Einführung von XP[1] in der Praxis

Seminar "Agile vs. klassische Methoden der Software-Entwicklung"

David Kocher, dkocher@sudo.ch
00-806-968

Institut für Informatik der Universität Zürich
Dozent: Prof. Dr. M. Glinz
Assistent: Ch. Seybold

28. Oktober 2003

[1] eXtreme Programming

Inhaltsverzeichnis

Kapitel 1

Einleitung

Extreme Programming (XP) bezeichnet sich selber als “leichte Methode” für kleine bis mittlere Entwicklungsteams. XP will Gemeinkosten von Software Projekten senken und gleichzeitig die Qualität und den Businesswert sicherstellen. Dieses Papier bezieht sich auf die Praktiken, welche XP dazu vorschlägt.

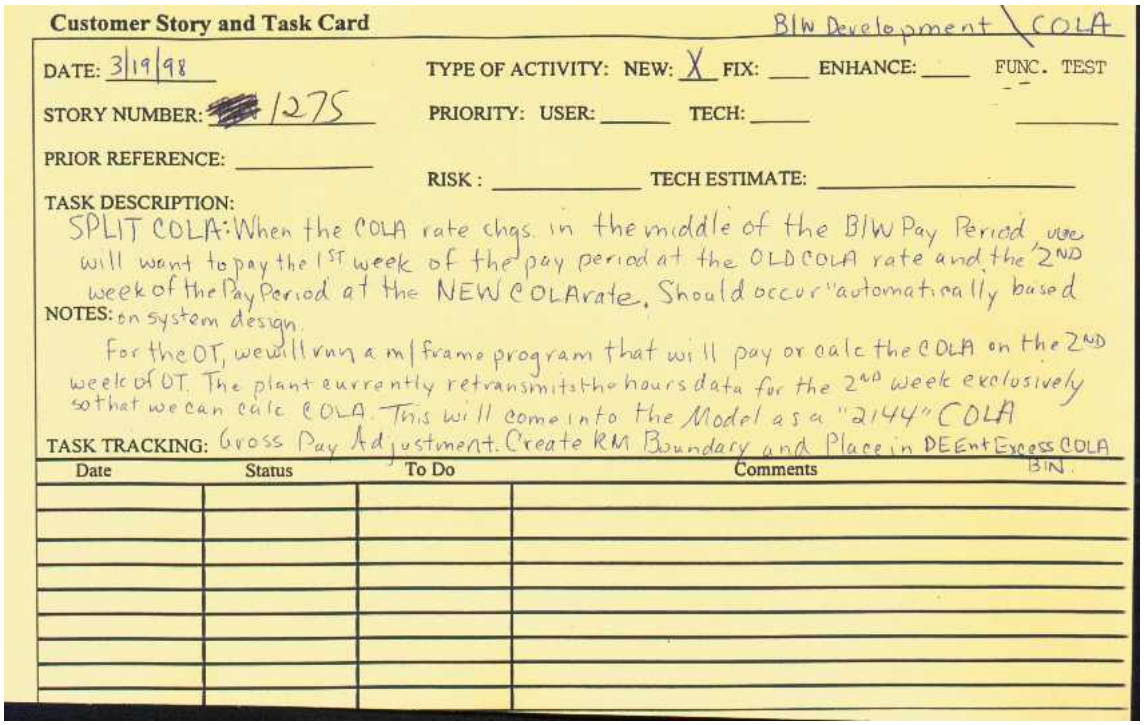
Kapitel 2

Planungsstrategie

2.1 Planungsspiel

Das Planungsspiel wird sowohl bei der Releaseplanung als auch bei der Iterationsplanung eingesetzt. Prinzipiell geht es beim Planungsspiel darum, die Anforderungen des Kunden bzw. die für eine Iteration vorgesehenen Aufgaben mit der zur Verfügung stehenden Programmierkapazität in Einklang zu bringen. Dazu wird der Aufwand für die Implementierung der Benutzergeschichten (oder Tasks in der Iterationsplanung) geschätzt. Wenn zu viele Benutzergeschichten für das Release bzw. für die Iteration geplant sind, muss im Dialog mit dem Kunden geklärt werden, was weggelassen werden kann. Sind umgekehrt noch Kapazitäten frei, kann entsprechend mehr gemacht werden.

2.2 Die Benutzergeschichte



Customer Story and Task Card — B|W Development \ COLA

DATE: 3|19|98 — TYPE OF ACTIVITY: NEW: X FIX: ___ ENHANCE: ___ FUNC. TEST

STORY NUMBER: [redacted] 1275 — PRIORITY: USER: ___ TECH: ___

PRIOR REFERENCE: ___ — RISK: ___ TECH ESTIMATE: ___

TASK DESCRIPTION:
SPLIT COLA: When the COLA rate chgs. in the middle of the B|W Pay Period we will want to pay the 1ST week of the pay period at the OLD COLA rate and the 2ND week of the Pay Period at the NEW COLA rate. Should occur "automatically based on system design.

NOTES:
For the OT, we will run a m|frame program that will pay or calc the COLA on the 2ND week of OT. The plant currently retransmits the hours data for the 2ND week exclusively so that we can calc COLA. This will come into the Model as a "2144" COLA

TASK TRACKING: Gross Pay Adjustment. Create RM Boundary and Place in DEEntExpress COLA BIN.

Date	Status	To Do	Comments

Abbildung 2.1: Beispiel einer Story Card

Eine Benutzergeschichte (engl. User Story) ist eine Beschreibung einer Teilfunktionalität des Systems und gleichzeitig das Funktionalitätsmass in einem XP Projekt. Sie bilden auch die Grundlage für die

Akzeptanztests (Black Box Tests). Eine Benutzergeschichte wird vom Kunden verfasst, ist wenige Sätze kurz und *nicht* technischer Natur. Der Fortschritt in einem XP Projekt wird gemessen an getestetem und integriertem Code, der eine Benutzergeschichte implementiert. Eine Benutzergeschichte muss sowohl für den Kunden als auch für den Entwickler verständlich sein.
Eine Benutzergeschichte ist ein Stück Funktionalität (Feature) welches einen direkten Wert für den Kunden hat. Es soll eine einfache Art sein für die Entwickler und Kunden die Anforderungen an das System in kleine Stücke zu splitten, die so unabhängig ausgeliefert werden können. Im Gegensatz zu anderen Ansätzen im Requirements Engineering setzt XP den Schwerpunkt auf Einfachheit. Es wird versucht, die simpelste Vorgehensweise umzusetzen.
Alternativen zu einer physischen Karte (engl. Story Card) sind ein Wiki[1] oder eine simple Tabelle.

2.2.1 Prinzipien von guten Benutzergeschichten

Einige Grundsätze die beim Schreiben von Stories zu beachten sind:

- Benutzergeschichten müssen für den Kunden verständlich sein; in einer natürlichen Sprache geschrieben. Je kürzer die Benutzergeschichte, desto besser. Die Benutzergeschichte ist nur ein Konzept und keine detaillierte Spezifikation. Im Grundsatz ist eine Benutzergeschichte nichts anders als ein Agreement zwischen Kunde und Entwickler über ein gewünschtes Feature zu sprechen.
- Jede Benutzergeschichte muss dem Kunden einen direkten Nutzen (Funktionalität) bringen.
- Entwickler schreiben keine Benutzergeschichten. Ein Entwickler soll zwar seine Ideen einbringen, es ist aber immer in der Verantwortung des Kunden, Stories auszwählen. Entwickler haben sich auf die technischen Aspekte zu konzentrieren.
- Eine Benutzergeschichte sollte klein genug sein, dass mehrere davon in einer Iteration implementiert werden können.
- Stories sollten unabhängig voneinander sein, damit sie in unbestimmter Reihenfolge implementiert werden können.
- Jede Benutzergeschichte sollte testfähig sein.

2.2.2 Wann ist man fertig mit Stories schreiben?

Zu Beginn des Projekts werden sicher mehr Benutzergeschichten geschrieben als gegen Ende des Projekts, doch während der Projektzeit kommen neue dazu, werden alte verworfen oder abgeändert. Die ist keineswegs negativ, sondern ein zentraler Punkt der XP Philosophie. Das Ziel ist Software zu liefern, die die Anforderungen in jedem Release erfüllt und nicht die Anforderungen, wie sie zu Beginn des Projekts missverstanden wurden.

2.3 Releaseplanung

Bei einem Release Planning Meeting zusammen mit dem Kunden wird das Projekt auf die gesamte Dauer geschätzt. In der Releaseplanung wählt der Kunde fokussiert auf ein Public Release Stories, die einige Monate Arbeit ergeben. Zwei Aspekte des Projektes sind zu synchronisieren: Datum und Umfang. Das Datum eines Release kann extern bestimmt sein; beispielsweise durch eine Austellungsmesse, vertragliche Abmachungen oder finanzielle Gründe. Der nächste Releasezeitpunkt ist also fast immer durch Geschäftsgründe gegeben und nicht durch technische Aspekte bestimmt.
Es besteht häufig der Zielkonflikt zwischen zu vielen oder zu weit auseinanderliegenden Releases. Häufige Releases sind oft mit dem Ziel verbunden der Konkurrenz voraus zu sein. Die Unterschiede zur letzten Iteration sind dann aber vielleicht so klein sind, dass die Leute von der Marketing Abteilung nicht wissen, was sie in der Pressemiteilung schreiben sollen.
Jedoch ist im XP ein Releaseplan alles andere als statisch. Der Plan ist lediglich eine Momentaufnahme der aktuellen Sicht auf die zu lösenden Aufgaben. Releaseplanung ist kontinuierlich und wird korrigiert sowohl wenn der Kunde seine Ansicht bezüglich der Anforderungen und Prioritäten ändert, als auch wenn

[1] Collaborative Web Tool, `http://c2.com/cgi/wiki?WikiWikiWeb`

das Entwicklerteam mehr Zeit für die Implementation braucht.
Das Wesentliche bei einem Release Planning Event bleibt es den Aufwand der Benutzergeschichten in Ideal Programming Weeks zu messen.

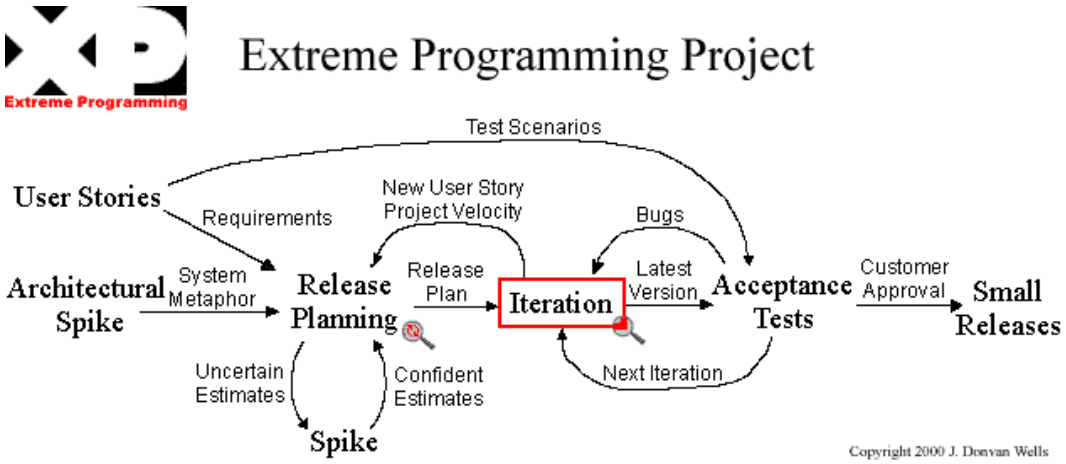


Abbildung 2.2: Projektplanung

2.3.1 Release Planning Meeting

Der Ablauf eines Release Planning Meetings gestaltet sich folgendermassen:

1. *Explorationsphase.* Herausfinden, welche neuen Anforderungen (Requirements) an das System gestellt werden sollen. Der Kunde schreibt eine Benutzergeschichte, welche eine Anforderung des Systems beschreibt. Das Entwicklungsteam schätzt den benötigten Aufwand zur Implementierung nach den "Ideal Programming Days"; also ohne Unterbrechungen oder Meetings. Eine Benutzergeschichte wird eventuell aufgesplittet, wenn der Task zu gross ist, so dass der Aufwand nicht geschätzt werden kann.
   Auch XP kennt kein Patentrezept für eine exakte Aufwandschätzung (engl. Story Estimation). Die beste Grundlage nach XP ist der Vergleich mit einer ähnlichen Benutzergeschichte. XP nennt drei Schlüssel für eine effektive Schätzung:

   (a) Einfach halten (*Keep it simple*)

   (b) Vergleiche mit frühreren Projekten (*Use what happend in the past*)

   (c) Erfahrungen (*Learn from the experience*)

   Es wird angenommen, dass ähnliche Stories einen ähnlichen Arbeitsumfang mit sich bringen.
   Aufwandschätzung ist in XP eine Teamarbeit. Das Team diskutiert die Benutzergeschichte und fällt schliesslich gemeinsam eine Entscheidung. Es wird grundsätzlich immer die kürzeste Zeit genommen; das heisst Optimismus gewinnt. Dadurch soll bewirkt werden, dass die Schätzungen nicht unnötig gross werden und gleichzeitig das Team nicht zu optimistisch ist.
   Sollten keine Erfahrungen vorliegen, kann ein sogenannter Spike geschrieben werden, um die Einschätzung über die Entwicklungszeit zu verbessern. Spikes sind kleine Programme, die ein riskantes technisches Problem, oder eine riskante Designfrage implementieren. Das Problem wird abstrahiert, alles was nichts damit zu tun hat weggelassen.

2. *Commitment Phase.* Das Ziel der Commitment Phase ist es, den Umfang und das Datum des nächsten Release festzulegen.
   Im klassischen Projektmanagement bestimmen die Abhängikeiten zwischen den Tasks die Planung. XP jedoch vernachlässigt diese Abhängikeiten beziehungsweise ist der Ansicht, dass diese fast immer zu umgehen sind. XP setzt auf andere Faktoren zur Sequenzierung der Stories:

   - *Business Value.* Stories, welche dem Kunden einen höheren Nutzen bringen sind zu bevorzugen. (high-value first)

- *Technisches Risiko.* Stories mit höherem Risiko sind vorzuziehen, damit noch genügend Zeit zum Handeln bleibt. (high-risk first)

Um die Entscheidung zu vereinfachen, werden die Stories sortiert nach Wert, Risiko und Geschwindigkeit.

(a) *Wert.* Die Stories werden auf drei Stapel verteilt. (1) Tasks, ohne welche das System nicht funktioniert, (2) weniger Essentielle aber trotzdem einen hohen Geschäftswert, (3) Nice to have.

(b) *Risiko.* Auch hier werden die Stories auf drei Stapel verteilt, je nachdem wie genau deren Risiko geschätzt werden kann.

(c) *Geschwindigkeit.* Die Entwickler legen fest, wie viele "Ideal Programming Days" pro Monat drin liegen.

(d) *Umfang.* Der Kunde wählt die Tasks, die in diesem Release ausgeführt werden sollen woraus sich das Release Datum ergibt.

3. *Steuerung.* Das Ziel der Steuerungsphase ist die fortlaufende Aktualisierung des Plans.

   - *Iteration.* Zu Beginn jeder Iteration werden die Benutzergeschichten ausgewählt, die implementiert werden sollen.
   - *Korrektur.* Sollte die Arbeitsgeschwindigkeit überschätzt worden sein, muss entschieden werden, welche Benutzergeschichten im aktuellen Release Priorität haben und welche verschoben werden.
   - *Neue Benutzergeschichte.* Wird festgestellt, dass eine zusätzliche Anforderung implementiert werden muss, kann diese auf Kosten einer anderen Benutzergeschichte in das aktuelle Release aufgenommen werden.
   - *Neueinschätzung.* Wenn festgestellt wird, dass der jetzige Plan nicht mehr dem aktuellen Entwicklungsstand entspricht, können alle Benutzergeschichten neu geschätzt werden.

2.3.2 Wie weit voraus soll geplant werden?

Je weiter geplant wird, desto ungenauer ist er Plan selbst. Darum plant XP lediglich zwei Iterationen und zwei Releases im voraus.

2.3.3 Planen der Infrastruktur

XP baut immer nur soviel Infrastruktur auf wie gerade nötig. Für jede Iteration wird nicht mehr Infrastruktur - darunter fallen beispielsweise das GUI Framework oder die Datenbankpersistenz die keinen direkten Kundennutzen bringen - gebaut, als für die Stories gerade nötig. Dadurch soll vermieden werden, dass eine komplexere Infrastruktur aufgebaut wird als später überhaupt nötig.

2.4 Iterationsplanung

Im Gegensatz zur Releaseplanung sind bei der Iterationsplanung nur die einzelnen Entwickler anwesend. Statt ganze Story Cards werden Tasks Cards ausgefüllt, die eine einzelne Iteration abdecken. Im Ablauf ist die Iterationsplanung ähnlich die Releaseplanung, die Zeiteinteilung beschränkt sich aber auf eine bis vier Wochen - eine Iteration. Während die Benutzergeschichten in der Sprache des Kunden geschrieben sind, werden Tasks mit einem technischen Vokabular formuliert.

1. *Exploration.* Die Stories welche in der Releaseplanung für diese Iteration ausgewählt wurden werden in Tasks umgeschrieben. Häufig ergibt eine Benutzergeschichte mehrere Tasks, da der Umfang eines Tasks nicht mehr als ein paar Tage beanspruchen sollte. Auch kann es möglich sein, dass ein Task keine direkte Beziehung zu einer Benutzergeschichte hat, wenn zum Beispiel zu einer neuen Version des Application Server migriert werden soll.

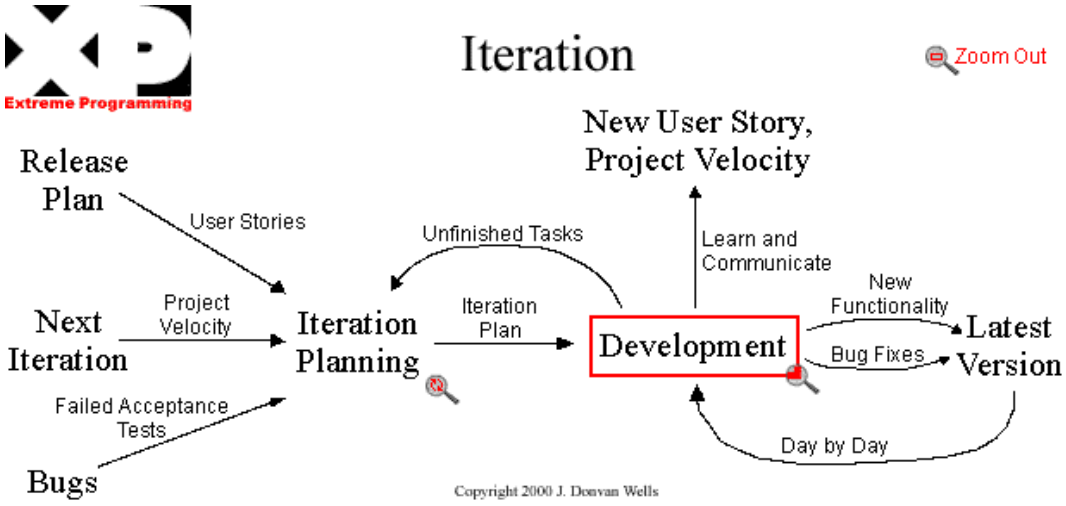


Abbildung 2.3: Iterationsplanung

2. *Commitment.* Ein Task wird einem Programmierer zugewiesen. Der verantwortliche Programmierer schätzt die Anzahl "Ideal Programming Days"[2] um den Task implementieren zu können.
   Jeder Task sollte nicht länger als bis zu drei "Ideal Programming Days" in Anspruch nehmen. Tasks kleiner als ein Tag sollten gruppiert werden, Tasks länger als drei Tage sollten weiter aufgeteilt werden. Der Auslastungsfaktor ergibt sich aus dem Verhältnis zur Länge der Iteration. In einer Iterationsphase von zwei bis drei Wochen sollten nicht mehr als sieben "Ideal Programming Days" angenommen werden.
   Die Zeitschätzung in "Ideal Programming Days" sollte die Projektgeschwindigkeit der letzten Iteration nicht überschreiten. Beinhaltet die Iteration zu viele Aufgaben, muss der Kunde Benutzergeschichten auf eine spätere Iteration verschieben (Snow Plowing).

3. *Steuerungsphase.* Zur Umsetzung eines Tasks gehört es einen Partner zu finden (Pair Programming), die Test Cases schreiben, die tatsächliche Implementierung und anschliessendes Release des Codes wenn die Test Suite erfolgreich durchlaufen wird. Sollte ein Programmierer überlastet (overcommited) sein, kann (1) der Umfang des Tasks verkleinert werden oder (2) der Kunde gebeten werden den Umfang der ganzen Benutzergeschichte zu verkleinern oder gar auf ein späteres Release zu verschieben.

[2]Ideal Programming Days sind die Anzahl Tage, die es brauchen würde um den Task zu erfüllen, wenn keine anderen Aufgaben (Sitzungen, Teamarbeit) wären

Kapitel 3

Entwicklungsstrategie

Die Entwicklungsstragie in XP unterscheidet sich am meisten von konventionellen Ansätzen des Software Engineering. XP setzt auf eine bestmögliche Lösung für heute und klammert die Zukunft bewusst aus. Die Entwicklungsstragie beginnt mit einer Iterationsplanung. Kontinuierliche Integration soll Konflikte bei der Entwicklung reduzieren. Collective Code Ownership ermutigt das ganze Team das System zu verbessern. Pair Programming soll den Prozess zusammenhalten.

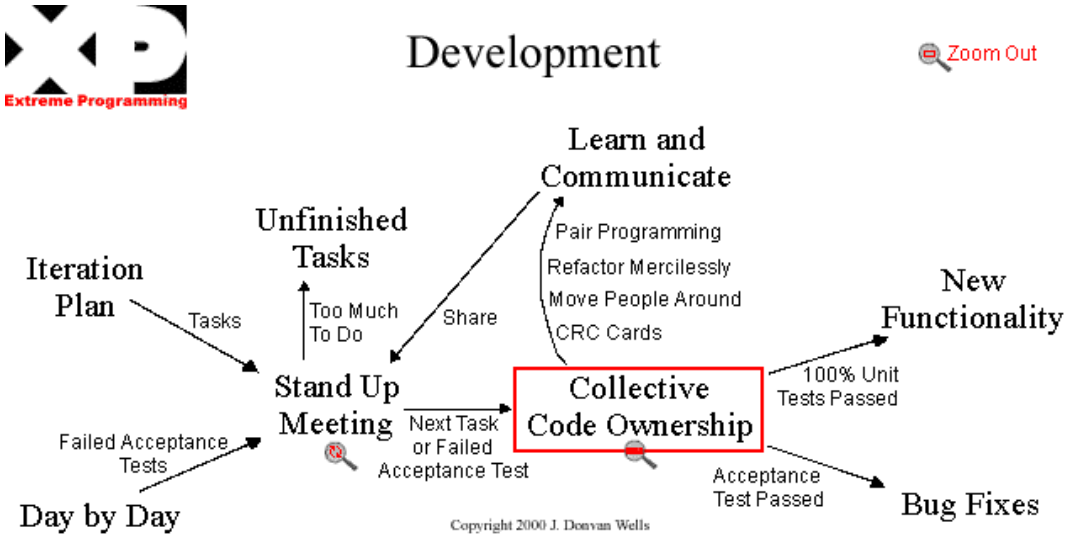


Abbildung 3.1: Arbeitsablauf in einem XP Projekt

3.1 Schlichtes Design

Bei traditionellen Vorgehensweisen wird häufig ein grosser Aufwand in das Software-Design gesteckt, lange bevor die erste Zeile Code geschrieben wurde. Dies führt aber zum Problem, dass sich während der Implementierung neue Einsichten ergeben, die häufig eine Designänderung erfordern. Die Ursachen für solche Änderungen sind vielfältig:

- Das ursprüngliche Design führt zu Performanceproblemen.
- Das ursprüngliche Design ist zu komplex.
- Es gibt eine neue Anforderung des Kunden, die in das System integriert werden muss.

Im Gegenteil dazu geht XP davon aus, dass solche Änderungen der Normalfall sind. In Verbindung mit dem Grundsatz immer den geringstmöglichen Aufwand zu betreiben, erstellen XP Programmierer immer nur das minimal nötige Design. Über die Zeit wird das Design weiterentwickelt, so dass genau das Design entsteht, dass für das System ausreichend ist.

3.2 Fortlaufende Integration

Nachdem eine Aufgabe vollständig implementiert wurde - nachweisbar durch erfolgreiches Absolvieren der Testfälle -, wird die Codeänderung in das System integriert. Dafür wird ein separater Arbeitsplatz verwendet, so dass zu einem beliebigen Zeitpunkt immer nur genau eine Änderung in das System integriert werden kann.
Es ist wichtig, dass Tools vorhanden sind, welche diesen Zyklus von schneller Integration, Build und Test unterstützen.

3.3 Kleine Releases

Mit grossen Releases kann zwar viel Funktionalität auf einmal geliefert werden. Aber typischerweise werden die Anforderungen zu Beginn der Implementierung, also am Anfang des Releasezyklus, festgelegt. Während der Projektlaufzeit verändern sich die Anforderungen. Für Kunden wird es auch immer wichtiger, dass sich eine Investition möglichst früh rentiert. Je eher also ein System produktiv eingesetzt werden kann, umso eher kann damit auch Geld verdient werden. Der Kunde hat auch die Möglichkeit, den weiteren Projektverlauf zu beeinflussen, z.B. indem er die Prioritäten verändert, oder auch Funktionalitäten hinzufügt oder aus den Anforderungen herausnimmt.

3.4 Collective Code Ownership

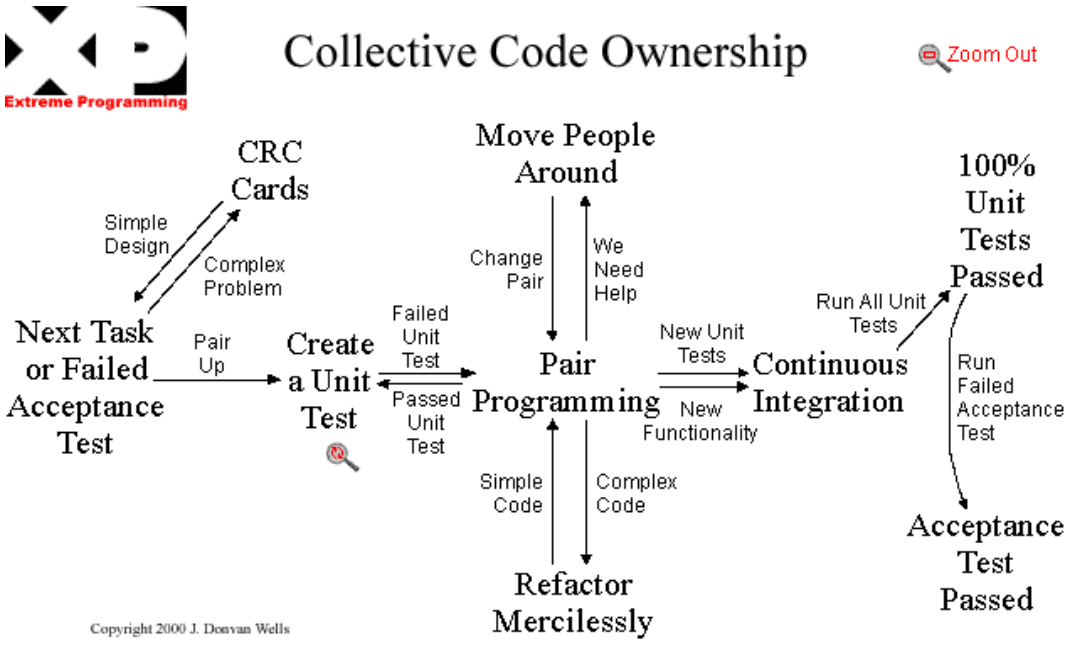


Abbildung 3.2: XP Entwicklungsstrategie

Häufig ist die Arbeitsaufteilung in traditionellen Softwareprojekten dergestalt, dass jede Komponente einem Ingenieur zugeordnet ist. Dieser Ingenieur ist verantwortlich für das Detaildesign, die Implementierung sowie die Wartung dieses Subsystems zuständig. Mit anderen Worten, dieser Ingenieur ist "Eigentümer" des Codes mit allem was dazugehört.
Auf den ersten Blick macht dies sehr viel Sinn, denn im Normalfall ist es so, dass die Fähigkeiten und das Wissen nicht gleichmässig im Team verteilt sind. So gibt es Experten für Datenbanken, für Benutzeroberflächen, für bestimmte Algorithmen.
Doch ergeben sich daraus erhebliche Projektrisiken. Was passiert wenn jemand im Urlaub ist, eine Weiterbildungsmassnahme besucht, krank wird oder sogar das Unternehmen verlässt? Wer kümmert sich dann um das Subsystem dieses Ingenieurs?

Collective Code Ownership hat die Idee, dass jeder Programmierer jedes Stück Code im System jederzeit editieren darf. Dies bringt einige Vorteile mit sich:

- Komplexer Code wird schneller eliminiert, da solcher Code schneller gefunden wird.
- Es wird verhindert, dass komplexer Code überhaupt geschrieben wird. Wenn ein Programmierer weiss, dass jemand anderes das selbe Stück Code auch anschauen wird, überlegt er sich zweimal ob er zusätzliche Komplexität einbauen soll.
- Collective Code Ownership gibt dem Programmier nicht das Gefühl, dass er sein Problem lösen könnte, wenn nur nicht das andere Codestück wäre, das er nicht editieren kann. Statt dessen ändert er es gleich selber.
- Das Projektrisiko wird reduziert, da die Gefahr, dass nur jemand ein Codestück editiert geringer ist (und mit Pair Programming ausgeschlossen werden kann).
- Aus Sicht der Programmierer ergibt sich der Vorteil, eben nicht immer am "ungeliebten" Subsystem arbeiten zu müssen. Statt dessen ergibt sich eine Abwechslung, in dem immer wieder in unterschiedlichen Bereichen gearbeitet werden kann. Dies führt in der Regel auch zu einer Marktwertsteigerung, weil die Basis aus Wissen und Fähigkeiten kontinuierlich verbreitert wird.

3.5 Pair Programming

Jeder Code, der in das Produktionsrelase eingefügt werden soll, wird von zwei Personen am selben Computer erarbeitet. Pair Programming soll die Softwarequalität erhöhen, ohne die termingerechte Auslieferung zu tangieren. Dies scheint entegegen der Intuition zu sein. Doch zwei Personen werden zusammen ebenso viel Funktionalität implementieren, wie wenn diese separat arbeiten würden. Gleichzeitig wird dabei ein höherer Qualitätsstandard erreicht. Die Qualität steigt, weil während der eine am tippen ist, der andere sich Gedanken auf einem strategischen Level machen kann.
Das Programmieren läuft nach einem einfachen Schema ab:

1. Komponententest schreiben
2. Komponententest laufen lassen; schlägt fehl
3. Implementieren
4. Komponententest laufen lassen; schlägt der Test fehl, zurück zur Implementation.

Da jeweils nur einer - nämlich der Fahrer - an der Tastatur arbeiten kann, nimmt der Navigator automatisch eine breitere Perspektive an. Während der Fahrer tippt, kann der Navigator mitlesen, und viele der typischen Codierfehler sofort erkennen, wie z.B. Tippfehler, falsche Variablennamen, aber auch Verletzungen der Programmierrichtlinien.
Schliesslich ergibt sich durch das paarweise Programmieren auch ein paarweises Lernen. Die Partner können voneinander lernen, aber natürlich auch gemeinsam experimentieren, z.B. eine Refaktorierung ausprobieren. Der Unerfahrene kann vom Erfahrenen lernen; die Wissensunterschiede im Team werden reduziert. Die Verwendung von paarweisem Programmieren unterstützt somit die Entwicklung des Teams zu einer lernenden Organisation.
Pair Programming ist ein Dialog zwischen zwei Entwicklern, die zusammen programmieren, anlysieren, designen und testen. Pair Programming soll die Produktivität, Qualität und Zufriedenheit der Entwickler fördern.

3.6 Refactoring

Der Quellcode wird dabei auf die Weise umformuliert, so dass sich funktional nichts ändert, dass aber der Quellcode an Qualität in verschiedener Hinsicht gewinnt. Zum Beispiel wird versucht, ein "switch"-Statement durch Polymorphismus zu ersetzen. Refactoring hat also das Ziel, für eine gegebene Implementierung eine einfachere Formulierung in der Programmiersprache zu finden.

3.7 Kunde vor Ort

Lehrbuchmässig bedeutet dies, dass der Kunde Bestandteil des Programmierteams ist (On-site Customer). Die typische Aufgabe des Kunden ist es, seine Anforderungen mit Hilfe von Benutzergeschichten zu formulieren.
Eine weitere wichtige Aufgabe des Kunden vor Ort ist es, für die Programmierer jederzeit ansprechbar zu sein, um Detailfragen in Bezug auf Benutzergeschichten zu klären. Für das Planungsspiel werden die Benutzergeschichten nur bis zu einem Detaillierungsgrad definiert, der für die Auswahl zur Implementierung bzw. die Aufwandsabschätzung erforderlich ist. Wenn es dann an die konkrete Implementierung geht, ist es normal, dass weiterer Klärungsbedarf entsteht.
In der Praxis stellt sich aber oft heraus, dass der Kunde nicht immer vor Ort sein 'kann'. In der Regel wird dadurch zusätzlicher Kommunikationsaufwand erforderlich sein. Wenn ein Kunde vor Ort sich nicht realisieren lässt, so sollte doch zumindest ein Mitglied des Programmierteams diese Rolle einnehmen und als "Anwalt des Kunden" arbeiten.

Kapitel 4

Teststrategie

> *The idea is that the developers are responsible for proving to their customers that the code works correctly; it's not the customer's job to prove that the code is broken. – Kent Beck*

Über das Testen will im Software Engineering niemand sprechen: Jedermann weiss, dass Testen wichtig ist; jedermann weiss aber auch, dass zu wenig getestet wird. Ein Programm ohne automatisierten Test existiert in XP grundsätzlich nicht. Testen umfasst sowohl die Unit Tests als auch die Akzeptanztests durch den Kunden. Testen in XP verfolgt zwei Ziele:

- Die Modifikation eines Programms ist einfacher. Es kann einfach verifiziert werden, dass die Änderung keine Nebeneffekte hervorruft.
- Der Kunde ist überzeugt von der Qualität der Software, wenn seine funktionalen Tests erfolgreich durchlaufen werden.

4.1 Unit Tests

Der Programmierer schreibt Tests pro Methode in folgenden Fällen[1]:

1. Das Interface für eine Methode ist unklar.
2. Die Implementation einer Methode könnte kompliziert sein.
3. Es gibt eine seltene Ausnahme im Code. Der Test zeigt die Umstände dazu auf.
4. Ein Problem wird gefunden und der Test isoliert das Problem.
5. Bevor ein Refactoring vorgenommen wird und unklar ist, wie es sich verhalten wird.

4.2 Funktionale Tests

Akzeptanztests ermöglichen dem Kunden zu prüfen ob das System funktioniert und zeigt dem Entwickler auf was fehlt. Die Akzeptanztests werden vom Kunden spezifiert. Wichtig ist dass auch hier die Tests automatisiert werden. Funktionale Tests müssen im Gegensatz zu Unit Tests nicht zu 100% durchlaufen, wobei sie sich natürlich gegen Ende des Release dazu tendieren sollten.

[1] xUnit Testing Framework, `http://xprogramming.com/software.htm`

Kapitel 5

Rollenverteilung

5.1 Programmierer

Der Programmierer gilt als das Herzstück der XP Philosphie. Ein XP Programmierer sollte kommunikativer sein als der typische Programmierer. Pair Programming ohne Worte funktioniert schlecht. Eine andere Eigenschaft die nicht fehlen sollte, ist die Liebe zur Einfachheit. Lösungen möglichst simpel umzusetzen ist ein Kernpunkt der XP Philosophie. Ein XP Programmierer sollte mit den Kernkonzepten der XP Umsetzung vertraut sein wie dem Refactoring und der Idee des Shared Code Ownership.

5.2 Kunde

Eines der wenigen Anforderungen um XP umsetzen zu können ist dass der Kunde bei Projekten von wesentlicher Grösse jederzeit verfügbar ist. Nicht nur um das Entwicklerteam zu unterstützen, sondern selbst Mitglied des Entwicklerteam zu sein. Jede Phase eines XP Projekts benötigt die Kommunikation mit dem Kunden vor Ort.
Die wichtigste Aufgabe des Kunden ist es, die Anforderungen mit Benutzergeschichten abzudecken und Prioritäten zu setzen. Der Kunde hat auch die Aufgabe bei der Release Planning Sitzung die Stories für das nächste Release auszuwählen.
Weil Details auf der Story Card ausgelassen werden, wird der Kunde auch bei der Umsetzung (Programmierung) benötigt.
Weiter wird der Kunde benötigt um die funktionalen Tests zu spezifieren und die Ergebnisse zu kontrollieren.

5.3 Tester

Da der grosse Teil der Verantwortung für das Testen beim Programmierer selbst lieg, ist die Rolle des Testers im XP Team auf den Kunden fokussiert. Der XP Tester ist verantwortlich den Kunden beim auswählen und schreiben der funktionalen Tests zu unterstützen. Sollten die funktionalen Tests nicht Teil der Integration sein, liegt bei ihm die Verantwortung diese regelmässig selbst zu fahren.

5.4 Coach

Der Coach ist verantwortlich für den Prozess als Ganzes. Der Coach sollte die XP Prinzipien am besten kennen und eingreifen, wenn das Team vom Weg abkommt.

Kapitel 6

Lebenszyklus eines XP Projekts

Ein ideales XP Projekt hat eine kurze initiale Entwicklungsphase gefolgt von einigen Jahren ständigen Produktsupport und Verbesserungen. Das Projekt endet im Idealfall wenn es keinen Business Value mehr liefert.

6.1 Exploration

Ein Projekt in unproduktivem Zustand ist ein unnatürlicher Zustand und sollte darum so kurz wie möglich gehalten werden. Nicht in der Produktion sein, heisst Ausgaben ohne entsprechende Einnahmen. Jedoch müssen die Entwickler auch überzeugt sein, dass ihr System produktionstauglich ist.
In der Explorationsphase werden alle Technologien benutzt, die später in der Produktion eingesetzt werden. Dieses Experimentierfeld beinhaltet beispielsweise Performancetests oder die unterschiedliche Implementierung einer Architekturidee.
Während das Team sich mit der Technologie vertraut macht, schreibt der Kunde erste Stories oder macht sich zumindest mit dem Konzept vertraut.

6.2 Planung

Das Ziel der Planungsphase ist, dass sich Kunde und Entwickler auf ein erstes Releasedatum für die kleinsten aber wertvollsten Stories einigen. Der Plan für das erste Release sollte zwischen zwei und sechs Monaten liegen; in einem kleineren Zeitrahmen ist wohl kein ernsthaftes Business Problem zu lösen. Erstreckt sich aber der Plan länger als auf sechs Monate, ist das Risiko einer Fehlschätzung zu hoch.

6.3 Iterationen bis zum ersten Release

Jede Iteration sollte für jede Benutzergeschichte ein Set an funktionalen Testfällen liefern, welche idealerweise alle erfolgreich durchlaufen. Werden zwischen den Iterationen Abweichungen vom Projektplan festgestellt wird entweder der Projektplan (Hinzufügen oder entfernen von Stories), der Prozess (Umgang mit der Technologie) oder aber die Arbeit mit XP selbst revidiert. Am Ende der letzten Iteration ist das System parat für die Produktion.

6.4 Pflege

Dies ist der eigentliche 'normale' Zustand eines XP Projekts. Es wird gleichzeitig neue Funktionalität hinzugefügt, das aktuelle System gewartet und neue Programmierer in das Team einbezogen und bisherige verabschiedet. Neu entwickelte Programme sollten fortlaufend in die Produktion einfliessen.

6.5 Tod

Zwei Gründe gibt es sich von einem Projekt zu verabschieden:

1. Der Kunde findet keine neuen Stories mehr. Dies ist der Zeitpunkt ein fünf- bis zehnseitiges Dokument über das System zu fertigen, das ein anderer Entwickler in einigen Jahren froh sein wird vorzufinden.

2. Die schlechtere Variante für ein (vorzeitiges) Ende ist, wenn das System die Erwartungen nicht efüllt. Der Kunde braucht Funktionen, die das Team nicht imstande ist, ökonomisch vertretbar zu integrieren. Auch ein XP Projekt ist nicht gefeit davor.

Kapitel 7

Grenzen von XP

Die exakten Grenzen von XP sind wohl heute noch nicht klar. Wann aber XP sicher nicht funktioniert - und nach den Initianten selbst auch gar nicht dazu gedacht ist - ist bekannt:

7.1 Unternehmenskultur

Die wohl grösste Barriere für XP ist die Unternehmenskultur. Wenn beispielsweise der Kunde auf eine komplette Spezifikation insistiert bevor auch nur eine einzige Zeile Code geschrieben wird, ist es schlicht nicht möglich XP einzusetzen. Ein anderes Beispiel für eine grundlegende Kulturbarriere ist ein Unternehmen, wo Überzeit ein Zeichen der Loyalität zur Firma ist. XP (und in Wahrheit wohl nicht nur XP) jedoch duldet keine Müdigkeit.

7.2 Grosse Entwicklerteams

XP ist auch nicht dazu gedacht in grossen Entwicklersteams eingesetzt zu werden. XP funktioniert nicht in einem Projekt mit hundert Programmieren. Zehn Programmierer gilt in der Literatur als die obere Grenze. Auch in XP ist der produzierte Funktionalitätsumfang nicht linear zur Anzahl Programmierer im Projekt.

7.3 Lange Feedbackzeiten

Lange Feedbackzeiten sind ein weiterer Hemmschuh für XP. Wenn ein System 24h zum Kompilieren benötigt, kommen Integration und Tests zu kurz. Oder wenn die Qualitätskontrolle eine ganzen Monat benötigt bevor die Software in Produktion geht, kommt das Feedback zu spät.

Literaturverzeichnis

[1] Kent Beck: eXtreme Programming eXplained - Embrace Change, Addison-Wesley, 2000

[2] Kent Beck, Martin Fowler: Planning eXtreme Programming, Addison-Wesley, 2000

[3] Ron Jeffries, Ann Anderson, Chet Hendrickson: eXtreme Programming Installed, Addison-Wesley, 2000

[4] `http://www.extremeprogramming.org`

[5] `http://www.xpexchange.net`

[6] `http://clabs.org/xpprac.htm`