Die Evolution von eXtreme Programming

Wieso eXtreme Programming so ist, wie es ist.

Von Michael Kircher

Der aktuelle Trend bei der Software-Entwicklung wird derzeit von Java und E-Commerce Projekten getrieben. Es gibt momentan immermehr Projekte, die in immer kürzerer Zeit fertig gestellt werden müssen. Aus diesem Grund werden häufig mächtige Frameworks, wie EJB und Servlets eingesetzt. Diese allein können aber nicht einen ausreichenden Leistungsgewinn in Bezug auf die Entwicklungszeit bringen. Es stellt sich die Frage wie man Software im Internetzeitalter schnell und trotzdem mit guter Qualität ausliefern und dabei noch das Risiko minimieren kann.

Bei mächtigen Entwicklungsprozessen, wie der Rational Unified Process [Jacobsen], wird auf die Hilfe von OOA, OOD Werkzeugen, wie Rational Rose, gesetzt. Leider sind die Entwicklungsprozesse dadurch sehr aufwendig und erfordern damit einen sehr hohen Grundaufwand. Für kleinere Teams sind daher komplexe Entwicklungsprozesse oft nicht gerechtfertigt. Hier sind leichtgewichtige Entwicklungsprozesse gefragt.

eXtreme Programming (XP) ist ein solcher leichtgewichtiger Entwicklungsprozess. Er basiert auf den pragramatischen Vorgehensweisen der Entwickler, aber auch der Projektleiter. Die Erfahrungen bis zum heutigen Tag zeigt, dass der Prozess sich bis zu Teamgrößen von 10-15 Leuten anwenden läßt. Ist das Team größer, so steigt der Kommunikationsaufwand [Brooks] überproportional. Für XP heißt das, dass der Austausch von Wissen über direkte Gespräche, auf denen die Kommunikation in XP hauptsächlich beruht, nicht mehr effizient gestaltet werden kann und damit der enge Zusammenhalt des Entwicklungsteams verloren geht.

Im Folgenden werden die Grundzüge von eXtreme Programming anhand von Problem-Lösungs Paaren dargestellt. Damit lässt sich zeigen wie aktuelle Probleme bei der Software-Entwicklung von XP gelöst werden.

Risikomanagement durch interative Entwicklung

Trotz vieler Jahre Erfahrung in Software Engineering und den Büchern von Frederick P. Brooks bergen Software-Entwicklungs Projekte noch immer sehr hohe Risiken. Diese Risiken sind vielfältig. Besonders hervorzuheben ist die Schätzung des Aufwandes und die schlecht

Kontrollmöglichkeiten. Was sich nicht fassen lässt, ist schwer zu messen und zu kontrollieren.

Um das Risiko bei der Software-Entwicklung zu reduzieren ist es wichitg viele kurze Iterationen zu haben. Innerhalb einer Iteration wird das Software-System um vorher festgelegte Leistungsmerkmale erweitert. Am Ende einer jeden Iteration steht ein lauffähiges System, das in jeder Iteration mehr von der eigentlichen Funktionalität implementiert. Kurze Iterationen erlauben es in den Entwicklungsprozess eingreifen zu können und ihn an Ereignisse anzupassen. Solche Ereignisse können das Ausfallen eines Entwicklers, drastische Qualitätsverluste im Quellcode, oder eine Änderung der Anforderungen des Kunden sein. Je kleiner die atomaren Tätigkeiten einer Iteration sind, desto sanfter kann der Entwicklungsprozess gesteuert werden. Die Steuerung eines solchen Entwicklungsprozesses ist ähnlich dem Steuern eines PKWs auf einer Straß. Eines der wichtigsten Dinge, die ein Fahrschüler lernen muss ist das Gefühl für das Lenkrad, um ein Übersteuern in eine Richtung zu vermeiden.

In traditionellen Entwicklungsprozessen spielt der Kunde nur zu gewissen Zeitpunkten eine Rolle. Hauptsächlich sind diese die Analysephase zu Beginn eines Projektes und die Abnahme der Software durch den Kunden am Ende des Projektes. Die geringe Involvierung des Kunden während der eigentlichen Entwicklung stellt jedoch ein erhebliches Risiko dar. XP versucht dieses zu minimieren. Zu diesem Zweck fordert XP den sogenannten On-Site-Customer. Das bedeutet, dass der Kunde, bzw ein von ihm ernannter Vertreter, in das Entwicklungsteam integriert ist. Der On-Site-Customer arbeitet bei dem Entwicklungsteam vor Ort im gleichen Raum wie die Entwickler.

Der On-Site-Customer hilft bei der Steuerung des Entwicklungsprozess, indem er dem Team schnell Feedback über die akutelle und geplante neue Richtung geben kann. Der Kunde ist schließlich derjenige, der das Ziel am klarsten vor sich sieht.

Eine der wichtigsten Regeln ist ständig ein lauffähiges System zu behalten. Die erste Iteration in einem neuen Projekt ist dafür verantwortlich ein solches zu erstellen. Die fortlaufende Integration neuer Funktionalität während den darauf folgenden Iterationen muss gewährleisten, dass das System weiterhin lauffähig bleibt.

Planungs-Spiel statt unrealistischer Meilensteine

Unrealistische Planung von Meilensteinen durch den Projektleiter führt bei den Entwicklern zu dem Gefühl Unmögliches erreichen zu

müssen und schon zu Anfang verloren zu haben. Letztendlich führt es zu unzufriedenen und frustrierten Entwicklern. Die Folge solcher Umstände ist das Überwechseln der Entwickler zu anderen Firmen, vielleicht sogar zur Konkurrenz. Dass dies ein enormer Wissensverlust für das Team bedeutet und ab einer gewissen Zahl von Verlusten das gesamte Projekt gefährden kann wird oft vergessen, bzw ignoriert.

Das Planungs-Spiel, ein weiterer Hauptbestandteil von XP, soll dafür sorgen, dass weder nur Geschäftsinteressen, zumeist vertreten durch den Kunden und den Projektleiter, noch die Interessen der Entwickler einseitig oder vorrangig behandelt werden. Vielmehr wird eine Aufteilung beschlossen: Personen die Geschäftsinteressen vertreten entscheiden über den Fokus der Iterationen, die Priorität von Leistungsmerkmalen und welche Leistungsmerkmale zu welchem Zeitpunkt zur Verfügung stehen sollen. Die Entwickler hingegen entscheiden über den Aufwand, der notwendig ist um ein Leistungsmerkmal zu implementieren.

Der Kunde formuliert seine Erwartungen über die Leistungsmerkmale in Form von Benutzungsbeispielen. Diese Anwendungsbeispiele werden auf Karten (größere Karteikarten) handschriftlich festgehalten.

Während der Kunde Prioritäten auf die einzelnen Karten vergibt, schätzen die Entwickler den Aufwand, der notwendig ist um das entsprechende Leistungsmerkmal zu implementieren. Der Kunde und der Projektleiter respektieren und akzeptieren diese Schätzungen.

Danach wählt der Kunde die Karten aus, welche er in der nächsten Iteration implementiert haben möchte.

Die Entwickler schätzen dabei in Ideal-Tagen, die noch mit einem Last-Faktor multipliziert werden, um sie als absolute Schätzung abzugeben. Der Last-Faktor ist zu Beginn eines Projektes mit einem neuen Team meist unbekannt und muss daher zuerst ermittelt werden. Bei Teams, die schon zuvor an ähnlichen Projekten gearbeitet haben, liegen meist schon Erfahrungswerte vor. Er ergibt sich aus dem Verhältnis der eigentlichen Dauer einer Tätigkeit zu der Schätzung, wie sie die Entwickler anfangs genannt hatten.

Die Entwickler lernen dabei das richtige Schätzen von Aufwänden und gewinnen das Vertrauen in den Prozess. Sie hängen also nicht mehr von den gutmütigen Schätzungen und Versprechungen des Projektleiters ab, sondern sind selbst für ihre Schätzungen verantwortlich.

Durch das Planungs-Spiel hat der Kunde bei jedem Release Einfluss darauf, welche Funktionalität enthalten sein wird.

Teil von XP ist die Regel, dass den Entwicklern nicht mehr als eine 40-Stunden Woche zugemutet werden sollte. Die Praxis zeigt, dass Entwickler, die über einen längeren Zeitraum mehr als 40 Stunden pro Woche arbeiten, eine geringere Konzentration bei ihren Aufgaben haben. Viele Überstunden führen in der Regel zu vermehrten Fehlern im Programm-Code und dies nicht nur durch Nachlässigkeit beim Testen.

Wartung durch einfache Architektur auf Basis von Entwurfsmustern erleichtern

Die Wartung von großen Software-Systemen bereitet häufig große Probleme. Dass ein Stück Software nicht, oder nur mit sehr großem Aufwand, wartbar ist hat häufig mehrere Gründe. Unter anderem sind dies über die Jahre gewachsene, komplexe Architekturen, die meist noch schlecht dokumentiert sind. Duplizierter Quellcode, oft durch Cut&Paste Programmierung entstanden, und verwaister Quellcode tun ihr übriges um ein solches System schwer verständlich zu machen.

XP spiegelt eine Grundregel erfahrener Programmierer wieder, welche besagt, dass die Architektur einer Applikation immer so einfach wie möglich gehalten werden sollte. Einfache Architekturen sind jedoch nicht immer einfach zu erreichen. Bei der Entwicklung ist es wichtig das Rad nicht immer neu zu erfinden, sondern von Expertenwissen in Form von Muster-Lösungen, sogenannten Design Patterns, zu profitieren. Viele bekannte Design Patterns erlauben es eine einfache, aber trotzdem erweiterbare, Architektur zu erarbeiten.

Der schrittweise Umbau einer existierenden Software-Architektur wird sehr gut durch Refactoring [Fowler99] durchgeführt. Damit lassen sich nicht nur komplizierte Architekturen vereinfachen, sondern auch schon einfache Architekturen weiter umbauen. Auch Refactoring baut auf den Regel des sanften Lenkens auf. Refactoring empfiehlt Programmänderungen in sehr kleinen Schritten zu machen und diese sofort durch einen Test zu verifizieren. Eine Liste von elementaren Änderungen, wie sie sich für objekt-orientierte Programmiersprachen ergeben, ist in Martin Fowler's Refactoring Buch beschrieben.

Eine Eigenschaft von einfachen Architekturen ist auch, dass sie die DRY-Regel [Hunt99] beachten. DRY steht hierbei für "Don't repeat yourself," und besagt, dass eine Funktionalität nur einmal im Quellcode implementiert werden sollte. Findet man zwei äquivalente Stellen, so sollten diese durch Refactoring zu einer reduziert werden.

Neben des Bestrebens einfache Architekturen zu erstellen ist es

hilfreich, die Tests vor dem eigentlichen Applikations Quellcode zu schreiben. Mit dieser Regel wird gewährleistet, dass der Entwickler sich auf das eigentliche Problem fokusiert und nicht falsche oder unnötige Funktionalität implementiert. Wichtige Features werden dabei von einem sogenannten Unit-Tests überprüft. Ein Unit-Test stellt die kleinste Testeinheit eines System-Tests dar und überprüft genau ein Feature. Ein sehr interessanter Nebeneffekt ist auch, dass damit die Test Suite des Projektes, die sämtliche Unit-Tests enthält, ständig wächst und sich ergänzt. Bei der Integration von neuem bzw. geändertem Quellcode stehen sämtliche Tests aller Entwickler zur Verfügung. Es kann also während der Integration gesehen werden, ob und welche Test-Fälle durch den integrierten Quellcode nicht mehr korrekt ablaufen. Die Integration ist erst beendet, wenn sämtliche Testfälle der Test Suite korrekt durchlaufen werden.

Gemeinsame Code-Verantwortung statt Abhängigkeit von einzelnen Entwicklern

Welcher Entwickler wünscht sich nicht unersetzbar zu sein? Manche Entwickler schaffen es tatsächlich für die Entwicklung einer Software unersetzbar zu sein. So gut dies für den einzelnen Entwickler sein mag, z.B. bei Gehaltsforderungen, so schlecht ist es für das Projekt selbst. Denn sollte dieser Entwickler ausfallen, verliert das Team zunächst ersatzlos die Betreuung für ein wesentliches Stück Quellcode.

Die Lösung zu diesem Problem ist die gemeinsame Code-Verantwortung. Dies bedeutet, dass das Team gemeinsam für den gesamten Quellcode verantwortlich ist und jeder an jeder Stelle Änderungen machen kann. Um nicht Chaos in das System zu bringen ist es wichtig, dass diese durch ein leistungsfähiges Versionsverwaltungs-Tool koordiniert wird.

Der Transfer von Wissen erfolgt im Wesentlichen über die Paar-Programmierung. Paar-Programmierung funktioniert folgendermaßen: Zwei Entwickler entwickeln zusammen an einem Rechner, d.h. eine Maschine, ein Monitor, eine Tastatur und eine Maus. Während der eine schreibt, kann der andere schon die nächsten Ideen, bzw. Schritte, ausdenken und/oder den Quellcode reviewen. Die Rolle desjenigen der schreibt ist nicht fest vorgegeben, sollte der andere schon ein komplettes Refactoring gedanklich vorbereitet haben, so übernimmt dieser.

Durch den ständigen Wechsel der Paare wird das Wissen zwischen allen Mitgliedern des Teams verteilt. Jedes Teammitglied kann daher kompetent Änderungen an einem Stück Quellcode vornehmen, das ursprünglich ein anderes Teammitglied entwickelt hat.

Falls "smelling code", sogenannter "riechender Quellcode", von einem Teammitglied entdeckt wird, so bemüht sich dieses Teammitglied den Quellcode mit Hilfe von Refactoring so zu verändern, dass er nicht mehr riecht.

Bei gemeinsamer Code-Verantwortung besteht die Gefahr, dass Entwickler viel Zeit mit der Anpassung des Quellcodes an ihren Formatierstil verbringen, und dies teilweise nur um eine geschweifte Klammer nicht am Ende der vorhergehenden Zeile zu haben, sondern am Anfang der nächsten. Darum ist es nicht nur wegen der gemeinsamen Code-Verantwortung sinnvoll Codier-Richtlinien aufzustellen, sondern auch wegen der Verständlichkeit und der Qualität.

Da XP versucht den Entwicklungsprozess so schlank wie möglich zu halten, wird dabei auf viel traditionelle Dokumentation verzichtet. Sogenannte "sprechende Bezeichner" sind die Grundlage eines sich selbst dokumentierenen Quellcodes. Natürlich unter der Prämisse, dass der Quellcode den Codier-Richtlinien entspricht.

System-Metaphern helfen, das Ganze im Blick zu behalten

Während der Entwicklung einer Lösung sind die Entwickler oft so vertieft in die Details ihres Problemes, dass sie keinen Überblick über das gesamte Projekt haben und es ihnen schwer fällt sich in andere Probleme, als das an dem sie gerade arbeiten, hinein zu denken. Dies kann unter anderem zu dupliziertem Quellcode und nicht benötigter Funktionalität führen. Ziel muss es sein dies zu vermeiden aber trotzdem den Entwickler nicht aus seiner Welt zu reißen.

Um dies zu erreichen schlägt XP vor eine System-Metapher für das Projekt zu wählen. Diese System-Metapher beschreibt den Kontext und die Komponenten innerhalb des Projektes konsistent mit Begriffen, die z.B. aus der realen Welt stammen können. Sind keine solchen Assoziationen mit der realen Welt möglich, wie es bei Frameworks der in der Regel der Fall ist, so eignen sich dafür auch Design Patterns und daraus entstehende Pattern Languages.

Eine eindeutige Begriffswelt hilft nicht nur bei der aktuellen Arbeit, sondern auch um neue Teammitglieder in das Team zu integrieren - sogenanntes "Insourcing". Intensives Mentoring durch Paar-Programmierung und die Verwendung der System-Metapher bieten den Neulingen einen schnellen Einstieg in das Projekt.

Zusammenfassung

Bei der Darstellung der Gründe, wieso XP so ist, wie es ist, dürfte klar geworden sein, dass es oft nicht eine bestimmte Lösung für ein gegebenes Problem gibt. Oftmals sind es vernetzte Zusammenhänge, die eine Lösung für ein Problem darstellen. Daher ist es schwierig sich Teile von XP heraus zu nehmen und sie einzeln in Isolation mit einem anderen Prozess zu verwenden. Am besten funktioniert dies noch mit Unit-Tests und dem Refactoring. Sie können auch der Anfang einer langsamen Migration eines traditionellen Software-Projektes hin zu einem XP Projekt darstellen.

[Brooks75] F. P. Brooks, The Mythical Man-Month. Reading, MA: Addison-Wesley, 1975

[Beck99] Kent Beck, Extreme Programming Explained: Embrace Change , Addison-Wesley, 1999

[Beck00] Kent Beck and Martin Fowler, Planning Extreme Programming, Addison-Wesley, 2000

[Fowler99] Martin Fowler, Refactoring, Addison Wesley, 1999

[Jeffries00] Ron Jeffries, Ann Anderson, and Chet Hendrickson, Extreme Programming Installed, Addison-Wesley, 2000

[Hunt00]Andrew Hunt and David Thomas, The Pragmatic Programmer: From Journeyman to Master, Addison-Wesley, 1999

[Jacobsen98] Ivar Jacobsen, Grady Booch, James Rumbaugh: The Unified Software Development Process, Addison-Wesley, 1998