Schlanke Produktionsweisen in der modernen Softwareentwicklung

Die Autoren

Frank Padberg
Walter Tichy

Dr. Frank Padberg und
Prof. Dr. Walter Tichy
Universität Karlsruhe
Fakultät für Informatik
Am Fasanengarten 5
76131 Karlsruhe
{padberg | tichy}@ipd.uni-karlsruhe.de

1 Einleitung

1.1 Stand der Softwareentwicklung

Die traditionellen Prozesse, mit denen Software heutzutage oft noch entwickelt wird, zeigen deutliche – manche würden sagen: erschreckende – Ähnlichkeit mit dem Stand der Produktionsprozesse in der europäischen und amerikanischen Automobilindustrie Ende der 80er-Jahre, wie er in der bekannten MIT-Studie [WoJR91] gut verständlich beschrieben wurde. Viele Softwareprozesse sind immer noch eng an Phasen orientiert und schwerfällig, und das trotz der Einsicht, dass Softwareentwicklung ein iterativer Prozess ist [LaBa03]. Schon den Begründern des bekannten Wasserfallmodells war das Auftreten von Iterationen bewusst [Royc70].

In traditionellen Softwareprozessen werden nicht nur die Softwareprodukte der einzelnen Phasen (wie Anforderungen, Entwurf oder Code), sondern auch das Vorgehen selbst mit großem Aufwand dokumentiert (Prozessformulare). Trotz aller Planung kann man beobachten, dass in der Entwicklung häufig „den Problemen hinterhergelaufen“ wird. Das gilt nicht nur für das Beheben von Fehlern, die in früheren Phasen der Entwicklung entstanden sind, sondern auch für die offenkundigen Schwierigkeiten, mit Anforderungen umzugehen, die instabil sind. Instabile Softwareanforderungen sind heute eher die Regel als die Ausnahme; sei es, weil die Anforderungen auf dem Markt des Kunden schnell wechseln oder weil sich erst nach und nach im Dialog mit dem Kunden herausarbeiten lässt, wie sich seine Arbeitsabläufe durch Software am besten unterstützen lassen.

Als Folge dieser traditionellen, starren Entwicklungsweise dauern viele Softwareprojekte zu lang und scheitern zu oft. Häufig ist die gelieferte Software fehlerhaft und erfüllt nicht die wirklichen Anforderungen des Kunden; nachträgliche Änderungen an der Software verursachen dann hohe Kosten. Trotz gelegentlicher Kritik an seinen Umfragen zeigt der regelmäßig veröffentlichte CHAOS-Report der Standish Group diese anhaltenden Probleme deutlich auf [Stan04].

1.2 Schlankes Produzieren

Die Grundgedanken einer schlanken Produktionsweise (*lean production*) wurden im Japan der Nachkriegszeit im Produktionssystem bei Toyota entwickelt. Die Studie [WoJR91, 49–55] beschreibt die we-

Kernpunkte

Der Beitrag zeigt am Beispiel von Extreme Programming (XP), dass agile Softwareentwicklungsmethoden auf den Ideen der Lean Production beruhen.

- XP erreicht eine extreme Qualitätssicherung durch Paarprogrammierung, testgetriebenes Entwickeln, häufiges Refaktorisieren und laufende Codeintegration.
- XP arbeitet mit extremer Kundenorientierung durch ständige Einbindung eines Kundenmitarbeiters, iteratives Planungsspiel, häufige Akzeptanztests und inkrementelle Auslieferung.
- XP erreicht eine extreme Verschlankung des Entwicklungsprozesses durch das konsequente Weglassen von Entwurf, Dokumentation und aufwendigen Koordinierungsmechanismen.

Die Verankerung in der Lean Production liefert eine konzeptionelle Erklärung, warum agile Methoden kostengünstig zu hochwertiger Software führen. Erkenntnisse über schlanke Vorgehensweisen in anderen Wirtschaftsbereichen können auf die Softwareentwicklung übertragen werden.

Stichworte: Extreme Programming, Agile Methoden, Schlanke Produktion, Softwareprozessmodelle

sentlichen Umstände, die dazu geführt haben, dass man bei Toyota nicht versuchte, die damals vorherrschende Massenproduktion [WoJR91, 30–43] einzuführen, sondern sich auf eine grundlegend neue Ausrichtung der Produktionsabläufe [Kraf88] konzentrierte. Wir fassen diese Umstände im folgenden Absatz kurz zusammen, um die weiter unten formulierten schlanken Prinzipien zu motivieren.

Zur Massenproduktion waren hochspezialisierte, teure Maschinen nötig, die mit hohen Stückzahlen laufen mussten, um rentabel zu sein. Ein schneller Wechsel in der Produktion auf andere Produkttypen war nicht möglich. Die Arbeiter waren auf wenige Handgriffe zur Bedienung einer bestimmten Maschine festgelegt und daher leicht austauschbar. In Japan hingegen war nach dem Krieg wenig Geld für Investitionen vorhanden. Der japanische Automobilmarkt war klein und verlangte nicht nach hohen Stückzahlen, sondern nach einer Vielzahl verschiedener Modelltypen, also Flexibilität in der Produktion. Bei Toyota waren die Arbeiter als Ergebnis eines Streiks unkündbar und stellten ein wertvolles Gut für das Unternehmen dar.

Die Kernidee der schlanken Produktionsweise besteht darin, *Verschwendung* im Produktionsprozess zu vermeiden und alle Abläufe soweit wie möglich auf die eigentliche Wertschöpfung auszurichten. Als Beispiele für Verschwendung gelten große Materialpuffer in der Produktionskette, große Lagerbestände, Materialausschuss und mangelhafte Produkte, die nachgebessert werden müssen. Bei Toyota wurden sieben unterschiedliche Arten der Verschwendung ausgemacht [Ohno93, 46]. Die grundlegenden Prinzipien einer *schlanken* Produktionsweise wurden wie folgt formuliert:

P1. Build only what is needed.
P2. Eliminate anything which does not add value.
P3. Stop if something goes wrong.

Das erste Prinzip zielt darauf ab, nur solche Dinge zu produzieren, für die ein Abnehmer vorhanden ist. Das bezieht sich sowohl auf das Endprodukt für den Kunden, als auch auf alle Zwischenschritte im Produktionsprozess. Es soll keine Lagerhaltung und kaum noch Materialpuffer in der Produktion geben; nichts soll auf Vorrat produziert werden.

Das zweite Prinzip will erreichen, dass nur produziert wird, was für den Kunden tatsächlich einen Wert hat, und dass nur solche Tätigkeiten in der Produktion ausgeführt werden, die zu diesem Wert beitragen. Alles andere soll weggelassen werden.

Das dritte Prinzip soll dazu führen, dass Probleme und Fehler so früh wie nur möglich entdeckt und nachhaltig an der Stelle behoben werden, an der sie entstehen. Das setzt voraus, dass Verantwortung im Produktionsprozess zum Teil an die Arbeiter übergeben wird. Sichtbares Zeichen dieses Prinzips bei Toyota war die Möglichkeit für jeden Arbeiter, das Band bei Problemen anzuhalten.

1.3 Agile Methoden

Agile Entwicklungsmethoden bieten eine Alternative zu herkömmlichen, planungs- und dokumentationszentrierten Vorgehensmodellen in der Softwareentwicklung. Die bekannteste agile Methode ist *Extreme Programming* (XP) [Beck99; Beck04]. Bei einer Umfrage unter 200 IS- und IT-Managern aus einem weiten Querschnitt von Ländern und Industriezweigen gaben 38 Prozent der Manager, in deren Firma agile Methoden eingesetzt werden, an, XP zu verwenden [Char01]. Andere bekannte agile Methoden sind ASD [High00], Scrum [ScBe01], FDD [PaFe02] und Crystal [Cock02].

Alle agilen Methoden haben als gemeinsames Ziel, dass sie Software schnell, flexibel und mit hoher Qualität entwickeln wollen. Alle agilen Methoden versuchen, sich besonders flexibel auf die Kundenwünsche einzustellen und möglichst viel Ballast in der Softwareentwicklung abzuwerfen – das erinnert doch sehr an die Prinzipien einer schlanken Produktionsweise! Wichtige Vertreter der agilen Methoden haben sich 2001 auf gemeinsame Wertvorstellungen geeinigt, die im „Agilen Manifest“ festgehalten sind [Agil01].

1.4 Methodik dieses Beitrags

Die wissenschaftlich-empirische Bewertung von XP und seinen Einzeltechniken (wie Paarprogrammierung oder testgetriebenes Entwickeln) steht noch am Anfang [MüPT05]. Das gilt auch für die anderen agilen Methoden. Gleichzeitig nimmt die Verbreitung agiler Methoden immer weiter zu [Char01], auch in großen deutschen Firmen [MaSc04; Ster06]. Deshalb ist es notwendig, trotz des Mangels an gesicherten empirischen Erkenntnissen schon jetzt zu einem fundierten Verständnis und einer tragfähigen Einschätzung agiler Methoden zu kommen – gerade für den Wirtschaftsinformatiker in der Praxis, der gefragt wird, ob und in welchem Umfang sein Unternehmen agile Entwicklungsmethoden einsetzen soll. Dafür bieten wir mit diesem Beitrag eine *konzeptionelle* Grundlage an.

Während bisherige wissenschaftliche Studien versuchen, jede einzelne XP-Technik für sich betrachtet zu verstehen, nehmen wir in diesem Beitrag eine ganzheitliche Betrachtungsweise ein. Wir zeigen auf, dass XP insgesamt betrachtet in hohem Maß auf den bewährten Prinzipien einer schlanken Produktionsweise beruht. Wir sehen in dieser konzeptionellen Verankerung von XP (und anderen agilen Methoden) in der *Lean Production* den tieferen Grund dafür, dass XP in der Praxis erfolgreich ist, also zu einer kostengünstigen und effizienten Entwicklung von hochwertiger Software führen kann. Eine konzeptionelle Verankerung von XP halten wir für ein wichtiges Gegenstück zur rein ökonomischen Bewertung von XP-Projekten [MüPa03]. Durch den Vergleich mit den Ideen und Prinzipien der *Lean Production*

- ist es möglich, XP auf konzeptioneller Ebene zu verstehen und einzuschätzen;
- wird deutlich, dass die XP-Techniken einen inneren Zusammenhang haben;
- wird es möglich, Erkenntnisse über schlanke Vorgehensweisen aus anderen Wirtschaftszweigen auf die Softwareentwicklung zu übertragen.

Die Ideen und Prinzipien der *Lean Production* sind nicht auf die Fertigung beschränkt, sondern werden in verschiedenen Wirtschaftsbereichen erfolgreich umgesetzt [WoJo03]. Beispiele sind der Gesundheitssektor [JoMi06], die Logistik [Baud05], die Bauindustrie [Howe99] und die Luftfahrtindustrie [MuAC02]. Schon bei Toyota wurden schlanke Prinzipien auch auf die technische Entwicklung von Fahrzeugen angewandt [WoJR91, 112–119]. Es ist also durchaus sinnvoll, die Übertragung schlanker Prinzipien auf die Softwareentwicklung zu untersuchen, auch wenn es sich bei der Entstehung von Software um einen Entwicklungsprozess und nicht um einen Produktionsprozess handelt.

1.5 Verwandte Arbeiten

Vor uns haben erst wenige Autoren den Zusammenhang zwischen agilen Softwareentwicklungsmethoden und den Konzepten der *Lean Production* betrachtet [Popp03; Harv04]. Es ist aber anzunehmen, dass die frühen Protagonisten agiler Methoden über schlanke Produktionsweisen im Bild waren, auch wenn das in den frühen Publikationen nicht erkennbar wird [C3Te98; Beck99]. Erst in der zweiten Auflage des Standardwerkes über XP [Beck04, 135–137] wird in einem dreiseitigen Abschnitt auf das Pro-

2071038 Wirtschaftsinformatik Heft 3/2007 3B2 Art. Pos. 23 (Autor: Frank Padberg) w:/p_3/wir/wir07_03/00613/wir00613u.3d
Stand: 10. 4. 2007 (gelieferte Daten) Bearbeiter: Ha./Storch

duktionssystem von Toyota hingewiesen. Lesenswert ist das Buch zweier US-Unternehmensberater über *Lean Software Development* [PoPo03]. Die Autoren orientieren sich an den Prinzipien des *„lean thinking“*, die im Nachfolgewerk [WoJo03] der Studie aus den 80ern über Toyota entwickelt wurden. Das Buch ist allerdings nicht so konkret und detailliert auf einen bestimmten agilen Ansatz ausgerichtet wie unser vorliegender Beitrag über XP. Neben den oben genannten Arbeiten (s. Abschnitt 1.4) über schlanke Methoden in anderen Wirtschaftszweigen gibt es auch Studien, die sich mit der Einführung schlanker Ideen in der Produktentwicklung befassen, zum Beispiel [KaAh96].

2 Verschwendung in der Softwareentwicklung

Auch in der Softwareentwicklung lassen sich Formen der Verschwendung im Sinn des schlanken Produzierens aufdecken. Allen voran enthält Software nach Umfragen zum CHAOS-Report des Jahres 2000 einen erstaunlich hohen Anteil an Funktionen, die gar nicht (45 %), selten (19 %) oder nur manchmal (16 %) verwendet werden [John02]. Nicht alle selten genutzten Funktionen einer Software sind wirklich unnötig – wer möchte schon auf die Möglichkeit verzichten, gelegentlich die Einstellungen in seinem Internet-Browser anzupassen? Trotzdem sind solche Umfrageergebnisse aufschlussreich.

Unstrittig bleibt, dass die Entwicklung unnötiger Funktionen eine Verschwendung von Ressourcen auf Seiten des Kunden und der Softwarefirma bedeutet. Das schließt das in der Softwaretechnik bekannte „gold plating“ durch die Entwickler mit ein, also das unnötige Polieren und Erweitern von Code, der die Anforderungen bereits gut erfüllt [McCo96]. Daraus ergibt sich als *Forderung* für eine schlanke Produktionsweise in der Softwareentwicklung, die Liste der wirklich nötigen Funktionen gemeinsam mit dem Kunden zu erarbeiten und vom Kunden priorisieren zu lassen; schließlich muss der Kunde nachher mit der Software arbeiten. Nur für den Kunden wichtige Funktionen dürfen entwickelt werden.

Eine ähnliche Argumentation gilt für Anforderungen, die nicht zügig umgesetzt werden [Popp03]. Das führt häufig zu unvollständigen Teilprodukten, die meist eine zeitlang „mitgeschleppt“, später aber oft verworfen werden. Die Arbeit war dann umsonst. Daraus ergibt sich als *Forderung*, nur an solchen Funktionen zu arbeiten, die sich bereits zum gegenwärtigen Zeitpunkt zügig und vollständig realisieren lassen.

Eine klare Quelle für Verschwendung sind Softwarefehler (nicht nur im Code, sondern auch in der Spezifikation oder dem Entwurf). Es ist bekannt, dass der Aufwand zum Beheben eines Softwarefehlers umso größer ist, je später er entdeckt wird. Daraus ergibt sich als *Forderung* für eine schlanke Produktionsweise, Softwarefehler so früh wie nur möglich in der Entwicklung zu finden und zu beheben.

Eine andere mögliche Quelle von Verschwendung sind Aktivitäten im Softwareprozess, die nicht unmittelbar zur Entstehung des Produktes – letzten Endes also des Programmcodes – beitragen. Dazu gehören organisatorische Tätigkeiten wie das Ausfüllen von Statusberichten, aber auch die Änderungsverwaltung (*change management*) und die Übergabe- und Abnahmeprozeduren für Teilprodukte. Viele dieser Aktivitäten liegen auf dem kritischen Pfad des Projekts und verzögern die Fertigstellung des Codes. Aus diesen Überlegungen ergibt sich als *Forderung* für ein schlankes Produzieren, nur solche Aktivitäten zu erlauben, die unverzichtbar sind, um das eigentliche Softwareprodukt herzustellen. Welche Aktivitäten wirklich unverzichtbar sind, ist aber nicht klar – schon über Art und Umfang der Aktivitäten zur Qualitätssicherung wird seit jeher in der softwaretechnischen Forschung und Praxis diskutiert.

3 Extreme Programming als eine Form der schlanken Produktionsweise

XP besteht aus einem Bündel von Techniken, die wir in diesem Abschnitt vorstellen und im Hinblick darauf bewerten, welche der schlanken Prinzipien (P1–P3) sie umsetzen – also wie schlank sie sind. Die grundlegenden Referenzen für XP und diesen Abschnitt sind [Beck99; Beck04]. Die meisten XP-Techniken waren in der einen oder anderen Form schon vorher bekannt, ihre Anwendung und ihre Kombination sind bei XP aber extrem. Das kann man zum Beispiel an der Technik des testgetriebenen Entwickelns (s. Abschnitt 3.1.2) im Vergleich zum herkömmlichen Testen gut erkennen.

XP zielt darauf ab, den oben beschriebenen Formen von Verschwendung in der Softwareentwicklung drastisch entgegenzuwirken: durch eine extreme *Kundenorientierung*, eine extreme *Qualitätssicherung* und eine extreme *Verschlankung* der Softwareentwicklung. Einer der Väter von XP hat die Essenz von XP so formuliert [Beck98]:

„Listening, Testing, Coding, Designing. That's all there is to software.“

Um Missverständnissen vorzubeugen: Beck versteht unter *„designing“* nicht einen expliziten, eigenständigen Entwurf der Software, sondern das Entstehen einer guten inneren Struktur des Codes während des Programmierens.

3.1 Paarprogrammierung und testgetriebenes Entwickeln

3.1.1 Paarprogrammierung

Paarprogrammierung (*pair programming*) bedeutet, dass zwei Entwickler vor einem Rechner sitzen (eine Tastatur, eine Maus) und die Programmieraufgabe gemeinsam lösen. Paarprogrammierung wurde erstmals von [Copl94] beschrieben. Die zugrunde liegende Idee ist nach [Beck04] einfach: Inspektionen (oder in vereinfachter Form: Durchsichten) sind ein bewährtes Mittel, um Fehler in Softwaredokumenten aller Art zu finden, nach dem Prinzip „Vier Augen sehen mehr als zwei“. Warum also nicht eine Art *ständige* Durchsicht während des Codierens einführen? Bei der Paarprogrammierung sitzt der eine Entwickler an der Tastatur und programmiert, der andere Entwickler macht Vorschläge, diskutiert Entscheidungen und macht auf mögliche Fehler aufmerksam, sobald sie entstehen. Das Ziel der Paarprogrammierung liegt also in einer massiven Verbesserung der Codequalität.

Die Paarprogrammierung macht Spaß, bringt Abwechslung in den Programmieralltag und wird gerne von den Entwicklern angenommen [WKCJ00; MüTi01]. Trotzdem ist Paarprogrammierung nicht ohne Probleme. Bei aller Begeisterung der Entwickler am Anfang gibt es Anzeichen, dass sich der positive Effekt bald abnutzt [AbKo04]. Die neue Technik ist irgendwann nicht mehr spannend, sondern wird genauso zur Routine wie andere Techniken. Außerdem hält die Arbeit zu zweit bisweilen auf. In Situationen, bei denen der Weg zur Problemlösung klar ist, arbeitet man alleine oft schneller.

Es ist weitgehend unklar, wann zwei Entwickler ein gutes Programmiererpaar bilden [PaMü04]. Die Entwickler dürfen nicht zu ungleich sein, sonst dominiert der

Entwickler, der erfahrener ist oder mehr Kenntnisse im Anwendungsgebiet und der gerade benutzten Technologie hat. Das Wissen und die Erfahrungen der beiden Entwickler dürfen aber auch nicht zu ähnlich sein, sonst kommen keine neuen Ideen auf; vor allem neigen die Entwickler dann zu denselben Fehlern. Zu dieser Frage sind erste ethnographische Studien erschienen [ShRo04; SSAD06; BrRB06].

3.1.2 Testgetriebenes Entwickeln

Bei XP wird die Programmierung von ständigem Testen „angetrieben“ (*test-driven development*). Es gibt zwei Arten von Tests: Programmierertests (*unit tests*) und Benutzertests (*acceptance tests*).

Die Programmierertests entsprechen den Modultests im herkömmlichen Entwicklungsprozess. Der entscheidende Unterschied zum klassischen Codieren mit Testen besteht darin, dass der Entwickler *zuerst* die Testfälle schreibt, die ein Codestück später bestehen muss; erst danach schreibt der Entwickler das eigentliche Codestück. Testfälle und Produktivcode werden dabei inkrementell und verzahnt erstellt. Damit soll erreicht werden, dass viel intensiver getestet und die Codequalität massiv verbessert wird.

Nur solcher Code darf in das Versionskontrollsystem eingegeben werden, der alle zugehörigen Testfälle besteht. Testgetriebenes Entwickeln führt zu extrem kurzen Zyklen aus Testen und Codieren (oft sogar im Fünfminutentakt), also zu einer extremen Form des iterativen Entwickelns [LaBa03]. Testgetriebenes Entwickeln ist nicht einfach zu vermitteln, denn Programmierer tun sich schwer mit der „Umkehrung“ der Reihenfolge von Codieren und Testen [MüHa02; MaSc04].

Um zu prüfen, ob der Code die Anforderungen erfüllt, werden vom Kunden vor Ort Akzeptanztests (*customer tests, acceptance tests*) ausgeführt, die auf den Benutzungsszenarien aufbauen. Die Tests sollen mit jeder neuen Version der Software (meist täglich, s. Abschnitt 3.3.2), zumindest aber am Ende jeder Iteration des Planungsspiels (s. Abschnitt 3.2.1) ausgeführt werden. Beim Erstellen der Tests wird der Kunde von den Entwicklern unterstützt. Der Kunde weiß, was auf welche Weise funktionieren soll, der Entwickler hilft, ausführbare Testfälle zu bauen.

3.1.3 Testautomatisierung

Testgetriebenes Entwickeln mit seinen extrem kurzen Zyklen aus Testen und Codieren ist nur dann praktikabel, wenn die Tests *automatisiert* sind; das manuelle Testen wäre einfach zu zeitraubend. Daher ist schon früh ein Werkzeug entstanden, mit dem die Modultests des Programmierers automatisierbar sind (sUnit für Smalltalk; später das bekannte jUnit für Java). Das Ziel ist eine starke Vereinfachung des Testablaufs.

Die Testautomatisierung macht es einfach, den eigenen (oder auch fremden) Code häufig zu testen, führt also in der Regel zu viel intensiverem Testen als es beim klassischen Entwicklungsprozess mit einer nachgelagerten Testphase üblich ist (die häufig auch noch dem Zeitdruck im Projekt zum Opfer fällt). Es könnte daher sein, dass ein erheblicher Teil des Nutzens der testgetriebenen Entwicklung schon durch die Testautomatisierung allein entsteht und erst in zweiter Linie durch das Voranstellen des Schreibens der Testfälle. Diese Tatsache wird in der Diskussion häufig übersehen und führt zu einem typischen Fehler in vielen empirischen Studien [MüHa02]. Dort wird testgetriebenes Entwickeln einschließlich Testautomatisierung gegen „normales“ Testen *ohne* Testautomatisierung verglichen. Somit hat man *zwei* unabhängige Variablen im Experiment und kann dann nicht mehr sicher schließen, ob beobachtete Qualitätsunterschiede nur auf das Voranstellen der Tests zurückgehen.

3.1.4 Testauswahl

Eine empirische Studie [MüHa02] mit Studenten zeigt, dass ein gewisses Risiko besteht, sich durch das Schreiben der vielen Testfälle zu früh in Sicherheit zu glauben. Es ist und bleibt eine grundsätzliche Schwierigkeit des Testens, „die richtigen“ Testfälle aus den Anforderungen (bzw. der Spezifikation oder dem Entwurf) abzuleiten, also solche Testfälle, mit denen auch tatsächlich Fehler entdeckt werden können. Daran hat sich durch die testgetriebene Entwicklung nichts geändert [MaSc04]. Es ist aus gutem Grund ein wichtiges Prinzip beim herkömmlichen Testen, dass der Tester (der die Testfälle schreibt) und der Entwickler des Codes verschiedene Personen sein sollen. Dieses Problem ist bisher in XP ungeklärt.

Dazu kommt, dass die Anzahl der Testfälle erfahrungsgemäß stark ansteigt. Es ist trotz Testautomatisierung nicht sinnvoll, beliebig große Testsuiten häufig ablaufen zu lassen. Es ist daher notwendig, geeignete Testfälle auszuwählen. Das Problem der richtigen Testauswahl ist schwierig und Gegenstand vieler wissenschaftlicher Untersuchungen in den letzten Jahren, wird aber in XP bisher ignoriert.

3.1.5 Fehlen von Entwurf und Dokumentation

In XP *ersetzen* die Testfälle sowohl eine genaue Spezifikation der Anforderungen als auch den Entwurf. Man muss sich klarmachen, dass es *in XP keinen eigenständigen Entwurf gibt*. Die Testfälle werden direkt aus den Benutzungsszenarien („user stories“, s. Abschnitt 3.2.1) abgeleitet. Die Entwurfsprobleme werden iterativ direkt beim Programmieren gelöst.

Natürlich entsteht beim Programmieren eine innere Struktur der Software, die man als „inhärenten Entwurf“ bezeichnen und aus dem Code extrahieren könnte. Es ist offen, welche Qualität (etwa bezüglich Modularität und Wartbarkeit) dieser inhärente Entwurf im Vergleich zu vorab erstellten Entwürfen aufweist.

XP geht aber noch weiter und hinterfragt *jedes* Nebenprodukt der Entwicklung, das nicht Code ist. XP verlangt *keine Dokumentation* außer den Testfällen und den Benutzungsszenarien. Wie schon in den Anfängen der Programmierung gilt der Code als die einzige verlässliche Quelle [KePl74]. Dadurch spart XP den Aufwand, Code, Entwurf und Dokumentation auf dem gleichen Stand zu halten.

3.1.6 Bewertung

Paarprogrammierung und testgetriebenes Entwickeln sind extreme Formen der Qualitätssicherung – beide Techniken wollen Fehler im Produkt (also dem Code) unmittelbar nach ihrem Entstehen entdecken oder gleich ganz vermeiden, ganz im Sinn der Forderungen für eine schlanke Produktionsweise („stop if something goes wrong“). Beide Techniken sind betont informell und auch so gesehen schlank. Prozessdokumente gibt es im Gegensatz zu Inspektionen und klassischen Testphasen nicht, den beiden gängigsten Techniken der Qualitätssicherung.

Die Idee, Fehler in der Software möglichst gar nicht erst entstehen zu lassen, ist nicht neu. Zum Beispiel wurde bei IBM mit dem *Cleanroom*-Prozess [MiDL87] versucht, durch verstärkten Einsatz von formalen Spezifikationen und Inspektionen von vornherein nur fehlerfreie Zwischenprodukte zu erzeugen. Dieser Ansatz hat sich aber wegen der hohen Kosten nicht durchgesetzt. Es stellt sich die Frage, wie kosteneffektiv die XP-Techniken der Paar-

programmierung und des testgetriebenen Entwickelns sind [PaMü03; MüPa03].

Ein großer Teil der Qualitätssicherung ruht bei XP auf den Testfällen. Entsprechend viel Testcode entsteht während der Entwicklung und das ist nicht gerade schlank, auch wenn das Durchlaufen der Tests durch Werkzeuge zur Testautomatisierung stark verschlankt ist. Dazu kommt das Problem der geeigneten Wahl der Testfälle.

Das Fehlen eines eigenständigen Entwurfs und der üblichen Dokumentation ist ausgesprochen schlank und folgt dem Prinzip „eliminate anything which does not add value“. Ein wichtiges Argument ist, dass es sehr aufwendig und in der Praxis oft erfolglos, also Verschwendung ist, den Code, den Entwurf und die Dokumentation auf dem gleichen Stand zu halten. Es bleibt dabei offen, wie sich das Fehlen der Dokumentation auf die Kosten einer späteren Wartung des Codes auswirkt. Auch noch so „schlank“ produzierter Code wird sich nach Fertigstellung mit neuen Anforderungen des Kunden weiterentwickeln müssen; spätestens dann sollte sich eigentlich der Wert von Entwürfen und von Dokumentation zeigen. Eine neuere empirische Studie etwa belegt den Nutzen von UML-Entwurfsdiagrammen bei Wartungsaufgaben [ABHL06].

Die Meinungen gehen hier weit auseinander. Ist maßvolle Dokumentation wirklich Verschwendung oder doch ein notwendiger Teil der Software? Die Vertreter agiler Methoden nehmen dazu eine extreme Position ein: alles außer lauffähigem Code und den für die Qualität unverzichtbaren Testfällen wird als Verschwendung gesehen.

3.2 Planungsspiel, inkrementelle Auslieferung und Kunde im Team

3.2.1 Planungsspiel

Das Planungsspiel (*planning game*) dient zum Erfassen der Anforderungen. Die Entwickler erarbeiten dabei mit dem Kunden die Benutzungsszenarien (*user stories*) für die Software. Die Szenarien werden auf Karteikarten (*index cards*) festgehalten. Der Kunde priorisiert die Szenarien und wählt diejenigen aus, die als nächstes realisiert werden sollen. Erste Studien deuten darauf hin, dass das Planungsspiel eine gute Methode ist, um Anforderungen zu erfassen und zu priorisieren [KTRB06]. Das Planungsspiel wird regelmäßig wiederholt (Iterationen), um weitere Anforderungen zu klären und neue Informationen über die Anforderungen zu berücksichtigen.

Ein Benutzungsszenario muss auf eine Karteikarte passen, sonst muss die zugehörige Anforderung zerlegt werden. Da in den Iterationen die Anforderungen nur schrittweise herausgearbeitet werden, werden die Karteikarten oft umgeschrieben, aufgeteilt und verworfen. Das führt häufig auch zu Umstrukturierungen im Code.

3.2.2 Inkrementelle Auslieferung

In XP werden dem Kunden in kurzen Abständen (Tage bis wenige Wochen) neue Versionen der Software übergeben (*incremental delivery*). Durch den kurzen Abstand sind die Unterschiede zwischen zwei aufeinander folgenden Versionen eher klein (man spricht hier von *small releases*). Trotzdem muss jede Version für den Kunden nützlich sein und eine Verbesserung der Funktionalität gegenüber der früheren Version bieten. Das geht weit über die herkömmliche Prototypentwicklung hinaus, bei der ein Prototyp nur als Hilfsmittel dient, um zu prüfen, ob die Software wie gewünscht aussieht oder um die Benutzungsszenarien mit dem Kunden festzulegen.

Inkrementelle Auslieferung und die Iterationen des Planungsspiels wirken zusammen und richten die Entwicklung stärker auf den Nutzen für den Kunden aus als bei herkömmlichen Projekten. Die wichtigsten Funktionen sollen zuerst implementiert werden; dann kann man auch immer wieder eine nützliche neue Version erstellen. Je nach der Art der Software arbeitet man aber auch bei XP in der Praxis eher in Zyklen von mehreren Wochen [ElSc02].

3.2.3 Kunde im Team

Ein Mitarbeiter des Kunden muss ständig für die Entwickler verfügbar sein (*onsite customer*) – der „Kunde vor Ort“ ist Teil des Teams (*whole team*). Im Idealfall sitzen alle Entwickler zusammen mit dem Kunden im selben Raum. Der Kunde soll Fragen der Entwickler über die Anforderungen ohne Umwege klären und Alternativen auswählen. Das macht es leichter, den Code ohne Entwurf gleich aus den Anforderungen heraus zu schreiben und Fehler beim Planungsspiel kurzfristig auszugleichen. Der Kunde vor Ort muss deshalb zu den späteren Benutzern gehören. Das kostet den Auftraggeber die Arbeitszeit des Mitarbeiters, soll sich aber über eine schnelle Fertigstellung der Software auszahlen, die dann auch besser auf die Bedürfnisse des Auftraggebers passt. Allerdings kann der Kunde vor Ort die aufgetretenen Fragen nicht immer schnell und verlässlich klären [ElSc02] und ist häufig schlecht ausgelastet (laut der Studie [AbKo04] nur zu einem Viertel seiner Zeit). Das ist zwar Verschwendung, aber Rückfragen und Kundensitzungen in herkömmlichen Projekten sind auch nicht kostenfrei.

3.2.4 Bewertung

Bei XP ist alles stark auf den Nutzen der lauffähigen Software für den Kunden ausgerichtet. Der Kunde soll möglichst genau das bekommen, was er braucht („build only what is needed“). Durch das iterative Planungsspiel und den Mitarbeiter im Team ist der Kunde in extremer Form im Projekt präsent. Die Anforderungen werden mit dem Kunden in Zyklen ermittelt und priorisiert. Gerade bei unklaren und instabilen Anforderungen kann das effektiver sein, als mit großem Aufwand zu versuchen, alles vorab festzulegen. Unklarheiten werden direkt mit dem Kunden vor Ort geklärt, ohne Umwege über Dokumente oder umständlich anberaumte Sitzungen („eliminate anything which does not add value“). Die Produktentwicklung soll möglichst ohne Verzögerungen laufen können. Der Code wird ständig vom Kunden daraufhin geprüft, ob er die Anforderungen umsetzt („stop if something goes wrong“). Die inkrementelle Auslieferung ist ein starker Anreiz, schnell die wirklich wichtige Funktionalität zu implementieren. Insgesamt bietet XP hier eine ausgesprochen schlanke Vorgehensweise, und zwar mit Blick auf alle drei schlanken Prinzipien.

Problematisch bleiben die häufigen Umstrukturierungen, die sich an den Benutzungsszenarien und am Code dadurch ergeben, dass die Anforderungen in vielen Iterationen parallel zur Programmierung geklärt werden. Es ist offen, ob das Vorgehen bei XP schlanker und kosteneffektiver ist als die herkömmlichen Techniken der Anforderungsermittlung. Es ist auch offen, welche Auswirkungen die Iterationen des Planungsspiels auf die Struktur der Software haben.

3.3 Refaktorisieren und ständige Integration

3.3.1 Refaktorisieren

Die wichtigste Technik in XP, um trotz des fehlenden Entwurfs eine gute und möglichst einfache Codestruktur (*simple design*) zu erreichen, ist das Refaktorisieren

(*refactoring*) [Fowl00]. Darunter versteht man ein Umstrukturieren des Codes, wobei seine Funktionalität unverändert gelassen wird. Unter XP wird der Code beim Programmieren immer wieder refaktorisiert, sodass er leichter zu verstehen, zu ändern und zu erweitern ist. Nach jeder Refaktorisierung müssen alle Testfälle erneut bestanden werden. Das Refaktorisieren ist auch notwendig, um den Code nach einer Iteration des Planungsspiels an neue oder geänderte Anforderungen anzupassen.

In gewissem Sinn stößt XP hier an seine Grenzen. Da immer wieder neue Funktionalität hinzukommt, kann es passieren, dass der Code (auch bei erfolgten inkrementellen Refaktorisierungen) früher oder später *in großem Ausmaß* umstrukturiert werden muss. Große Umstrukturierungen verursachen einen erheblichen Aufwand und sind fehleranfällig. Eine bekannte industrielle Fallstudie beschreibt das drastisch: „large refactorings stink“ [ElSc02].

3.3.2 Ständige Integration

Unter XP integriert jedes Entwicklerpaar seinen Code mindestens einmal täglich, oft aber mehrmals pro Stunde, mit dem Rest der Software (*continuous integration*). Bei der Integration werden auch die zugehörigen Testfälle ausgeführt. Auf diese Weise hält XP immer eine komplett lauffähige, aktuelle Version der Software bereit. XP will sicherstellen, dass alle Teile der Software zusammenpassen und die gefürchteten Integrationsprobleme spät im Projekt vermieden werden. Außerdem ist jederzeit und ohne Zusatzaufwand eine nutzbare, testbare Version für den Kunden lieferbar.

Die ständige Integration veränderter oder neuer Codestücke in die Software verursacht einen gewissen Aufwand, der aber durch Werkzeugunterstützung vertretbar gehalten werden kann. Meist wird ein dedizierter Server (*build server*) betrieben, der im Hintergrund regelmäßig das Versionskontrollsystem auf Änderungen der Codebasis abfragt und dann automatisch einen Erstellungsvorgang für die Software durchführt. Die Testfälle werden dabei automatisch ausgeführt.

3.3.3 Bewertung

Die Vorgabe, dass der Code gerade so komplex sein darf, wie unbedingt nötig, um die derzeitigen Anforderungen zu implementieren, ist eine schlanke Sichtweise („build only what is needed“). Unnötig komplizierter Code wird als Verschwendung gesehen. Im Code sollen keine Vorkehrungen für „zukünftige Erweiterungen“ getroffen werden, die zum jetzigen Zeitpunkt keinen Nutzen für den Kunden bringen („eliminate anything which does not add value“). Es ist ja nicht einmal klar, ob und in welcher Form solche Erweiterungen später kommen werden.

Die ständige Integration des Codes zu einem lauffähigen System ist eine extreme Form der Qualitätssicherung. Integrationsprobleme sollen nicht erst gegen Projektende, sondern so schnell wie möglich aufgedeckt werden. Das passt zum schlanken Prinzip des „stop if something goes wrong“.

Die Refaktorisierungen sollen eine einfache Codestruktur sicherstellen, dienen also schlanken Prinzipien. Auch in der herkömmlichen Softwareentwicklung sind Umstrukturierungen von Entwürfen und Code an der Tagesordnung; typische Refaktorisierungen [Fowl00] wirken sich hingegen eher lokal aus. Es ist offen, inwieweit sich durch Refaktorisieren eine tragfähige Gesamtstruktur der Software „evolutiv“ einstellt. Die traditionelle Ansicht in der Softwareforschung ist, dass man ohne Entwurfsarbeiten auf einer höheren Abstraktionsebene, der Softwarearchitektur, nicht auskommt. Es stellt sich deshalb die Frage, ob das häufige Refaktorisieren in XP nicht sogar zum Teil Verschwendung ist, also den Prinzipien einer schlanken Produktionsweise zuwiderläuft.

3.4 Gemeinschaftlicher Code und Programmierstandard

3.4.1 Gemeinschaftlicher Code

In XP kann jeder Entwickler jedes Codestück ändern – man sagt, „der Code gehört dem gesamten Team“ (*collective ownership*). Ein Codestück wird in der Praxis aber nicht angefasst, wenn gerade ein anderes Entwicklerpaar daran arbeitet (das lässt sich am Versionskontrollsystem erkennen). Zudem gilt die Regel, dass neuer oder geänderter Code erst dann in das Versionskontrollsystem eingegeben werden darf, wenn er *alle* Testfälle besteht.

Ein Entwickler wechselt bei XP die Aufgabe nach wenigen Tagen, oft täglich [ShRo04]. So wird Spezialistentum vorgebeugt, aber auch der Gefahr, durch die Arbeit an immer demselben Codestück blind für Fehler zu werden. Da jeder Entwickler an den Iterationen des Planungsspiels teilnimmt, in Paaren programmiert wird und für jedes Codestück Testfälle vorliegen, ist die Einarbeitungszeit gering. Die Tatsache, dass andere Kollegen an demselben Code weiterarbeiten werden, führt auch dazu, dass unschöne Angewohnheiten, wie unsauberes Programmieren (Stichwort „quick and dirty“), eingedämmt werden. Außerdem ist das Wissen über den Code weiter im Team verbreitet. Der Ausfall oder Wechsel eines Entwicklers kann dadurch leichter vom Team ausgeglichen werden.

3.4.2 Programmierstandard

Alle Entwickler müssen einen gemeinsamen Programmierstandard (*coding standard*) einhalten [Jeff01]. Das ist notwendig, wenn ein Entwickler nahtlos den Code eines anderen Entwicklers weiterbearbeiten soll, was bei XP häufig vorkommt. Die genauen Regeln des Standards sind dabei nicht ganz so wichtig. Es kommt vor allem auf ihre einheitliche Anwendung an. Der Wert von Richtlinien für einen guten Programmierstil ist schon lange bekannt und mit der Objektorientierung nicht kleiner geworden. Bei XP ist ein gemeinsamer Standard im Team unverzichtbar.

3.4.3 Bewertung

Bei XP entfallen die sonst üblichen, aufwendigen Mechanismen zur Übergabe von Teilprodukten, wie zum Beispiel Abnahmen, Qualitätstore (*quality gates*) oder zentralisierte Freigaben. Spezifikationen und Entwürfe gibt es nicht. Jeder Entwickler kommt an jeden Teil des Codes (und an alle Testfälle) heran. Neuer Code wird mit minimalem Aufwand freigegeben – der Entwicklungsprozess wird insgesamt massiv schlanker („eliminate anything which does not add value“). Offen ist aber, bis zu welcher Projektgröße diese minimalen Mechanismen ausreichen, um die Arbeiten zu koordinieren.

3.5 Die XP-Techniken als Paket

In der schlanken Produktion bei Toyota gibt es praktisch keine Puffer zwischen den Verarbeitungsschritten. Die nötigen Vorprodukte werden bei Bedarf angefordert, kurzfristig geliefert und dann weiterverarbeitet („pull“-Modell). Das funktioniert nur dann reibungslos, wenn die gelieferten Teile von sehr hoher Qualität sind. Das „pull“-Modell kann nur im Paket mit Maßnahmen zur Steigerung der Qualität erfolgreich betrieben werden.

In ähnlicher Weise greifen bei einem typischen XP-Projekt alle beschriebenen Techniken ineinander [ShRo04]. Die Vertreter von XP behaupten, dass ihr Ansatz

2071038 Wirtschaftsinformatik Heft 3/2007 3B2 Art. Pos. 23 (Autor: Frank Padberg) w:/p_3/wir/wir07_03/00613/wir00613u.3d
Stand: 10. 4. 2007 (gelieferte Daten) Bearbeiter: Ha./Storch

nur dann funktionieren kann, wenn *alle* Techniken als Paket verwendet werden [Beck99; HiCo01]. Bei unvoreingenommener Betrachtung trifft das so natürlich nicht zu. Zum Beispiel kann man Paarprogrammierung oder testgetriebenes Entwickeln sehr wohl mit Gewinn einzeln einsetzen, und das wird in der Praxis auch gemacht. Das gilt auch für das Planungsspiel als eine Methode zum Erfassen instabiler Anforderungen und die Technik der ständigen Integration.

Dennoch können bestimmte XP-Techniken nur zusammen vernünftig wirken – es gibt einen *inneren Zusammenhang* der XP-Techniken, ähnlich wie bei den Maßnahmen, die im Produktionssystem von Toyota zusammenwirken. Zum Beispiel ist das enge Einbinden des Kunden in den gesamten XP-Entwicklungsprozess notwendig, wenn ohne vollständig festgelegte Anforderungen, also flexibel entwickelt wird. Das Fehlen eines Entwurfs wird durch testgetriebenes Entwickeln, Refaktorisieren und ständige Integration aufgefangen. Das häufige Wechseln der Programmieraufgabe ist ohne Programmierstandard kaum machbar. Welche XP-Techniken nur zusammen mit anderen erfolgreich sind, muss noch empirisch untersucht werden.

3.6 Das Problem des Skalierens

Vor allem das Fehlen eines Entwurfs wirft die grundlegende Frage nach dem Skalieren von XP auf: in welchem Maß ist XP überhaupt für große Projekte geeignet? Damit mehrere Teams gleichzeitig an einer großen Software arbeiten können, ist es unverzichtbar, klare Grenzen zwischen den einzelnen Teilsystemen zu ziehen und die Schnittstellen festzulegen. Aus der Praxis ist nur allzu gut bekannt, dass sich ungeplante Änderungen in der Software sonst unkontrolliert ausbreiten (*ripple effects*) und die Vorteile des parallelen Arbeitens zunichte machen. Genauso fraglich ist, ob das Planungsspiel und die minimalen Mechanismen, mit denen bei XP die Arbeiten am Code koordiniert werden, skalieren. Diese Probleme mit XP sind aus Fallstudien bekannt [ElSc02].

Eigentlich sollte eine schlanke Produktionsweise das Skalieren der Produktion erleichtern. Hier tritt der Konflikt zwischen den herkömmlichen, managementzentrierten Vorgehensweisen in der Softwareentwicklung und den agilen Methoden besonders deutlich zutage. Auf das Problem des Skalierens geben XP und andere agile Methoden bisher keine befriedigende Antwort. Mancher Verfechter von XP gibt im Gespräch zu, dass XP nur für kleine bis mittelgroße Projekte mit nicht mehr als 20 Entwicklern geeignet ist. Es gibt zwar schon Workshops über agile Methoden, die sich mit diesem Problem beschäftigen [ReME03]; eine Lösung steht aber noch aus.

3.7 Der Stellenwert des Entwicklers

Es gehört zu den Kennzeichen einer schlanken Produktionsweise, dass der einzelne Arbeiter mehr Verantwortung und mehr Einfluss auf die Produktionsabläufe hat als bei der herkömmlichen Massenproduktion (*empowerment*). Auch bei XP haben die Entwickler einen großen Einfluss auf den Prozess. XP verlangt von ihnen ein hohes Maß an Eigenverantwortlichkeit und hohes Können, ähnlich dem Produktionssystem bei Toyota. Das kommt der Haltung vieler Entwickler entgegen, die ihr Können in die Software einbringen wollen [ShRo04].

Bei XP gibt es keine Trennung mehr in Tester, Programmierer, Analytiker oder Architekt. Das testgetriebene Entwickeln verlangt ein hohes Können im Bereich Softwaretesten. Jeder Entwickler hat vollen Zugriff auf den Code – das erfordert Verantwortungsbewusstsein und Disziplin. Das direkte Umsetzen der natürlichsprachlich formulierten Anforderungen in Code, ohne einen expliziten Entwurf, erfordert viel Erfahrung mit der Strukturierung von Software.

Jeder Entwickler arbeitet mit dem Kunden. Die ständige Anwesenheit des Kunden im Entwicklungsteam ist eine Herausforderung für die Entwickler. Alles liegt offen für den Kunden, auch die Schwierigkeiten und kleinen Misserfolge. Es erfordert eine offene Einstellung aller Beteiligten, um positiv mit dieser Situation umgehen zu können: der Kunde muss als Partner gesehen werden, nicht als jemand, der einem bei der Abnahme das Leben schwer macht.

Die sehr intensive Form des Entwickelns mit XP ist nur eine begrenzte Zahl von Stunden am Tag machbar. Daher versucht XP, mit einem gleichmäßigen, für die Entwickler langfristig verträglichen Tempo zu arbeiten (*sustainable pace*), anstatt – wie so oft gegen Ende eines typischen Softwareprojekts – mit vielen Überstunden zu versuchen, Zeit aufzuholen. Diese XP-Technik ist auch bekannt als „40-Stunden-Woche“ (*40-hour week*) und passt zum ethischen Wert des schlanken Produzierens, wonach diejenigen, welche die Arbeit machen, wertzuschätzen sind [Ohno93, 19].

4 Übertragen von Erkenntnissen

Durch die konzeptionelle Verankerung von XP in der *Lean Production* ist es grundsätzlich möglich, Erkenntnisse über schlanke Vorgehensweisen aus anderen Wirtschaftsbereichen auf die Softwareentwicklung zu übertragen. Ein wichtiges Thema sind Erkenntnisse über die Erfolgsfaktoren schlanker Vorgehensweisen, vor allem über die nötige Unternehmenskultur [Like04; BoSp99].

Es ist auffällig, dass vor allem *kleine* Softwarefirmen bereit sind, XP in ihren Projekten zu versuchen. Das hängt wohl damit zusammen, dass eine so grundlegende Umstellung des Softwareprozesses in einem informellen Umfeld mit kurzen Kommunikationswegen, jungen Mitarbeitern und flachen Hierarchien leichter möglich ist als in großen Firmen. An sich spricht aber nichts dagegen, schlanke Entwicklungsmethoden auch in großen Softwarefirmen einzusetzen [MaSc04]. Hier könnte die Softwaretechnik von den Erfahrungen mit schlanken Methoden in anderen großen Unternehmen profitieren.

Ein häufig genannter Faktor für den Erfolg von XP ist ein „dynamisches Umfeld“ [HiCo01]. Durch seine enge Kundeneinbindung und kurzen Zyklen kann XP besonders schnell und flexibel auf Änderungen der Anforderungen oder ihrer Priorität für den Kunden reagieren. Außerdem versucht XP, Entwurfsentscheidungen möglichst so lange hinauszuschieben, bis die zugehörigen Anforderungen klar und stabil sind. Das Verzögern von Entscheidungen in Kombination mit einer Parallelisierung verschiedener Entwicklungsarbeiten ist eine gängige Methode in der *Lean Production* [WoJR91, 116/117]. Im Fall von Software ist aber noch weitgehend unklar, welche Entwurfsentscheidungen sich längere Zeit aufschieben lassen – ohne Schaden für die Qualität der Software und ihrer Architektur. Auch bei dieser Frage könnte die Softwaretechnik von den Erfahrungen in Entwicklungs- und Produktionsbereichen anderer Industriezweige profitieren.

Weitere Themen, bei denen ein Übertragen von Erkenntnissen aus schlanken Projekten anderer Industriezweige aussichtsreich erscheint, betreffen die Frage, wie groß ein Softwareprojekt sein darf, damit

es noch mit agilen Methoden durchführbar ist (s. Abschnitt 3.6), oder die Frage der geeigneten Führung, Motivation und Weiterbildung der Entwickler in agilen Softwareprojekten (s. Abschnitt 3.7). Bei allen genannten Themen sind noch umfangreiche Forschungsarbeiten nötig.

5 Zusammenfassung und Ausblick

In diesem Beitrag haben wir Formen der Verschwendung in der Softwareentwicklung im Sinn der *Lean Production* aufgezeigt. Durch eine konzeptionelle Analyse der Techniken des *Extreme Programming* (XP) wird deutlich, dass XP in den Prinzipien der *Lean Production* verankert ist. XP verbindet eine extreme Verschlankung der Softwareentwicklung mit einer extremen Kundenorientierung und extremen Formen der Qualitätssicherung, ganz im Sinn der *Lean Production*. XP steht hier stellvertretend auch für die anderen agilen Methoden in der modernen Softwareentwicklung.

Wir haben auch Schwierigkeiten beim praktischen Einsatz von XP aufgezeigt, zum Beispiel Abnutzungseffekte bei der Paarprogrammierung oder das Problem des Ableitens wirkungsvoller Tests direkt aus den Anforderungsszenarien. Dazu kommen offene, eher konzeptionelle Fragen über das Vorgehen bei XP: die Frage, in welchem Maß XP auf große Projekte und große Firmen skaliert; die Frage, welche Entwurfsentscheidungen sich ohne Schaden hinauszögern lassen; oder die Frage nach der Unternehmenskultur für einen erfolgreichen Einsatz von XP. Bei solchen konzeptionellen Fragen könnte die Softwaretechnik von Erkenntnissen über schlanke Produktions- und Entwicklungsmethoden in anderen Wirtschaftsbereichen profitieren.

Der Bedarf an Software und ihre Komplexität steigen immer mehr. Gleichzeitig wird es schwieriger, hochqualifizierte Entwickler zu finden. Die Softwareentwicklung in Unternehmen muss in Zukunft mehr mit den *verfügbaren* Entwicklern erreichen.

„Lean production is ‚lean' because it uses less of everything compared with mass production." [WoJR91, 13]

Dieser Grundgedanke einer schlanken Produktionsweise ist sehr gut auf Softwareentwicklungsprozesse übertragbar. Der Entwicklungsprozess muss flexibler und schneller – kurz: schlanker – werden. Die großen Verbesserungen der Dauer, Qualität, Kosten und Flexibilität der Fertigung, die bei Toyota und anderen Unternehmen durch *Lean Production* erreicht werden [WoJR91, 81/92; Kraf88], sind ein starker Anreiz, schlanke Prinzipien auch in der Softwareentwicklung einzusetzen. Agile Ansätze wie *Extreme Programming* sind deshalb ein wichtiger Schritt in die richtige Richtung.

Literatur

[ABHL06] *Arisholm, E.; Briand, L.; Hove, S.; Labiche, Y.:* The Impact of UML Documentation on Software Maintenance: An Experimental Evaluation. In: IEEE Transactions on Software Engineering TSE 32 (2006), S. 365–381.

[AbKo04] *Abrahamsson, P.; Koskela, J.:* Extreme Programming: A Survey of Empirical Data from a Controlled Case Study. In: International Symposium on Empirical Software Engineering ISESE (2004), S. 73–82.

[Agil01] *Agile Alliance:* Manifesto for Agile Software Development. http://agilemanifesto.org/, 2001, Abruf am 2007-03-15.

[Baud05] *Baudin, M.:* Lean Logistics. The Nuts and Bolts of Delivering Materials and Goods. Productivity Press, 2005.

[Beck98] *Beck, K.:* Why 'Extreme'? http://c2.com/cgi/wiki?ExtremeProgramming, 1998, Abruf am 2007-03-15.

[Beck99] *Beck, K.:* Embracing Change with Extreme Programming. In: IEEE Computer (October 1999), S. 70–77.

[Beck04] *Beck, K.:* Extreme Programming Explained. 2. Aufl., Addison-Wesley, 2004.

[BoSp99] *Bowen, H. K.; Spear, S.:* Decoding the DNA of the Toyota Production System. In: Harvard Business Review (September/October 1999), S. 96–106.

[BrRB06] *Bryant, S.; Romero, P.; du Boulay, B.:* The Collaborative Nature of Pair Programming. In: XP/Agile 7 (2006), S. 53–64.

[C3Te98] *C3 Team:* Chrysler Goes to 'Extremes'. In: Distributed Computing 10 (1998), S. 24–28.

[Char01] *Charette, R.:* The Decision is in: Agile versus Heavy Methodologies. In: Cutter Consortium Executive Update 2 (2001) 19, S. 2–3.

[Cock02] *Cockburn, A.:* Agile Software Development. Prentice Hall, 2002.

[Copl94] *Coplien, J.:* Developing in Pairs. In: Pattern Languages of Programming PLOP (1994), S. 183–238.

[ElSc02] *Elssamadisy, A.; Schalliol, G.:* Recognizing and Responding to 'Bad Smells' in Extreme Programming. In: International Conference on Software Engineering ICSE 24 (2002), S. 617–622.

[Fowl00] *Fowler, M.:* Refactoring: Improving the Design of Existing Code. Addison-Wesley, 2000.

[Harv04] *Harvey, D.:* Lean, Agile. In: Workshop on The Software Value Stream, Object Technology OT (2004), http://www.davethehat.com/articles/LeanAgile.pdf, Abruf am 2007-03-15.

[HiCo01] *Highsmith, J.; Cockburn, A.:* Agile Software Development. In: IEEE Computer (September 2001), S. 120–122.

[High00] *Highsmith, J.:* Adaptive Software Development. Dorset House, 2000.

[Howe99] *Howell, G. A.:* What Is Lean Construction? In: Annual Conference of the International Group for Lean Construction IGLC 7 (1999), S. 1–10.

[Jeff01] *Jeffries, R.:* What is Extreme Programming? In: XP Magazine (November 2001), http://www.xprogramming.com/xpmag/whatisxp.htm, Abruf am 2007-03-15.

[John02] *Johnson, J.:* ROI – It's Your Job! In: Hauptvortrag auf der XP (2002), http://ciclamino.dibe.unige.it/xp2002/talksinfo/johnson.pdf, Abruf am 2007-03-15.

[JoMi06] *Jones, D.; Mitchell, A.:* Lean Thinking for the NHS. National Health Service (NHS) Confederation Report, UK 2006.

Abstract

Lean Production Methods in Modern Software Development

This paper shows that Extreme Programming (XP) is rooted in the principles of Lean Production. XP drastically slims down the development process, but adds extreme customer orientation and extreme ways of quality assurance to the process. The fact that XP and other agile methods in modern software development are based on lean principles explains why agile methods can produce high-quality software in a cost-effective way. The paper also contains a discussion of problems that come up when using XP in practice, and raises important questions about lean methods in software development; for example, whether lean methods scale to large software projects and large software organizations. When trying to answer such questions, software engineering could draw from experience with lean production and lean development in other fields.

Keywords: Lean Production Methods in Modern Software Development

[KaAh96] *Karlsson, C.; Ahlström, P.:* The Difficult Path to Lean Product Development. In: Journal of Product Innovation Management 13 (1996), S. 283–295.

[KePl74] *Kernighan, B.; Plauger, K.:* The Elements of Programming Style. McGraw-Hill, 1974.

[Kraf88] *Krafcik, J. F.:* Triumph of the Lean Production System. In: Sloan Management Review 30 (Fall 1988) 1, S. 41–52.

[KTRB06] *Karlsson, L.; Thelin, T.; Regnell, B.; Berander, P.; Wohlin, C.:* Pair-wise Comparisons versus Planning Game Partitioning. Experiments on Requirements Prioritisation Techniques. In: Journal of Empirical Software Engineering 11 (2006) 2, (online), Abruf am 2007-03-15.

[LaBa03] *Larman, C.; Basili, V.:* Iterative and Incremental Development: A Brief History. In: IEEE Computer (June 2003), S. 47–56.

[Like04] *Liker, K. J.:* The Toyota Way. McGraw-Hill, 2004.

[MaSc04] *Manhart, P.; Schneider, K.:* Breaking the Ice for Agile Development of Embedded Software: An Industry Experience Report. In: International Conference on Software Engineering ICSE 26 (2004), S. 378–386.

[McCo96] *McConnell, S.:* Rapid Development. Kap. 3, Microsoft Press, 1996.

[MiDL87] *Mills, H.; Dyer, M.; Linger, R.:* Cleanroom Software Engineering. In: IEEE Software (September 1987), S. 19–25.

[MuAC02] *Murman, E. M.; Allen, T.; Cutcher-Gershenfeld, J.:* Lean Enterprise Value. Insights from MIT's Lean Aerospace Initiative. Palgrave Macmillan, Juni 2002.

[MüHa02] *Müller, M.; Hagner, O.:* Experiment about Test-first Programming. In: IEE Proceedings Software 149 (2002), S. 131–136.

[MüPa03] *Müller, M.; Padberg, F.:* On the Economic Evaluation of XP Projects. In: European Software Engineering Conference ESEC 9 (2003), S. 168–177.

[MüPT05] *Müller, M.; Padberg, F.; Tichy, W.:* Ist XP etwas für mich? Empirische Studien zur Einschätzung von XP. In: GI-Fachtagung Software Engineering (2005), S. 217–228.

[MüTi01] *Müller, M.; Tichy, W.:* Case Study: Extreme Programming in a University Environment. In: International Conference on Software Engineering ICSE 23 (2001), S. 537–544.

[Ohno93] *Ohno, T.:* Das Toyota-Produktionssystem. Campus, 1993.

[PaFe02] *Palmer, S.; Felsing, J.:* A Practical Guide to the Feature-Driven Development. Prentice Hall, 2002.

[PaMü03] *Padberg, F.; Müller, M.:* Analyzing the Cost and Benefit of Pair Programming. In: International Software Metrics Symposium METRICS 9 (2003), S. 166–177.

[PaMü04] *Padberg, F.; Müller, M.:* An Empirical Study about the Feelgood Factor in Pair Programming. In: International Software Metrics Symposium METRICS 10 (2004), S. 151–158.

[PoPo03] *Poppendieck, M.; Poppendieck, T.:* Lean Software Development. Addison-Wesley, 2003.

[Popp03] *Poppendieck, M.:* Lean Software Development. In: C++ Magazine (Fall 2003), http://www.poppendieck.com/pdfs/Lean_Software_Development.pdf, Abruf am 2007-03-15.

[ReME03] *Reifer, D.; Maurer, F.; Erdogmus, H.:* Scaling Agile Methods. In: IEEE Software (July 2003), S. 12–14.

[Royc70] *Royce, W.:* Managing the Development of Large Software Systems. In: IEEE WESCON (1970), S. 1–9.

[ScBe01] *Schwaber, K.; Beedle, M.:* Agile Software Development with Scrum. Prentice Hall, 2001.

[ShRo04] *Sharp, H.; Robinson, H.:* An Ethnographic Study of XP Practices. In: Journal of Empirical Software Engineering 9 (2004), S. 353–375.

[SSAD06] *Sfetsos, P.; Stamelos, I.; Angelis, L.; Deligiannis, I. S.:* Investigating the Impact of Personality Types on Communication and Collaboration-Viability in Pair Programming. In: XP/Agile 7 (2006), S. 43–52.

[Stan04] *Standish Group:* The CHAOS Report. The Standish Group, 2004.

[Ster06] *Sterlicchi, J.:* Kontinuierliche Zusammenarbeit mit den Kunden. In: SAP Info (Juli 2006), http://www.sap.info/index.php4?ACTION=noframe&url=http://www.sap.info/public/DE/de/index/Category-12613c61affe7a5bc-de/7, Abruf am 2007-03-15.

[WKCJ00] *Williams, L.; Kessler, R.; Cunningham, W.; Jeffries, R.:* Strengthening the Case for Pair-Programming. In: IEEE Software (July/August 2000), S. 19–25.

[WoJo03] *Womack, J. P.; Jones, D. T.:* Lean Thinking. Free Press, 2003.

[WoJR91] *Womack, J. P.; Jones, D. T.; Roos, D.:* The Machine That Changed the World. Perennial, 1991.