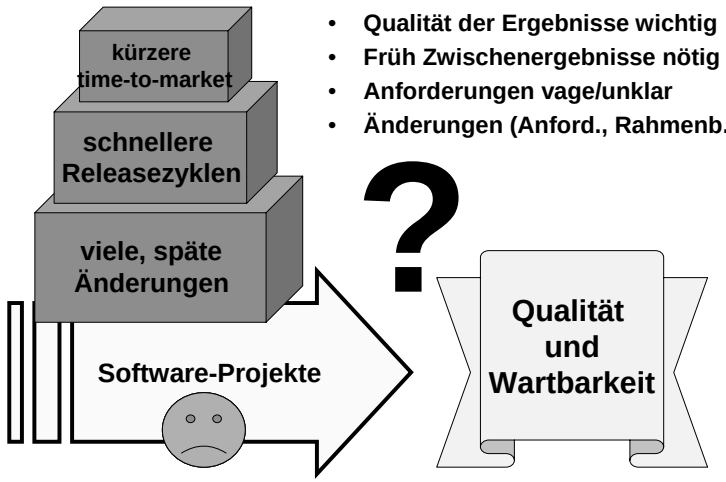
Flexible und agile Software-Prozesse, Extreme Programming

Dr. Kurt Schneider
Leiter Erfahrungsbasierte Prozessverbesserung
DaimlerChrysler Forschungszentrum Ulm
kurt@acm.org

Einführung in agile Methoden

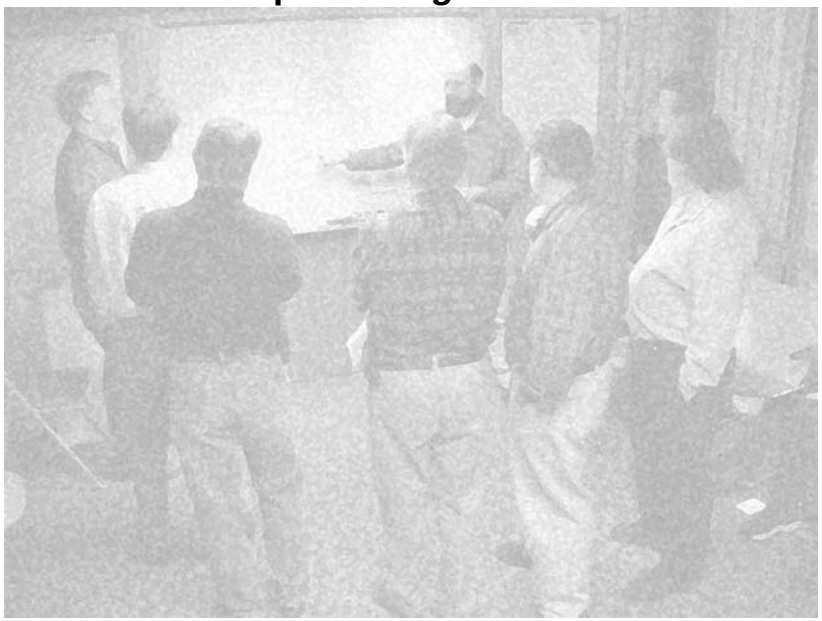
- **Wozu agile Methoden?**
- **Was sind agile Methoden?**
  - **keine (richtige) Definition**
  - **aber charakteristische Eigenschaften**
- **Zwei verschiedene, konkrete Beispiele**
  - **eXtreme Programming (XP): Praktiken**
  - **SCRUM: Prinzip**
- **Agile Methoden anwenden - nicht immer, aber ...**

Wozu agil? Softwareprojekte unter Druck



- **Großer Zeitdruck**
- **Qualität der Ergebnisse wichtig**
- **Früh Zwischenergebnisse nötig**
- **Anforderungen vage/unklar**
- **Änderungen (Anford., Rahmenb.)**

Konspirativ: *Agile Alliance*



Was ist agil? Manifesto for Agile Software Development

www.agilemanifesto.org

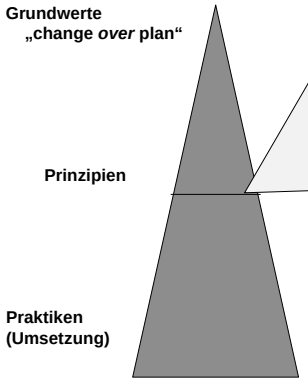
"We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

Individuals and interactions *over* processes and tools
Working software *over* comprehensive documentation
Customer collaboration *over* contract negotiation
Responding to change *over* following a plan

That is, while there is value in the items on the right, we value the items on the left more. "

Kent Beck, Alistair Cockburn
Martin Fowler, Jim Highsmith, Ron Jeffries
Ken Schwaber und 11 andere

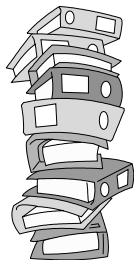
Was ist agil? Struktur agiler Methoden



- **Höchste Priorität: Kundenzufriedenheit**
  - **frühe und häufig laufende Software**
- **Einfachheit**
  - **unnötige Arbeit vermeiden**
- **Auch späte Änderungen willkommen**
  - **bedeuten Verbesserungen für den Kunden**
- **...**
- **Direkte Mensch-zu-Mensch Kommunikation**
  - **Beste Möglichkeit zum Informationsaustausch**
- **Regelmäßige Überprüfung und Verbesserung der eigenen Effektivität**
  - **Feedback auf mehreren Ebenen**

Wieviel Planung braucht ein Projekt?
nach Barry Boehm

feingranulare Verträge
++ klare Arbeitsgrundlage
+ finanzielle Sicherheit
- - enorm aufwändig
- - Änderungen auch für AG schwer durchsetzbar



Wieviel Planung braucht ein Projekt?
nach Barry Boehm

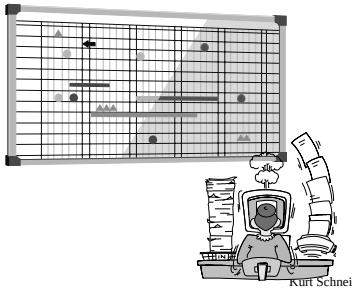
ad hoc
+ wenig Planungsaufwand
+ individuelle Freiheit
- - Ergebnis unvorhersehbar
- - abhängig von "Helden"

feingranulare Verträge

Wieviel Planung braucht ein Projekt?
nach Barry Boehm

Meilenstein- u. Plangetrieben
+ langfristige Vorhersagen
+ gute Zustandskontrolle
- - Änderungen aufwändig
- - unrealistische Annahmen
schwer zu eliminieren

ad hoc — **feingranulare Verträge**

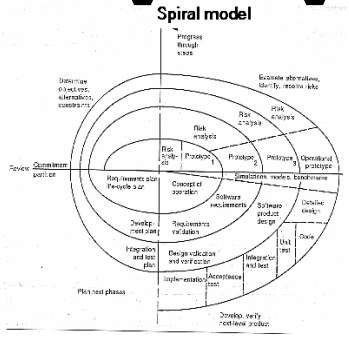


Wieviel Planung braucht ein Projekt?
nach Barry Boehm

Meilenstein- u. Risikogesteuert
++ Risiken aktiv ausgeräumt
+ Teilergebnisse früh
- kaum langfristig planbar
- relativ aufwändig

ad hoc — **Meilenstein- u. Plangetrieben** — **feingranulare Verträge**

Spiral model



Wieviel Planung braucht ein Projekt?
nach Barry Boehm

eXtreme Programming
+ früh Kernergebnisse
++ Änderungen unproblem.
- in sehr großen Proj. Zusatzplanung nötig
- viel Selbstdisziplin nötig

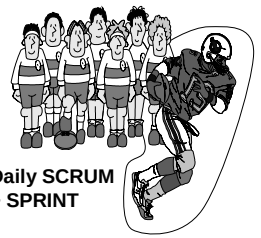
ad hoc — **Meilenstein- u. Risikogesteuert** — **Meilenstein- u. Plangetrieben** — **feingranulare Verträge**

Was bedeutet agil? Planungshorizont nach Maß
nach Barry Boehm

Agiles Beispiel: SCRUM
+ mittelfristig geplant
+ reagiert schnell
- hängt an MA-Qualifikation
- Endprodukt nicht spezifiziert

ad hoc — **eXtreme Programming** — **Meilenstein- u. Risikogesteuert** — **Meilenstein- u. Plangetrieben** — **feingranulare Verträge**

Agile Methoden



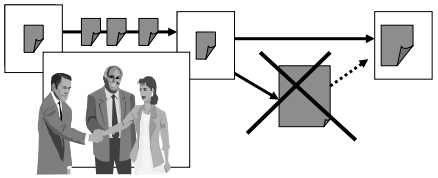
Daily SCRUM
+ SPRINT

Prinzip und Denkweise von Agilen Methoden

nach Frühauf, Conquest 2001	**Bisheriger Ansatz**	**Agiler Ansatz**
Mitwirkung des Kunden	unwahrscheinlich	kritischer Erfolgsfaktor
Etwas Nützliches wird geliefert	erst nach einiger (längerer) Zeit	mindestens alle sechs Wochen
Das Richtige entwickeln durch	langes Spezifizieren, Vorausdenken	Kern entwickeln, zeigen, verbessern
Nötige Disziplin	formal, wenig	informell, viel
Änderungen	erzeugen Widerstand	werden erwartet und toleriert
Kommunikation	über Dokumente	zwischen Menschen
Vorsorge für Änderungen	durch Versuch der Vorausplanung	durch „flexibel Bleiben“

Wie *wird* ein Projekt agil(er)?

- **Druck reduzieren durch leicht-gewichtige Ansätze**
  - **Unnötige Dokumente einsparen**
  - **Vorgaben und Templates “verschlanken”**
- **Vage Anforderungen und Änderungen einkalkulieren**
  - **schnell zum Kernsystem, Weiterentwicklung inkrementell**
- **Besseres Feedback**
  - **konkrete organisatorische und technische Maßnahmen**
  - **engere Kundeneinbindung**



Agil oder nicht agil: Das ist hier die Frage

(fiktives Fallbeispiel)

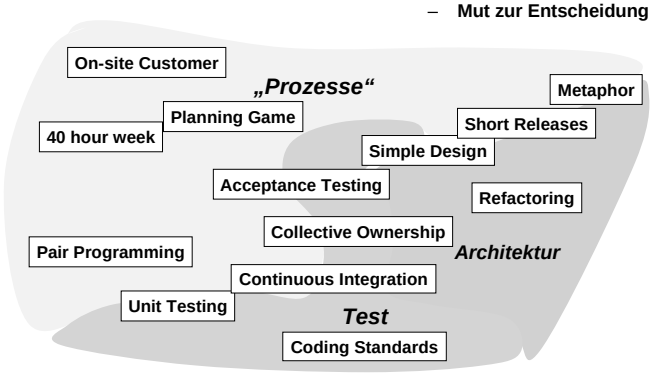
- **Gleiches OO-Projekt; die Hälfte der Leute oo-erfahren**
  - **Alle Entwickler 100% im Proj.**
  - **Offen das Erfahrungsgefälle angehen**
- **Erste Fachliche Analyse**
  - **Mit den Anwendern**
  - **Auf Schmierzetteln und Tafeln**
  - **Bei Bedarf abfotografiert**
- **Machbarkeitsphase**
  - **Zentrale Fragen mit Prototyp klären (Labormuster/Demo-PT)**
- **Erste Iteration (4 Wochen)**
  - **Kunden wünschen Funktionen**
  - **Entwickler schätzen**
  - **Kunden priorisieren**
  - **Entwickler implementieren genau das – nicht mehr, nicht weniger**
  - **Fragen an den Kunden direkt**
- **OO-Projekt; 12 Monate; die Hälfte der Leute oo-erfahren**
  - **Alle Entwickler nur 50% im Proj.**
- **Gleich zu Anfang Technologie auswählen,**
  - **um sichere Basis zu haben**
  - **Case-Tool etc. kaufen und installieren**
- **Fachliche Analyse**
  - **Domain-Experten modellieren im Case-Tool Sequenz-Diagramme**
  - **Legen sie Anwendern vor; bestätigen Korrektheit**
- **Implementierung**
  - **Mit Case-Tool technisch zerlegt**
  - **Viele Diagrammarten eingesetzt**
  - **Arbeitsteilig und iterativ**
  - **Entwickler bewerten die Modelle**
  - **Kommentare direkt eingearbeitet**

Agil oder nicht agil: Das ist hier die Frage

- **Nach der Iteration**
  - **Fehlen noch Funktionen, aber die wichtigsten sind da**
  - **Gab es immer eine auslieferbare Version**
  - **Geht es in die nächste usw.**
- **Am Ende**
  - **Viel (sinnvolle) Kommunikation**
  - **Anwender entscheidet über Erfolg**
  - **Entwickler bleiben „bescheiden“**
  - **Feedback bei allen Iterationen, richtig einsetzbar nur Inkremente (alle 2-3 Iterationen)**
- **Aus dem Case-Tool Code generieren**
  - **Rümpfe dann ausfüllen**
  - **Entwickler bekommen Teile der Klassen zugewiesen**
- **Am Ende**
  - **Nicht rechtzeitig fertig, weil Schnittstellen nicht passen**
  - **Immer höherer Druck auf Entwickler**
  - **Setzen schnell Änderungen um, schludern, verderben Struktur**

Beispiel „eXtreme Programming“ (XP)

- **Praktiken**
  - **Verstärken sich gegenseitig**
  - **Zusammenspiel von Prozess, Testen und Architekturfragen**
- **Werte und Prinzipien**
  - **Kommunikation u. Kundeneinbindung**
  - **Feedback und inkrementelle Entwicklung**
  - **Einfachheit / Simplicity**
  - **Mut zur Entscheidung**



Was heißt hier extrem?

- **Agile Entwickler sind keine Anarchisten**
- **eXtreme Programming erfordert** *extreme Disziplin*
- **Bewährte Praktiken aus der SW-Entwicklung...**
  ... werden ins Extrem geführt
- **Beispiel: Code-Reviews haben sich bewährt...**
- **... Daher: <u>Ständige</u> Code-Reviews durch Pair Programming**

Nach Jan von Hunnius, DaimlerChrysler, 2002

Planning Game / Planungssitzung

- **Anforderungen werden auf User Story Cards gesammelt**
  - **vom Kunden geschrieben**
  - **umgangssprachlich, mit Skizzen**
- **Entwickler schätzen Kosten für jede Story Card**
  - **abstraktes Maß (z.B. 1-5 Punkte)**
  - **Schätzung durch Erfahrung (ähnlichen Karten)**
  - **Nachfragen und Feedback an Kunden**
  - **Neue Erkenntnisse auf Story Cards notieren**
- **Kunde wählt Story Cards für nächste Release(s) aus**
  - **(Meistens) nach Nutzen priorisiert**
  - **Bisherige Produktivität des Entwicklungsteams ist Limit**
- **Story Cards sind Grundlage für Akzeptanz-Tests**

Motor starten
12.4.2002, D.Autor

Anforderung

1. Fahrer dreht Zündschlüssel
2. Zündung an
3. Motor startet
4. Fahrzeug fährt

Aufwand,
Datum der Erledigung
geschätzt: 3 Gu.

Planning Game / Planungssitzung: Tips

- **Anforderungen werden auf Story Cards gesammelt**
  - **DIN A5 (auch DIN A4 ok), möglichst informell**
- **Entwickler schätzen Kosten für jede Story Card**
  - **Bisherige Produktivität des Entwicklungsteams ist Limit**
    - **Entwickler notieren also, wie lange sie gebraucht haben**
    - **Vergleich schult die Schätzfähigkeit**
  - **Vergangene Zeit ist nicht gleich eingesetzte Zeit!**
    - **Telefonate, email, Pausen**
    - **Verhältnis: „Velocity“, „Load Factor“ : oft 3-5**
      - **Möglichst auf 1,5-2 drücken**
    - **Ebenfalls nach jeder Iteration errechnen**
  - **Ideale Zeit/Netto-Zeit ist Schätzbasis**
  - **Relative, abstrakte Schätzung oft besser**
  - **Maximalwert festlegen (5 Gu.): alles größere wird geteilt**
  - **Wichtig: jede Iteration hat gleich viel Zeit!**

Motor starten
12.4.2002, D.Autor

Anforderung

1. Fahrer dreht Zündschlüssel
2. Zündung an
3. Motor startet
4. Fahrzeug fährt

Aufwand,
Datum der Erledigung
geschätzt: 3 Gu.

Planning Game / Planungssitzung
wie es danach weitergeht

- Planungs“spiel“ ist für Deutschland zu unernst
- Kunde wählt Story Cards für nächste Release(s) aus
  - (Meistens) nach Nutzen priorisiert
  - Dabei kann – und muss – man oft rückfragen
  - Für diesen Schritt müssen Story Cards physisch vorliegen (nicht im Computer)
- Entwicklerpaar nimmt sich ***oberste*** Story Card vom Stapel
  - Dadurch keine Überschneidung
- Story Cards sind Grundlage für Akzeptanz-Tests
  - Müssen daher testbar sein
  - Tests entwickeln sich ebenfalls im Laufe der Zeit
  - Es gibt zusätzlich Task Cards: feinere/technisch motivierte Aufgaben
- Streitfall: Was geschieht mit Story Cards weiter?
  - Variante 1: Dokumentation der Anforderungen
  - Variante 2: Wegwerfen! Anforderungen ändern sich ja.

Planning Game / Planungssitzung
eine wichtige Verpflichtung

- Jedes Inkrement muss so definiert werden, dass es dem Auftraggeber/Kunden Nutzen stiftet!
- Welchem Auftraggeber würde das nicht gefallen?
- Aber Achtung: das ist eine Verpflichtung an den Auftraggeber, nicht an den Entwickler!
  - Sonst nimmt er dem Entwickler die Chance zum Erfolg
  - Die Entwickler werden den Kunden daran erinnern (evtl. im Vertrag)
  - Und ihm bei der Auswahl helfen

Planning Game / Planungssitzung
Auf Dauer

- Arbeiten auf Basis von Story- und Task-Cards fordert Konzentration
  - Man muss das umsetzen, was gefordert ist
  - Weitergehende Ideen sind nicht gefragt, und man hat keine Zeit
  - Die Arbeit ist sehr intensiv und anstrengend
- Gefahr: Ausbrennen von Entwicklern
  - Zu lange am engen Zügel geführt, kein „gold plating“ mehr erlaubt
  - Zu lange im Stress
  - Zu lange keine neuen technischen Herausforderungen mehr
- Abhilfe: Gold Cards
  - Ein „Joker“ für einen lange angespannt arbeitenden Mitarbeiter
  - Muss mal (z.B. ein, zwei Tage lang) an einer neuen technischen Frage knobeln
  - Darf rumprobieren und Grundlagen legen
  - In der Schätzung gehen die automatisch ein, sofern regelmäßig

On-site Customer / Kunde vor Ort

- Kunde ist über die gesamte Projekt-Laufzeit vor Ort
  - zumindest leicht und jederzeit erreichbar
- Beantwortet Fragen der Entwickler
  - Kein Zeitverlust bei Unklarheiten
- Schreibt User Stories (oder hilft dabei, sie zu schreiben)
  - Ständige Weiterentwicklung der Anforderungen
- Trifft Entscheidungen
  - Kein Zeitverlust bei Entscheidungen
  - Keine Fehlentscheidungen durch Entwickler
- Voraussetzung: Kompetenz und Entscheidungsgewalt

On-site Customer / Kunde vor Ort

- **Schreibt User Stories**
  - **Bezeichnung betont: informale kleine Geschichten aus seiner Sicht**
    - **Weder formal noch vollständig**
    - **Festgehalten auf Story Cards**
- **Kunde ist über die gesamte Projekt-Laufzeit vor Ort**
  - **Nachfragen sind so schnell zu klären: Erfolgsentscheidend!**

- **Empfehlungen**
  - **Muss nicht 100% vor Ort sein, aber kurzfristig erreichbar**
    - **Z.B. tägliches „Stand-up Meeting“ (ähnlich SCRUM)**
    - **Auf lange Frist könnte 100%-anwesender Kunde sonst sogar die „Kundigkeit“ verlieren**
    - **Oft sind verschiedene „Kunden-Arten nötig“ (z.B. Multi-Channeling)**
  - **Wichtige Unterscheidung**
    - *Anwender* **beantwortet fachliche Rückfragen**
    - *Kunde* **priorisiert Story Cards**
  - **Künstlicher Kunde bündelt Perspektiven, ersetzt echten On-Site-C.**

On-site Customer / Kunde vor Ort

- **Verhaltensweisen von Kunden**
  - **Oft werden Soll- mit Istprozessen verwechselt**
  - **Altsystem wird trickreich „umgangen“**
  - **Stellen sich oft nur lokale Änderungen vor**

- **Daher: Entwickler helfen Kunden beim Story Card-Schreiben**
  - **Erinnern an die Prüfbarkeit, fragen gleich nach**

- **Ganz wichtig**
  - **Anwenderkontakt ist essenziell.**
  - **Wenn Entwicklerteam fachliche Frage hat: nicht selbst raten, fragen!**
  - **Anwender entscheidet über Alternativen**
  - **Überlegen und klären:**
    Welche „Kundenarten“ sind nötig, woher kommen sie?

Metapher

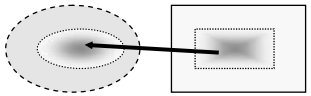
- **Bildhafte Beschreibung der Grundidee des Systems**
- **Verständlich und verbindlich für alle Beteiligten**
  - **Schafft mehr gemeinsame Terminologie als explizit vereinbart**
  - **„Richtschnur, falls gerade keine Klärung möglich“**
  - **Relevanter Modellkern?**

- **Relativ umstritten: „nicht unbedingt nötig...“**
  - **Kreative Leistung, nicht jedermanns Sache**
  - **Oder Repertoire fertiger Metaphern?**

- **Metaphern**
  - **Für das zu entwickelnde System**
  - **Aber auch für den Prozess dorthin**

Metapher

- **Bildhafte Beschreibung der Grundidee des Systems**
  - **Beispiele: Handy / Schreibmaschine / Sekretärin**



- **Modelltheoretisch:**
  - ***Original*** gibt es noch nicht („neues System“)
  - ***Relevante Original-Attribute*** sucht man
  - ***Modell*** hat einige „prägnante“ Eigenschaften: als relevant angenommen!
  - ***Für*** Entwickler und Kunden, ***ab*** Projektbeginn, ***um*** reichhaltiges, gemeinsames Verständnis des zu entwickelnden Systems zu entwickeln, darüber zu reden – und um zu entscheiden!
  - ***Stärke der Metapher ist also:***
    „Durch Vertrautheit mit dem als Metapher verwendeten Ding (=Modell) gewinnt das noch vage Original an Plastizität (d.h. mehr Eigenschaften)“
- **Verständlich und verbindlich für alle Beteiligten**
  - Fördert Kommunikation hin zur Architektur
  - Kent Beck: ist Architektur andere: führt zu Architektur
  - Bei Bedarf mehrere Teil-Metaphern kombinieren

Metapher

- **Herausforderungen**
  - Verstehen alle das gleiche unter der gewählten Metaphern?
    - Woher wissen wir das?
  - Können alle metaphorisch denken?
    - Erfahrung: manche sind Fans, andere mögen es gar nicht
    - Auch Experten haben oft wenige Metaphern im Repertoire
  - Metaphern-Finden erfordert Fantasie und Routine darin
  - Erfahrung: Metaphern kommen oft sehr kurz (zu kurz)
- **Fertig-Metaphern: auch nicht das Wahre**
  - Streitfall: Produktmetapher, die den Bildcharakter zunehmend verliert
  - Beispiel: Schreibtisch (mit Papierkorb drauf?)
- **Beispiele für Projektmetaphern**
  - Brückenbauprojekt vs. Expedition
  - Reihenhaus, Stadthalle oder Millionärsvilla bauen?
  - ***Was geht Ihnen da durch den Kopf?***

Kurze Release-Zyklen

- **Erste Release nach 1-2 Monaten, dann alle 2-4 Wochen**
- **Häufiges Feedback durch den Kunden**
  - durch Erfahrung mit dem laufenden Code
- **Projektfortschritt wird nach fertigen Story Cards gemessen**
  - Kunde sieht Projektfortschritt
- **Nach Release neues Planungsspiel für nächste Release**
  - eventuell neue oder geänderte Story Cards
  - eventuell neue Kostenschätzung
  - Auswahl der Story Cards
- **Selbst bei Abbruch ist ein Nutzen vorhanden**
  - Geld für Nutzen sinnvoll eingesetzt
  - kein gescheitertes Projekt: irgendwas Nützliches ist da!

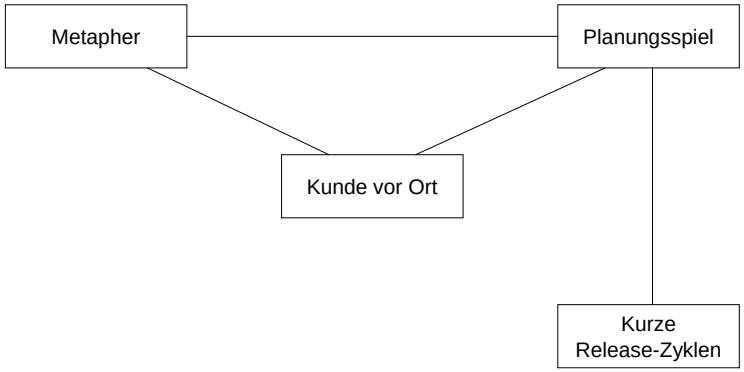
Kurze Release-Zyklen

- **Ziel: Häufiges Feedback durch den Kunden**
  - durch Erfahrung im operativen Einsatz – nicht bloß beim Rumspielen!
  - Teillieferung scheint oft kaum möglich, ist es aber meist
- **Begriffsklärung**
  - „*Release*“ = „Inkrement“ = Auslieferung zur operativen Nutzung
  - Verlangt einiges an Doku, Handbücher, Zusatzaufwand
  - Dazwischen *Iterationen*: auch lauffähig, aber weniger Aufwand
- **Nach Release: Feedback, Planungsspiel für nächstes Release**
- **Standard-Einwand: „Saubere Spezifikation ist besser“**
  - Antwort: ist aber fast nie zu erreichen

Kurze Release-Zyklen

- **Bedenken vieler Entwickler**
  - Release = Stress; häufige Releases = Dauerstress
  - Abhilfe: technische Unterstützung + XP-Techniken
  - Gute Testabdeckung hilft z.B. sehr
- **Häufige Kundenbedenken**
  - „wie sollen sich die Anwender ständig an Neues gewöhnen?“
    - Aber: sind nur kleine Schritte, eben gerade kein big bang
  - Dennoch: in großen Firmen ist Roll-out sehr teuer
    - Lösung: Einige Pilotanwender
  - Da so wichtig: <u>operative</u> Nutzung im Vertrag fixieren
- **Teilsysteme sind keine Prototypen!**
  - Prototypen (Labormuster) heißen in XP „Spikes“, sind nur zum Lernen – und nicht sehr häufig oder beliebt
    - Denn sie bergen Risiken: „schon fertig?“/“Aspekt vergessen“
  - PTs sind sinnvoll besonders in „Vorprojektphase“ zur Machbarkeit
    - Die baut auch Vertrauen auf, was wichtig ist

Praktiken Kommunikation



Nach Jan von Hunnius, DaimlerChrysler, 2002

Test First / zuerst Testen

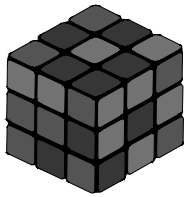
- **Erst Testfall codieren, dann Funktionalität**
- **Solange implementieren, bis Test erfolgreich läuft**
- **Test als Kriterien für Anforderungserfüllung**
  - **Stellt Kundenzufriedenheit sicher**
  - **Hilft Mehrarbeit zu vermeiden**
- **Reihenfolge ist wichtig!**
  - **Zuerst über das „Was soll diese Funktion?“ nachdenken: Unit Test**
  - **Dann das „Wie“ behandeln: Implementierung, die Unit Test erfüllt**
  - **Oft testen**
- **Unit Tests sind Voraussetzung für Refactoring**
  - **Wichtig: Ausführbare, automatisierte Unit Tests**

Test First / zuerst Testen

- **Begriffe**
  - **Akzeptanztests: Black Box, „Funktionstests“**
  - **Komponententests: auch White Box, „Unit Tests“**
- **Vorteile**
  - **Automatisch ausführbare Tests**
  - **Entwickler achten auf Testbarkeit**
  - **Einfachheit wird nahegelegt**
  - **Unerwartetes Verhalten gefunden => Testfall schreiben, der es abfängt**
- **Schwierigkeiten**
  - **Nicht immer einfach: GUI, nicht-funktionale Anforderungen etc.**
  - **„und wenn der Testcode Fehler hat?“**

Refactoring

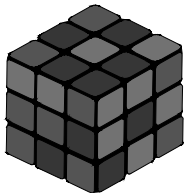
- **Änderung des Codes ohne Änderung des Verhaltens**
  - **Man macht sich Gedanken über Architektur**
    ... aber nicht unendlich lange
  - **Änderungen kommen ohnehin - vorbereitet sein!**



- **Vorgehen nach fest definierten Refactorings**
  - **kleine Schritte**
  - **häufiges Übersetzen und Testen**
- **Verbesserung der Architektur**
  - **Priorität: Bessere Verständlichkeit**
  - **Ermöglichen von Erweiterungen**
- **Testfälle sichern konstantes Verhalten**

Refactoring

- **Änderung des Codes ohne Änderung des Verhaltens**
  - **Widerspricht der Regel: „never change a running system“**
    - **Die aber zu veralternden Systemen führt!**
- **Vorgehen nach fest definierten Refactorings**
  - **kleine Schritte, je nur wenige Minuten**
    - **große Änderungen machen viel Ärger**
  - **Bei kleinen: optimistische Sperre**
    - **Prüfung beim Wieder-Einbringen**
    - **Pessimistische verstößt gegen Werte, ist langsam**
  - **häufiges Übersetzen und Testen**
  - **Vor und nach Stories (kleines) Refactoring erwägen**



- **Verbesserung der Architektur**
  - **Aber auch nur so viel, wie gerade derzeit nötig!**
- **Testfälle sichern konstantes Verhalten ohne Seiteneffekte**
- **Refactoring-Papst: Martin Fowler (Buch „Refactoring“)**

Refactoring

Beispiel: Umbenennen einer Operation in Oberklasse

- **Situation**
  - **In grafischem Editor gibt es Oberklasse Figur über Oval, Rechteck etc.**
  - **Jede Klasse habe Methode „zeichne“**

```
Public class Figur
  {
  public void zeichne ()
    { ... }
  }
```

- **Änderung**
  - **Neuen Parameter für `zeichne()` einfügen; bei vielen Unterklassen geht das nicht auf einmal.**
  - **Ein „Refactoring-Browser“ kann das**
  - **Oder eben manuell:**
    1. **Alte neben neuen Methoden**
    2. **Alte ruft neue Methode auf**
    3. **Alte Methode als „veraltet“ markieren**
    4. **In Unterklassen implementieren**
    5. **`Zeichne()` von Figur löschen**
    6. **Alte Aufrufe mit Werkzeug ändern**

```
public void zeichne (int groesser)
```

```
Public class Figur
  {
  public void zeichne (int groesser)
    { ... }

  public void zeichne ()
    { ...
    This.zeichne (1);
    }
  }
```

`Alt!`

Einfacher Entwurf

- **Einfachheit ist Trumpf: Unnötige Mehrarbeit vermeiden**
  - **kein „Vorausschauen“**
- **Einfacher Entwurf ist**
  - **einfach zu implementieren**
  - **einfach zu ändern**
  - **einfach zu testen**
  - **einfach zu verstehen**

OK. Aber was einfach bedeutet, ist nicht so einfach!

- **Vor Implementierung kleinen Entwurf diskutieren**
  - **Ideen kurz skizzieren, eine Idee auswählen**
  - **Wichtig: nicht lange aufhalten**
- **bei Streitigkeiten mehrere Ideen verfolgen**
  - **alle Kandidaten** kurz **ausprobieren**
  - **besten übernehmen, andere verwerfen**

Einfacher Entwurf

- **Heißt das: „Eínfach drauf los?“**
  - **Nein: Architektur hilfreich; exzessives Entwerfen dagegen nicht**
  - **Aber immer Möglichkeit zum Optimieren**
  - **Design läuft stets mit!**
- **XP-Designs sollen sein:**
  - **auf einen Aspekt zugeschnitten, nicht allumfassend**
  - **Auf Änderung angelegt – da man nicht weiß, was kommt**
- **Kriterien für Einfachen Entwurf**
  - **Code und Testfälle sollen alles verständlich machen, was drin steckt (z.B. durch gute Bezeichner)**
  - **Kein duplizierter code, keine Redundanz**
  - **So wenige Klassen wie möglich**
  - **So wenige Methoden wie möglich**
  - **Aber: alle Unit Tests laufen**

Einfacher Entwurf

- **Einer der häufigsten Einwände überhaupt:**
  - **„Wenn man nicht gleich mit einplant, dass ..., wird man sich später sehr viel schwerer tun“**
    - **Das lernt man seit jeher im Software Engineering**
  - **Hier braucht man einen XP-Coach, um bei der Stange zu bleiben**
- **Oft wirklich hilfreich: Standardarchitektur für Bereich**
  - **Die ergänzt das („Flexibilität durch tragende Struktur“)**
  - **Darin geht es dann gut**
  - **Aber natürlich: Refactorings/Aufräumen nicht zu lange aufschieben**
- **Empfehlungen**
  - **Ein gutes Design geht auf ein Whiteboard (oder zwei)**
  - **Keine Designsitzung über einen halben Tag; 3-5 Leute**
  - **Schlechte Designs rasch ändern (Refactoring)**
    - **Dadurch auch Performance immer im grünen Bereich**

Gemeinsame Code-Verantwortung

- **Jeder darf jederzeit an jeder Stelle Code ändern**
  - **Auch den, den anderer geschrieben hat!**
  - **notwendig wegen Aufteilung der Arbeit in Story Cards**
    - **Den „Verursacher“ ausfindig zu machen, wäre zu aufwändig. Und wozu?**
    - **Zumal, wenn der nur Teilzeit im Projekt hat: man müsste lange warten**
  - **Voraussetzung für Refactoring**
- **Schützt gegen personelle Ausfälle**
  - **jeder kann für jeden einspringen**
  - **Aber oft nicht über eine Teilgruppe hinaus: Spezialisierungen**
  - **Aber nicht personalisierte Spezialisierungen!**
- **Sitzordnung gleich Code-Struktur? Großraumbüros!**
  - **Wichtig: (möglichst professionelles) Versionsverwaltungssystem**

Code Standards

- **Eine relativ *konservativ-agile* Praktik!**
- **Voraussetzung für gemeinsame Verantwortung**
  - **Leserlichkeit**
- **Struktur**
  - **Einrückung**
  - **Klammerung**
  - **Methodenlänge**
- **Namenskonventionen**
  - **Großschreibung**
- **Kommentare**

Code Standards

- **Legen äußere Form des Quelltexts fest**
  - **Damit sieht Code einheitlich aus (*wie*, ist nicht so wichtig)**
    - **Ebene: wo steht { ?**
  - **Qualitätskriterium: Autoren nicht am Code unterscheidbar**
  - **Das ist für Pair Programming wichtig: keine Umgewöhnung**
- **Ohne Code Standards geht nichts**
  - **Die reduzieren unnötige Lern- und Verständnishürden**
- **Dabei beachten**
  - **Richtlinien müssen kurz sein (unter 10 Seiten)**
  - **Beispiels-Quellcode**
  - **Schreiben Sie im Imperativ!**
  - **Bestehende Richtlinien ruhig mutig anpassen**
  - **Früher sagte man: Regeln müssen automatisch prüfbar sein**

Kontinuierliche Integration

- **Integration mehrfach am Tag, mit optimistischer „Sperre“**
  - **Jeder hat immer aktuellen Code**
  - **Doppelte Arbeit wird vermieden**
- **Voraussetzung für gemeinsame Code-Verantwortung**
  - **Parallele Arbeit**
  - **Eventuell Refactorings**
- **Jederzeit lauffähige, getestete Version**
  - **Feedback durch Kunden**
  - **Basis für Entscheidungen**
  - **Rückfallposition**

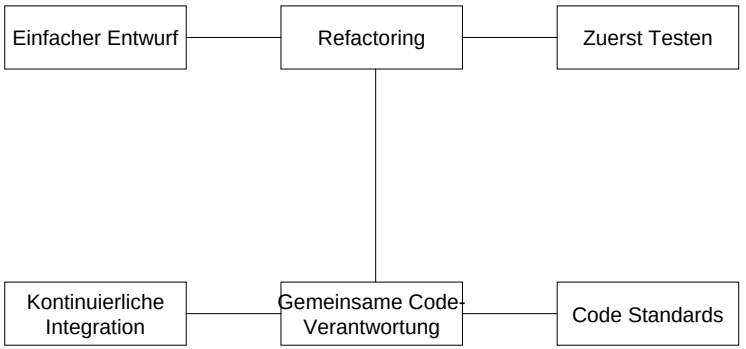
Kontinuierliche Integration

- **Integration mehrfach am Tag, mit optimistischer „Sperre“**
  - **Jeder hat immer aktuellen, lauffähigen Code, weil jeder sofort Änderungen integriert (*kleine* Änderungen!)**
    - **Wenn alle Tests einwandfrei: drinlassen.**
    - **Sonst: rücksetzen, alle Spuren tilgen**
  - **Eigener Integrationsrechner - wird dauernd gebraucht (gut 10 Mal/Tag)**
    - **Aber nur einer! Es darf nur eine Referenzintegration geben.**
  - **Konflikte werden gleich beim Einspielen gesucht und behoben, nicht beim Herausholen (optimistisch)**
  - **Integration muss technisch sehr schnell gehen: daily statt nightly build**
    - **Schneller Rechner**
    - **Gute Source-Code und Versions-Verwaltung**
    - **Evtl. Komponentenstruktur (nicht alles neu kompilieren)**
- **Jederzeit lauffähige, getestete Version**
  - **Feedback durch Kunden**
  - **Refactorings werden kleiner**
- **Bewährte, tragende Stütze von XP**

Kontinuierliche Integration
Empfehlungen

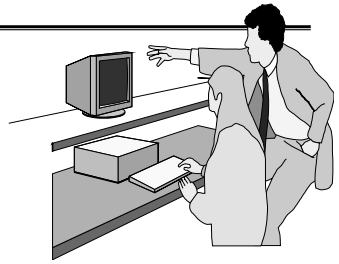
- **Mindestforderungen fürs Einspielen**
  - **Vorversion muss gesichert sein**
    - **Falls doch was schief geht**
  - **Alle vorherigen Änderungen müssen eingespielt sein**
    - **Auch bei nicht-zentralem Integrationsrechner**
  - **Ggf. auftretende Konflikte wurden aufgelöst**
  - **Alle Testklassen sind erfolgreich durchgelaufen**
  - **In der Zwischenzeit hat niemand etwas anderes eingespielt**
- **Dann ist ein Source Code Management System nicht unverzichtbar**
- **Geschwindigkeit ist wichtig**
  - **Zwei Stunden für einen Integrationsvorgang ist viel zu viel**
  - **Wenige Minuten ist besser, aber immer noch langes Warten**

Praktiken Software



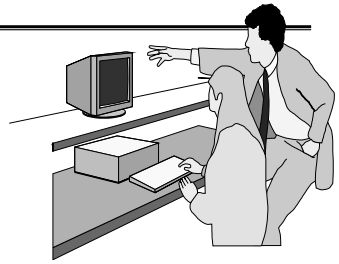
Nach Jan von Hunnius, DaimlerChrysler, 2002

Pair Programming



- Metapher: Fahrer und Beifahrer
- Jedes Paar arbeitet an einer Story Card
  - oder Task Card
  - häufiges Abwechseln der Rollen
  - möglichst tägliches Tauschen der Paare
- Ständiges Review
  - Vermeidet vor allem Flüchtigkeitsfehler
- Ständiges Lernen voneinander
  - Hebt Entwickler kontinuierlich auf ein einheitliches Niveau
- Effektiv und effizient
  - nach Studien
  - nach eigener Erfahrung

Pair Programming



- Jedes Paar arbeitet an einer Story Card
  - oder Task Card
  - Nach dem 4-Augen-Prinzip
- Pair Programming heißt: Keine Arbeitsteilung!
  - Tastatur geht schnell hin und her (20 Min)
- Auch die Paar-Zusammenstellung wechselt ständig
  - Spezialwissen verteilt sich so im Team („Truck-Factor“)
  - Einheitlicherer Code
  - Mindestens 50% gute Leute sind nötig
  - Das hat aber Grenzen: Spezialisten und Domain-Subteams
- Sparpotenzial: Rechnerkauf
  - Aber gute Idee: Arbeitsplatz- vs. Pair-Rechner

Pair Programming

- Programmieren in Paaren: Wirkungen
  - Zwang zur Disziplin (keine emails, keine Telefonate!)
  - Zwang zur Teamfähigkeit (mit allen können)
    - Auch soziale Kontrolle in Stresszeiten
  - Stärkere Konzentration auf die Aufgabe
  - Lerneffekt
    - Meist erklärt der mit der Tastatur, warum er das tut, was er tut
    - Der Schlechtere lernt schnell hinzu
    - Aber der bessere wird langsamer
- Nebenwirkungen
  - Zu viele Paare auf zu engem Raum: schwierig
  - Aber zwei oder drei sind ok; Beifahrer hören sich gegenseitig
  - Mobiliar muss so aufgestellt sein, dass Wechsel einfach ist

Pair Programming
Empfehlungen

- Pair Programming lernen
  - Ist nötig! Es dauert, sich daran zu gewöhnen
  - Ständiges Feedback: was ist gut gelaufen, was weniger?
- In der Durchführung
  - Pausen nicht am Arbeitsplatz
  - Unterbrechungen minimieren
  - Konzeptdiskussionen begrenzen
  - Management muss explizit verdeutlichen, dass es Pair Programming will
  - An das geeignete Mobiliar denken (Pairs ermöglichen)
- Wenn einer sich nicht einfügt
  - Kann man ihn evtl. nicht zwingen
  - Möglichst aus dem Team nehmen
  - Lange nicht so gut: einer ohne Pairs; dann aber viele Reviews

40 Stunden-Woche

- **Grundidee: Es gibt eine Grenze**
  - **Mehr Arbeiten bringt weniger**
  - **In Deutschland auch: „35 Stunden-Woche“**
  - **Oder: „5 h/Tag Pair Limit“**
  - **Limit für effektive, effiziente Arbeit**
- **Überstunden (durch Zeitdruck) führen zu**
  - **Fehlern im Code**
  - **Nachlässigkeiten im Prozess**
  - **Fehlende Testfälle**
  - **Schlechte Architektur (kein Refactoring)**
- **Wegen IG Metall und Betriebsrat wohl kaum ein Thema**
  - **(oder doch?)**

40 Stunden-Woche

- **Für Entwickler in Deutschland ein Problem?**
  - **Vielleicht**
  - **Oft aber für „Berater“: Externe Unterstützer**
    - **Haben oft besonders abwegige Zeitvorgaben**
    - **Woher? Von Managern, die das normal finden**
- **Fazit**
  - **Heiße Phasen sind normal im SW-Projekt**
    - **Dagegen hat auch XP nichts; nur nicht ständig**
  - **Nicht alle machen Überstunden**
    - **Erstmal Überblick verschaffen**
  - **Auch mal entspannende Zeiten einplanen**

Praktiken Menschen

40 Stunden-Woche

Pair Programming

Nach Jan von Hunnius, DaimlerChrysler, 2002

Prinzip und Denkweise von Agilen Methoden

nach Frühauf, Conquest 2001	**Bisheriger Ansatz**	**Agiler Ansatz**
Mitwirkung des Kunden	unwahrscheinlich	kritischer Erfolgsfaktor
Etwas Nützliches wird geliefert	erst nach einiger (längerer) Zeit	mindestens alle sechs Wochen
Das Richtige entwickeln durch	langes Spezifizieren, Vorausdenken	Kern entwickeln, zeigen, verbessern
Nötige Disziplin	formal, wenig	informell, viel
Änderungen	erzeugen Widerstand	werden erwartet und toleriert
Kommunikation	über Dokumente	zwischen Menschen
Vorsorge für Änderungen	durch Versuch der Vorausplanung	durch „flexibel Bleiben“

Anwendungsspektrum agiler Methoden

? *Sicherheitskritische* Embedded Software
? Entwicklung von *reinen* Software*architekturen*

+ „Oberflächen-intensive Systeme“
  – in Verbindung mit Usability Engineering

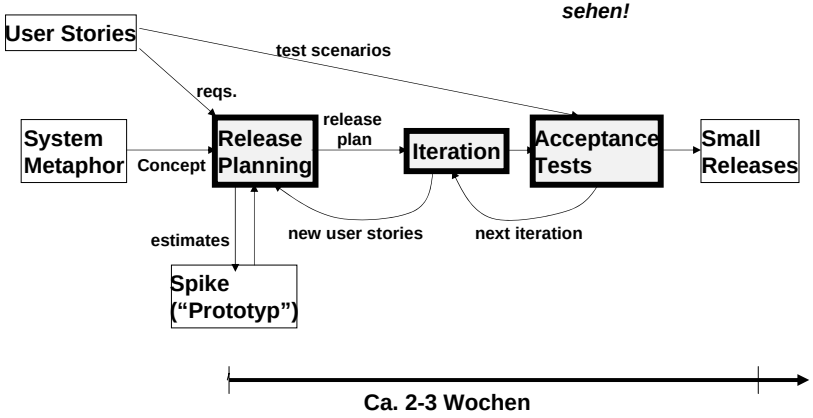
✦ Web-basierte Informationssysteme mit folgender Charakteristik:
  – Unklare Anforderungen – hoher Zeitdruck
  – Schnelle Auslieferung wichtigster Funktionen
  – Inkrementeller Ausbau akzeptiert
  – schrittweise Freigabe von Entwicklungsbudget
  – Hohe Qualität von Anfang an

+ Kooperationsunterstützende IT-Systeme
  – Gemeinsame Informationsräume
  – Koordinierte Interaktion
  – Geschäftsprozessunterstützung

+ dazu gehören: eBusiness-, Telematik-Anwendungen

Abläufe in XP (Prozesse) *auf Basis der Praktiken*

Viele Praktiken sind hier nicht explizit zu sehen!



Frei nach J. Donovan Wells, Copyright 2000

Aufgabe: Wie agil sind Sie schon?

- Agile Methoden behaupten nicht, etwas ganz Neues zu sein
- Viele Praktiken kennt man
  – Unter anderem Namen
  – Ohne eigenem Namen
  – Aus eigener Erfahrung
- Bitte teilen Sie die Praktiken von XP in drei Gruppen:
  – „schon selbst ziemlich genau so gemacht“
  – „schon mal etwas ähnliches probiert“ (was?)
  – „noch nie in den Sinn gekommen oder gleich wieder verworfen“ (wieso?)
- Berichten Sie kurz, was bei Ihnen wieso in welcher Gruppe gelandet ist!

Zusammenfassung

- Die Entwicklung geht zu anderen Ansätzen
  – Keine großen Prozesse
  – Aufmerksamkeit für die persönliche Ebene
  – Inkrementell
  – Mit anderen Werten
- Grundwerte, Prinzipien, Praktiken
  – Ideologie, Leitsätze und konkrete Anleitung
- XP: der bekannteste Vertreter im Überblick
  – Es gibt auch SCRUM und diverse andere agile Methoden!
- Vergleich der Denkansätze

Literatur

- ***Extreme Programming Explained: Embrace Change*, Kent Beck, Addison-Wesley, 1999.**
- ***Refactoring: Improve the Design of Existing Code*, Martin Fowler, Addison-Wesley, 1999.**

- **http://www.extremeprogramming.org**
- **http://www.xprogramming.com**
- **http://www.pairprogramming.com**
- **http://www.refactoring.com**

- **http://www.junit.org**
- **http://www.objectmentor.com**