XTREME Programming

FH Mannheim - Fakultät Informatik
Seminar MSI – Extreme Programming

gehalten von:
Markus Boos und Ingo Stilz

Betreuer:
Prof. Dr. Schramm, Prof. Dr. Knauber

17.05.2005

Übersicht

1. Was bedeutet „Extreme Programming“ ?

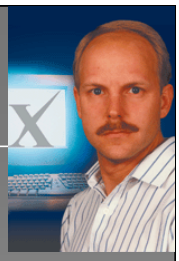
2. Die 12 XP-Techniken

3. „Implementierung“ und Organisation von XP-Projekten

4. Erfahrungen und Bewertung

1. Was bedeutet „Extreme Programming“ ?

Entstehung von eXtreme Programming…



- …agile Software-Entwicklungs-Methode
- …1996 von Kent Beck, Ward Cunningham und Ron Jeffries erstmals im Lohnbuchhaltungsprojekt C3 bei Daimler-Chrylser eingesetzt (nachdem ein schwergewichtiger Prozess daran gescheitert war)
- …einzelne XP-Techniken älter, jedoch noch nie zuvor aufeinander abgestimmt eingesetzt
- …Wurzeln liegen in der Tradition und Kultur von Smalltalk (OOP)
- …im Juni 2000 erste internationale XP Konferenz in Sardinien
- …agiles Manifest entstand im Jahre 2001 (Agile Software Developement Alliance)

Kent Beck über das C3-Projekt (aus [Plam03])**:**

Beck: „The project had serious problems -- everything from the **contractual arrangement, to exhausted people, to lowered voices.** After three days I told the CIO of Chrysler, Sue Unger, that she had three options: keep going, which no one wanted; cancel the project and fire everyone; or **give everyone a week off and start over**. She picked option "c" and said "but you're in charge.

They didn't work end-to-end. They worked their way through the system such that in 18 months they had done the data import part, had done some calculation work, and were getting around to the export part. Lo and behold when they spit out numbers, [the results] didn't reconcile with anyone else's numbers.

Their **biggest problem**: they treated the **requirements as a fixed set of tasks**. The tasks would grow over time, like a mirage. At some point the project recedes as fast as it progresses. **Instead, we made the project's scope visible with stories with costs and a fixed budget**.

So, I said that at the end of the first three weeks we'll print a live, cashable check. They said that's impossible -- we must import data from 18,000 systems. I asked if there was one person whose check we could print. "Well, for most people it's not that difficult." After three weeks, we printed a live check with correct data. They framed it and put it on the wall. **That's the XP creation myth**.

Up until then I believed better programming would solve all the world's ills. Yes, you can screw up the programming so badly you kill the project. **Usually**, however, **the problem concerns relationships between the business people and the programmers**, **the budget process, poor communications -- factors unrelated to the programming**. The **context in which the software development takes place proves as important to the project's success as the programming itself**. “

XP als agile Methode

traditionelle Vorgehensmodelle

Ingenieurwissenschaften

„Brücke bleibt Brücke“

eXtreme Programming

kreativer Lernprozess

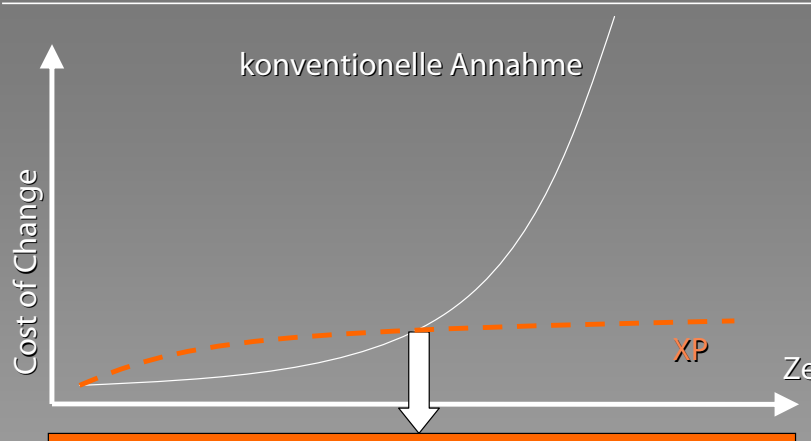
Beobachtung: Anforderungen sind oft zu Beginn nicht genau festzulegen

→ agile Reaktion auf veränderte Bedingungen

traditionelle Softwareentwicklung <-> Agile Softwareentwicklung

Kritiker von traditionellen Prozessen gehen davon aus, dass Software nichts anderes ist als ausführbares Wissen. Wissen lässt sich jedoch nicht ingenieursmäßig (wie eine Brücke) herstellen, sondern wird in einem kreativen Prozess gefunden.

Cost-of-Change Kurve



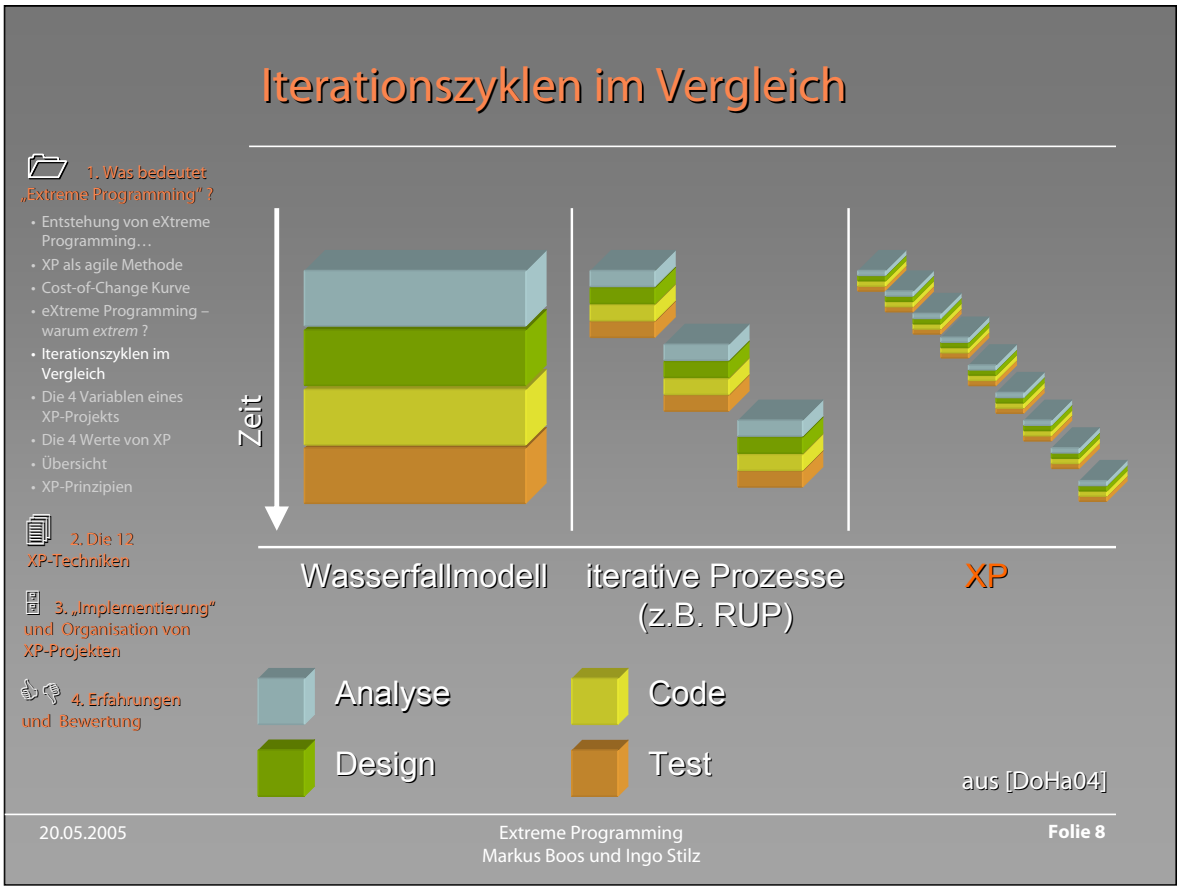
Konsequenzen für den Einsatz von XP:
- Später entscheiden
- Später implementieren
- Veränderungen willkommen heissen!

r

eXtreme Programming – warum *extrem* ?

BEWÄHRTE PRAKTIKEN	…INS EXTREME GEFÜHRT
„Code Reviews“ sind gut	Code wird andauernd begutachtet, Pair-Programming
Testen ist gut	Ständiges Testen
Design ist wichtig	Ständiges Redesign und Refactoring
Einfachheit ist wichtig	Alles so einfach wie möglich
Architektur ist wichtig	Ständige Architekturweiterentwicklung
Kurze Iterationszyklen sind gut	Extrem kurz: wenige Stunden oder Tage

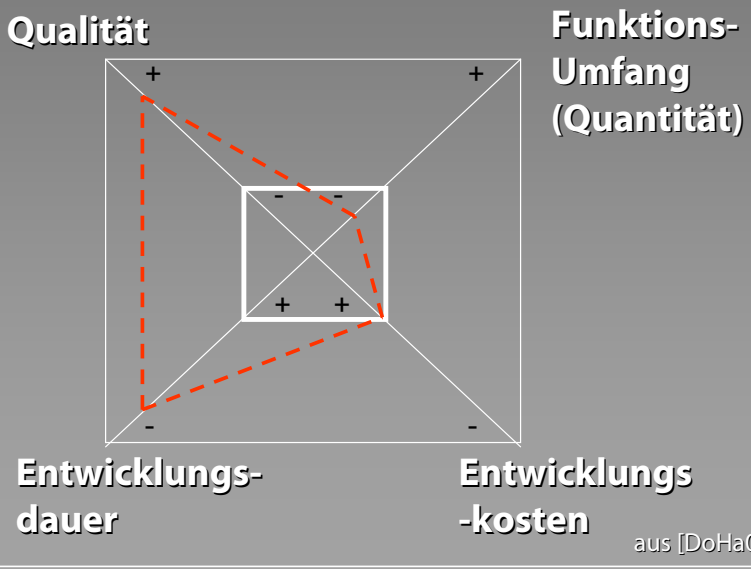
„Extreme“ bedeutet nicht mehr Risikobereitschaft oder „Hacken“ sondern: Bewährte, sinnvolle Praktiken werden ins Extreme geführt.



Im Gegensatz zu traditionellen Vorgehensmodellen gibt es bei XP keine einzelnen Phasen. XP ist ein iterativer Prozess, bei dem laufend Analyse, Design, Coding und Testing stattfindet. Der Vorteil ist, dass nie wegen Zeitmangel kurz vor einer Deadline die Test- und Integrationsphase kürzer oder ganz ausfällt. Das Ergebnis jeder einzelnen Wiederholung ist eine getestete und lauffähige Software.

Die 4 Variablen eines XP-Projekts

- …oder das „Teufelsquadrat“ nach Sneed



aus [DoHa04]

Das Quadrat in der Mitte entspricht den vier Variablenbelegungen zu Projektbeginn. Ändern sich ein bis drei Variablen im Laufe des Projekts, so bewegen sich die entsprechenden Eckpunkte des Quadrats (gestrichelte Linien). Es gilt die 3+1-Regel: Wenn drei Werte gegeben sind, steht die 4. Variable automatisch fest. D.h. wenn durch den Kunden mehr Qualität gefordert wird, schneller entwickelt werden soll und die Kosten gleichbleiben müssen, kann dies nur auf Kosten des Funktionsumfangs geschehen. Hier muss dann zurückgesteckt werden.

Einfachheit

→die zum aktuellen Zeitpunkt möglichst einfache Lösung anstreben

→keine Annahmen über zukünftige Implementierungen treffen

Kommunikation

→ durch direkte Kommunikation werden Planungsdokumente weniger wichtig

→ Vermeidung von Missverständnissen

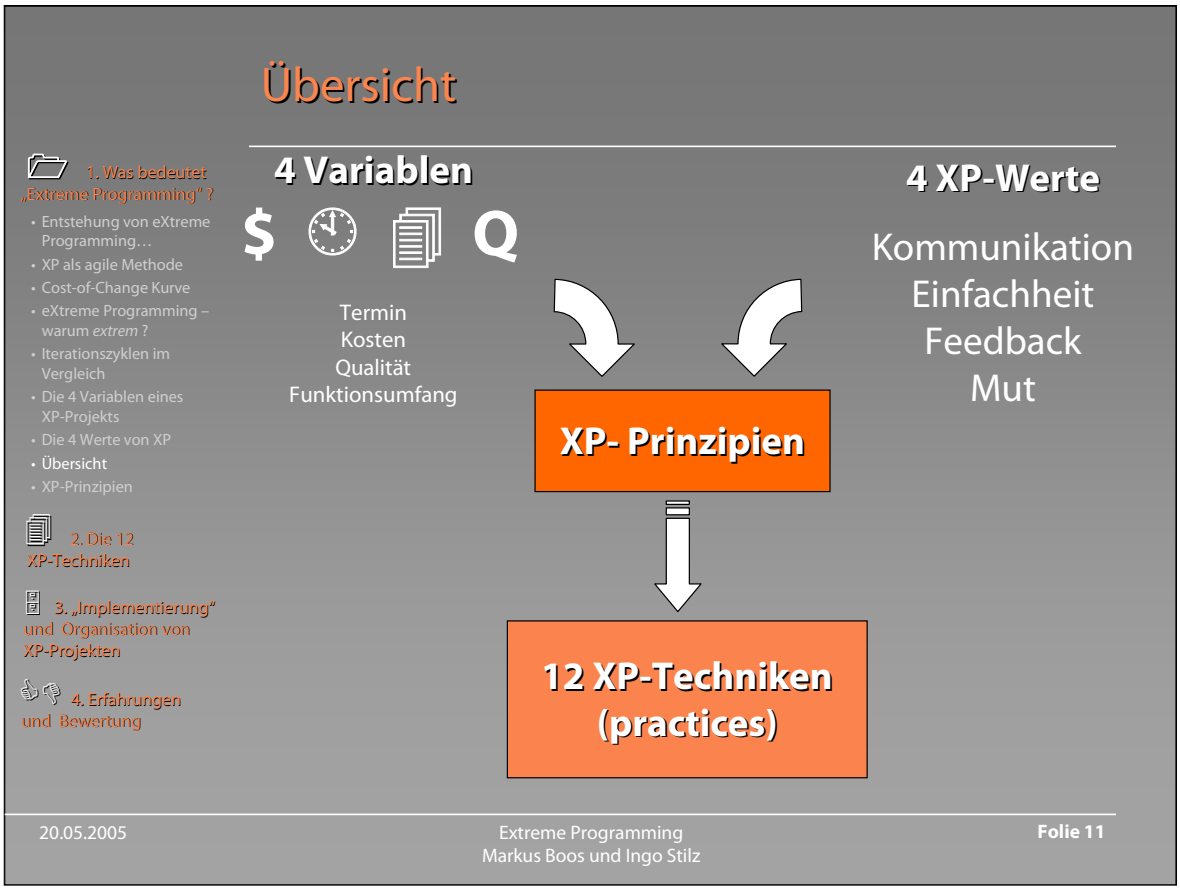
Feedback

→ schnelles Feedback → Qualitätssicherung

Mut

→ einfache Lösungen erfordern Mut

→ Kommunikation & Feedback erfodert Mut, denn sie können leicht als destruktive Kritik aufgefasst werden



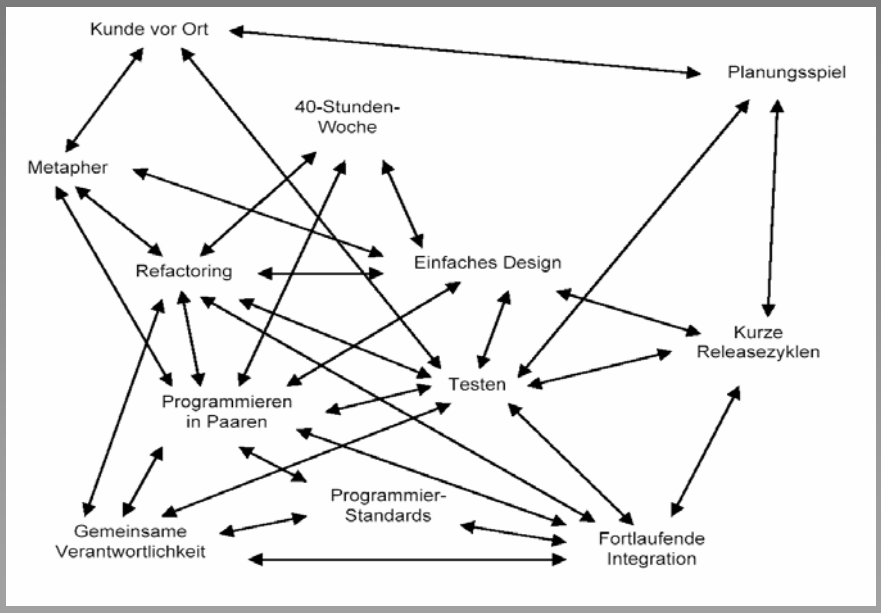
Aus den XP-Werten und den 4 Projekt-Variablen sind die XP-Prinzipien abgeleitet. Die praktische Umsetzung dieser Prinzipien findet durch die Anwendung der 12 konkreten XP-Techniken statt.

XP-Prinzipien

- unmittelbares Feedback
- Einfachheit anstreben
- inkrementelle Veränderungen
- Änderungen willkommen heißen
- Qualitätsarbeit leisten
- Lernen lehren
- kleine Anfangsinvestitionen
- auf Sieg spielen
- gezielte Experimente
- offene, ehrliche Kommunikation
- Verantwortung übernehmen
- an örtliche Gegebenheiten anpassen
- mit leichtem Gepäck reisen
- ehrliches Messen
- die Instinkte des Teams nutzen
- nicht dagegen arbeiten

2. Die 12 XP-Techniken

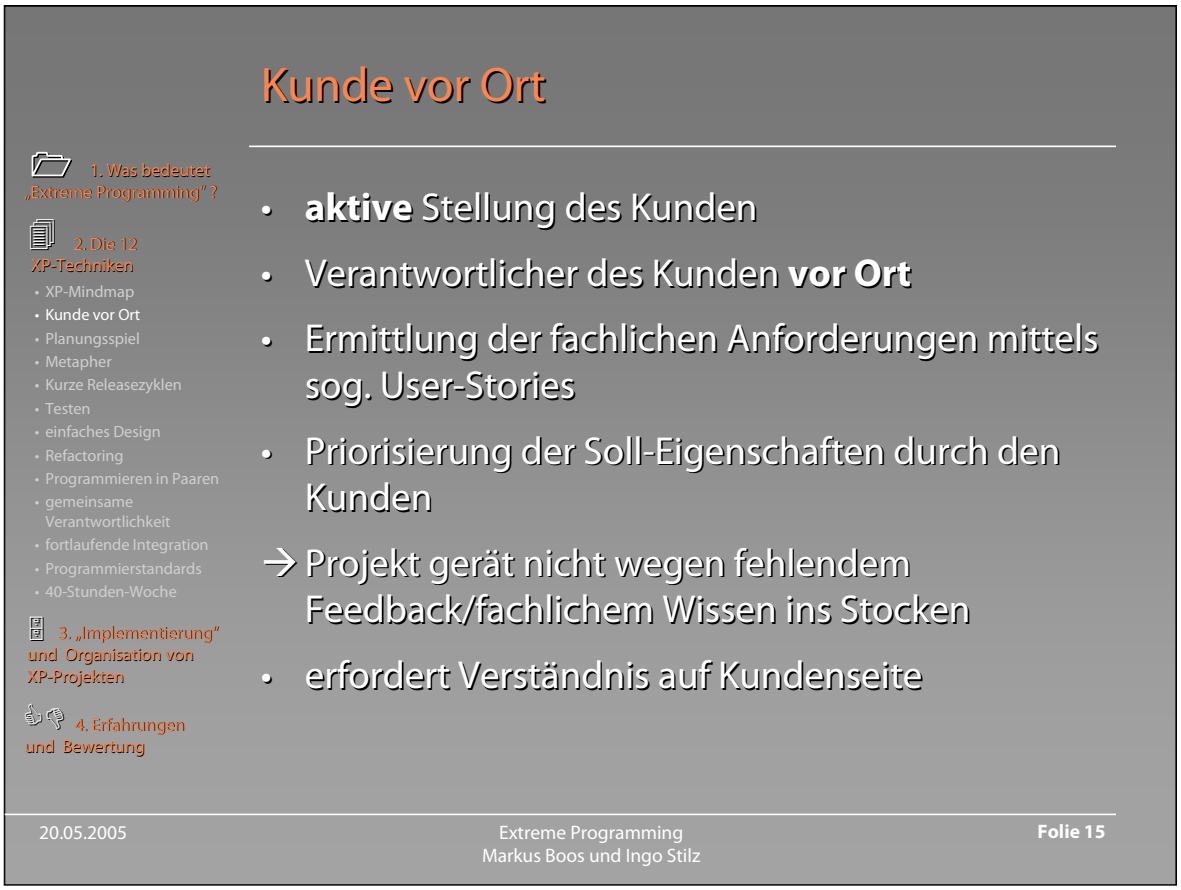
Übersicht



20.05.2005 Extreme Programming Markus Boos und Ingo Stilz **Folie 14**

Die 12 XP-Techniken sind eng miteinander verwoben und beeinflussen sich gegenseitig. Die Schwäche einer Technik wird durch die Stärke einer anderen wieder kompensiert. Dadurch ergibt sich durch Anwendung der Einzel-Praktiken eine sinnvolle und robuste Gesamtmethode.

Beispiel: Pair Programming beansprucht die Entwickler sehr stark und verlangt von ihnen höhere Konzentration über einen längeren Zeitraum hinweg. Dies führt möglicherweise dazu, dass Entwickler nach der Arbeit ausgelaugter sind, als wenn sie alleine Pausen nach Bedarf einlegen und sich zwischendurch erholen können. Dies wird kompensiert durch die 40-Stunden-Woche, in der den Entwicklern bewusst genügend Zeitraum zur Erholung gegeben wird, damit sie produktiv bleiben können.



Die Verfügbarkeit des Kunden/Anwenders ist oft entscheidend für Projekterfolg und sollte unbedingt beachtet werden. Ist eine 100-prozentige Vor-Ort-Verfügbarkeit nicht erreichbar, so sollte der Kontakt zum Kunden unmittelbar und direkt sein. Es sollte Erreichbarkeit gewährleistet sein, damit Rückfragen immer sofort beantwortet werden können.

Schwierigkeiten:

- „AUSFALL“ des Mitarbeiters auf Kundenseite unerwünscht

- es gibt selten Verantwortliche, die die Bedürfnisse von unterschiedlichen Anwendergruppen kennen (z.B. Lagerarbeiter und Management) → Erarbeitung der User-Stories mit verschiedenen Anwendern auf mehreren Ebenen

- Verantwortliche, die über Jahre hinweg nicht mehr ihrer ursprünglichen Berufspraxis nachgehen, lassen Fachwissen fehlen

- Fachwissen des Kundenmitarbeiters lässt mit der Zeit nach, denkt eher in IT-Kategorien anstatt fachlich

→der Kunde ist nicht immer der Anwender ! bei Diskrepanzen zwischen den beiden Parteien müssen die Entwickler als Moderatoren wirken

→**Austausch/Wechsel** der Mitarbeiter vor Ort bei längeren Projekten

Bei Standardprodukten ohne direkten Kunden muss entweder von einem Teammitglied die Rolle des Entwicklers übernommen werden oder es müssen Mitarbeiter mit entsprechendem Fachwissen „hinzugekauft“ werden.

Planungsspiel („planning game“)

- Entwickler und Kunde planen zusammen das nächste Release/die nächste Iteration
- welche Story-Cards und Task Cards werden als nächstes umgesetzt ?
- Priorisierung der umzusetzenden Anforderungen durch Kunden
- Schätzung des Realisierungsaufwands durch Entwickler (Achtung, selten gilt: geschätzter Aufwand = tatsächlicher Aufwand)
- Metrik: *Load factor* = $\frac{\text{Realzeit}}{\text{Idealzeit}}$
- Load factor · geschätzter Aufwand = Entwicklungsdauer für die nächste Iteration

- Story Cards: Titel, Datum, Autor, Art der Karte (fachliche Anforderung, technisches Refactoring, Bug), Beschreibung der Anforderung, geschätzter Aufwand, tatsächlicher Aufwand, Entwickler, Datum der Erledigung etc.

- Karteikarten haben sich als gutes System zur Aufgabenverteilung bewährt

- Task Cards dienen zum „Herunterbrechen“ / Parallelisieren der Story-Card-Aufgaben (hier sind auch Spezifizierung von technischen Anforderungen möglich)

- zum tatsächlichen Aufwand gehören auch: Besprechungen, Telefonanrufe, Supportanfragen usw., diese müssen beim Schätzen des Entwicklungsaufwands miteinkalkuliert werden

- Beispiel zu Load Factor: 3 Tage Gesamtaufwand für Umsetzung einer Story Card (inkl. Meetings usw.), 1 Tag tatsächlicher Entwicklungs-Aufwand → Load Factor 3

- Ziel: Load Factor verringern, gute Teams schaffen bis zu 1,2

- Dauer der Iteration richtet sich nicht nach Anforderungen, sondern die umzusetzenden Anforderungen werden nur in die nächste Iteration aufgenommen, wenn diese in den festgesetzten Rahmen passt (z.B. 2 Wochen)

- abstrakte und relative Einheiten zur Messung des Aufwands können die Schätzgenauigkeit verbessern

Der Begriff „Planungs-Meeting“ ist dem Kunden gegenüber als Bezeichnung für den Vorgang zu bevorzugen, da es sich nicht um ein Spiel handelt, bei dem Zeit vertrödelt wird.

Metapher

- bildhafte Darstellung von abstrakten Zusammenhängen / Vorgängen / Konzepten
- gemeinsame Idee und Zielvorstellung
- Basis für Architektur
- enger Bezug zu Entwurfsmustern
- handlungsleitend für Entwickler beim Treffen von Entscheidungen
- kurze Einarbeitungszeit für Anwender
- Sorgfältige Auswahl der Metapher ist wichtig, sie bestimmt die Zielvorgabe und sollte gut auf das Problem passen.

Beispiele für Metaphern: Kalkulationsblatt, Pinsel, Radiererfunktion, Papierkorb

Die Strukturen der technischen Objekte ähneln in der OO-Programmierung oft denen der entliehenen Metapher → aus Metaphern lässt sich eine Basis für die Architektur ableiten. Metaphern dienen dazu, eine gemeinsame Idee *der Entwickler untereinander* als auch *zwischen Kunde und Entwickler* zu etablieren.

→ Die sorgfältige Auswahl der Metapher ist wichtig, sie bestimmt die Zielvorgabe und sollte gut auf das Problem passen. Wenn keine 100%-ig passende Metapher gefunden werden kann, sollte sie besser weggelassen werden.

Kurze Releasezyklen

- Ziel: Software-Releases in kurzen Abständen
- Releases (< 3 Monate) und Iterationen (1-4 Wochen)

→ weniger Änderungen pro Release, trotzdem erkennbarer Fortschritt für Anwender

- voll einsetzbare (Teil-)Systeme
- Ziele:
  - schnelles Feedback
  - Erkennung von Fehlentwicklungen
  - → Risiko-Minimierung (kein Big-Bang)
- Planungsspiel nach jeder Iteration
- „Spike Solutions“

- Modul-/Klassentests und technische Tools vereinfachen die Release-Erstellung

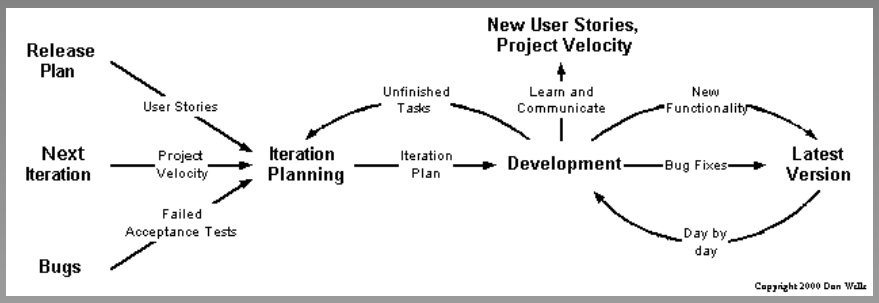
- kurze Releasezyklen anfangs oder bei bestimmten Projekten nicht immer zu verwirklichen (wg. Einarbeitungsaufwand). Dazu dienen gezielte Spike Solutions zum Ausloten potentiell schwieriger Felder oder technischer Probleme.

- keine BigBang-Ablösung eines antiquierten Legacy-Systems. Möglichkeit der Kapselung und Ablösung in kleinen Schritten → hier ist Kreativität gefragt !

- durch fortlaufende Integration entfällt eine lang andauernde Integrationsphase am Ende der Entwicklung

→ schnelles Feedback und Abgleich mit den Anforderungen des Kunden

Iterationsplanung im Überblick



Testen

- „testgetriebene Entwicklung“
- 1. Komponententests
  - häufiges Ausführen → schnelle Rückkopplung über Fehler
  - Entwürfe tendenziell einfacher
  - häufig aussagekräftigere Bezeichner für Operationen
  - sehr gute Automatisierungsmöglichkeiten (z.B. JUnit)
  - alle Entwickler müssen Tests schreiben (siehe „Broken Window Theory“)
- 2. Akzeptanztests (= Funktionstests)
  - wird das System vom Anwender akzeptiert ?
  - wenn möglich vom Anwender selbst „formuliert“
  - auch Indikator für den Projektfortschritt

- Entwürfe sind tendenziell einfacher, da komplexe Klassen auch schwierig zu testen sind.

- Es werden sinnvollere Bezeichner verteilt, da die Sicht zuerst aus der „nutzenden Klasse“ (=Testklasse) betrachtet wird.

- pro Klasse eine Testklasse

- Eine Klasse wird nur in gemeinsamen Bestand des Projekts aufgenommen, wenn alle Tests erfolgreich ablaufen.

- Eine Studie zeigt: häufiges Durchlaufen der Tests löst Erfolgserlebnis beim Entwickler aus → positiver Effekt durch das Testen

- kontinuierliches Testen von jedem Entwickler notwendig – Zeitdruck oder „Untestbarkeit“ sind höchstens schlechte Ausreden

- Broken Window Theory basiert auf Feldversuchen in USA. Es wurden Autos mit offener Motorhaube abgestellt → nach wenigen Minuten waren Reifen geplündert. Bei einem anderen Auto wurde gezielt eine Scheibe eingeschlagen → Scheiben anderer Autos wurden auch eingeschlagen, Reifen geklaut usw. → deshalb werden Graffitis in NY sofort entfernt, damit sich keine anderen „dazugesellen“ → Auf das Testen bezogen, erhöht sich der Aufwand ungemein, wieder vollständige Testabdeckung zu erreichen, wenn einmal für einen bestimmten Zeitraum keine Tests geschrieben wurden. Deshalb muss immer konsequent getestet werden. Schlampigkeit eines Mitarbeiters zieht oft Schlampigkeit anderer nach sich.

- Akzeptanztests können von versierten Kunden bzw. bei speziellen Projekten vom Anwender selbst durchgeführt werden, der Normalfall ist jedoch, dass die vom Kunden spezifizierten Test-Fälle vom Entwickler getestet werden.

Einfaches Design

- einfachstes, funktionierendes Design
- ≠ einfach darauflosprogrammieren
- Design begleitet Entwicklung, Revision jederzeit möglich

→ zu jedem Zeitpunkt optimales Design

- Designs beschreiben kleine Ausschnitte oder spezifische Probleme
- durch schnelle Umsetzung werden Fehler im Design frühzeitig erkannt
- 4 Richtlinien von Kent Beck
- kein Design und Implementierung auf Vorrat
- Kent Beck: „do not make two designs in a row“

Bei XP hat der Begriff „Architektur“ einen schlechten Beigeschmack und wird oft mit „kompliziertem Design“ und „Technologie auf Vorrat“ in Verbindung gebracht. Design ist bei XP vergleichbar mit Lagerhaltung in der Warenwirtschaft (Lagerzeiten kurz halten). Designs sollten just-in-time sehr kurz auf Vorrat gemacht werden, damit die „Lagerkosten“ (= erhöhter Implementierungsaufwand durch noch nicht benötigte Funktionalitäten) nicht unnötig steigen. Auf die Softwareentwicklung bezogen bedeutet dies: die Ware (=Design) wird im Optimalfall direkt nach der Produktion verkauft (= implementiert) → kein „Big Upfront Design“

Falsch ist nach XP-Theorie folgender Ansatz: „Das ist jetzt nicht für diese Story-Card wichtig, aber das brauchen wir ja sowieso bald. Und wenn wir jetzt schonmal dabei sind…“ → Es ist nicht egal, **wann** eine Eigenschaft realisiert wird ! Es werden Ressourcen zum falschen Zeitpunkt gebunden ! Eigenschaft wird bei Praxistests nicht erfasst und Implementierung kann in Zukunft womöglich einfacher und eleganter realisiert werden → **weniger Aufwand, wenn man den Zeitpunkt so weit wie möglich in die Zukunft schiebt.**

Der Einwand für zu wenig Vorausschau / Vorausplanung ist natürlich berechtigt, die Etablierung eines Gesamtkonzepts ist auch bei XP notwendig (grobe Makro-Architektur)→ hier ist Einsatz von Metaphern sehr sinnvoll. Die Architektur muss natürlich angesprochen werden, da Entwickler implizit schon eine Architekturvorstellung im Hinterkopf besitzen → wenn diese sich nicht decken, kann dies zu Schwierigkeiten führen. XP-Trainer können bei der Einführung einfacher Designs bei unerfahrenen Entwicklern helfen.

Refactoring und kontinuierliche Design-Verbesserungen machen aus „zusammengestückelten“ Einzel-Systemen und -Designs ein performantes Gesamtsystem (getreu dem Motto von Kent Beck: „make it run, make it right, make it fast“).

-es wird kein finales Design am Anfang erstellt sondern ein Design

-diese Designs beschreiben auf keinen Fall ein komplettes System, sondern immer kleine Ausschnitte oder spezifische Probleme, welche leicht überschaubar sind. Sie sollten ein kleines Problem gut lösen und schnell umsetzen können.

-Bei schnellem Einsatz eines Entwurfs können frühzeitig Erfahrungen damit gemacht und Fehler erkannt werden.

-4 Richtlinien von Kent Beck (Reihenfolge der Aufzählung impliziert eine Priorisierung):

-Das System (Source Code und Testklassen) sollte all das verständlich ausdrücken, was in ihm enthalten sein sollte.

-Das System sollte keinen duplizierten Code enthalten („Once and only once“-Prinzip)

-Das System sollte so wenig Klassen wie möglich besitzen.

-Das System sollte so wenig Methoden wie möglich enthalten.

-Zur Design-Erstellung dienen sog. Quick Design Session-Runden (schnell, kurz, auf 1 Whiteboard beschränkt) mit wenigen Entwicklern → eher ein Design kurz implementieren und ausprobieren als in Sackgassen zu laufen

-Kent Beck: „do not make two designs in a row“: erst nach der Implementierung der ersten Entscheidung wird weiterdiskutiert

Refactoring

- Umstrukturierung unter Beibehaltung der Funktionalität
- Verbesserung der Struktur zur Minimierung der laufenden Änderungen
- Umgestaltung nach den folgenden Gesichtspunkten:
  - Lesbarkeit
  - Übersichtlichkeit
  - Verständlichkeit
  - Erweiterbarkeit
  - Vermeidung von Redundanz
  - Testbarkeit
- Mögliche Refakturierungen
  - Änderung eines Symbolnamens
  - Verschieben eines Symbols in ein anderes Modul

Alle kennen das Prinzip „Never change a running System". Eine Änderung kann zu ganz unerwünschten Effekten an vielen Stellen führen und viel Arbeit und Arger mit sich bringen. Besonders Änderungen die rein von interner Struktur sind, werden prinzipiell nicht ausgeführt. Die Folge ist erhöhter Aufwand in der Wartung und es stehen keine Ressourcen mehr für die Neuentwicklung zur Verfügung.

Hier stellt XP das Refactoring entgegen. Unter Refactoring versteht man die Umstrukturierung eines Softwaresystems unter Beibehaltung der Funktionalität. Ziel ist die Verbesserung der Struktur um die Aufwandskurve im Projekt niedrig zu halten. Voraussetzung hierfür ist die Verwendung von Komponententests um unerwünschte Seiteneffekten vorzubeugen. Große Refactorings sollten im Möglichst viele kleine zerlegt werden. Gewünschte Änderungen in viele kleine Schritte zerlegen.

Vorsicht: Refactoring bringt dem Kunden keine weitere Funktionalität oder einen direkten Gewinn, sondern Verbessert nur die interne Struktur eines Programms.

Nur so viele Refactorings durchführen, dass sich die nächste Anwendungsanforderung leicht umgesetzt werden kann. Aber Refactorings sollten auch nicht aufgeschoben werden um nicht später Probleme zu bekommen.

<u>Empfehlungen:</u>

-Never believe in „Never change a running system"

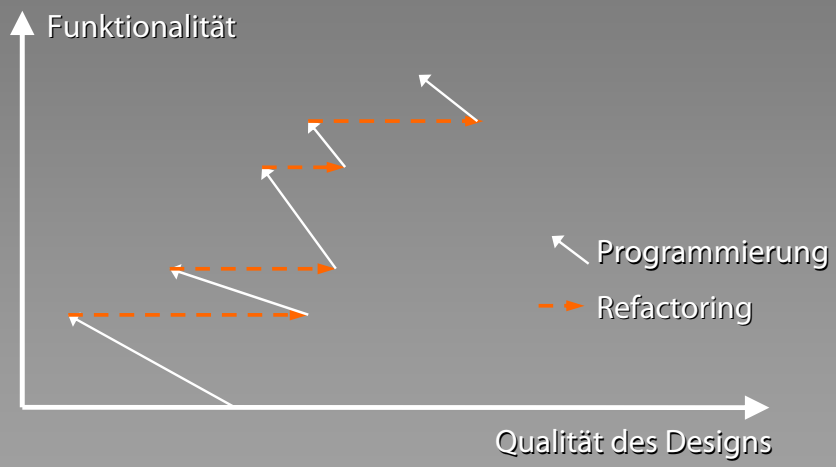
-Refaktorisierung muss zur Grundlage werden

-Quelltextverwaltungssystem mit optimistischen Sperrmechanismus

-Größere Änderungen in möglichst kleine Refactorings zerlegen

-Refactorings rechtzeitig für größere Änderungen planen

Refactoring



Der Zyklus von Refactoring und Implementierung erhöht Funktionalität und stabilisiert die Designqualität.

Programmieren in Paaren

- 4-Augen-Prinzip
- keinerlei Aufgabenteilung oder Hierarchie
- Tastaturwechsel klar vorgegeben
- Programmierwissen, guten Programmierstil und Erfahrungen vermitteln
- höhere Netto Programmierzeit
- Minimierung des Truck-Faktors
- Arbeit macht mehr Spaß, wird effektiver

→ höhere Qualität

Pair-Programming bedeutet paarweise Programmieren an einem Rechner. Es existiert keinerlei Aufgabenteilung und beide Entwickler sind absolut gleichberechtigt. Alles wird zusammen erarbeitet und implementiert. (4 Augen Prinzip). Dadurch ist die Kommunikation direkt bei der Entwicklung möglich. Genau wie Reviews auf Code und Designebene. Da die Tastatur nicht von beiden Leuten gleichzeitig genutzt werden kann, sind hier 2 Rollen zu vergeben. Währen der Entwickler seinen Code eintippt und erläutert warum und wieso er was macht, kann sein Partner über diese Lösung, eine Vereinfachung und Testfälle nachdenken.

Das Arbeiten macht so den Entwicklern mehr Spaß, die Qualität der erzeugenden Produkte steigt merklich an und die Arbeit wird effektiver.

Durch XP Technik wird auch der Ausfall eines Teammitglieds deutlich minimiert (Truck Faktor), da nicht nur eine spezielle Person über ein Aufgabengebiet die nötigen Informationen und Wissen verfügt. So konzentriert sich das Spezialwissen nicht auf eine einzige Person. Auch kann so Wissen innerhalb des Unternehmens besser verbreitet werden.

Programmieren in Paaren setzte aber auch einen ständigen Wechsle der einzelnen Paare voraus! Der Tastaturwechsel sollte klar vorgegeben werden, sonst gibt der erfahrene Entwickler die Tastatur nur ungern an einen Beginner ab. Der Wechsel sollte auch ohne Änderung der Sitzposition gut möglich sein (Gute Anordnung der Tische – am besten runde Tische)

Durch das Paarweise Programmieren wird auch die Netto-Programmierzeit deutlich erhöht und somit die Produktivität und die Konzentration auf die anstehende Arbeit. Das zwischendurch kurz surfen, privat telefonieren, beantworten von Emails oder sonstige Ablenkungen reduzieren sich auf ein Minimum. Das Programmieren in Paaren ist auch optimal dazu geeignet um praktische Programmierwissen, guten Programmierstil und Erfahrungen zu vermitteln. Dadurch gibt es eine deutlich steilere Lernkurve! Auch werden Programmierfehler stark reduziert.

Durch die ständige Kommunikation innerhalb eines Paares sollte der Raum nicht all zu klein sein und sollten auch alle Entwickler eines Teams in einem Raum sitzen. So können Missverständnisse schnell aufgeklärt werden.

Die Teams müssen gut harmonieren, da Kritik von vielen Leuten oft nicht so leicht angenommen wird. So muss sich das Team am Besten als ein Ganezes verstehen und eine gemeinsame Aufgabe lösen.

<u>Empfehlungen:</u>

-Programmieren in Paaren muss erlernt werden

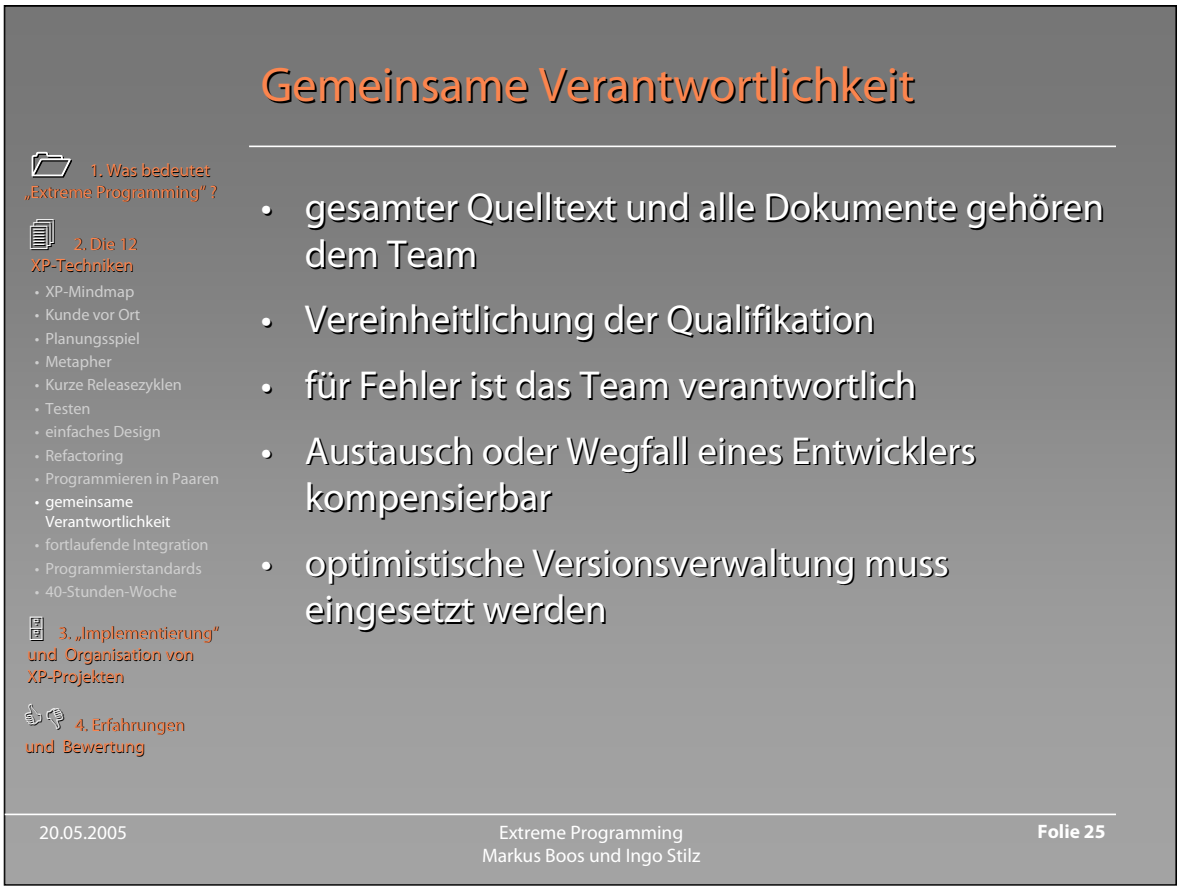
-Feedback geben

-Mehrere Pausen machen, da viel anstrengender

-Schneller PC und großer Monitor

-Beinfreiheit und arbeiten auf gleicher Höhe

-Rollenwechsel muss einfach möglich sein



In einem XP-Projekt gehören der gesamte Quelltext und alle Dokumente dem ganzen Team. Das bedeutet dass jeder zu jedem Zeitpunkt den Quelltext oder Dokumente abändern kann, was im Wesentlichen aber einen ähnlichen Wissensstand voraussetzt. Hieraus ergibt sich die Forderung dass alle Mitglieder auf den gleichen Qualifikationsstand gehoben werden.

Gemeinsame Verantwortlichkeit heißt hier dann dass das ganze Team für Fehler verantwortlich ist und nicht eine zum Sündenbock gestempelt wird. So sollte der jenige der zuerst Kenntnis vom Problem hat, dieses beseitigen. Fühlt sich niemand verantwortlich, werden Probleme und Fehler erst gar nicht behoben. Eine Rückverfolgung zum Verursacher kostet zu viel Zeit, ergäbe keinen Sinn und wäre in den meisten Fällen gar nicht mehr möglich.

Auch ist auch ein Austausch oder Wegfall eines Entwicklers in einem laufenden Projekt durch die gemeinsame Codeverwaltung innerhalb eines Teams möglich ohne viel Kostbare Zeit zu verlieren.

Die räumliche Situation spielt hier auch eine entscheidende Rolle, je weiter die Teams auseinander sitzen desto weniger Kommunikation wird zwischen den Gruppen stattfinden.

Nach Möglichkeit sollte auch ein Entwickler nicht an mehreren Projekten gleichzeitig arbeiten.

Die die Umsetzung der gemeinsamen Verantwortlichkeit ist eine Versionsverwaltung noch wichtiger als in konventionellen Entwicklungsumgebungen.

Die einfachste Lösung ist oft die beste.

<u>Empfehlungen:</u>

-Vereinheitlichung der Qualifikation

-Bei unterschiedlichem Wissensstand gemischte Teams einsetzen

-Möglichst räumliche Nähe innerhalb eines Teams

-Versionsverwaltung muss unbedingt eingesetzt werden

Fortlaufende Integration

- jeder kann am Gesamtsystem Änderungen vornehmen
- Verwendung eines Integrationsservers
  - stellt immer lauffähige Version zur Verfügung
  - Übernahme nur bei erfolgreichen Tests
  - Änderungen immer möglichst klein halten
- sofortiges Feedback
- große Änderungen werden auf ein Minimum reduziert
- Motivation innerhalb eines Teams
- Einhaltung der Konventionen

Alle Entwickler die an einem Softwareprojekt beteilt sind, haben das gleiche Recht am Gesamtsystem Änderungen vorzunehmen, wenn er es für sinnvoll hält. Um dies zu unterstützen, müssen die Änderungen den beteiligten schnell zugänglich gemacht werden. XP beschreibt hierzu die Technik der fortlaufenden Integration.

Die Änderungen einer Gruppe werden zuerst auf einen Integrationsrechner geladen, der einzig und alleine für die Integration von neuen Quellen verantwortlich ist. Spielt ein Team von Entwicklern eine neue Quelle ein, so muss auf dem Integrationsrechner das Gesamtsystem mittels Komponententests geprüft werden. Nur wenn alle Tests erfolgreich durchlaufen wurden ist die Integration erfolgreich und der Code darf auf dem Rechner belassen werden. Wenn die Tests nicht erfolgreich waren muss das Team den alten Zustand wieder herstellen oder den eigenen Code dazu bringen alle Tests erfolgreich zu absolvieren. Erst dann ist die Arbeit am Integrationsrechner vollendet. Der Integrationsrechner stellt damit sicher dass immer eine lauffähige und getestete Version auf dem Rechner vorhanden ist. Änderungen sollten nach Möglichkeit immer klein gehalten werden. Fortlaufende Integration löst somit den „nigthly build“ ab. Entwickler bekommen somit auch ein sofortiges Feedback vom Gesamtsystem.

Als Leitlinie in der XP-Gemeinde hat sich eine Integration pro Paar pro Tag als Minimum herausgebildet. Durch die laufenden Integrationen ist das Verwenden eines Versionsmanagers unabdingbar. Denn die Frage wann ein Quelltext bearbeitet werden kann, ist hier von entscheidender Bedeutung. Für kleine Änderungen müssen alle Quellen immer zugreifbar und änderbar sein. Somit eignet sich die Verwendung von optimistischen Sperrmechanismen am besten. Bei kleinen Änderungen von 2 Paaren an der gleichen Quelle, ist die Zusammenführung, dann auch nicht besonders aufwendig. Somit führt die fortlaufende Integration auch zu kleinen Refactorings, was als sich als sehr positiver Effekt herausgestellt hat und große problematische Änderungen auf ein Minimum reduziert. Je schneller ein Paar somit seine Änderungen auf dem Integrationsrechner einspielt und in einem lauffähigen Zustand bringt, desto weniger müssen aufwendige Zusammenführungen von 2 verschiedenen Paaren durchgeführt werden. Das motiviert die unterschiedlichen Paare vor allem wenn an den gleichen Quellen gearbeitet wird. Hier entsteht am Integrationsrechner ein natürlicher Flaschenhals. Somit sollte eine durchschnittliche Integration nicht länger als 10-20min dauern. Bei größeren Projekten kann dies aber zu Problemen führen und somit ist der physikalische Integrationsrechner nicht mehr praktikabel und es sollte auf deine Server/Client – Struktur umgestellt werden. Nur wenn alle Konventionen erfüllt sind, darf man eine Integration ausführen, wobei auch hier nur eine Integration gleichzeitig ausgeführt werden kann.

Auch wenn kein physikalischer Integrationsrechner mehr genutzt wird, sollte zum Beispiel alle Daten auf einen Referenzrechner überspielt werden (z.B. jeden Montag). Hier kann dann insbesondere das Zusammenspiel mit den verschiedenen Systemkomponenten wie Datenbank, Applikationsserver und Webserver getestet werden. Somit findet hier einregelmäßiger Gesamttest statt.

Empfehlungen:

-Überlegung ob sich ein physikalischer Integrationsrechner lohnt oder ein virtueller Integrationsrechner mit Versionsverwaltung reicht?

-Konventionen aufstellen für die Integration (z.B. nur dann wenn alle Tests auf dem eigenen Rechner erfolgreich waren)

Programmierstandards

- Äußere Form des Quelltextes wird festgelegt
- standardisierte Darstellung / Richtlinien
- Einheitliche Programmierumbebung
- Nur so kann auch eine gemeinsame Verantwortlichkeit praktiziert werden
- Programmierstandards dürfen nicht zu lang werden
- Tools zur Erleichterung einsetzen

Legen die Äußere Form des Quelltextes fest (Klammer, Kommentare, Leerzeichen, Werkzeuge). Dadurch erhält man eine standardisierte Darstellung / Richtlinien innerhalb eines Teams und man muss sich nicht verschiedene Layouts einprägen. Einheitliche Programmierumbebung sehr entscheidend für das paarweise Programmieren um sich im gesamten Quelltext schnell zurechtzufinden, sonst ist die Arbeit unangenehm und Änderungen fallen schwerer. Nur so kann auch eine gemeinsame Verantwortlichkeit in einem Team gegeben sein. Ein Beispiel hierfür kann bei SUN für die Programmierrichtlinien bezogen werden. Diese können innerhalb des Teams in gemeinsamer Absprache auch modifiziert und erweitert werden. Stellt sich in einem Projekt eine Richtlinie als Umständlich heraus, darf man sich nicht scheuen diese zu Ändern, auch wenn viel Quelltext davon betroffen ist. Ansonsten wird diese wohl nicht von allen Entwicklern umgesetzt und es entsteht Wildwuchs.

Sollten noch keine Programmierstandards bestehen, sollte man nicht zu viel Zeit dafür investieren und endlosen Diskussionen um Kleinigkeiten aus dem Weg gehen.

Auch die Programmierumgebung sollte innerhalb eines Teams gleich gewählt und konfiguriert werden um den wechselnden Paaren eine Eingewöhnungs- oder Umstellungszeit zu ersparen.

<u>Empfehlungen:</u>

-Formulieren Sie die Programmierstandards im Imperativ

-Programmierstandards dürfen nicht zu lang werden

-Anreichern der Standards mit Beispielquelltext

-Mut bereits verwendete Standards zu ändern

-Tools suchen die die Umsetzung erleichtern oder helfen (z.B. JIdent)

40-Stunden-Woche

- Starke Verdichtung der Arbeitszeit durch
  - kurze Iterationszyklen
  - kontinuierliche Integration
- → Höhere Netto-Programmierzeit
- Einzuhaltende Regeln
  - Trennen von Programmierzeit, Verwaltungszeit und Pausen
  - Keine persönlichen Dinge während der Arbeitszeit
  - Konzentration bei der Arbeit
  - viele kurze Pausen, aber auf keinen Fall vor dem Rechner

Eine notwendige Diskussion warum nicht mehr als 40 Stunden gearbeitet werden sollte ist fast überflüssig, trotzdem sieht die Realität für viele Entwickler meist anders aus. Besonders Berater in Projekten die meist einen festen Termin haben, kennen diese Situation. Ständige Überstunden sind ein Indiz für schlechtes Management des Entwicklungsprozesses. Oft wird der Mangel an qualifizierten Softwarenetwicklern als Grund für Überstunden angegeben. Letzten Endes sind Überstunden aber kontraproduktiv, weil die dauernde Überbelastung der Entwickler zu Krankheit, Demotivation oder geringerer Leistungsfähigkeit münden. Daher nicht härter arbeiten sondern besser!

Kurz vor Abgabe eines Projektes kann es immer mal wieder zu so genannten heißen Phasen kommen, wo 2-3 Wochen Überstunden geleistet werden müssen. Diese sollten aber nicht länger werden und regelmäßig vorkommen. Zum Glück setzen heute die meisten Kunden mehr auf Qualität als auf einen festen Termin. Oft kommen auch Probleme auf wegen des Termins wenn der Projektverantwortliche gegenüber dem Kunden nicht zugeben möchte, dass das Projekt im Verzug ist. Daher ist auf ein gutes Verhältnis zum Kunden zu achten, da es innerhalb der Entwicklung immer zu unvorhersehbaren Problemen kommen kann. Aber kann das funktionieren?

-XP Projekte sind sehr zielgerichtet und erfolgsorientiert, machen also auch nicht vor einer 40h Woche halt.

-Kurze Auslieferzyklen kleinerer Systemversionen. Das bedeutet ständigen Termindruck. Reicht hier eine 40h aus?

-Durch die ständigen kurzen Iterationszyklen, die kontinuierliche Integration wird die Arbeitszeit extrem stark verdichtet. Somit wird deutlich mehr Arbeit verrichtet als in anderen üblichen Projekten. Damit aber diese Verdichtung funktioniert müssen ein paar zusätzliche Regeln eingehalten werden.

-Trennen von Programmierzeit, Verwaltungszeit und Pausen

-Keine persönlichen Dinge (Email, Aktien) während der Arbeitszeit

-Konzentration bei der Arbeit und keine Gespräche über sachfremde Dinge

-Machen sie viele kurze Pausen, aber auf keinen Fall vor dem Rechner

-Nicht mehr als eine Arbeitsstunde pro Tag für die Verwaltung!

-Die Bereitschaft für Mehrarbeit hängt danach ganz deutlich von der Unternehmenskultur ab. Mitarbeiter wollen befragt und um ihre Einwilligung gebeten werden.

<u>Empfehlungen:</u>

-Überblick über die geleisteten Mehrstunden verschaffen

-Kompensation für Überstunden anbieten

-Sollten die 8h an mehreren Tagen nicht reichen. Freitag nur halbtags?

-Keine Verplanung des Freitags

3. „Implementierung“ und Organisation von XP-Projekten

XP-relevante Artefakte

- Artefakte = Sammlung von Werkzeugen mit Funktion im Entwicklungsprozess
- **Story Cards (Umfang: 3-9 Paartage)**
  - fachliche Anforderungen
  - evtl. Akzeptanztests formuliert oder referenziert
  - Adaptionen: Interviews, Szenarios, Glossar, Projekttagebuch
- **Task-Cards (~ 2 Paartage)**
  - technische Anforderungen
  - Priorisierung durch Entwickler/Versuch der Zuordnung zu Story Cards
- **Release-Plan**
  - Termin und Funktionen des jeweils nächsten Release
  - Adaptionen: Projektetappen, Kernsystem mit Ausbaustufen
- **Iterations-Plan**
  - Zuordnung und Priorisierung von Story- und Taskcards
  - Adaption: Referenzlinien

Bei den Story Cards ist die physikalische Vergegenständlichung der Aufgaben nützlich:

- Ablage auf „erledigt" Stapel ergibt befreiendes Gefühl beim Entwickler
- Aufnahme der Karte vom Stapel = bildhaft für: neue Aufgabe annehmen
- Priorisierung der Karten mit der Reihenfolge, in der sie auf dem Stapel liegen
- nur ein Team kann jeweils dieselbe Karte besitzen, dadurch ist eine gleichzeitige Bearbeitung ausgeschlossen
- Notizen sind jederzeit möglich und erwünscht
- Projekteindruck und –fortschritt „greifbar"

Die **Interviews** und **Szenarios** gehen den Story Cards voraus, Szenarios werden von den Entwicklern geschrieben und dienen zur Verdeutlichung und zum Verstehen der fachlichen Vorgänge (sind allerdings nicht zwingend erforderlich), ähnlich USE-Cases.

Ein **Glossar** dient zur Vereinheitlichung der Projektsprache → Erklärung der Begriffe in ihrem Kontext, Ausschluss von Mehrdeutigkeiten, Grundlage für Dokumentation, Einsteigerhilfe

Das **Projekttagebuch** (nicht zwingend erforderlich) wird täglich von allen Mitarbeitern gepflegt (so kompakt wie möglich, so ausführlich wie nötig) und enthält: Ergebnisse von Untersuchungen, Projektfortschritt, Offene Probleme, Gelöste Probleme, Anweisungen von Vorgesetzten, Entscheidungen von Kunden

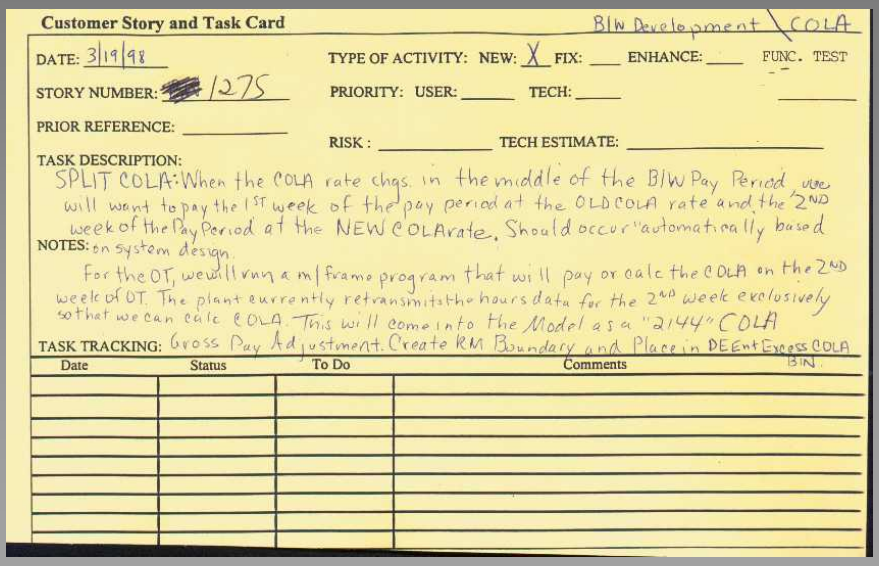
ein **Beispiel zu Task Cards**:
Story-Card zur Kundensuche nach Name, Kundennummer, Vertragsnummer, Anschrift:

Task-Card 1: Klasse Kundenverwaltung so erweitern, dass man Kunden suchen kann
Task-Card 2: Oberfläche für Kundenfinde entwickeln
Task-Card 3: Kundenfinder mit Oberfläche aus Task Card 2 und Kundenverwaltung aus TaskCard 1 entwickeln

Projektetappen definieren die groben Etappen des Projekts (1-3 Monate).

Referenzlinien: Feinplanung innerhalb der Projektetappen, keine Daten für Fertigstellung, sondern tabellarische Darstellung der Story- und Task-Cards, sie dienen als Stützräder während der Einführung von XP, da sie explizite Aufgaben zuteilen (bei XP jedoch nicht notwendig)

Story Card aus dem C3-Projekt



Customer Story and Task Card — B1W 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 B1W 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

NOTES: on system design.
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 DEEntExcess COLA BIN.

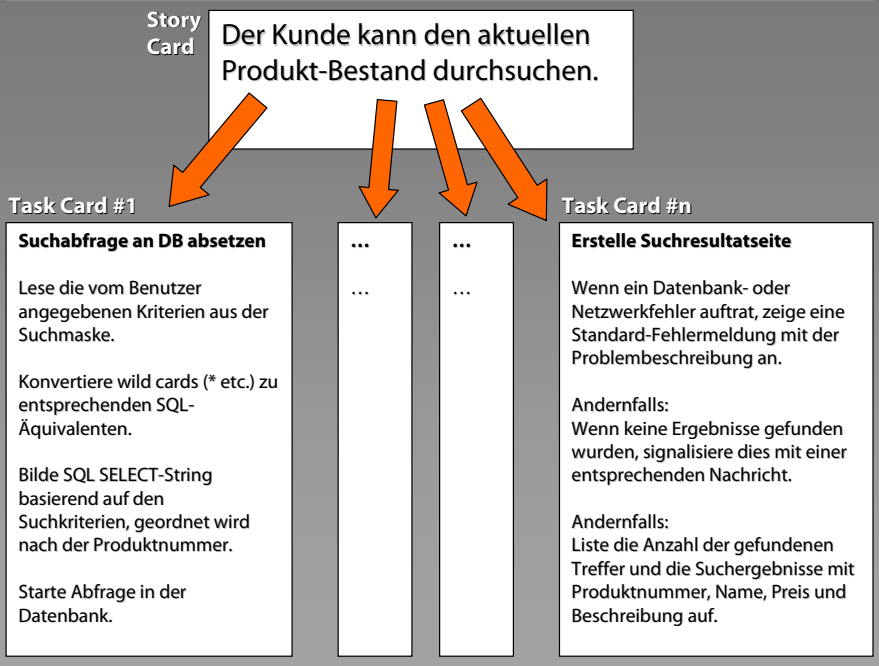
Date	Status	To Do	Comments

Story-Card Beispiel aus dem C3 Projekt von Daimler-Chrysler

→nur wenige der vorgesehenen Felder wurden effektiv genutzt

→wichtig sind lediglich die deutliche Beschreibung der fachlichen Anforderungen und Platz für Notizen, es gilt auch hier das Prinzip der Einfachheit

Task Cards



Die wichtigsten Rollen im XP-Projekt

- Der Kunde
  - bestimmt, was programmiert werden soll
  - legt Prioritäten fest
- Der Programmierer
  - übernimmt im XP Modell mehrere Rollen
  - koordiniert, entwirft, dokumentiert und testet das System
- Der Verfolger
  - verfolgt den Entwicklungsprozess und weist auf Probleme hin
  - sammelt Informationen, stellt Kennzahlen auf

Eine Person kann in einem XP-Projekt mehrere Rollen erfüllen. Genauso kann eine Rolle von mehreren Personen ausgefüllt werden.

Der Kunde:

bestimmt was programmiert werden soll. Oft ist zum Start des Projekts nicht genau klar was alles umgesetzt werden soll. Daher ist Softwareentwicklung ein Lernprozess für Entwickler und Kunden. Zunächst muss festgelegt werden welche Systemunterstützung überhaupt möglich und sinnvoll ist. Der Kunde definiert hieraus dann die Reihenfolge in der Entwickelt werden soll. Kunde sollte den Entwicklern jederzeit als Ansprechpartner zur Verfügung stehen. Nach XP sollte der Kunde die StoryCards schreiben. Hat er noch keine Erfahrung ist meinst nur ein Suboptimales Ergebnis zu erwarten. Die Entwickler haben dann die Aufgabe dem Kunden Feedback zu geben und dessen Lernprozess zu unterstützen.

Der Programmierer:

Der Programmierer übernimmt im XP Modell mehrere Rollen. Sie koordinieren, entwerfen, dokumentieren und testen das System. Vorgegangen wird nach den StoryCards des Kunden. Die benötigte Dokumentation des Entwurfs soll sich soweit wie möglich im Quelltext wieder finden. Dadurch wird verhindert dass Quelltext und Dokumentation auseinander laufen.

Die Kenntnisse umfassen:

Kenntnis aller im System verwendeten Technologien

Kommunikationsfähigkeit

Konfliktfähigkeit (mit Kritik richtig umgehen)

Der Tester:

In XP sind die Komponententests so stark mit der Programmierung verknüpft, dass diese die Tests selbstständig ausführen. Daher wird die Rolle des Testers in XP auf die für den Anwender relevanten Akzeptanztests gesetzt. Der Tester definiert den Akzeptanztest zusammen mit dem Kunden. In den meisten Projekten gibt es daher keinen einzelnen Tester, sondern der Entwickler übernimmt auch diesen Schritt, da Tester auch einen guten Einblick in das Projekt brauchen, da sonst die Gesamtfunktionalität vielleicht beim Testen nicht beachtet wird.

Der Verfolger:

Der Verfolger ist das Gewissen des Teams. Er verfolgt den Entwicklungsprozess und weißt auf Probleme hin. Hierzu sammelt der Verfolger Informationen während des Entwicklungsprozess durch Gespräche mit den Entwicklern und durch die Sammlung quantitiver Daten. Hier zählt das Prinzip: „Sammle keine Daten mit denen Du nichts anzufangen weist".

Typische Daten:

Budgetkontrolle (Verbrauchte reale Aufwände->Gesamt, diese, nächste Iteration)

Projektfortschritt

Effektivität (load factor - Programmierzeit zur Gesamtzeit)

für nicht fachliche Angelegenheiten, wie zum Beispiel Finanzierung des Projekts Die gesammelten Daten dienen der weiteren Projektsteuerung und werden für erforderliche Maßnahmen und zur Orientierung im Prozess verwendet. Der Verfolger sammelt nicht nur die Daten, sondern entscheidet auch welche Daten in welchem Abschnitt des Projektes relevant sind. Der Verfolger ist verantwortlich den Loadfaktor zu beobachten und neu zu ermitteln wenn dieser ansteigen sollte.

Anforderungsermittlung und Planung

- kontinuierliche Anforderungsermittlung über die gesamte Projektdauer hinweg
  - offene Interviews
  - Identifizierung der Stakeholder
  - Bestimmung der Verantwortlichen
  - Risikoabschätzung für Nichtbeachtung
  - Szenarios
  - Glossar
- Planung des Projekts damit ebenfalls zyklisch

Die Anforderungsermittlung ist keine Phase (es gibt weder Anfang noch Ende) sondern ist eine Aktivität, die immer wieder durchgeführt wird und ändert bereits existierende Ergebnisse (zyklische Vorgehensweise: es werden nur immer die wirklich benötigten Anforderungen ermittelt)

Offene Interviews:

-persönliche Gespräche mit den Anwendern (nach Möglichkeit Einzelinterviews)

-auch hier gilt wieder: Kunde ist nicht gleich Anwender → getrennte Befragung

-Manager neigen dazu, von idealisierten Arbeitsabläufen auszugehen oder sind zuweit von der Praxis entfernt

-Zwei Gesprächspartner auf Entwicklerseite (einer alleine wäre überfordert mit Fragenkatalog und Protokollführung, …)

-wenn möglich am Arbeitsplatz des Anwenders (zur Vergegenständlichung des Arbeitskontextes z.B. Antragsformular oder Beatmungsgerät in einer Klinik)

Zyklische Anforderungsermittlung (d.h. anfangs sind nicht alle Details bekannt) widerspricht der Planbarkeit eines Projekts → deshalb muss die Planung ebenfalls zyklisch betrieben werden

Zielgruppen der Planung:

- Auftraggeber: in der Regel nicht an Details der Planung interessiert; wichtig ist, dass das Projekt im finanziellen und zeitlichen Rahmen fertiggestellt wird.
- Anwender
- Programmierer

Vertragsgestaltung

- konventionelle Abwicklung:
  - Festpreisprojekt (Pflichtenheft)
  - nach Aufwand (monatliche Rechnungen)
- Problem: Anforderungen oft nicht vollständig und eindeutig formulierbar !
- Alternativen für XP-Projekte:
  - Vertrag auf Budgetbasis
  - Festpreis nach Vorprojekt
  - „Proviant und Prämie“

Die Anforderungen an nichttriviale Softwaresysteme sind oft nicht vollständig und eindeutig formulierbar.

→ Anforderungen werden erst während des Projekts geklärt bzw. ändern sich

→ missglückte Projekte enden daher oftmals vor Gericht

→ beide Parteien sind **Verlierer ! (≠ XP-Prinzip „play to win“)**

Bei der zyklische Anforderungsermittlung (XP) wird deutlich: Softwareentwicklungsprojekte sind nur schwer nach Festpreis abzurechnen. Es gibt deshalb 3 Alternativen der Vertragsgestaltung:

Vertrag auf Budgetbasis:

- Auftraggeber gibt ein Budget für Projekt vor
- ggf. Realisierung eines Prototyps mit Teil des Budgets

Festpreis nach Vorprojekt

- Vorprojekt / Prototyp vor Vertragsvergabe
- **Schätzrisiko** enorm für Projekte mit mehr als 200 Personentagen

„Proviant und Prämie“

- Mischform aus Festpreis- und Aufwandsprojekt
- Auftraggeber zahlt monatlich den Entwicklungsaufwand
- bei Ablieferung einer Komponente wird zusätzlich eine Prämie gezahlt (die Höhe wird dadurch bestimmt, ob sie vorzeitig, rechtzeitig oder verspätet abgeliefert wird)

Sprints

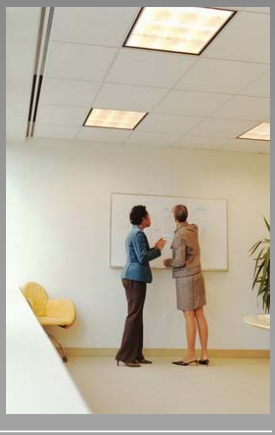
- kurze Zeiträume (max. 30 Tage)
- konzentrierte Entwicklung
- Organisation verhindert Störeinflüsse von außen
- für interne Störfaktoren (email, Aktienkurse, ebay, …) werden ebenfalls Regeln vereinbart
- bewusst mehr Pausen zur Erholung und Reflektion über die Arbeit

Von Entwicklern wird eigtl. erwartet, immer konzentriert zu arbeiten, jedoch ist dies häufig nicht der Fall. Gründe dafür sind: Urlaub, Fortbildungen, Anwender-Schulungen, Wartungsaufgaben, Fehlerkorrekturen usw.

Bei Sprints werden von der Organisation Störeinflüsse von außen auf das Entwicklerteam verhindert (z.B. eine Person übernimmt den Telefondienst). Wenn dies notwendig ist, kann sogar der Entwicklungsort gewechselt werden (Hotel, Seminarraum, andere Niederlassung) und für interne Störfaktoren (private und geschäftliche emails, Aktienkurse, ebay, …) werden ebenfalls Regeln vereinbart.

Stand-Up-Meetings

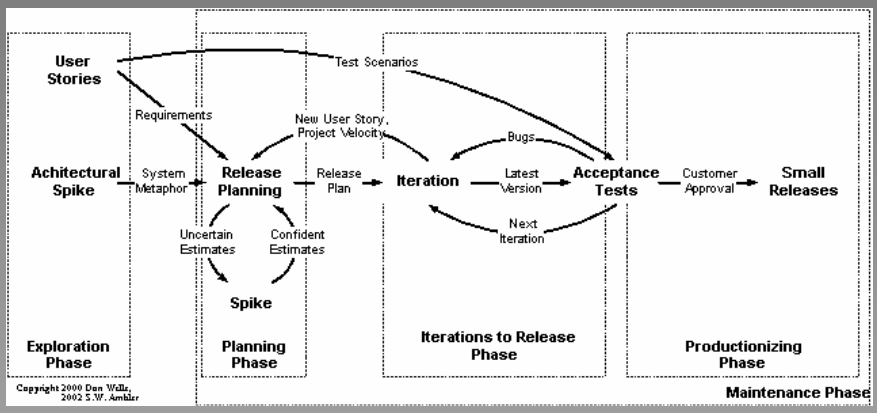
- gewöhnliche Wochenmeetings ufern oft aus
- Detaildiskussionen zwischen wenigen Beteiligten für die meisten irrelevant
- → tägliche Stand-Up-Meetings
  - festgelegte Zeit und Dauer (max. 15 Minuten)
  - „**Steh**ung“ anstatt „**Sitz**ung“
  - keine Diskussionen erlaubt, Rückfragen höchstens kurz zum Verständnis



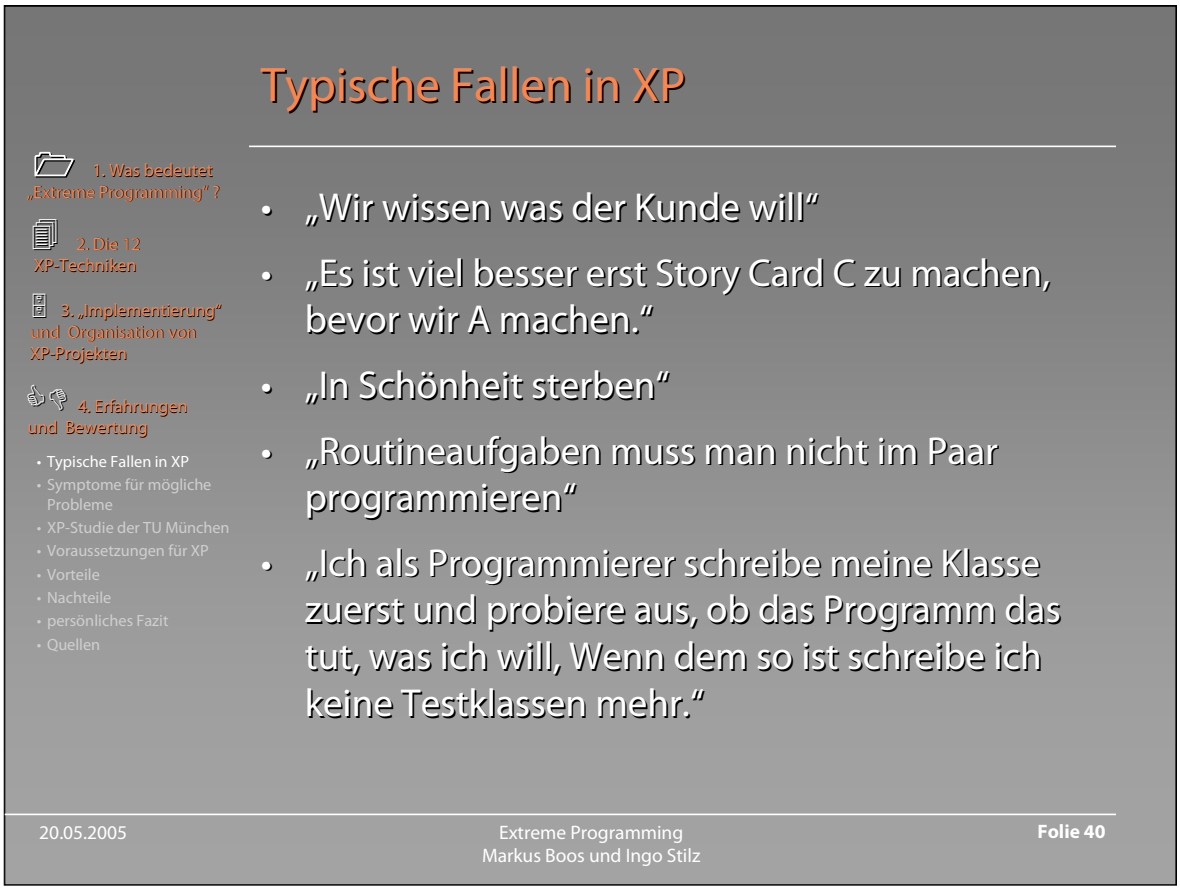
„**Steh**ung“ anstatt „**Sitz**ung“: es geht nicht um Kaffeetrinken und Kekseessen, sondern um Projektfortschritt und Wissensverbreitung

→ keine Diskussionen erlaubt, Rückfragen höchstens kurz zum Verständnis (Details werden danach in Einzelgesprächen behandelt)

XP Development Lifecycle im Überblick



4. Erfahrungen und Bewertung



Typische Fallen und wie man sie umgeht

Gerade in der Einführungsphase kann es zu Fehlern mit schwerwiegenden Folgen kommen. Nicht selten scheitert man alleine schon an der Umsetzung von XP.

Typische Fehler sind:

„Wir wissen was der Kunde will“, meist wird dann der Kontakt zum Kunden vernachlässigt

„Es ist viel besser erst Story Card C zu machen, bevor wir A machen, auch wenn der Kunde der andere Prioritäten setzt.“ Warum macht der Kunde dann überhaupt Priorisierungen? Kann das dass Team nicht besser einschätzen? NEIN! Bei einem Abbruch des Projekts sind so nicht die entscheidenten Funktionalitäten umgestzt und im schlimmsten Fall der der Kunde nichst mit de System anfangen.

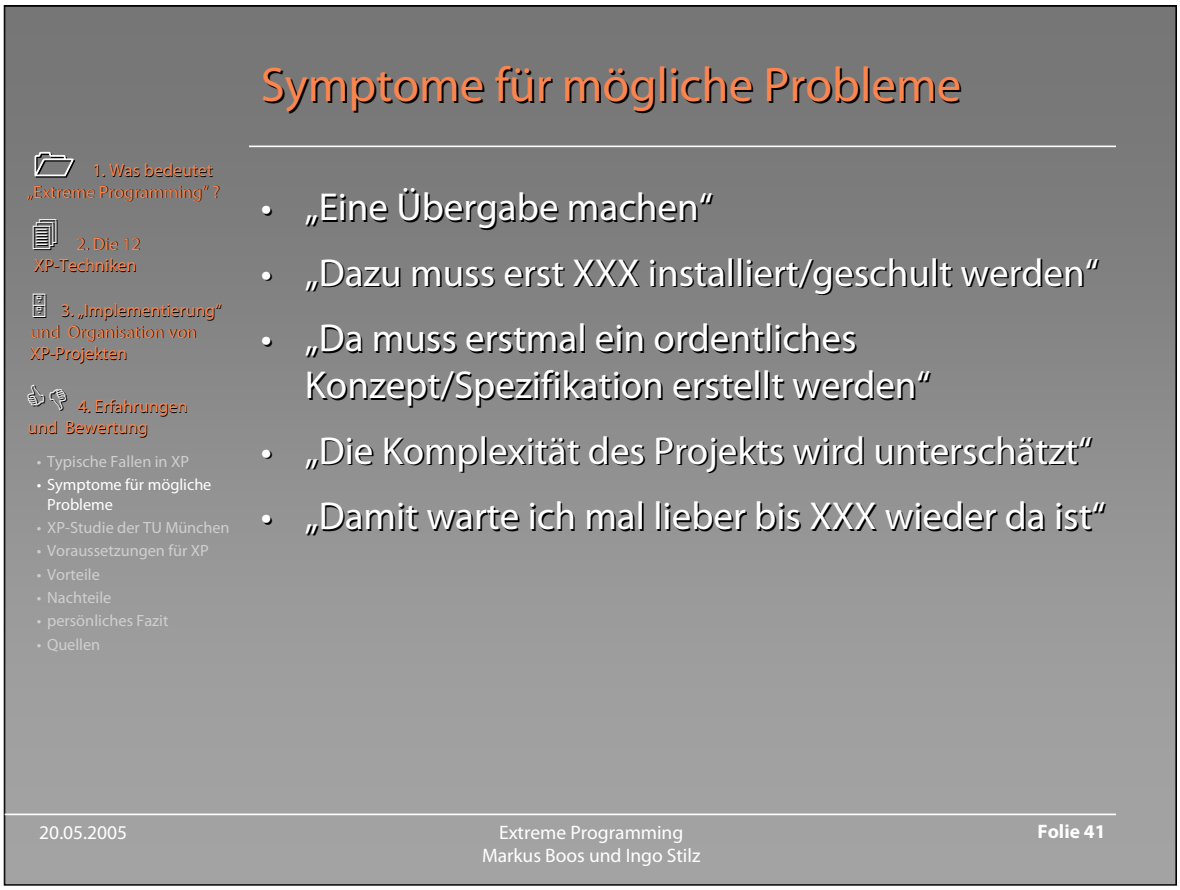
„Deadlines verschieben anstatt Funkionalität zu reduzieren“ Häufig werden Deadlines mehrfach verschoben und verursachen einen enormen Stress in der Entwicklung. Das kann kann und soll in einem XP Projekt nicht passieren, da keine einzige Deadline verschoben wird. Es kann höchstes passieren, dass nicht die komplette Funktionalität realisiert worden ist.

„Ich als Programmierer schreibe meine Klasse zuerst und probiere aus, ob das Programm das tut, was ich will, Wenn dem so ist schreibe ich keine Testklassen mehr.“. Als Folge werden immer weniger Testklassen geschrieben was starke negative Effekte hat. Siehe XP-Prinzip Testen.

„Das Refactoring lässt sich nicht zerlegen“ Ein Trugschluss, der sich nach der Realisierung des großen Refactorings herausstellt und sich doch in kleinere Refactorings hätte zerlegen lassen.

„In Schönheit sterben“ Auf der Suche nach dem schönsten Code vergessen manche Programmierer oft die Priorisierung. Sie verlieren sich in endlosen Designverbesserungen. Achten Sie darauf dass dies nicht passiert. XP Entwickler sind Finisher!

„Routineaufgaben muss man nicht im Paar programmieren“ Gerade bei Routineaufgaben kann schnell man ein Flüchtigkeitsfehler unterlaufen. Daher ist das Programmieren in Paaren stets vorteilhaft!



Ob XP falsch verstanden oder angewendet wird kann man an bestimmten Symptomen erkennen (So genannte Gerüche eng. Bad Smell). Jeder hier genannte Ausspruch kann die Andeutung eines Problems bedeuten und man sollte hellhörig werden.

->>Eine Übergabe machen<<

-Wenn ein Team mit gemeinsamer Verantwortlichkeit handelt, muss keine Übergabe gemacht werden!

->>Dazu muss erst XXX installiert/geschult werden<<

-Ein XP-Team nimmt Herausforderungen an und verschafft sich in der Not selbst die notwendigen Qualifikationen selber. Verzögerungen werden in einem XP-Team auf Grund von fehlender Software oder Qualifikation nicht geduldet.

->>Da muss erstmal ein ordentliches Konzept/Spezifikation erstellt werden<<

-Die Abwesenheit großer konzeptioneller Vorarbeiten in XP-Projekten ist zunächst sehr gewöhnungsbedürftig. Die Herausforderung liegt darin nicht in alte Verhaltensweisen zurück zu fallen und Papier am laufenden Meter zu produzieren.

->>Die Komplexität des Projekts wird unterschätzt<<

-In XP werden immer nur die aktuellen Anforderungen umgesetzt und auf die einfachste mögliche Art und Weise. Durch den konsequenten Einsatz objektorientierter Technologien kann jedoch eine flache Aufwandskurve erreicht werden und somit sind später auftretende Anforderungen kein Problem.

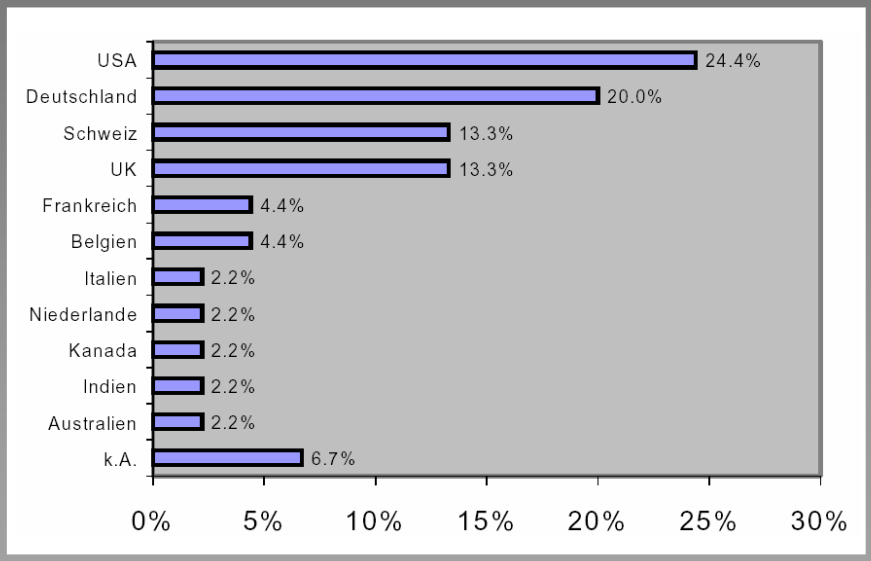
->>Damit warte ich mal lieber bis XXX wieder da ist<<

-Entwickler in einem XP-Team erledigen einfach jeweils die nächste Anforderung und verschiebt sie nicht nur weil ein Entwickler gerade nicht zur Verfügung steht. Stellt sich so eine Verhaltensweise ein, wird die gemeinsame Verantwortlichkeit nicht konsequent praktiziert.

XP-Studie der TU München

- 45 Teilnehmer aus der Softwareindustrie
  - 43 kommerzielle Projekte
  - 2 private Projekte
- gesamtes Alters- und *Grössenspektrum*
  - Neuling bis zur alteingesessenen Firma
  - Einmann-Betrieb bis zum gigantischen Konzern
- Studie entstand Ende 2001

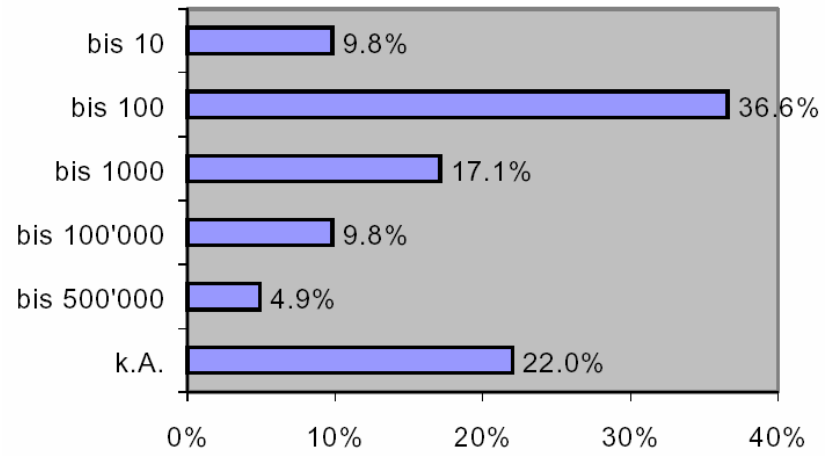
XP-Studie – Länderverteilung



aus [RuSc01]

Knapp zwei Drittel der Projekte wurden also im europäischen Raum durchgeführt, knapp ein Drittel in Nordamerika. Asien und Australien sind mit jeweils einem Projekt vertreten. Afrika gar nicht. Was auch auffällt, ist die geringe Präsenz von XP Projekten im asiatischen Raum, in dem doch viel Computerindustrie vertreten ist. Hier konnten bereits in der Recherche so gut wie keine Firmen und Projekte gefunden werden, denen der Bogen geschickt werden konnte.

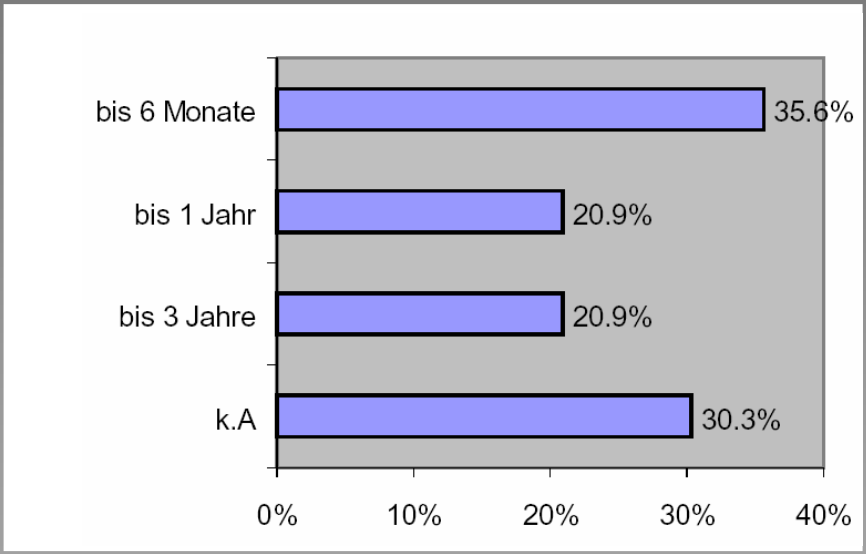
XP-Studie – Unternehmensgrössen



aus [RuSc01]

Bei der Mitarbeiterzahl findet sich eine Konzentration bei Firmen mit 10 bis 1000 Mitarbeitern. Über die Hälfte der Projekte wurden in Unternehmen dieser Größe durchgeführt. Sehr kleine Firmen zwischen 100.000 und 500.000 Mitarbeitern immerhin noch mit einem Anteil von knapp 5% vertreten.

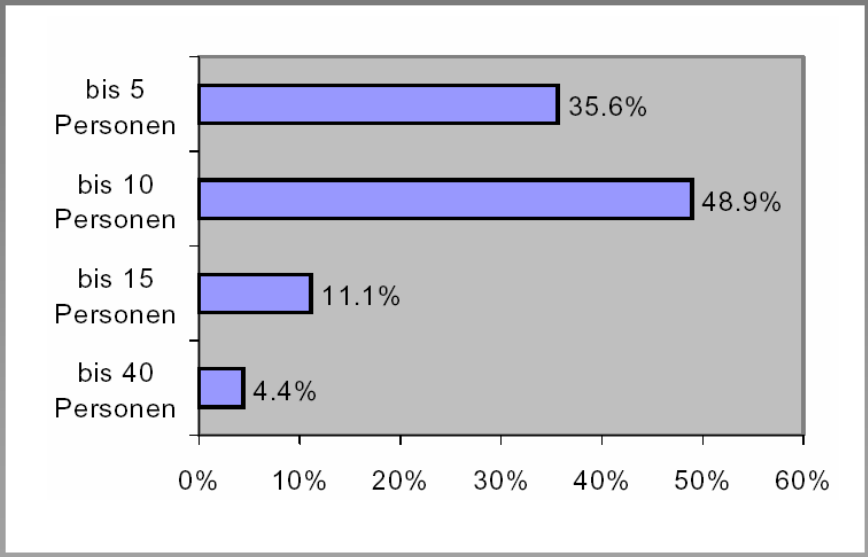
XP-Studie – Projektdauer



aus [RuSc01]

Die Projektdauer verteilt sich eher gleichmäßig auf längere und kürzere Projekte. 28% der Projekte sind eher kurz, Laufzeit bis zu einem halben Jahr. Jeweils 21% der Projekte laufen bis zu einem Jahr bzw. bis zu drei Jahren. 30% machen keine Angaben zur Projektlänge.

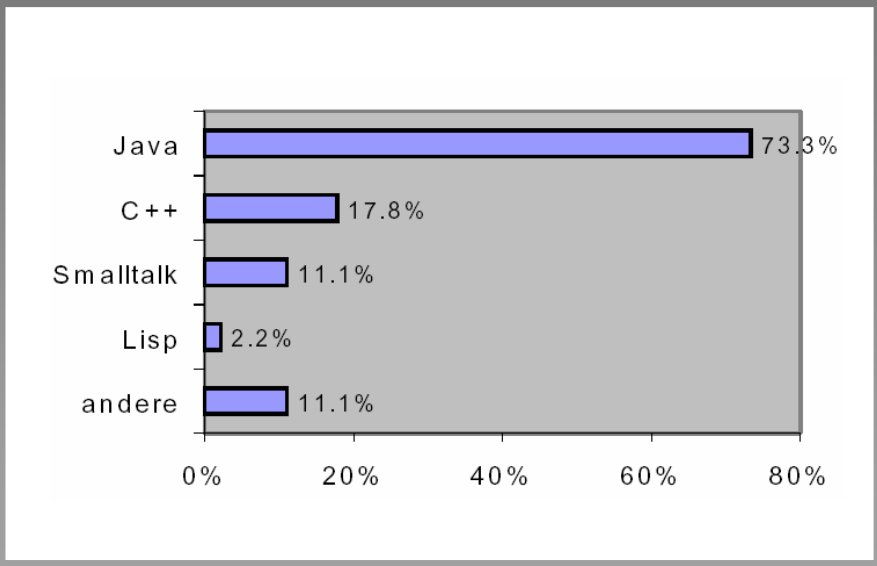
XP-Studie – Teamgrößen



aus [RuSc01]

Die Abbildung zeigt, dass die bei XP empfohlene Teamgröße von zwei bis zehn Teammitgliedern meist eingehalten wurde. In knapp 36% der Projekte betrugen Teamgrößen bis zu fünf Personen, in 49% der Fälle lagen sie zwischen fünf und zehn. In einem Unternehmen wurde ein XP Projekt nur von einer Person durchgeführt. Es gab jedoch auch zwei Projekte, die die empfohlene Teamgröße stark überschritten. Und jeweils ca. 40 Mitglieder im Team hatten. Dabei handelt es sich jedoch in einem Fall nicht um „pures“ XP.

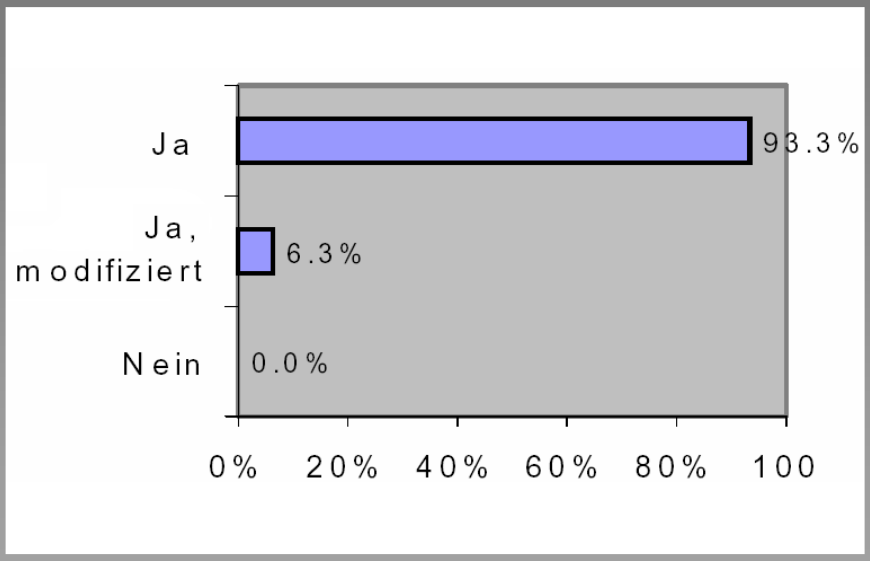
XP-Studie - Programmiersprache



aus [RuSc01]

Bei den benutzen Programmiersprachen sind Mehrfachnennungen möglich gewesen.

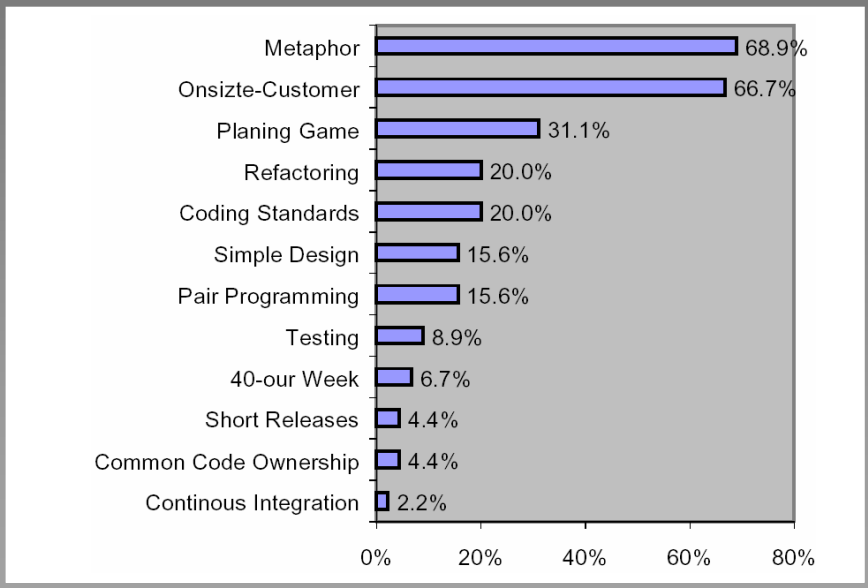
XP-Studie - Wiederverwendung



aus [RuSc01]

Bei der Frage nach dem weiteren Einsatz von XP sprechen die Zahlen eine klare Sprache. Ausnahmslos wollen alle XP wieder verwenden und vertreten es aktiv. In drei Fällen wurde angegeben, es in einer angepassten Form wieder zu verwenden.

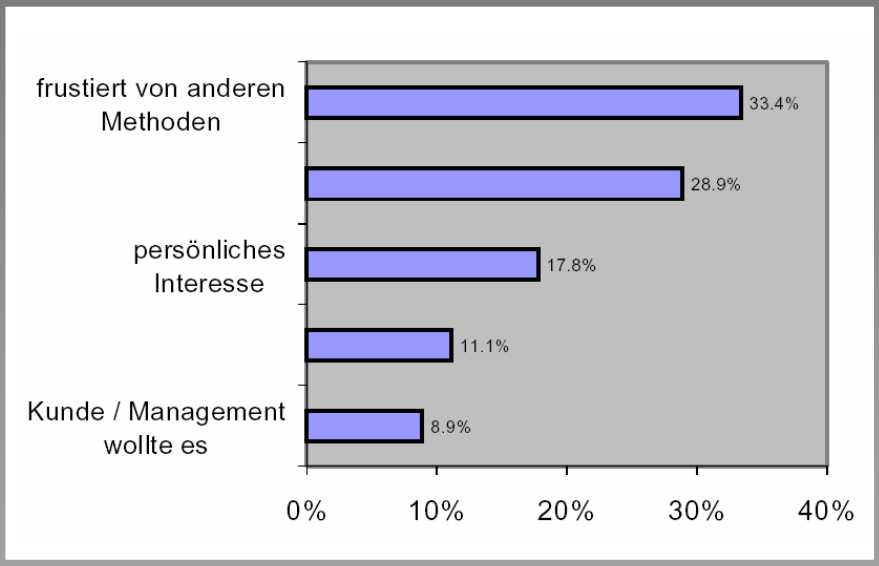
XP-Studie – nicht genutzte Elemente



aus [RuSc01]

Die Abbildung gibt eine Übersicht über die am wenigsten benutzten Elemente. Die drei am wenigsten genutzen Elemente sind Metapher, On-site Customer und Planing Game. Metapher , On-site Customer liegen hierbei mit Abstand vorne.

XP-Studie – Warum XP?



aus [RuSc01]

Das Schaubild gibt Übersicht, warum für XP entschieden wurde. Auch hier waren Mehrfachnennungen möglich. In über 33% der Fälle schien XP die sinnvollste Methode zu sein. Hier wurde vor allem nach einer einfachen und leichtgewichtigen Methode verlangt.

Mehrere waren konkret frustriert von herkömmlichen Methoden wie Wasserfall oder CMM; vorher angewandte Techniken hatten zu unbefriedigten Ergebnissen geführt.

In 29% der Fälle war das Projekt ein „typisches XP Projekt“, die Ziele von XP passten auf die Anforderungen, die das Projekt stellte.

In 18% spielten persönliches Interesse am XP Prozess eine Rolle bei der Entscheidung. Hierbei gefällt die Flexibilität an XP.

11% hatten bereits konkrete, gute Erfahrungen mit XP in anderen Projekten gemacht.

IN einigen Fällen wurde XP direkt vom Management oder Kunden nachgefragt; dass dies im Moment noch eher ungewöhnlich ist und aus dieser Richtung derzeit eher Widerstand gegen XP kommt.

Vorraussetzungen für XP

- Infrastruktur (Raumaufteilung, Hardware etc.)
- Einheitliche und praktische Entwicklungsumgebung
- Testsoftware für automatische Tests
- Konfigurationsmanagement- Tool
- Min. 50% der Entwickler Erfahrungen mit XP
- Mitarbeiterverhalten und -qualifikation:
  - Disziplin
  - Ehrlichkeit
  - Lernbereitschaft
  - Teamfähigkeit und realistische Einschätzung
- Enger und guter Kontakt zum Kunden

Vorteile von XP

- Verantwortlichkeiten sind klar verteilt
- motiviert mehr zum Arbeiten
- Fördert soziale, rücksichtsvolle und kommunikative Personen
- Sehr flexibel (z.B. auf Änderungen, Mitarbeiter)
- XP- Verfahren relativ einfach zu lernen
- Feedbacks zeigen große Beliebtheit
- Arbeitet wirtschaftlicher als traditionelle Entwicklungsansätze
- qualitativ hochwertige Software
  → jederzeit lauffähige Version
- Wissensverbreitung im Team

Nachteile von XP

- Keine Dokumentation für verantwortungsvolle Anwendungen
- Fehlendes Risikomanagement
- Nicht geeignet zum freien experimentieren neuer Techniken
- XP-Werte müssen gelebt werden
- Funktioniert nur mit harmonischen und relativ kleinen Teams
- Akzeptanz fehlt bei vielen Kunden/Managern

persönliches Fazit

- XP kann erfolgreich eingesetzt werden
- guter Einstieg für Neulinge ohne Berufserfahrung
- Einzelkämpfertum wird nicht unterstützt
- das Team ist mehr als nur die Gesamtheit der Mitarbeiter
- viel Kommunikation und Austausch
  → Mensch im Vordergrund
- gut geeignet für moderne Unternehmenskulturen und flache Hierarchien
- Fehlen objektiver Bewertungen
  → XP ist nicht immer die Lösung aller Probleme

UNSER FAZIT: XP führt nicht zu den Anfängen der Software-Engineering Kultur zurück, sondern versucht, manchen Ballast heutiger Vorgehensmodelle zu vermeiden.

Quellen

[Ambl01] Ambler, S.: Agile Modeling Throughout the XP Lifecycle, 2001, http://www.agilemodeling.com/essays/agileModelingXPLifecycle.htm, Abruf am 13.05.2005

[DoHa04] Dornberger, R.; Habegger, T.: Extreme Programming – eine Übersicht und Bewertung, Discussion Paper 04-05, Fachhochschule Solothurn, 2004

[LiRW02] Lippert, W.; Roock, S.; Wolf H.: Software entwickeln mit XP, 1. Auflage, dpunkt.verlag, Heidelberg, 2002

[Plam03] Plamondon, S.: Working smarter, not harder: An interview with Kent Beck, 2003, http://www-128.ibm.com/developerworks/java/library/j-beck/, Abruf am 13.05.2005

[RuSc01] Schröder, A.; Rumpe, B.: Quantitative Untersuchung des Extreme Programming Prozesses, Technischer Bericht TU München (I0110) und ViSEK/006/D2001, 2001, http://www.software-kompetenz.de/?4763, Abruf am: 10.05.2005

[Vett05] Vetter, M.: Vorlesung SME, WS 04/05– Softwareentwicklungsmethoden und –werkzeuge, FH Mannheim

[Zivk03] Zivkovic, S.: Extreme Programming – Theorie und praktische Erfahrungen, Diplomarbeit, Basel, 2003

http://www.extremeprogramming.org

http://www.x**programming**.com