Ist XP etwas für mich ?
Empirische Studien zur Einschätzung von XP

Matthias M. Müller Frank Padberg Walter F. Tichy
Fakultät für Informatik, Universität Karlsruhe,
Am Fasanengarten 5, 76131 Karlsruhe, Germany
{muellerm|padberg|tichy}@ira.uka.de

Abstract: Agile Software-Entwicklungsmethoden, und speziell Extreme Programming (XP), haben große Aufmerksamkeit unter Praktikern und Forschern auf sich gezogen. Dieser Artikel bietet eine Gesamtschau der bis jetzt bekannten, empirischen Erkenntnisse zu XP, mit dem Ziel, dem Praktiker fundierte Entscheidungshilfen zu bieten sowie die Lücken in der wissenschaftlichen Untersuchung aufzuzeigen. Verhältnismäßig gut untersucht sind die Praktiken der Paarprogrammierung und der testgetriebenen Entwicklung; weitere Techniken von XP wurden bis jetzt nur in Erfahrungsberichten evaluiert. Langzeituntersuchungen über die Nachhaltigkeit der XP-Praktiken stehen ebenfalls noch aus.

1 Einführung

Agile Methoden erfreuen sich enormer Aufmerksamkeit. Seit dem ersten Artikel von Beck (Bec99) über Extreme Programming (XP) gibt es über ein Dutzend Bücher zum Thema agile Methoden, zwei Konferenzserien ausschließlich für Agilität, Spezialausgaben bei mehreren hochrangigen Zeitschriften, Diskussionsveranstaltungen zu agiler Methodik auf nahezu allen Software-Konferenzen und eine Unmenge von Artikeln. Erfreulich ist, dass es auch eine ganze Reihe von Arbeiten gibt, die agile Methoden empirisch untersuchen. Die Ergebnisse dieser Untersuchungen sollen hier zusammengefasst werden, um zum einen potentiellen Anwendern eine Entscheidungshilfe zu bieten und zum anderen Lücken in der wissenschaftlichen Untersuchung agiler Methoden aufzuzeigen.

Agile Methoden wollen eine Alternative zu traditionellen, planungs- und dokumentationszentrierten Vorgehensmodellen bieten. Das Ziel agiler Methoden ist, nicht nur zuverlässige Software schnell und kostengünstig zu entwickeln, sondern Änderungswünsche während der Entwicklung möglichst rasch und vollständig berücksichtigen zu können. Extreme Programming ist die wohl bekannteste und laut Charette (Cha01) mit einem Marktanteil von 38 Prozent dominierende agile Methode. Dieser Artikel konzentriert sich daher auf XP, obwohl einige der XP-Methoden auch bei anderen agilen Vorgehensmodellen Anwendung finden.

Die am sorgfältigsten untersuchten Praktiken von XP sind *Paar-Programmierung* und *testgetriebene Entwicklung*. Gegenüber der Einzelprogrammierung kann Paar-Programmierung den Vorteil der schnelleren Fertigstellung, wenn genausoviele Paare wie Einzelentwickler eingesetzt werden, und der geringeren Fehlerdichte erreichen. Dieses Potenzial

wird in mehreren Studien bestätigt. Der Nachteil sind höhere Kosten und, jedenfalls initial, zusätzlicher Lernaufwand. Diese Vor- und Nachteile gegen den Zeitpunkt des Markteintritts und den somit zu erwartenden Markanteil abzuwägen ist ohne ein geeignetes Modell kaum möglich. Wir skizzieren in diesem Papier dazu die Kapitalwert-Rechnung von (MP03), die eine Verrechnung der genannten Vor- und Nachteile erlaubt. Weitere Fragen betreffen den Vergleich von Paar-Programmierung mit Inspektionen. Könnten zum Beispiel Einzelprogrammierer mit Inspektionen die gleiche Qualität liefern wie Paare? Erste Ergebnisse deuten an, dass bei vorgegebener Qualität die Kosten gleich sind. Die Leistung von Paaren ist bestimmt von einer besseren Problemabdeckung, einer durch die Paararbeit erzwungenen, höheren Arbeitskonzentration sowie dem Gefühl, nicht alleine gelassen zu sein. Wenig ist bekannt über die kausalen Zusammenhänge bezüglich der Eigenschaften einzelner Entwickler im Hinblick auf die Produktivität des Paares sowie der Nachhaltigkeit der Eigenschaften von Paar-Programmierung. Auch ist ungeklärt, warum der Anteil der Paar-Programmierung gegenüber der Einzel-Programmierung starken Schwankungen unterliegen kann (Hul04).

Über testgetriebene Entwicklung liegen ebenfalls brauchbare Studien vor, wenn auch nicht so viele wie über Paar-Programmierung. Zwei unabhängige Faktoren sind hier zu berücksichtigen. Der erste Faktor betrifft die automatische Durchführbarkeit von Tests. Diese Herangehensweise ist unter dem Begriff Regressionstest bekannt, wurde aber bis dato meist erst spät im Entwicklungszyklus eingesetzt. Der zweite Faktor betrifft den Zeitpunkt der Erstellung der Tests: Sie werden zwar inkrementell, aber immer *vor* der Entwicklung der zu testenden Software-Komponente geschrieben. Daher kommt das Schreiben der Tests auch einer impliziten Spezifikation gleich, die, wie Manhart & Schneider (MS04) berichten, eine Umstellung des organisationsweiten Testprozesses nach sich ziehen kann. Erste Ergebnisse zeigen, dass die testgetriebene Entwicklung eine lange Lernkurve erfordert. Ohne entsprechende Schulung kann testgetriebene Entwicklung zu oberflächlichem Testen führen, das die Entwickler in falscher Sicherheit wiegt. “Richtige” testgetriebene Entwicklung kann aber sowohl zu besser testbaren Komponenten als auch besserer Qualität führen. Eine Überwachung der Testabdeckung scheint aber weiterhin notwendig. Auch hier stehen Langzeituntersuchungen sowie Vergleiche mit anderen Qualitätssicherungsmaßnahmen noch aus.

Der Rest dieses Artikels konzentriert sich auf Aspekte der Paar-Programmierung und der testgetriebenen Entwicklung. Die beiden genannten Techniken werden in Isolation untersucht; um Synergien feststellen zu können, wären multifaktorielle Experimente notwendig, die noch ausstehen. Die anderen Bausteine von XP, z.B. Refaktorisieren oder inkrementelle Auslieferung, werden nicht weiter betrachtet, da hier bisher zu wenig verlässliche empirische Erfahrung vorliegt.

2 Experimente über Paar-Programmierung

Folgende Szenarien werden unterschieden: *Paar-Programmierung* (PP), das heißt, zwei Entwickler sitzen gemeinsam vor *einem* Rechner und lösen die Aufgabe; *Partner-Programmierung*, zwei Entwickler mit jeweils einem Rechner lösen die Aufgabe in konventioneller Gruppenarbeit (GAD02); *Einzel-Programmierung*, ein Entwickler löst die Aufgabe alleine.

Die *quantitativen* Studien untersuchen die ökonomischen Aspekte der Paar-Programmierung zu extrahieren. Dabei wird nach dem Ausgleich der bei der Paar-Programmierung verdoppelten Entwicklungskosten durch die schnellere Entwicklung und die höhere Qualität der Programme gesucht. Unter Qualität wird dabei die Fehlerdichte des entwickelten Programms verstanden. Ansonsten wurde noch die Lesbarkeit der Programme untersucht, siehe Wilson e.a. (WHN93) und Nosek (Nos98).

Die *qualitativen* Studien untersuchen die sozialen Effekte der Paar-Programmierung und die Auswirkungen der Paar-Programmierung auf die Lehre. Das Vertrauen der Entwickler in die gemeinsame Lösung und der Spass bei der Entwicklung spielen dort die größte Rolle. Es wird aber auch untersucht, welche Merkmale einzelner Programmierer einen Einfluss auf die Leistungsfähigkeit des Paares haben.

Die Tabellen[1] 1 und 2 geben einen Überblick über die Ergebnisse und Merkmale der Studien.

Tabelle 1: Ergebnisse der Experimente über Paar-Programmierung

Quantitative Ergebnisse

Studie	Ergebnisse
Nosek 1998	PP ist schneller
Williams e.a. 2000 Cockburn & Williams 2000	PP ist schneller und führt zu weniger Fehlern, hat aber Mehrkosten
Nawrocki & Wojciechowski 2001	PP ist nicht schneller
Tomayko 2002	PP führt zu Programmen mit geringerer Defektdichte
Heiberg e.a. 2003	PP ist gleichwertig mit Partner-Programmierung
Müller 2004	PP ist gleichwertig zu Einzel-Programmierung mit Durchsichten

Qualitative Ergebnisse

Wilson e.a. 1993	Paare haben mehr Vertrauen in Lösung und mehr Spass
Nosek 1998	Paarleistung hängt nicht alleine vom Leistungsniveau der Individuen ab
Williams e.a. 2003	Paare erzielen bessere Lernerfolge
Thomas e.a. 2003	Schwächere Studenten haben mehr Spass an PP
Puus e.a. 2004	Paare haben weniger Vertrauen in Lösung als bei Partner-Programmierung
Müller & Padberg 2004	Paarleistung könnte vom „Feelgood Factor“ bestimmt sein

2.1 Quantitative Ergebnisse

Aus den drei Studien von Nosek (Nos98), Williams e.a. (WKCJ00), sowie Nawrocki & Wojciechowski (NW01) lassen sich konkrete Werte für den Geschwindigkeitsvorteil der Paar-Programmierung ableiten. Der Geschwindigkeitsvorteil, siehe Abschnitt 4.2, bewegt sich in einem Bereich von 1,0 bei Nawrocki & Wojciechowski über 1,4 bei Nosek bis maximal 1,7 bei Williams e.a. Diese widersprüchlichen Werte könnten sich mit der gemeinsamen Programmiererfahrung der Paare erklären lassen. Die Paare von Williams e.a. hatten eine Eingewöhnungszeit, während den Paaren von Nawrocki & Wojciechowski diese nicht gewährt wurde. Mit Hilfe der Gewöhnungszeit allein lässt sich aber nicht der mittlere Wert von 1,4 von Nosek erklären, da in dieser Studie die Entwickler auch unvorbereitet in Paare

[1] Die Publikationen (WKCJ00) und (CW00) stellen beide gemeinsam eine Studie über Paar-Programmierung vor, und die Studie (PSSH04) basiert auf Daten, die in (HPSS03) gesammelt wurden.

Tabelle 2: Charakteristiken der Studien über Paar-Programmierung

Studie	Größe	PP vs.	Umfeld	Teiln.	exp. Schwächen
Wilson e.a. 1993	10 vs. 14	E	Kurs	S	Zeitbeschränkung
Nosek 1998	5 vs. 5	E	Büro	P	Zeitbeschränkung
Williams e.a. 2000 Cockburn & Williams 2000	14 vs. 13	E	HA	S	Hausaufgaben Gruppeneinteilung
Nawrocki & Wojciechowski 2001	5 vs. 6 vs. 6	E-PSP0 E-XP	Labor	S	Aufgabengröße
Thomas e.a. 2003	64		Labor	S	Selbsteinschätzung
Heiberg e.a. 2003	110	Part	Labor	S	Länge der Aufgabe
Tomayko 2002	4 vs. 4	TSP	Kurs	S	Prozesstreue
Williams e.a. 2003	1200	E	Kurs	S	
Müller 2004	20	D	Labor	S	Aufgabengröße
Müller & Padberg 2004	38	D	Labor	S	Aufgabengröße Selbsteinschätzung

E=Einzel, Part=Partner-Programmierung, D=Durchsichten, S=Studenten, P=Profis

eingeteilt wurden. Eine Studie über den Stellenwert des Grades der Gewöhnung aneinander, die z. B. ad-hoc Paare mit eingespielten Paaren vergleicht, existiert leider noch nicht. Andere quantitative Eigenschaften der Studien, wie z. B. der Umfang der Experimentaufgabe, könnten sich ebenfalls stark auf die Produktivität der Paare auswirken, da zwei Entwickler ein größeres System besser überblicken können als ein Einzelentwickler.

Williams e.a. und Cockburn & Williams berichten zudem über eine höhere Qualität der von Entwicklerpaaren erstellten Programme. Ihre 15-prozentige Erhöhung der bestandenen Testfälle wurde aber bisher von keiner weiteren Studie bestätigt.

Tomayko (Tom02), Heiberg e.a. (HPSS03), und Müller (Mül04) vergleichen Paar-Programmierung mit Einzel-Programmierern. Die einzelnen Entwickler bekommen dabei noch zusätzlich Unterstützung durch Inspektionen bzw. einen weiteren Entwickler zur Partner-Programmierung oder Durchsichten. Während Partner-Programmierung bzw. Durchsichten keinen Unterschied zur Paar-Programmierung bezüglich Kosten und Qualität bedeuten, beobachtet Tomayko, dass sich Inspektionen negativ auf die Kosten und die Qualität der Programme der Einzelentwickler auswirken. Der von Tomayko gemessene Effekt könnte jedoch auch andere Ursachen als den Gebrauch von Inspektionen haben. So werden dort Teams verglichen, die XP und den *Team Software Process* (TSP) (Hum99), einsetzen. Damit ist nicht auszuschließen, dass bei der TSP Gruppe die Probleme bei der Anwendung des TSPs alle anderen Effekte überlagert haben.

2.2 Qualitative Ergebnisse

Bei den Fragen nach den sozial-psychologischen Aspekten bei der Paar-Programmierung scheint sich ein Trend abzuzeichnen. So berichten Wilson (WHN93) und Nosek (Nos98), dass die Produktivität der Paare nicht allein vom Leistungsniveau der Individuen abhängt. Die Studie von Müller & Padberg (MP04) scheint dies zu bestätigen. Dort wird der "Feel-good Factor", also die Stimmung innerhalb des Paares, als mögliche Erklärung für die Produktivitätsunterschiede vorgeschlagen. Die Stimmung kann aber auch ein typischer Neuigkeitseffekt sein, der nach einiger Zeit verfliegt.

Bei der Frage, ob Paar-Programmierung das Vertrauen in die Lösung fördert, gibt es bisher

widersprüchliche Aussagen. Bei Wilson und Nosek berichten die Paare, dass sie mehr Zutrauen in die gemeinsame Lösung haben, während bei der Studie von Puus e.a. (PSSH04) dies gerade nicht der Fall ist. Eine Erklärung für diesen Widerspruch gibt es bisher nicht.

2.3 Gültigkeit

Es fällt auf, dass bis auf die Studie von Nosek, nur Studenten eingesetzt wurden. Damit stellt sich natürlich die Frage nach der Übertragbarkeit der Resultate auf Projekte in der Software-Industrie. Ein weiteres Problem der Übertragbarkeit der Resultate ist, dass sich ein Entwicklerpaar erst aneinander gewöhnen muss, bevor die höchste Produktivität erreicht wird. Über die optimale Dauer dieser "Eingewöhnungszeit" herrscht noch Unklarheit.

Weiterhin könnte auch der Umfang der Aufgabe eine Rolle beim Vorteil der Paare gegenüber einzelnen Entwicklern spielen. Zwei Entwickler können eine größere Aufgabe besser überblicken als ein einzelner. Die bisher in den Experimenten verwendeten "kleinen" Aufgaben favorisieren damit wahrscheinlich eher die Einzel- als die Paar-Programmierung.

Eine weitere Unsicherheit entsteht durch den Aufbau der meisten Studien. Viele Studien vergleichen Programme mit verschiedener Qualität. Damit können die beobachteten Effekte auf die Qualitätsunterschiede zurückzuführen sein. Besser sind die zwei folgenden Alternativen: Halte den Zeitrahmen, in dem die Aufgabe bearbeitet werden soll, *fest* oder stelle eine vergleichbare Programmqualität her, z.B. durch einen Akzeptanztest. In den Studien von Wilson und Nosek müssen die Aufgaben innerhalb einer festen Zeit bearbeitet werden. Allerdings ist fraglich, ob die Zeitschranke wirkte, da Angaben über unvollständige Programme fehlen. Müller (Mül04) verfolgt einen anderen Ansatz. Dort werden Programme mit vergleichbarer Qualität durch eine Akzeptanztestphase erreicht. Damit werden nur Programme mit annähernd gleicher Qualität untersucht und Unterschiede im Programmieraufwand lassen sich auf die eingesetzte Programmiermethode zurückführen.

3 Experimente über testgetriebene Entwicklung

Tabelle 3 gibt einen Überblick über die untersuchten Testtechniken. Die Zeilen unterschei-

Tabelle 3: Übersicht über die Untersuchten Testtechniken.

Testzeitpunkt	**Testautomation**	
	ja	**nein**
Test-danach	Iterativ Test-danach ITD	–
Test-zuerst	Testgetriebene Entwicklung [a] TGE	–
Komplett	Regressionstest	konventionell

[a] *engl. test-driven Development TDD*

den den Zeitpunkt, in dem das Testen stattfindet. Bei der Verwendung von *test-danach* wird direkt nach der Implementierung einer Funktion der entsprechende Test geschrieben und

ausgeführt. Bei *test-zuerst* wird der Testfall *vor* der Implementierung der Funktion spezifiziert. Beide Fälle beschreiben einen feingranularen Prozess, bei dem die Implementierung und die Tests in Scheiben von fünfminütiger Dauer sukzessive erweitert werden. Dagegen bedeutet der letzte Tabelleneintrag, dass die Tests erst nach abgeschlossener Implementierung der Gesamtaufgabe geschrieben werden. Die Spalten unterscheiden zwischen *automatischen* oder *manuellen* Testen. Ohne die Testautomatisierung sind *iterativ test-danach* und *testgetriebene Entwicklung* zu aufwändig, weshalb die entsprechende Einträge in der Tabelle leer sind. Die letzte Spalte repräsentiert den herkömmlichen Testprozess, der in einer separaten Phase nach der Implementierung stattfindet. Wenn dieser Prozess mit automatischen ausführbaren Tests durchgeführt wird, so nennen wir ihn *Regressionstest*, ansonsten *konventionell*.

Die Tabellen 4 und 5 geben einen Überblick über die Studien zur testgetriebenen Entwicklung.

Tabelle 4: Empirische Studien über testgetriebenes Entwickeln

Studie	Ergebnisse
Müller & Hagner 2002	TGE ist eher langsamer als Regressionstest und bringt keine Verbesserung der Qualität
Pancur e.a. 2003	TGE mit PP ist etwa gleichwertig zu ITD mit PP
George & Williams 2003	TGE mit PP eher langsamer bei verbesserter Qualität als PP
Geras e.a. 2004	kein Produktivitätsunterschied zwischen TGE und ITD

Tabelle 5: Merkmale der Studien über testgetriebene Entwicklung

Studie	Größe	TGE vs.	Umfeld	Teiln.	exp. Schwächen
Müller & Hagner 2002	10 vs. 9	Regression	Labor	S	Aufgabe
Pancur e.a. 2003	20 vs. 18	ITD	Kurs	S	Aufgabe, Korrektheit
George & Williams 2003	6 vs. 6	konventionell	Büro	P	Aufgabe, Korrektheit, Effekt
Geras e.a. 2004	7 vs. 7	ITD	Büro	P	Prozess, Korrektheit

S=Studenten, P=Profis

3.1 Ergebnisse

Test-zuerst verlangt ein Umdenken der Entwickler, die in der Regel nur mit nachträglichem Testen vertraut sind. Während die Paar-Programmierung in der Lehre reibungslos von den Studenten übernommen wird (MT01; Wil01; Fen03; MLSM04), gestaltet sich am Anfang die Umstellung auf die testgetriebene Entwicklung als schwierig (MT01; Wil01; MLSM04). Diese Schwierigkeiten werden jedoch im Laufe der Zeit überwunden, so dass die testgetriebene Entwicklung von den Studenten als wertvolle Erfahrung eingestuft wird.

Aus dieser Lehrerfahrung lassen sich Gefahren für die externe Gültigkeit von Studien formulieren, insbesondere wenn sie Studenten einsetzen. Bei unerfahrenen Teilnehmern liegt noch kein gefestigter *test-zuerst* Prozess vor, sodass ein möglicher gemessener Effekt mehr auf den noch nicht vollständig verinnerlichten Prozess als auf die *test-zuerst* Entwicklung selbst zurückzuführen ist. Experimente mit Studenten sind trotzdem hilfreich. Sie zeigen die Probleme auf, die bei vorschneller Anwendung testgetriebener Entwicklung entstehen können. Die Zeitspanne, die ein Student oder ein professioneller Entwickler benötigt, um

die *testgetriebene Entwicklung* vollständig zu erlernen, ist noch unbekannt. Die Schwierigkeit liegt in der Unterteilung des Problems in Teilaufgaben, für die in etwa 5 Minuten ein Test und die zugehörige Implementierung geschrieben werden können. Diese Technik ist ohne fremde Hilfe nur schwer zu erlernen.

Die Lernproblematik ist in allen Studien präsent. Zwar setzen George & Williams (GW03) und Geras e.a. (GSM04) professionelle Entwickler ein, bei George & Williams waren dies jedoch unerfahrene *test-zuerst* Entwickler; Geras e.a. machen keine Aussage über die Erfahrenheit der Subjekte.

Keine der Studien macht eine Aussage über die Auswirkungen der testgetriebenen Entwicklung auf die Struktur der erstellten Software. So ist zum Beispiel die Testbarkeit der Programme, die Benutzbarkeit der Schnittstellen sowie die Verwendung der Tests als Teil der Dokumentation noch nicht untersucht worden.

3.1.1 Überlagerung der Effekte

Das Zusammenspiel der beiden Aspekte *test-zuerst* und *Automatisierung* ist ein Problem bei der Isolierung der zugehörigen Effekte. Die einzelnen Effekte können nur durch ein multifaktorielles Experiment erkannt werden, in dem alle Testtechniken (konventionell, Regressionstest, iterativ test-danach und testgetriebene Entwicklung) miteinander verglichen werden. Studien, die *testgetriebene Entwicklung* mit konventionellen Testen vergleichen, wie z. B. George & Williams (GW03), können den Beitrag von Testzeitpunkt und Testautomation nicht differenzieren. Bei den Studien von Müller (MH02), Pancur e.a. (PCTV03) und Geras e.a. (GSM04) hingegen werden Gruppen verglichen, die sich jeweils nur im Testzeitpunkt unterscheiden. Keine dieser letzten Studien kann einen nennenswerten Vorteil der *testgetriebenen Entwicklung* gegenüber *Regressionstest* oder einem *iterativen test-danach* Prozess feststellen. Das könnte bedeuten, dass die von George & Williams beobachteten Verbesserungen auf die Testautomatisierung zurückzuführen ist.

3.1.2 Prozesstreue

Ein letztes Problem bei der Bewertung der *testgetriebenen Entwicklung* ist die Kontrolle, inwieweit der vorgeschriebene Prozess überhaupt befolgt wurde. Keine bekannte Studie macht hierzu eine überprüfbare Aussage. Abhilfe würde ein Protokoll schaffen, das Testzeitpunkte und die jeweils ausgeführten Tests und Implementierungen dokumentiert. Dieses Protokoll müsste dann nach der Durchführung des Experimentes auf die Prozesstreue hin untersucht werden. Geras e.a. (GSM04) versucht den Prozess durch Prozessskripte zu beschreiben. Er bleibt den Beleg jedoch schuldig, ob seine Teilnehmer diese Skripten auch befolgt haben.

Der Prozess der Kontrollgruppe muss auch einer Überprüfung unterzogen werden. So hat die Kontrollgruppe bei George & Williams (GW03) zu wenig (oder gar nicht) getestet. Damit ergibt sich eine weitere “verborgene” Variable, nämlich den Grad der Testüberdeckung.

3.2 Gültigkeit

Die Studie von Müller & Hagner (MH02) zeigt, dass die Korrektheit der abgegebenen Programme durch einen externen Akzeptanztest angeglichen werden muss, andernfalls sind die abgelieferten Programme inakzeptabel schlecht. So war die Gruppe der *test-zuerst* Entwickler bei der Implementierung schneller, wobei die abgegebenen Programme deutlich schlechter waren als die der Kontrollgruppe, die *Regressionstest* einsetzte. Prinzipiell sollte mit Hilfe eines Akzeptanztest eine *vergleichbare* und *hohe* Korrektheit der Programme angestrebt werden.

4 Ökonomisches Modell zur Bewertung von XP

Das Modell (MP03) dient zum Vergleich des ökonomischen Wertes von XP-Projekten mit konventionellen Softwareprojekten. Der Ausgangspunkt des Modells bilden die nahezu verdoppelten Personalkosten von Paar-Programmierung. Diese erhöhten Entwicklungskosten können allenfalls dadurch ausgeglichen werden, dass durch einen schnelleren Markteintritt ein höherer Marktanteil und damit ein höherer Gewinn möglich ist. Ein weiterer Pfeiler des Modells bilden die qualitativ höherwertigen Programme der Entwicklerpaare. Im Modell müssen die Einzelprogrammierer diesen Qualitätsunterschied vor Markteintritt durch Mehrarbeit ausgleichen, um eine vergleichbare Basis zu schaffen. Diese Mehrarbeit bedeutet einen Zeitverlust und damit wieder Verlust von Marktanteilen. Dieser Abschnitt skizziert die relevanten Eigenschaften des Modells.

4.1 Kapitalwert

Im Modell wird der ökonomische Wert eines Software-Projektes durch den *Kapitalwert* (*engl. net present value* NPV) modelliert.

$$\mathsf{Kapitalwert} = \frac{\mathsf{Aktivwert}}{(1 + \mathsf{Diskontsatz})^{\mathsf{Entwicklungszeit}}} - \mathsf{Entwicklungskosten}$$

Der Kapitalwert spiegelt die Eigenschaft wieder, dass sich die mit einem Projekt erzielbaren Marktanteile und Preise nach $\mathtt{T}$ Jahren um $(1 + \mathsf{Diskontsatz})^{\mathsf{T}}$ verringern. Der *Diskontsatz* wird in den Wirtschaftswissenschaften zur Modellierung des Marktdrucks verwendet. Werte für den Diskontsatz von bis zu 100 Prozent sind gängig. Der *Aktivwert* gibt den Wert des Projektes wieder, wenn es sofort vermarktet werden könnte. Projekte mit einem negativen Kapitalwert bedeuten einen Verlust.

4.2 Geschwindigkeit der Paare

In der hier betrachteten Variante unseres Modells haben XP und das konventionelle Projekt die gleiche Anzahl an Entwicklern zur Verfügung. Das XP Projekt setzt die Entwickler in Paaren ein. Wenn die Paare genau doppelt so schnell arbeiten wie die Einzelprogrammierer, dann ergibt sich keine zeitliche Differenz bei der Fertigstellung der Projekte. In der Realität sind die Paare zwar schneller aber nicht doppelt so schnell. Diesen Geschwindigkeitsunterschied zwischen Paaren die testgetrieben entwickeln und konventionellen Einzelentwicklern bei einer “Einheitsaufgabe” gibt der $\mathsf{XPSpeedFactor}$ wieder:

$$\mathsf{XPSpeedFactor} = \frac{\text{Zeit einzelner Programmierer ohne TGE}}{\text{Zeit Programmiererpaar mit TGE}}$$

Die Entwicklungstechnik im Zähler ist der Basisfall. Im Nenner steht die Dauer für die Vergleichstechnik. Im Modell wird hier die Kombination von Paar-Programmierung mit testgetriebener Entwicklung eingesetzt. Oft interessiert auch nur der Geschwindigkeitsvorteil von Entwicklerpaaren, die aber keine testgetriebene Entwicklung einsetzen: der PairSpeedAdvantage. Für diesen Wert wird im Nenner die Zeit für Paar-Programmierung ohne testgetriebene Entwicklung eingesetzt. Ein Wert von 1,0 für den Geschwindigkeitsvorteil bedeutet, dass die Paare doppelt so viel kosten wie Einzelprogrammierer bei gleicher Produktivität. Typischerweise variieren die Werte für den Geschwindigkeitsvorteil zwischen 1,0 und 1,7, siehe Nosek (Nos98) und Williams e.a. (WKCJ00).

Im Modell wird die Produktivität der Paare durch das Produkt der Produktivität der Einzelentwickler mit dem Geschwindigkeitsvorteil der Paare berechnet. Bei *gleicher* Entwickleranzahl und einem Geschwindigkeitsvorteil von 2,0 ergibt sich somit kein zeitlicher Unterschied zwischen der Fertigstellung beider Projekte. Bei kleineren Geschwindigkeitsunterschieden und gleicher Enwicklerzahl wird das XP Projekt immer eine längere Entwicklungszeit benötigen als das konventionelle Projekt.

4.3 Fehlerdichte der Programme

Entwicklerpaare können qualitativ höherwertige Programme entwickeln. Die höhere Programmqualität ist im Modell durch eine geringere Defektdichte dargestellt. Der Defektunterschied XPDefectFactor bildet sich aus den unterschiedlichen Defektdichten des XP und konventionellen Projektes (C).

$$\mathsf{XPDefectFactor} = \frac{\mathsf{DefectDensity}_{\mathsf{C}} - \mathsf{DefectDensity}_{\mathsf{XP}}}{\mathsf{DefectDensity}_{\mathsf{C}}} \quad .$$

Bei gegebener Projektgröße in Programmzeilen lässt sich somit die Anzahl der Fehler berechnen, die das konventionell entwickelte Programm mehr enthält als das XP-Programm. Diese Fehler müssen im Modell korrigiert werden. Die Korrektur bedeutet zusätzlichen Aufwand, der sich zur reinen Entwicklungszeit des konventionellen Projektes addiert. Damit können auch XP Projekte mit einem XPSpeedFactor kleiner 2,0 mit konventionellen Projekten mithalten, wenn sich durch die Korrektur der Mehrfehler die Auslieferung des konventionellen Projektes entsprechend verzögert.

4.4 Kapitalwerte der Studien

Tabelle 6 gibt einen Überblick über die Kapitalwerte von XP und konventionellen Projekten für verschiedene Werte von XPSpeedFactor (XPSF) und XPDefectFactor (XPDF). Für die Berechnung wurde ein Aktivwert von 1.000.000 € angenommen. Das XP Projekt benötigt einen sehr hohen Geschwindigkeitsvorteil von 1,8 und einen Defektvorteil von 25 Prozent, um besser als das konventionelle Projekt abzuschneiden. Der Vorteil des XP Projektes berechnet sich nach

$$\frac{\mathsf{NPV}_{\mathsf{XP}} - \mathsf{NPV}_{\mathsf{C}}}{\mathsf{NPV}_{\mathsf{C}}} \quad .$$

Tabelle 6: Kapitalwerte (NPV) in Euro von XP und konventionellen Beispielprojekten.

		Diskontsatz 25 %		Diskontsatz 75 %	
XPSF	XPDF	NPV_C	NPV_{XP}	NPV_C	NPV_{XP}
1,4	15 %	626.026	524.093	474.817	341.932
1,8	15 %	626.026	627.851	474.817	477.233
1,8	25 %	600.509	627.851	441.177	477.233

Bei niedrigen Marktdruck (Diskontsatz von 25 Prozent) liegt der Vorteil bei etwa 4,5 Prozent, bei hohem Marktdruck (Diskontsatz von 75 Prozent) bei etwa 8 Prozent. Die Werte in Tabelle 6 für XPSF und XPDF sind den Studien von Nosek (Nos98), Williams e.a. (WKCJ00) sowie Nawrocki & Wojciechowski (NW01) entonmmen. Unsere Berechnungen zeigen, dass selbst für die besten gemessenen Werte von Geschwindigkeits- und Defektvorteil, XP keinen nennenswerten ökonomischen Vorteil gegenüber einem konventionellen Projekt erreicht. Darüber hinaus scheint sich durch die anderen Studien abzuzeichnen, dass die tatsächliche Werte eher geringer ausfallen werden, so, dass sich der Bereich, in dem sich XP lohnt, weiter eingeschränkt wird.

Weiterhin geben erste empirische Erkenntnisse über testgetriebene Entwicklung Grund zur Annahme, dass der Einsatz von testgetriebener Entwicklung die Entwicklungszeit eher verlängert. Wie sich diese Ergebnisse jedoch in Kombination mit Paar-Programmierung verhalten, ist bisher noch nicht untersucht worden.

5 Fazit

Wenn wir die hier vorgestellten Ergebnisse als Grundlage nehmen, so stellt sich XP als Technik dar, die nicht einfach übernommen werden kann. Die vorhandene Literatur sieht XP zu optimistisch. Uns scheint der Einsatz der XP-Techniken in realen Industrieprojekten riskant.

Unsere Einschätzung basiert leider noch auf lückenhaften Wissen, denn es gibt noch offene Fragen. Testautomatisierung, der test-zuerst-Ansatz und Paar-Programmierung haben qualitätssteigernde Effekte. Kann eine dieser Techniken weggelassen werden, ohne dass die Qualität dabei sinkt ? Die Paar-Programmierung könnte z.B. durch Einzelprogrammierer mit zusätzlichen Durchsichten ersetzt werden. Existiert zu den anderen beiden Techniken auch ein konventionelles Gegenstück ? Weitere Lücke betreffen speziell die Paar-Programmierung. Welche Eigenschaften machen ein produktives Entwicklerpaar aus ? Kann das Potential der Paar-Programmierung auch ausgeschöpft werden, wenn Paare nur in frühen Stadien der Entwicklung, wie z.B. im Entwurf, eingesetzt werden ?

Ein letzter Punkt betrifft die Frage: Was war vor XP ? Ist der Erfolg von XP nicht letztendlich das Resultat der Anwendung eines definierten Software-Entwicklungsprozesses im Vergleich zu einem ad-hoc Prozess nach CMM Stufe 1 ? Paulk (Pau01) berichtet beispielsweise, dass XP einige Schlüsselbereiche der CMM Stufen 2 und 3 erfüllt. So beschreibt das Software-CMM Prozessbereiche die mit Leben gefüllt werden müssen, und XP gibt in einigen dieser Bereiche eine Antwort für ein kleines Entwicklerteam in einem Umfeld mit sich schnell ändernden Anforderungen.

Wir verstehen dieses Papier als einen rationalen Beitrag zur XP-Diskussion. Obwohl XP keinesfalls in allen Umständen einen Vorteil bewirkt, sehen wir in XP eine Bereicherung der Prozesslandschaft, die es vor allem auch wegen der hier aufgeworfenen Fragen weiter zu erkunden gilt.

Literatur

[Bec99] K. Beck. Embracing Change with Extreme Programming. *IEEE Software*, Seiten 70–77, Oktober 1999.

[Cha01] R. Charette. The Decision is in: Agile versus heavy Methodologies. *Executive Update, e-Project Management Advisory Service*, 2(19), 2001.

[CW00] A. Cockburn und L. Williams. The Costs and Benefi ts of Pair Programming. In *eXtreme Programming and Flexible Processes in Software Engineering (XP2000)*, Cagliari, Italy, Juni 2000.

[Fen03] J. Fenwick. Adapting XP to an Academic Environment by Phasing-In Practices. In *XP/Agile Universe*, Jgg. 2753 of *Lecture Notes in Coputer Science*, Seiten 162–171. Springer, 2003.

[GAD02] H. Gallis, E. Arisholm und T. Dyba. A Transition from Partner Programming to Pair Programming - an Industrial Case Study. In *Workshop "Pair Programming Installedïn Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Seattle, USA, November 2002.

[GSM04] A. Geras, M. Smith und J. Miller. A Prototype Empirical Evaluation of Test Driven Development. In *International Symposium on Software Metrics (Metrics)*, Seiten 405–416, Chicago, Illinois, USA, September 2004.

[GW03] B. George und L. Williams. An initial investigation of test driven development in industry. In *ACM symposium on Applied computing*, Seiten 1135–1139, Melbourne, Florida, USA, 2003.

[HPSS03] S. Heiberg, U. Puus, P. Salumaa und A. Seeba. Pair-Programming Effect on Developers Productivity. In *XP 2003*, number 2675 in LNCS, Seiten 215–224. Springer, 2003.

[Hul04] H. Hulkko. *Pair Programming and Its Impact on Software Quality: A Multiple Case Study*. Technical Research Centre of Finland, 2004.

[Hum99] W. Humphrey. *Introduction to the Team Software Process*. Addison-Wesley, 1999.

[MH02] M. Müller und O. Hagner. Experiment about test-fi rst programming. *IEE Proceedings Software*, 149(5):131–136, Oktober 2002.

[MLSM04] M. Müller, J. Link, R. Sand und G. Malpohl. Extreme Programming in Curriculum: Experiences from Academia and Industry. In *Conference on Extreme Programming and Agile Processes in Software Engineering (XP2004)*, Garmisch-Partenkirchen, Germany, Juni 2004.

[MP03] M. Müller und F. Padberg. On the Economic Evaluation of XP Projects. In *Joint European Software Engineering Conference (ESEC) and SIGSOFT Symposium on the Foundations on Software Engineering (FSE)*, Seiten 168–177, Helsinki, Finland, September 2003.

[MP04] M. Müller und F. Padberg. An Empirical Study about the Feelgood Factor in Pair Programming. In *International Symposium on Software Metrics (Metrics)*, Chicago, Illinois, USA, September 2004.

[MS04] P. Manhart und K. Schneider. Breaking the Ice for Agile Development of Embedded Software: And Industry Experience Report. In *International Conference on Software Engineering*, Seiten 378–386, Edinburgh, Scotland, UK, Mai 2004.

[MT01] M. Müller und W. Tichy. Case Study: Extreme Programming in a University Environment. In *International Conference on Software Engineering*, Seiten 537–544, Toronto, Canada, Mai 2001.

[Mül04] M. Müller. Are Reviews an Alternative to Pair Programming ? *Journal on Empirical Software Engineering*, 9(4):335–351, Dezember 2004.

[Nos98] J. Nosek. The Case for Collaborative Programming. *Communications of the ACM*, 41(3):105–108, Marz 1998.

[NW01] J. Nawrocki und A. Wojciechowski. Experimental Evaluation of Pair Programming. In *European Software Control and Metrics (Escom)*, London, UK, 2001.

[Pau01] M. Paulk. Extreme Programming from a CMM Perspective. *IEEE Software*, Seiten 19–26, November/Dezember 2001.

[PCTV03] M. Pancur, M. Ciglaric, M. Trampus und T. Vidmar. Towards empirical evaluation of test-driven development in a university environment. In *EUROCON 2003. Computer as a Tool. The IEEE Region 8.*, Jgg. 2, Seiten 83–86, September 2003.

[PSSH04] U. Puus, A. Seeba, P. Salumaa und S. Heiberg. Analyzing Pair Programmer's Satisfaction with the Method, the Result, and the Partner. In *Extreme Programming and Agile Processes in Software Engineering*, number 3092 in LNCS, Seiten 246–249. Springer, 2004.

[Tom02] J. Tomayko. A Comparison of Pair Programming to Inspections for Software Defect Reduction. *Computer Science Education*, 12(3):213–222, 2002.

[WHN93] J. Wilson, N. Hoskin und J. Nosek. The benefi ts of collaboration for student programmers. In *SIGCSE technical symposium on Computer science education*, Seiten 160–164, 1993.

[Wil01] D. Wilson. Teaching XP: A Case Study. In *XP Universe*, Raleigh, NC, USA, Juli 2001.

[WKCJ00] L. Williams, R. Kessler, W. Cunningham und R. Jeffries. Strengthening the Case for Pair-Programming. *IEEE Software*, Seiten 19–25, Juli/August 2000.