extreme programming

mehr zum thema:
www.xprogramming.com
www.agilealliance.org

von kerstin dittert

1+1 > 2
XP UND „PAIR PROGRAMMING"

„eXtreme Programming" (XP) erregt sowohl bei Softwareentwicklern als auch bei Projektmanagern seit dem letzten Jahr große Aufmerksamkeit. Entwicklungszyklen werden immer kürzer und Anforderungen ändern sich immer schneller. Leichte Prozesse sollen die Lösung bringen. Doch hält die Praxis wirklich das, was die Theorie verspricht? Der Artikel berichtet von Erfahrungen bei der Einführung von XP und diskutiert Vor- und Nachteile des „Pair Programming".

die autorin

Kerstin Dittert (E-Mail: kerstin.dittert@oocon.de) ist unabhängige Beraterin für objektorientierte Methoden und E-Business-Technologien. Ihre Schwerpunkte liegen in den Bereichen OOA/OOD, Softwarearchitektur, Projektmanagement und Methodik. Im vergangenen Jahr hat sie als Leiterin eines Design- und Entwicklungsteams einige der XP-Methoden erstmalig eingesetzt.

Was zeichnet XP-Projekte aus?

XP ist ein leichter Entwicklungsprozess, der besonders zur Durchführung objektorientierter Projekte geeignet ist **(siehe Abb. 1)**. Er wurde von Kent Beck entwickelt und von ihm im Projekt „C3" (vgl. [And98]) zum ersten Mal durchgängig eingesetzt.

Zentraler Bestandteil der XP-Methodik ist eine starke Einbeziehung des Kunden. Im Planspiel erarbeitet er gemeinsam mit dem Projektteam die Anforderungen an das Projekt. Seine wichtigste Aufgabe besteht darin, die minimalen fachlichen Anforderungen, die den wirtschaftlichen Nutzen des Projektes gewährleisten, zu definieren. Diese Anforderungen (und auch alle weiteren) priorisiert er zeitlich gemäß dem Gedanken des *Business value first* [1].

Während der gesamten Projektlaufzeit stehen dem Projektteam Mitarbeiter aus der Fachabteilung zur Klärung fachlicher Fragen zur Verfügung (*On-Site-Customer*-Bedingung). Idealerweise arbeiten diese sogar aktiv an der Spezifikation der Testfälle mit.

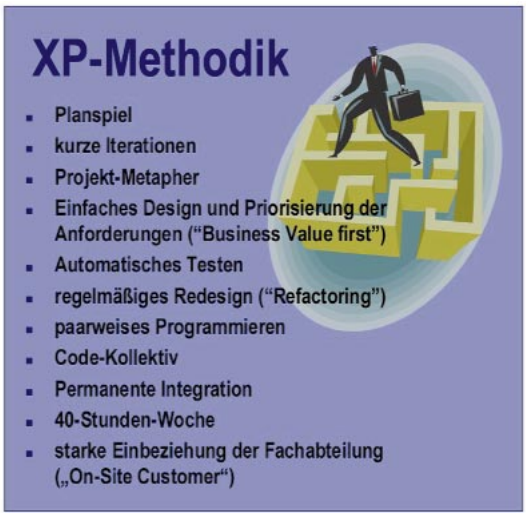

Abb. 1: XP-Methodik

Die Umsetzung der Anforderungen erfolgt gemäß der Priorisierung in kurzen, ungefähr zwei- bis vierwöchigen Iterationen. Die Softwareentwickler des Projektteams arbeiten paarweise zusammen, wobei sich diese Zwei-Personen-Teams immer wieder unterschiedlich zusammensetzen. Niemand hat „Rechte" an den vom ihm erstellten Programmteilen, jedes Teammitglied ist berechtigt Kritik zu äußern und den Code zu verbessern. Der regelmäßige Wechsel der Paar-Konstellationen sorgt für einen klaren Blick auf das implementierte Design. Darüber hinaus ist das Wissen über bestimmte Bereiche der Architektur auf mehrere Köpfe verteilt, was den Ausfall einzelner Mitarbeiter weniger kritisch macht **(vgl. Abb. 2)**.

Erstellter Code wird unmittelbar nach dem Programmieren mit automatischen Testfällen auf seine Lauffähigkeit und die Erfüllung der fachlichen Spezifikation überprüft. Diese Testfälle sind Bestandteil des Quellcodes und können mit Hilfe des Frameworks „JUnit" leicht erstellt werden. Idealerweise codiert der Entwickler die Testfälle sogar vor der Umsetzung der Fachlichkeit. Dies führt zu einer stärkeren fachlichen Zielorientierung im Sinne des „Business value first": Eine Anforderung ist dann umgesetzt, wenn die zugehörigen Testfälle fehlerfrei durchlaufen werden.

Anschließend erfolgt die Integration des neuen Code in die aktuelle Softwareversion. Die automatische Test-Suite minimiert die Gefahr von Seiteneffekten, da alle bisher umgesetzten Anforderungen erneut überprüft werden. Am Ende jeder Iteration steht eine kurze Redesign-Phase (*Refaktorierung*, vgl. [Fow99]). Sie gibt dem Entwicklungsteam die Möglichkeit, das Design und den Code von schnellen Lösungen zu bereinigen oder die Lösungen gleichartiger Probleme in abstrakteren Klassen zusammenzufassen.

Erste Schritte

Das Projekt, in dem ich zum ersten Mal nach der XP-Methodik vorging, hatte als Aufgabe die Neuentwicklung einer Anwendung zur Finanzierungsplanung. Es handelte sich um die Ablösung zweier unterschiedlicher Systeme, die durch die neue Anwendung vereinheitlicht und funktionell erweitert wurden. Die Anforderungsanalyse basierte auf einer Bestandsaufnahme der Altsysteme, Interviews mit den Mitarbeitern der Fachabteilung und resultierte in einer Anwendungsfall-Dokumentation.

Während der fachlichen Analyse stellten sich viele Anforderungen als vage und unklar heraus. Der ehrgeizige Einführungstermin (erstes Release vier Monate nach Projektstart) ließ jedoch eine Verschiebung des Realisierungsbeginns nicht zu. Aus diesem Grunde beschloss ich, einen Großteil der XP-Methoden einzusetzen, um eine schnelle Reaktion auf

[1] *"Business value first" bedeutet, dass im Projektverlauf die Dinge, die dem Kunden den größten wirtschaftlichen Nutzen bringen, zuerst umgesetzt werden.*

Abb. 2: Truck-Faktor

Abb. 3: Teambildung

die zu erwartenden Anforderungsänderungen zu ermöglichen.

Weniger ist mehr

Der Kunde ließ sich für die Durchführung des Planspieles leider nicht begeistern. Alle Projektbeteiligten sahen eine geeignete Priorisierung der Anforderungen zwar als notwendig an, die Erwartungshaltung an den Leistungsumfang des ersten Release war aber unrealistisch hoch. Erst kurz vor der Installation der Anwendung akzeptierte der Kunde eine stabile Version mit geringerem Leistungsumfang (d. h. weniger Masken), aber vollständig implementierter Funktionalität. Als Alternative hierzu war ursprünglich ein Release vorgesehen, das bereits alle Masken enthalten sollte, aber noch viele funktionale Lücken aufgewiesen hätte.

Das Tagesgeschäft lastete die Mitarbeiter der Fachabteilung stark aus, sodass ihnen wenig Zeit für eine aktive Beteiligung am Projekt blieb. Die „On-Site Customer”-Bedingung war also nicht erfüllt, sodass sich die schnelle und unkomplizierte Klärung von fachlichen Fragen sehr schwierig gestaltete. Änderungswünsche wurden erst mit zwei- bis dreiwöchigem Verzug im Rahmen von Abstimmungen oder informalen Präsentationen erkannt.

An eine Mitarbeit der Fachabteilung bei der Testspezifikation war aus den oben genannten Gründen gar nicht mehr zu denken. Das Entwicklungsteam war in diesem Punkt auf sich allein gestellt, was im weiteren Projektverlauf zu einer Verunsicherung bezüglich der tatsächlichen Erfüllung der fachlichen Anforderungen führte.

Auch die Planung kurzer Iterationen gestaltete sich schwierig, da sowohl die Gesamtprojektleitung als auch der Kunde einen Projektplan gemäß dem Wasserfallmodell erwarteten. Aus den gleichen Gründen stieß auch die Einführung von Refaktorierungsphasen auf Schwierigkeiten, sodass diese in zu geringem Umfang durchgeführt wurden. Als Resultat wurde der Code gegen Ende des Projektes deutlich instabiler.

Die Länge der Softwareentwicklungs-Iterationen wurde durch die Meilensteine der einzelnen Releases bestimmt und betrug zwei bis drei Monate. Im Sinne von XP hätten dem Projekt deutlich kürzere Entwicklungszyklen gut getan, um eine kurzfristigere Zieldefinition und -überprüfung zu ermöglichen.

Das Team einschwören

Das Java-Design- und -Entwicklungsteam setzte sich bezüglich objektorientierter Programmier- und Designerfahrung völlig heterogen zusammen: OO-Spezialisten, erfahrene strukturierte Programmierer und Berufsanfänger saßen in einem Boot. Das gesamte Projektteam befürwortete die Idee des paarweisen Programmierens, da sich alle davon einen effizienten Wissenstransfer versprachen.

Zu Beginn der Realisierung wurde der grundlegende Anwendungsrahmen gemäß einer 3-Schichten-Architektur entworfen und implementiert. Diese Phase dauerte gut zwei Wochen, wobei das gesamte Team den größten Teil der Zeit zusammen arbeitete. Die Design-Diskussionen verliefen zeitweise sehr lebhaft, da sich die Vorkenntnisse, Präferenzen und Ansichten über den besten Weg bei den einzelnen Teammitgliedern teilweise stark unterschieden.

Umso positiver war das Ergebnis dieser zweiwöchigen Startphase: Beim exemplarischen Durchgriff über alle Schichten anhand eines konkreten Anwendungsfalles entstand ein tragfähiger Anwendungsrahmen. Dieser diente als Basis und Vorlage für den Rest der Implementierung. Alle Teammitglieder hatten sich schließlich auf das gemeinsam entworfene und implementierte Design eingeschworen und betrachteten es am Ende der zwei Wochen als „ihr Baby” **(siehe Abb. 3)**. Jeder Entwickler kannte alle Schichten der Anwendung aus eigener Erfahrung, da jeder Einzelne an der Erstellung von Serverkomponenten, Masken und *Controllern* beteiligt war.

Ein zunächst befürchteter Zeitverlust („vier Leute an einer Maschine, bei solchem Termindruck!”) wurde im weiteren Projektverlauf mehr als wettgemacht, da das ständige Aufflackern ideologisch geführter Designdiskussionen ausblieb.

Anschließend erfolgte die Realisierung sowohl paarweise als auch einzeln. Ein Teammitglied äußerte ein größeres Bedürfnis, Dinge allein auszuprobieren und zu implementieren. In diesem Fall fanden sich die Partner nur sporadisch zusammen.

Paare und Singles

Das paarweise Programmieren verblüffte besonders durch seinen ausgeprägten Review-Effekt: Ein „krummes” Design führte unmittelbar zu Diskussionen mit dem Partner und logische Fehler wurden meist schnell aufgedeckt. Entwurfsentscheidungen, die nicht hinreichend erklärt werden konnten, bargen meist noch Tücken in sich. Die Diskussion führte in diesem Fällen häufig zu einer effi-

Abb. 4: Aufgabenorientierung im Team

zienteren Lösung. Der Partner achtete ebenso auf die Einhaltung der Programmierrichtlinien, was den Verzicht auf formale Code-Reviews ermöglichte. Darüber hinaus ging die Zahl fehlerhafter Kompilierversuche nahezu auf Null zurück: In der paarweisen Konstellation war es praktisch unmöglich, eine Zeile Code mit fehlerhafter Syntax einzugeben, da der Partner dies sofort bemängelte.

Der Synergieeffekt zwischen Partnern mit unterschiedlichem Kenntnisstand war sehr gut, es profitierte keineswegs nur der „Anfänger" vom „alten Hasen". Gerade die Notwendigkeit, die Dinge unkompliziert und leicht verständlich zu machen, führte zu den im Sinne des „Business value first" willkommenen einfachen Lösungen. Die im Alleingang erstellten Programmteile hatten ihre Licht- und Schattenseiten. Diese Anwendungsteile waren stabil und in extrem kurzer Zeit erstellt worden, d. h. die Produktivität war hier zunächst sehr hoch. Das Design wich jedoch teilweise von den Mustern ab, die zu Anfang gemeinsam erstellt worden waren. Das Fehlen des Partners in seiner Rolle als Design-Reviewer machte sich hier stellenweise bemerkbar. Darüber hinaus blieben diese Klassen immer das „Geheimnis" des jeweiligen Programmierers, zu dem das restliche Team wenig Zugang hatte. Der Truck-Faktor **(siehe Abb. 2)** war bei diesen Komponenten also sehr hoch, was sich im weiteren Projektverlauf rächte, als einzelne Teammitglieder das Projekt verließen.

Beim Programmieren in Paaren stellten alle Beteiligten fest, dass die Kräfte durch die ununterbrochene konzentrierte Arbeit gegen Nachmittag erschöpft waren. Darüber hinaus empfanden einige im Team den Verlust an Privatsphäre in ihrem Arbeitsbereich als unangenehm: Plötzlich fanden jede E-Mail und jeder Anruf Zuschauer und Zuhörer! Auch die freie Arbeitszeiteinteilung des Einzelnen stieß plötzlich an ihre Grenzen: Wenn Frühaufsteher und Nachteulen ein Paar bildeten, waren Konflikte vorprogrammiert.

Das Team einigte sich deshalb auf eine tägliche Kernarbeitszeit von 10 bis 16 Uhr. Innerhalb dieser Zeitspanne arbeiteten die Paare zusammen, davor und danach blieb Freiraum für Routinearbeiten oder das eigenständige Tüfteln am Code.

Programmers love writing Tests?

Für die Durchführung der automatischen Tests wurde das Framework „JUnit" (vgl. [JUnit]) verwendet. Die Integration des Frameworks in die Entwicklungsumgebung und die Erstellung eines allgemeinen Testrahmens erfolgte innerhalb von einem Tag.

Das Schreiben automatischer Tests erwies sich als einfach und effizient — soweit es um das Testen der Fachlogik oder einzelner Hilfsklassen (Konvertierungen, Formatierungen etc.) ging. Zum Abprüfen ereignisgesteuerter Verarbeitung eignete sich das Framework jedoch nicht, sodass ein automatisches Testen der einzelnen Controller scheiterte. Die Maskensteuerung wurde deshalb nach wie vor manuell getestet.

Entgegen der Behauptung von Kent Beck und Erich Gamma „Programmers Love Writing Tests" (aus [Bec]) zeigte sich die Mehrheit der Projektmitarbeiter wenig begeistert beim Schreiben der automatischen Tests. Die meisten wollten an den anstehenden Problemen weiterarbeiten. Das Schreiben eines Testfalls — und das sogar noch vor der Implementierung der eigentlichen Programmlogik — betrachteten viele als unnötige Bremse.

Da die im Projekt erstellte Anwendung wenig Verarbeitungslogik aufwies und äußerst datenzentriert war, konnte mit Hilfe des Frameworks nur ein geringer Teil der eigentlichen Testfälle abgedeckt werden. Die Gefahr von unbemerkten Seiteneffekten war also nicht gebannt. Das eigentliche Ziel der automatischen Tests, mehr Sicherheit bei der permanenten Integration zu gewinnen, wurde deshalb verfehlt.

Licht und Schatten

Einen Kunden für das Planspiel und die anfängliche Reduzierung des Leistungsumfanges zu begeistern, ist extrem schwierig. Im Allgemeinen sieht er zunächst weniger Leistung (d. h. weniger Masken), die funktionale Qualitätssteigerung ist ihm jedoch nur schwer zu vermitteln. Die Priorisierung der Anforderungen lief im Projekt auf eine Salami-Taktik hinaus, bei der der Kunde erst kurz vor Einführung des ersten Release einer vorläufigen Reduzierung des Leistungsumfanges zustimmte. Diese Vorgehensweise belastete das Verhältnis zwischen Projektteam und dem Kunden und kostete alle Beteiligten einiges an Zeit und Nerven. Nach der erfolgreichen Einführung des ersten Release entspannte sich die Situation jedoch.

Einen Auftraggeber von den Vorteilen der schrittweisen Produktentwicklung und -einführung zu überzeugen, scheint eine der größten Herausforderungen im XP-Umfeld zu sein.

Der „Business value first"-Gedanke lieferte den roten Faden, um sich auf die wesentlichen Projektziele zu konzentrieren. Goldene Wasserhähne und niemals benötigte generische Ansätze konnten so vereitelt werden. Vereinzelt wurde diese Richtschnur jedoch an der Projektperipherie mit Hacken und „code like hell" verwechselt, was sich in ungenügenden Zeiträumen für die Refaktorierungsphasen bemerkbar machte. Dadurch sank die Qualität der bereits programmierten Komponenten mit zunehmenden Projektfortschritt und steigender Komplexität, was wiederum zur Unzufriedenheit im Entwicklungsteam führte. Der Einschub einer sehr kurzen Refaktorierungsphase nach Auslieferung des ersten Release brachte nur eine geringe Verbesserung. Insgesamt führte der Verzicht auf regelmäßiges Refaktorieren zu Motivationsproblemen und vermehrten Aufwänden für die Fehlersuche und war damit kontraproduktiv.

Die Vorzüge des automatischen Testens blieben hinter den Erwartungen zurück. Ein Großteil der Tests musste weiterhin manuell durchgeführt werden, da das Framework für den Test grafischer Benutzeroberflächen ungeeignet ist. In hinreichend großen Projekten empfiehlt sich deshalb der zusätzliche Einsatz eines Test-Tools mit der Möglichkeit zur Aufzeichnung und automatischen Wiedergabe von Oberflächen-Ereignissen.

Abb. 5: Erfahrung in diesem Projekt

Die Beteiligung des gesamten Entwicklungsteams am Schreiben der Tests ist unerlässlich. Existieren die Tests nur für ausgewählte Klassen einzelner Entwickler, so gewinnt dieser zwar eine höhere Sicherheit beim Modultest. Das schwierigere Problem der Seiteneffekte bei der Integration der einzelnen Komponenten bekommt man mit diesen „sporadischen” Tests jedoch nicht in den Griff.

Ein voller Erfolg war hingegen die Einführung des paarweisen Programmierens. Besonders positiv wirkte sich die Methode auf den Teambildungsprozess, die Identifikation mit den Designentscheidungen, den Wissenstransfer und die Reduzierung des Truck-Faktors aus **(siehe auch Abb. 2)**. Auf den Einsatz formaler Code- und Design-Reviews konnte verzichtet werden, da diese Aufgabe permanent vom Partner übernommen wurde. Die Produktivität war sehr hoch und ließ gegenüber der Arbeit zweier einzelner Spezialisten nichts zu wünschen übrig. Der Wissenstransfer war deutlich höher als bei einer konventionellen Vorgehensweise. Durch das paarweise Zusammenarbeiten erfolgte er implizit, ohne dass die erfahrenen Teammitglieder ständig in ihrer Arbeit unterbrochen wurden.

Durch den Wechsel der Paar-Konstellationen konnten fast alle Entwickler erstmalig in Bereichen mitarbeiten, in denen sie vorher nicht als Spezialisten galten. Diese Erweiterung des eigenen Wissensspektrum wurde von allen Mitarbeitern als sehr positiv bewertet.

Die Anforderungen an den Arbeitsplatz sind beim paarweisen Programmieren höher als sonst. Der Arbeitsplatz muss so ergonomisch sein, dass beide Partner bequem am Schreibtisch sitzen und jeder den Bildschirm gut sehen kann. Notebooks sind hierfür z. B. ungeeignet, da der optimale Blickwinkel zu schmal ist. Keyboard und Maus sollten von beiden Partnern gut erreichbar sein und den Wechsel zwischen aktiver und passiver Rolle einfach erlauben.

Die Arbeitsweise des paarweisen Programmieren erfordert starke Konzentration und ist durch den Wegfall üblicher Störungen und Ablenkungen — wie Telefonate, -E-Mail etc. — sehr produktiv. Dadurch entfallen allerdings Erholungspausen, wie sie z. B. das Erledigen von Routinearbeiten darstellen. Die tägliche paarweise Zusammenarbeit sollte deshalb maximal sechs Stunden betragen.

Jedes Teammitglied benötigt fest definierte Arbeitszeiten, in denen sein Arbeitsplatz wieder „ihm gehört”. Es empfiehlt sich deshalb, zweimal täglich feste Zeiten zu reservieren, in denen jeder Mitarbeiter seinen Schreibtisch für sich allein hat. Eine noch bessere Alternative stellt das Einrichten von „Paar-Arbeitsplätzen” dar, die zusätzlich zu den persönlichen Schreibtischen der einzelnen Mitarbeiter bestehen.

Paarweises Programmieren erfordert von jedem Einzelnen eine hohe Bereitschaft, Wissen weiterzugeben und die eigene Arbeit zur Diskussion zu stellen. Nicht jeder Mitarbeiter ist dazu bereit oder in der Lage. Manche Menschen eignen sich auf Grund ihrer persönlichen Vorlieben und Arbeitsweisen besser für XP-Projekte als andere. Die Implementierung allgemeiner Lösungsansätze und das unerbittliche Tüfteln an den letzten Details sind in diesen Projekten nicht gefragt. Der Quellcode wandert durch viele Hände, Lösungen können von jedem verworfen und verbessert werden. Aus diesem Grund sind Detaillisten und Sammler in einem XP-Projekt fehl am Platz **(siehe Abb. 4)**. Entwickler mit einer anderen Aufgabenorientierung finden sich einfacher in einem XP-Projekt zurecht, wobei es sowohl fatale als auch besonders produktive Paar-Kombinationen gibt. Die Kombination einer Ideenschleuder mit einem Arbeitspferd birgt auf Grund der unterschiedlichen Arbeitsweise meist viel Konfliktpotenzial in sich. Die Zusammenarbeit eines Strategen und eines Prototypers ergänzt sich hingegen sehr gut und kann besonders produktiv sein. Eine gute Mischung der verschiedenen Typen bringt das Projekt voran, hierin unterscheidet sich die XP-Methodik nicht von anderen Vorgehensmodellen.

Literatur & Links

[And98] A. Anderson, R. Beattie, K. Beck et al., Chrysler Goes to “Extremes”, in: Distributed Computing, Oktober 1998

[Bec] K. Beck, E. Gamma, Test Infected: Programmers Love Writing Tests (siehe http://members.pingnet.ch/gamma/junit.htm)

[Bec99] K. Beck, extreme programming explained, Addison-Wesley, 1999

[Bec00] K. Beck, M. Fowler, Planning Extreme Programming, Addison-Wesley, 2000

[Coc00] A. Cockburn, L. Williams, The Costs and Benefits of Pair Programming, in: Proc. of Conference of eXtreme Programming and Flexible Processes in Software Engineering, 2000

[Fow99] M. Fowler, Refactoring, Addison-Wesley, 2000

[JUnit] K. Beck, E. Gamma, Test-Framework JUnit (siehe http://sourceforge.net/projects/junit)

[Kel00] H. Kellner, Projekte konfliktfrei führen, Carl Hanser, 2000

[Wil] L. Williams, Pair programming questionnaire (siehe http://limes.cs.utah.edu/questionnaire/questionnaire.htm)

[Wil00] L. Williams, R. Kessler, The Effects of “Pair-Pressure” and “Pair-Learning” on Software Engineering Education, in: Proc. of Conference of Software Engineering Education and Training 2000

Bewertung und Ausblick

Insgesamt überwiegen die positiven Erfahrungen, wobei die Methode umso besser funktioniert, je mehr der Bestandteile zusammen angewendet werden **(siehe Abb. 5)**. Das Herausgreifen nur weniger Bestandteile wirkt eher kontraproduktiv. Die einzige Ausnahme hiervon stellt das paarweise Programmieren dar. Sofern die Arbeitsumgebung stimmt und die Teammitglieder Bereitschaft für diese Arbeitsweise zeigen, ist paarweises Programmieren auch ohne die restlichen XP-Elemente äußerst produktiv.

Auch andere Projekte haben gute Erfahrungen mit dem paarweisen Programmieren gemacht. Laurie Williams und ihre Koautoren haben sich unter anderem in [Wil00] und [Coc00] intensiv mit diesem Thema beschäftigt. Wer eigene Erfahrungen zum Thema beitragen möchte, kann sich an einer Internet-Umfrage (siehe [Wil]) beteiligen, deren Ergebnisse regelmäßig aktualisiert und veröffentlicht werden. ■