichtete Systeme zu entwerfen. Schließlich bieten moderne Werkzeuge große Unterstützung bei Änderungen im Design.

Heute kann man sich also berechtigt die Frage stellen, was wäre, wenn Software plötzlich änderbar wäre? Ist das Dogma der auf Anhieb korrekten Software noch zu rechtfertigen? Oder würde

---

[2] Ich übersetze hier den englischen Begriff „Lightweight Processes“ bewußt mit „Leichten Prozessen“ und nicht mit „Leichtgewichtigen Prozessen“, weil „leichtgewichtig“ als reiner Anglizismus keine andere Bedeutung hat als „leicht“ (trotz dreier zusätzlicher Silben), während im Englischen „lightweight“ auch an das Fliegengewicht im Boxen erinnert.

man einen Entwicklungsprozeß ganz anders aufsetzen, wenn man die Gewißheit hätte, daß die Software änderbar bleibt? Oder anders formuliert: Wie sieht ein Vorgehen aus, das nicht darauf optimiert ist, Fehler möglichst früh aufzudecken, sondern sie möglichst günstig beheben zu können? Aus diesem Gedankenspiel sind die leichten Prozesse entstanden. Vorreiter waren hier Alistair Cockburn mit „Chrystal Clear“ [Coc98], Jim Highsmith mit dem „Adaptive Software Development“ [Hig00], sowie Ward Cunningham und Kent Beck mit ihrem „eXtreme Programming“ [Bec00].

In diesen Prozessen liegt der Schwerpunkt nicht mehr darauf, die Anforderungen möglichst vollständig zu erfassen und in ein möglichst vollständig entworfenes Design zu gießen, bevor man mit der Realisierung beginnt. Statt dessen wird ein großer Teil des Aufwandes dazu verwendet, die Software änderbar zu gestalten. Automatisch ablaufende Testfälle sind in fast allen diesen Methoden Pflicht, auf Codequalität wird bei weitem mehr Wert gelegt, als auf CASE-Modelle. Werden Schwächen oder Redundanzen im Code entdeckt, so werden diese sofort behoben, statt sie durch das weitere Projekt hindurch zu schleifen. Die automatischen Testfälle stellen dabei sicher, daß solche Änderungen nicht zu „Verschlimmbesserungen“ führen. Statt eines groß angelegten Dokumentations- und Berichtswesens setzen diese Prozesse auf direkte Kommunikation innerhalb eines kleinen Teams. Mißverständnisse werden durch gemeinsame Entwicklung vermieden, statt durch Dokumentation.

Diese technischen Aspekte sind gepaart mit hohem Pragmatismus: „Bau das einfachste, was möglicherweise funktionieren könnte“ rät Kent Beck. Die so erstellte Software kann dann dem Bedarf entsprechend erweitert werden. So werden „goldene Henkel“ vermieden, welche die Entwicklungszeit verlängern, ohne echten Mehrwert zu schaffen. Zusätzlich steht das Projekt unter ständigem Auslieferungsdruck. Die Inkrementlängen reichen von wenigen Wochen bis zu drei Monaten, so daß das Team die Auslieferung als routinemäßigen Teil der Projektarbeit erledigt und große Überstundenlasten vermieden werden.

Die bisher mit solchen Vorgehen gemachten Erfahrung zeigen, daß die sich so ergebende Software meist gut änderbar ist. Schon das Originalteam geht binnen weniger Monate in den Wartungsmodus über, so daß es schnell merkt, wo sich das System gut warten läßt und wo es noch verbessert werden muß. Es ist projektentscheidend, daß solche Verbesserungen dann nicht als Rucksäcke auf einem möglicherweise maroden Design aufgesetzt werden, sondern daß stets das System als ganzes weiterentwickelt wird. Es gibt keine sakrosankten Stellen im System, jeder Code ist freigegeben, verbessert zu werden. Dadurch entwickeln sich schnell sehr schlanke und flexible Architekturen. Das Team hat gar keine Zeit, Designs zu entwickeln, die aufwendiger sind, als nötig.

Leichte Prozesse zielen also auf minimalistische, änderbare Software, die in relativ kurzer Zeit bereitgestellt werden kann. Wird bei schweren Prozessen fast jeder Preis bezahlt, um Fehler möglichst früh zu vermeiden, so scheuen leichte Prozesse jede Form von Redundanz. Redundanz

führt dazu, daß Änderungen teuer werden. Dies kann Redundanz innerhalb des Codes sein, wie sie zum Beispiel zwischen verschiedenen SQL-Abfragen an eine Datenbank auftreten, aber auch Redundanz innerhalb der Dokumente. Aus dieser Überlegung heraus wird Entwicklungsdokumentation (wenn sie überhaupt für notwendig erachtet wird) nur als Zwischenergebnis angesehen, das nicht weiter gepflegt wird[3]. Lediglich Bedienungsanleitungen werden als Teil des Endprodukts gepflegt. Das setzt stabile Teams voraus, erleichtert aber die Änderbarkeit des Systems deutlich.

Ich möchte im folgenden auf die einzelnen Bestandteile eines auf Änderbarkeit ausgerichteten Entwicklungszyklus eingehen. Dabei ist es wichtig, daß diese einzelnen Phasen nicht für das gesamte Projekt gleichzeitig gelten. Vielmehr befinden sich die einzelnen Themen, an denen in einem Projekt zu einem Zeitpunkt gearbeitet wird, in verschiedenen Phasen. Eine Synchronisation findet üblicherweise nur für Auslieferungen statt.

Planung

Die Planung besteht normalerweise aus zwei Teilen: Einem sehr groben Gesamtplan, der die Aufteilung von Funktionalität auf die einzelnen Inkremente beschriebt und einem Inkrementenplan, der die Aktivitäten bis zu nächsten Auslieferung beschreibt. Während der Gesamtplan vor allem zur Kommunikation nach außen dient, arbeitet das Team praktisch ausschließlich mit dem Inkrementenplan. Beide Pläne werden im Rhythmus der Inkremente den jeweils neuen Gegebenheiten angepaßt.

Zu Beginn eines Inkrements sehen alle Prozesse einen Planungsworkshop (auch „Inkrementenworkshop“ oder „Planning Game“ genannt) vor, der zumindest den geplanten Umfang des Inkrementes festlegt. Basis für diese Planung sind Anforderungen oder Use Cases, die häufig sehr informell gefaßt sind. Das kann eine Sammlung von Moderationskarten sein in der Schubladen der Projektleiterin oder eine kleine Datenbank, in der alle Teammitglieder ihre Ideen eingetragen haben. Es spielt dabei keine Rolle, ob Anforderungen bereits seit Anfang des Projektes für das anstehende Inkrement vorgesehen worden sind, oder erst am Ende des vorherigen Inkrements aufgetaucht sind. Zu Beginn jedes Inkrements werden alle Karten neu gemischt.

In jedem Fall ist es Aufgabe des Workshops, die noch nicht implementierten Anforderungen zu priorisieren und den jeweils benötigten Aufwand abzuschätzen. Da es sich in der Regel um Aufgaben von wenigen Bearbeitertagen handelt, sind die Schätzungen meist recht brauchbar. Die Priorisierung muß vollständig sein, also eine klare Reihenfolge festlegen, in der die Aufgaben abzuarbeiten sind. Sie wird für jedes Inkrement neu gemacht, wodurch das Projekt flexibel auf geänderte Rahmenbedingungen reagieren kann. Um sicherzustellen, daß dabei die fachlichen

[3] Damit kalkulieren leichte Prozesse das ein, was auch bei schweren Prozessen ohnehin gelebt wurde, auch wenn man dort stets ein schlechtes Gewissen hatte, wenn die Dokumentation veraltet war.

Überlegungen den Ausschlag geben und nicht technische Spielereien, muß mindestens ein Vertreter der Anwender an dem Workshop teilnehmen – auch dies ein Grundmuster leichter Prozesse. Sie alle verlangen, daß ein Anwender oder Fachexperte Vollzeit im Team mitarbeitet. Langwierige Reviewrunden werden so vermieden.

Gegen Ende eines solchen Inkrementenworkshops ist nun die Reihenfolge festgelegt, in der im kommenden Inkrement die Anforderungen angegangen werden. Mit Hilfe der Aufwandsschätzungen ist zudem eine Aussage möglich, welche Themen sicher und welche Themen vermutlich in der nächsten Auslieferung erledigt sein werden. Die leichten Prozesse nutzen alle inhaltliche Puffer statt eines Zeitpuffers in der Planung. Das bedeutet, daß der Auslieferungstermin auf jeden Fall eingehalten wird und der Auslieferungsumfang zum Justieren dient. Niedriger priorisierte Themen fallen so während des Inkrements möglicherweise dem Rotstift zum Opfer und werden auf ein späteres Inkrement verschoben.

Ob die Planung als Netzplan notiert wird, oder wieder als sortierte Kartensammlung vorliegt, hängt vom verwendeten Prozeß ab und den Vorlieben der Projektleiterin. Alles, was funktioniert, ist erlaubt.

Analyse

Die Analyse unterscheidet sich in den einzelnen Prozessen recht stark. Während „eXtreme Programming“ sogenannte „User Stories“ vorsieht, eine sehr informelle Fassung fachlicher Use Cases, ziehe ich persönlich etwas ausführlichere Analysen vor. Unabhängig von der Dokumentation muß die Analyse bestimmte Anforderungen erfüllen, bevor mit Design und Programmierung begonnen werden kann:

- Die fachlichen Anforderungen müssen verstanden worden sein. Der Vollzeit-Anwender steht allen anderen Teammitglieder daher für Fragen zur Verfügung.

- Es muß klar sein, wie diese fachlichen Anforderungen in das System integriert werden sollen. Dabei muß die fachliche Architektur oder auch „Vision“ des Systems berücksichtigt werden. Es muß allen beteiligten Teammitgliedern klar sein, auf welchen fachlichen Modellen das System aufsetzt und wo diese Modelle jeweils erweitert und angepaßt werden können. Hilfreich ist dabei auch die Identifikation von „Streichkandidaten“, also Teilen der Umsetzung, die auch in spätere Inkremente verschoben werden können.

- Es muß klar sein, welche Änderungen an der Bedienoberfläche notwendig sind. Hier müssen alle Teammitglieder die zugehörigen Interaktionskonzepte und mentalen Modelle verstanden haben, auf denen das Oberflächendesign aufbaut.

Welcher Technik man sich bedient, um über diese drei Punkte Klarheit zu gewinnen, ist letztlich nebensächlich, solange der gesteckte Zeitrahmen eingehalten wird, der meist bei wenigen Tagen liegt. Die von mir betreuten Teams verwenden meist kurze Dokumente von fünf bis zehn Seiten,

in denen die Ergebnisse gemeinsamer Sitzungen zusammengefaßt werden. Diese Dokumente werden allerdings nicht gepflegt. Der Einsatz von UML oder CASE-Tools in dieser Phase ist bisher von keinem Team als hilfreich empfunden worden.

Wer monate- oder gar jahrelange Analysephasen gewöhnt ist, mag über die Vorgabe von wenigen Tagen für eine Analyse erstaunt sein. Hier ist es wichtig, sich in Erinnerung zu rufen, daß es ja nicht darum geht, ein Thema vollständig zu durchdringen, sondern einen möglichst einfachen Lösungsweg zu finden. Komplexe Sonderfälle können gegebenenfalls in späteren Inkrementen abgedeckt werden, ebenso Korrekturen, wenn weitere Erfahrungen mit den getroffenen Entscheidungen vorliegen. „Make it right the last time, not the first“ fordert Jim Highsmith.

Design, Programmieren und Ändern

Leichte Prozesse sehen in der Regel keine eigene Designphase vor. Ebenso erfolgt die Übergabe der Analyseergebnisse zur weiteren Realisierung nicht in Papierform, sondern durch einen gleitenden Übergang, während dem sowohl Analytiker als auch Realisierer gemeinsam an dem Thema arbeiten. Während der ganzen Zeit, in der ein Thema bearbeitet wird, müssen alle Beteiligten unabhängig von ihrer Rolle räumlich eng zusammen sitzen, möglichst im gleichen Zimmer, um optimale Kommunikation sicher zu stellen. Manche Prozesse sehen auch gar keine Trennung der Rollen vor. Statt dessen wird ein gesamtes Thema von einem Zweierteam bearbeitet, das alle Phasen gemeinsam durchläuft.

Das Design findet in der Regel nach Bedarf statt. Lassen sich Anforderungen ohne große Änderungen in das System integrieren, ist möglicherweise gar kein eigener Designschritt nötig. Bei Bedarf können Entwickler aber auch ein Designmeeting einberufen, um schwierigere Probleme anzugehen. Es gibt keine obligatorische Dokumentation des Designs, da diese häufig automatisch aus dem Code extrahiert werden kann. Obligatorisch ist aber das Streben nach Minimalismus. Sieht ein Entwickler Teile des Systems, die ähnliche Aufgaben lösen, wie die anstehende, so ist er verpflichtet, die fraglichen Bestandteile so umzubauen, daß sie sowohl für die alte als auch für die neue Aufgabe verwendet werden können. Das Konzept des „Code Ownership“ existiert in solchen Prozessen nicht, jeder ist berechtigt, alles zu ändern. Damit das funktioniert, müssen mehrere Voraussetzungen erfüllt sein:

- Gute Kenntnis des gesamten Codes: Jeder Entwickler muß zumindest Grundkenntnisse über das gesamte System besitzen. Zu jedem Teil des Systems muß es zumindest zwei Personen geben, die sich in dem Code auskennen. eXtreme Programming setzt dabei auf „Pair Programming“, eine Technik, bei der stets zwei Programmierer gemeinsam programmieren. Chrystal Clear empfiehlt eher Code Reviews im gesamten Team, um Wissen zu streuen.

- Gute Kommunikation in Team: Die Entwickler müssen in der Lage sein, Fragen zum Code schnell klären zu können. Die Kollegin, die ein fragliches Stück Code geschrieben hat, muß in Rufweite sitzen, um sie jederzeit hinzuziehen zu können. Die Teamkultur muß angst- und

vorwurfsfreies Diskutieren ermöglichen. Oft genug stellt man nach einigen abfälligen Bemerkungen über ein fragliches Stück Code fest, daß es sich um das eigene Elaborat handelt...

- Automatische Tests: Nach Änderungen an fremdem Code (oder Code, den man selbst vor Monaten oder Jahren geschrieben hat), muß es eine schnelle und einfache Möglichkeit geben, zu prüfen, ob das System noch so gut funktioniert, wie vorher. Testen sollte „nur ein Mausklick entfernt sein“ und die Testfälle sollten dicht genug sein, um Sicherheit zu bieten. Ich komme auf das Testen später nochmals zurück.

- Leistungsfähiges Konfigurationsmanagement: Das Konfigurationsmanagement muß unkompliziert mehrere parallele Threads unterstützen und den gesamten Code möglichst feingranular historisiert speichern. Wenn diese Voraussetzungen erfüllt sind, können die Entwickler auch Experimente wagen, ohne das gesamte System zu gefährden. Das von OTI entwickelte ENVY ist das Paradebeispiel für ein solches Konfigmanagement[4].

Im Laufe der letzten Jahre hat sich herausgestellt, daß immer wieder typische Änderungen am Code vorgenommen werden, um das Design zu verbessern. Diese Änderungen wurden unter dem Schlagwort „Refactoring“ bekannt und von Martin Fowler in einem Buch gesammelt [Fow99].

Eine interessante Variante ist das im „eXtreme Programming“ propagierte Testen vor dem Programmieren. Hier werden zunächst Testfälle geschrieben, die prüfen, ob die Anforderung erfüllt ist. Der eigentliche „Code“ besteht wenn überhaupt aus Breakpoints. Beim Ablaufen der Testfälle geht das System nun in den Debugger, in dem dann der notwendige Code geschrieben wird. Allerdings stellt dieses Vorgehen Anforderungen an die Entwicklungsumgebung, denen nicht jede Umgebung gewachsen ist.

Eine andere Technik, die Codequalität hoch zu halten, sind Codereviews, die vom gesamten Entwicklungsteam wechselseitig durchgeführt werden. Ziel ist nicht so sehr die Prüfung, ob Richtlinien eingehalten wurden, sondern die Frage, wie der Code noch einfacher und übersichtlicher gestaltet werden kann und wie weitere Synergien genutzt werden können. Ein solches Codereview kann den Umfang des Codes hin und wieder um dreißig oder gar fünfzig Prozent reduzieren.

Auch hier wird von einer alten Vorstellung Abstand genommen: Produktivität wird nicht mehr in Zeilen Code pro Zeiteinheit gemessen, vielmehr ist die Vermeidung von Code (und damit von Redundanz und Fehlern) oberstes Ziel.

---

[4] Man beachte, daß die in der klassischen Entwicklung recht aufwendige Integration durch tägliche Zusammenführung („Daily Build“) fast völlig entfällt, wodurch auch die Anforderungen an ein Konfigurationsmanagement anders sind.

Testen

Änderbarkeit setzt voraus, daß die Entwickler nach jeder Änderung sofort prüfen können, ob das System noch wie vorgesehen funktioniert. Dafür sind automatisch ablaufende Tests unabdingbar. Da automatische Oberflächentests sehr schwierig und zeitaufwendig sind, setzen viele Projekte nur automatisch ablaufende Komponententests ein, andere Projekte unterscheiden zwei Testebenen. Die Komponententests werden von den Entwicklern selbst erstellt und sind Bestandteil der Freigabe von Code. Die anfänglichen Tests können sehr pragmatisch erstellt werden, um die Aufwände für die Testfallerstellung im Rahmen zu halten. Wichtig ist es aber, jeden anschließend gefundenen Fehler auch als Lücke in den Testfällen zu begreifen, die zunächst geschlossen werden muß. Erst wenn ein Testfall den Fehler reproduziert, wird der Fehler selbst behoben. Mit dieser Technik entstehen im Laufe der Zeit ausgesprochen zuverlässige Testsuiten, die auch nach komplexen Änderungen noch die Funktion des Systems sicherstellen können.

Voraussetzungen an das Team

Die wichtigste Voraussetzung ist die Größe des Teams: Sie sollte 10 bis 12 Personen nicht übersteigen, weil sonst die direkte Kommunikation zwischen den Teammitgliedern nicht mehr gewährleistet ist. Nachdem die Kommunikationskomplexität mit dem Quadrat der Teamgröße steigt, sinkt die Gesamtproduktivität mit der Teamgröße auch überproportional. Es existieren zwar keine gesicherten Zahlen zu diesem Thema, doch gibt es Grund zu der Vermutung, daß ein gutes Team von 10 Personen mit einem leichten Prozeß eine höhere Gesamtproduktivität erreichen kann, als ein Team von 30 oder 40 Personen mit einem schweren Prozeß. [Cop95]

Weiterhin müssen die Teammitglieder sowohl in ihrer technischen als auch in ihrer sozialen Kompetenz gehobenen Anforderungen entsprechen. Leichte Prozesse erfordern ein hohes Maß an Disziplin, Selbständigkeit und Kooperationsfähigkeit. Das (noch nie zutreffende) stereotyp vom asozialen Computerfreak hat endgültig hier ausgedient.

Zusammenfassung

Um den Anforderungen der heutigen Wirtschaft gerecht zu werden, muß die Entwicklung von Software deutlich schneller und sicherer werden. Leichte Prozesse sind ein Ansatz, dieser Herausforderung zu begegnen. Sie nehmen fachliche Fehler bewußt in Kauf und lenken ihre Aufmerksamkeit statt dessen auf hohe Änderbarkeit. Zusätzlich findet die Kommunikation innerhalb des Teams nicht über Dokumente und Modelle, sondern über direkten Austausch, Code und gemeinsame Arbeit statt. Enge, automatische Tests und ein starker Fokus auf die fachliche Priorisierung zeichnet die derzeitigen Prozessmodelle ebenso aus, wie ein stark inkrementelles Vorgehen.

Literatur

[Boe81] **Barry Boehm:** *Software Engineering Economics;* Prentice Hall, 1981, ISBN 0-138-22122-7

[Bec00] **Kent Beck:** *Extreme Programming Explained - Embrace Changes;* Addison-Wesley, Reading, Massachusetts, 2000; ISBN 0-201-61641-6

[Coc98] **Alistair Cockburn:** *Crystal "Clear" - A Human-Powered Software Development Methodology For Small Teams*, 1998; erhältlich über http://members.aol.com/acockburn; erscheint voraussichtlich 2002

[Cop95] **James Coplien:** *A Generative Development-Process Pattern Language* in: **James Coplien, Doug Schmidt (eds.):** *Pattern Languages of Program Design;* Addison-Wesley, Reading, Massachusetts, 1995; ISBN 0-201-60734-4

[Fow99] **Martin Fowler:** *Refactoring - Improving the Design of Existing Code;* Addison-Wesley, Reading, Massachusetts, 1999; ISBN 0-201-48567-2

[GHJ+95] **Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides:** *Design Patterns - Elements of Reusable Object-Oriented Software;* Addison-Wesley, Reading, Massachusetts, 1995; ISBN 0-201-63361-2

[Hig00] **James A. Highsmith III:** *Adaptive Software Development - A Collaborative Approach to Managing Complex Systems;* Dorset House, New York, 2000; ISBN 0-932633-40-4

[SKR+94] **Peter Senge, Art Kleiner, Charlotte Roberts, Richard Ross, Bryan Smith:** *The Fifth Discipline Fieldbook - Strategies and Tools for Building a Learning Organization;* Doubleday Inc, New York, 1994; ISBN 0-385-47256-0