

Einführung in die Agilen Methoden

- Teil 1 -

Stephan Kiesling, Jil Klünder

{stephan.kiesling, jil.kluender}@inf.uni-hannover.de

19.11.2015

Ziele der heutigen Veranstaltung

- Verstehen, was agile Software-Entwicklung bedeutet
 - Später im Unternehmen wissen, wovon gesprochen wird
 - Später selber Veränderungen auswählen, planen und einführen können
- Praktiken und Werte der agilen Programmierung kennen lernen
- Verstehen, warum die Praktiken angewendet werden



Essentiell, um agiles
Vorgehen auf eigene
Situation anzupassen

Plakativ, konkret,
sofort anwendbar

Agile Methoden

- Wir wollen die Werte so gut wie möglich umsetzen
 - Dafür brauchen wir die Praktiken
- Wir wollen die negativen Sachen loswerden, und die positiven Sachen erhöhen
- Bisher bekannt: vor allem die Praktiken

Manifesto for Agile Software Development

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
Mike Beedle
Arie van Bennekum
Alistair Cockburn
Ward Cunningham
Martin Fowler

James Grenning
Jim Highsmith
Andrew Hunt
Ron Jeffries
Jon Kern
Brian Marick

Robert C. Martin
Steve Mellor
Ken Schwaber
Jeff Sutherland
Dave Thomas

- Kundenzufriedenheit hat höchste Priorität
 - Wird durch frühe und häufige Auslieferung laufender Software sichergestellt
- Einfachheit ist das wichtigste Konstruktionsprinzip
 - Möglichst wenig unnötige Arbeit
 - Nicht auf Vorrat arbeiten
- Auch späte (und spontane) Änderungen werden willkommen geheißen
 - Änderungen bedeuten Verbesserungen für den Kunden

- Manager und Entwickler arbeiten jeden Tag zusammen
- Projekte werden von motivierten Mitarbeitern getragen und durchgeführt
 - Wir schaffen eine möglichst ideale Umgebung und haben Vertrauen in die Entwickler
- Direkte „Mensch-zu-Mensch“-Kommunikation
 - Dadurch werden die Informationen effektiv und effizient ausgetauscht
- Jeder Mitarbeiter überprüft und verbessert regelmäßig seine eigene Leistung
- Feedback als Grundprinzip

Prinzip und Denkweise von Agilen Methoden

	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	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 Flexibilität

*nach Frühauf,
Conquest 2001*

Warum sollte man die Werte kennen?

- Die Praktiken erlauben einen schnellen Start in der agilen Entwicklung,
- aber die Praktiken...
 - ... müssen oft angepasst werden.
 - ... erreichen schnell ihre Grenzen.
- Die Werte geben stattdessen die Grundrichtung vor
 - Darauf basierend kann man selbst kreative Lösungen finden
 - Man kann daran festmachen, was schief läuft

eXtreme Programming

Eine agile Methode

Was heißt hier **extrem**?

- Agile Entwickler sind keine Anarchisten
- eXtreme Programming erfordert **extreme** Disziplin
- Bewährte Praktiken aus der SW-Entwicklung...
... werden ins **Extreme** geführt.

Was heißt hier **extrem**?

- Beispiel: Code-Reviews haben sich bewährt...
... daher: **Ständige** Code-Reviews durch Pair Programming.
- Beispiel: (Frühe) Tests haben sich bewährt...
... daher: **Ständig** mehrmals täglich testen.
...daher: **Vor dem Implementieren** die Testfälle erstellen (Test First).



- Grundwerte
 - Bestimmen die Grundrichtung
 - Langfristige und ganzheitliche Denkweise
 - Verfolgen die allgemeinen agilen Werte
- Prinzipien
 - Werte an sich sind sehr vage
 - Unterschiedliches Verständnis von Werten im Detail
 - Prinzipien füllen vage Werte mit konkreteren Vorstellungen
 - Dienen zur Guidance, z.B. bei der Entscheidung zwischen 2 Alternativen
- Praktiken
 - Prinzipien zeigen, was man erreichen will
 - *Wie* die Prinzipien erreicht werden, wird durch Praktiken angegeben
 - Konkrete Anwendung

Kommunikation

Einfachheit

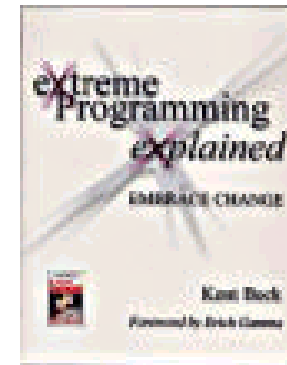
Feedback

Mut

Respekt

Prinzipien im XP

- Im Kern sind es bei XP 12, davon fünf zentrale:
 - Unmittelbares Feedback
 - Einfachheit anstreben
 - „Vorbeugen wollen“ ist sinnlos
 - (Das lernt man sonst im Studium anders!)
 - Inkrementelle Veränderung
 - Um Seiteneffekte von Änderungen zu beschränken
 - Damit Risikominimierung
 - Veränderung wollen
 - Technisch und in der Arbeitsweise
 - Qualitätsarbeit
 - Will jeder Entwickler leisten
 - Muss man ermöglichen
 - Die Maßstäbe setzt aber der Anwender/Auftraggeber!



Kent Beck

Weitere XP-Prinzipien – Lebensregeln

- Geringe Anfangsinvestition: großes Anfangsbudget ist zu riskant
- Auf Sieg spielen (play to win)
 - Nicht „Fehler vermeiden“, sondern „Erfolge anstreben“ (trotz Fehlern)
 - Was nicht mehr gewinnen kann, wird gestoppt
- Mit leichtem Gepäck reisen (Travel Light)
 - Mehr Hilfsmittel im Rucksack -> Expedition geht langsamer voran

„XP ist eine Disziplin der Software-Entwicklung, die sich durch

Einfachheit, Kommunikation, Feedback und Mut

auszeichnet. Dabei wird den jeweiligen Rollen [...] hohe Bedeutung eingeräumt und den Personen in diesen Rollen außerdem

Schlüsselrechte und –verantwortlichkeiten

zugeschrieben.“

Aus: Ron Jeffries et.al. Extreme Programming Installed

Rollen in XP

- **Kunde**
 - Wählt Funktionalität nach ihrem Geschäftswert aus
 - Priorisiert Features
 - Bestimmt Akzeptanztests
- **Programmierer**
 - Entwerfen und programmieren die Software
 - Schätzen den Aufwand für Features
- **Coach**
 - Kunde und Programmierer werden zu einem Team vereint
 - Ebnet dem Team den Weg

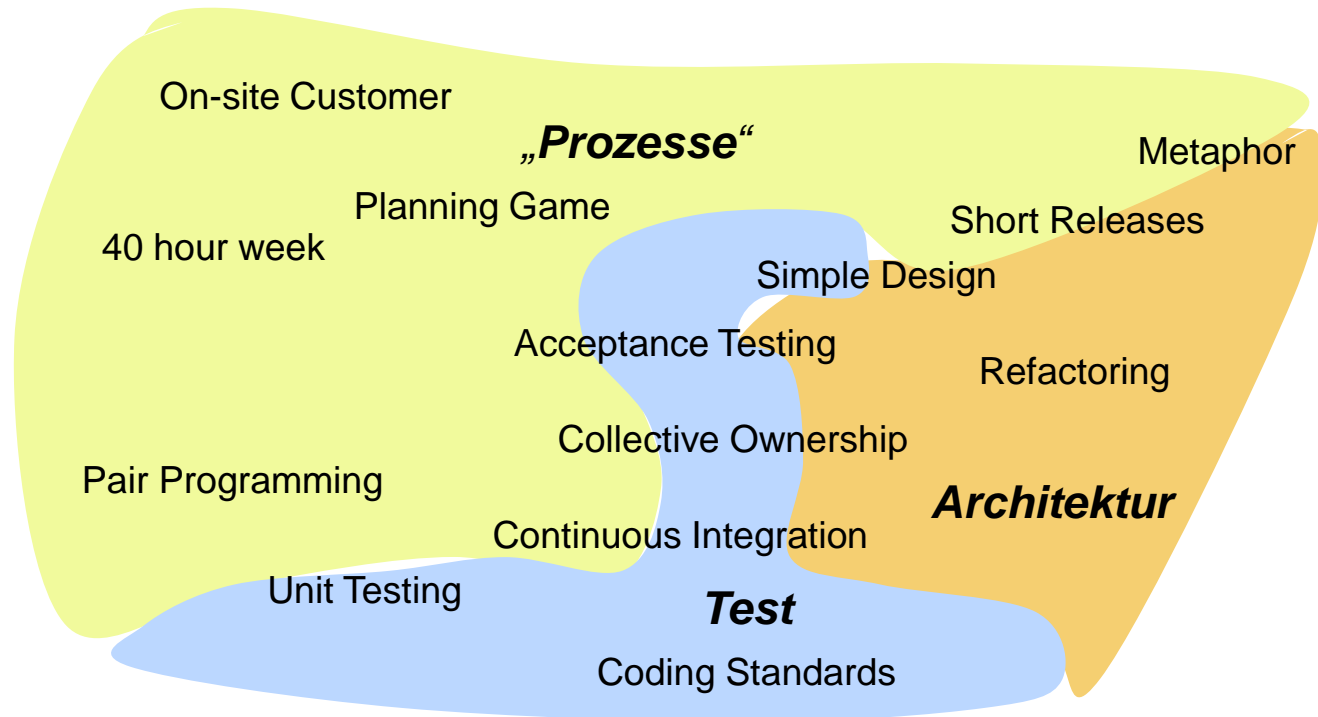
Rechte des Kunden und des Coaches

- Recht auf einen Gesamtplan
 - Was kann zu wann zu welchen Kosten fertiggestellt werden?
- Recht auf maximalen Wertezuwachs
 - Und das aus jeder Programmierwoche
- Recht auf ein lauffähiges System
 - Immer / Durch Tests belegt
- Recht auf Änderungen
 - Ändern von Prioritäten und Features
- Recht auf Information
 - Z.B. über terminliche Änderungen

Rechte der Programmierer

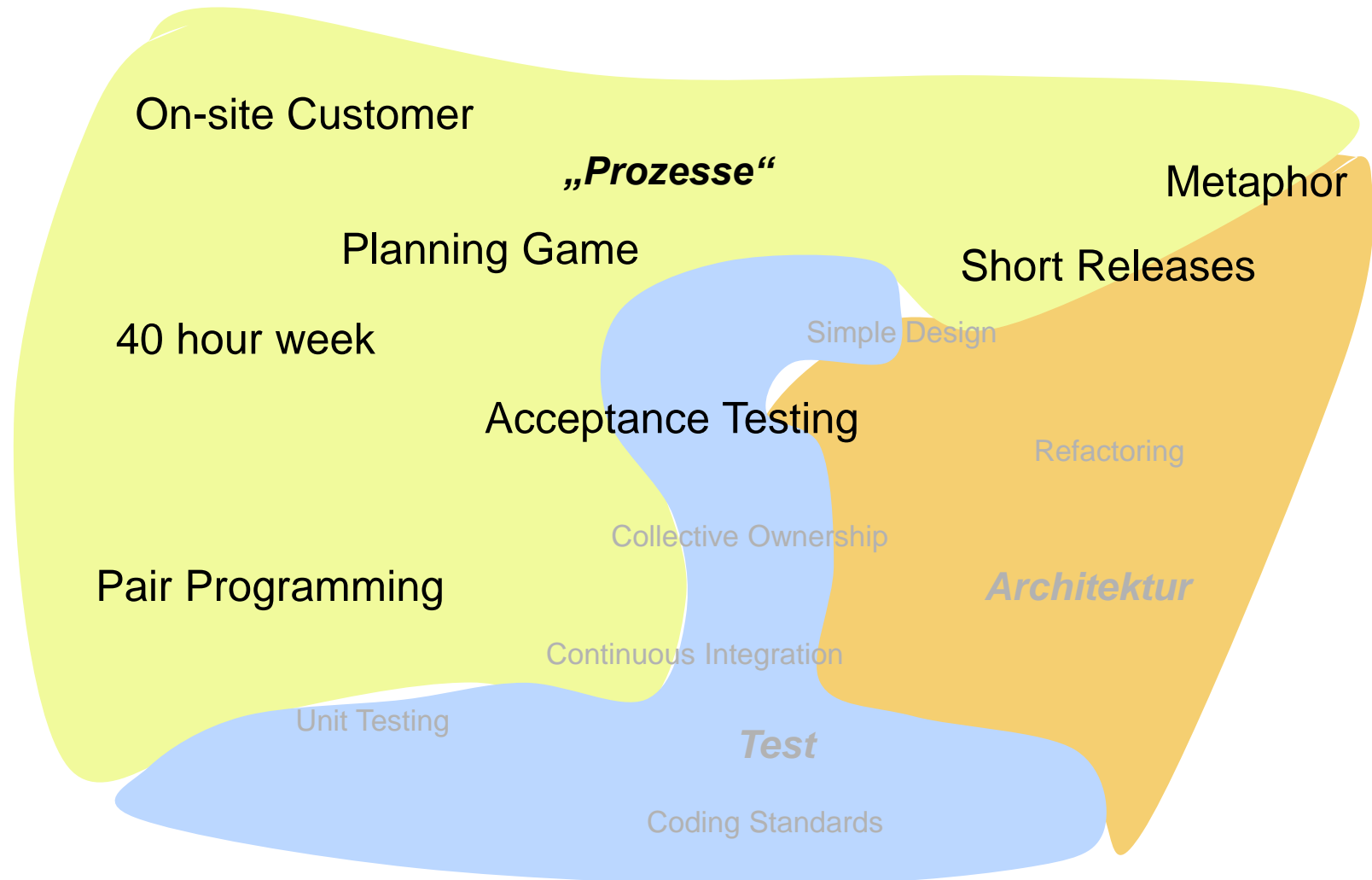
- Recht zu wissen, was als nächstes benötigt wird
 - Anhand von klaren Prioritäten
- Recht darauf, Qualität produzieren zu können
 - D.h. wenn es eng wird, wird an der Funktionalität gespart
- Recht auf Hilfe
 - Von jedem (andere Programmierer, **Kunde, Manager**)
- Recht, Einschätzungen abzugeben und zu ändern
 - Z.B. Schätzungen, die sich als unrealistisch herausstellen
- Recht darauf, Verantwortlichkeiten zu akzeptieren
 - Nicht zugewiesen zu bekommen

- Praktiken
 - Verstärken sich gegenseitig
 - Zusammenspiel von Prozess, Testen und Architekturfragen



Praktiken von „eXtreme Programming“

- Ausschnitt: Kommunikation



- Kunde ist über die gesamte Projekt-Laufzeit vor Ort
 - Zumindest leicht und jederzeit erreichbar
- Schreibt Story Cards (Entwickler dürfen helfen)
 - Ständige Weiterentwicklung der Anforderungen
- Beantwortet Fragen der Entwickler, auch unter Ungewissheit
 - Kein Zeitverlust bei Entscheidungen
 - Keine Fehlentscheidungen durch Entwickler
- Voraussetzung: Kompetenz und Entscheidungsgewalt

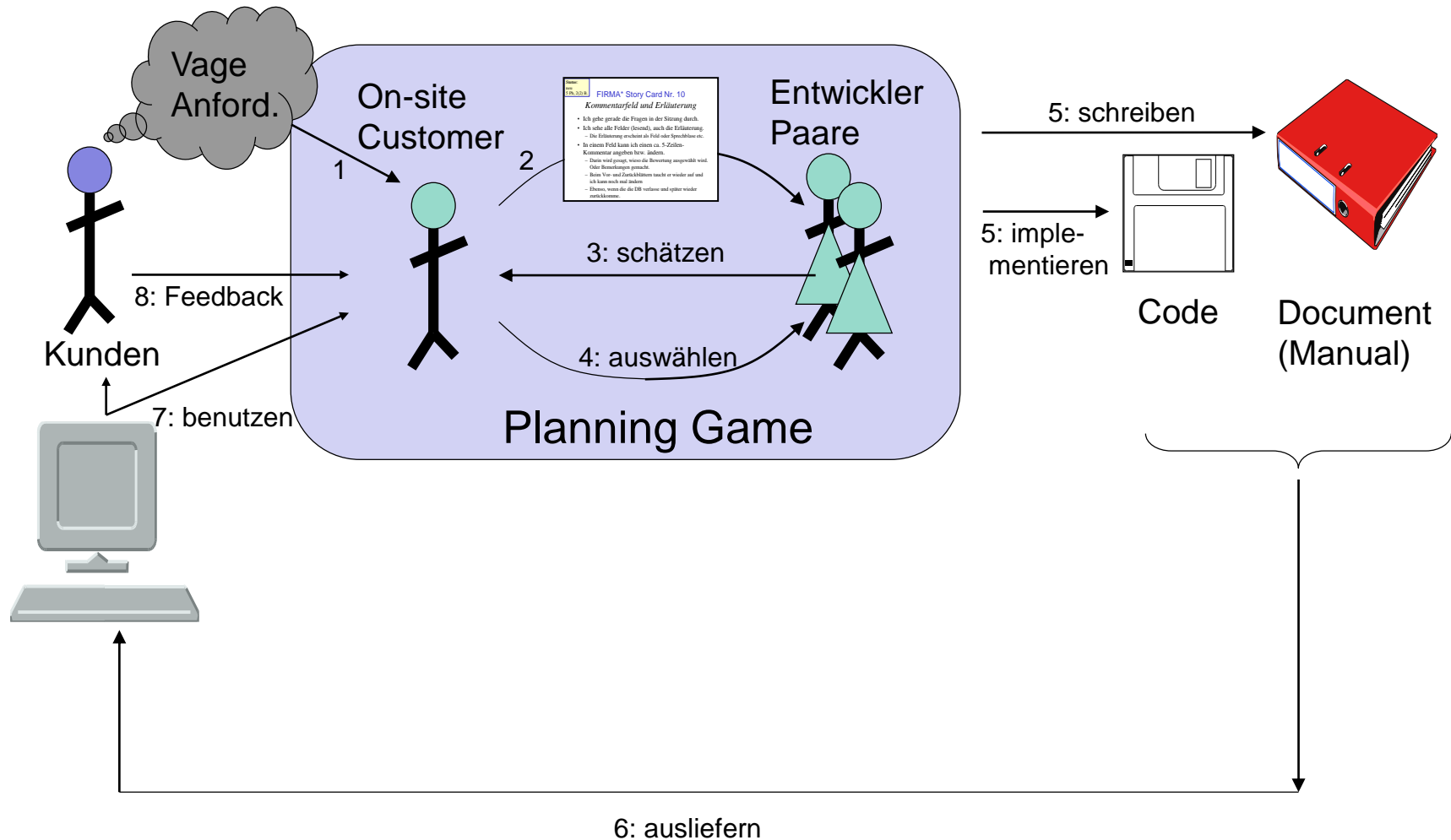


- Verhaltensweisen von Kunden
 - Oft werden Soll- mit Istprozessen verwechselt
 - Altsystem wurde lange „trickreich umgangen“
 - Stellen sich oft nur lokale Änderungen dazu vor
- Daher: Entwickler helfen Kunden beim Story Card-Schreiben
 - Erinnern an die Prüfbarkeit
 - Fragen nach drastischeren Alternativen
- Ganz wichtig für den Erfolg
 - Anwenderkontakt ist unverzichtbar für XP
 - Wenn Entwicklerteam fachliche Frage hat: nicht selbst raten, fragen!
 - Anwender entscheidet über Alternativen

On-site Customer – Tipps

- 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“: Anwender/Kunden?
- Wichtige Unterscheidung
 - *Anwender* beantwortet fachliche Rückfragen
 - *Kunde* priorisiert Story Cards
- Häufig versuchen Projekte, den Kunden zu ersetzen
 - Kunden haben wenig Zeit
 - Kompetente, entscheidungsfähige erst recht
 - On-Site Customer sitzt zeitweise untätig im Projekt herum
 - Konsequenz: Selbst „On-Site Customer“ spielen!

Planen, Arbeiten und Feedback



Planning Game / Planungssitzung

- Sitzung zur Planung der nächsten Iteration
- Ablauf
 1. Anforderungen auf Story Cards sammeln (zusammen)
 2. Aufwand schätzen (Entwickler)
 3. Story Cards für Iteration auswählen und priorisieren (Kunde)

Planning Game / Planungssitzung

1. Anforderungen auf Story Cards sammeln (zusammen)
2. Aufwand schätzen (Entwickler)
3. Story Cards für Iteration auswählen und priorisieren (Kunde)

ID 1: Der Benutzer kann nach Büchern suchen.

Im Suchfeld können Titelteile, Autoren, ISBNs eingetragen werden.

- **Story Cards beschreiben umgangssprachlich, was der Benutzer mit dem System tun kann**
- **Informell, evtl. mit kleinen Skizzen...**
 - Story Cards sollen zur Kommunikation anregen, nicht alle Details erschlagen
- **Story Cards bilden die Anforderungen. Es gibt keine gewohnte ausführliche Spezifikation**

Planning Game / Planungssitzung

1. Anforderungen auf Story Cards sammeln (zusammen)
2. Aufwand schätzen (Entwickler)
3. Story Cards für Iteration auswählen und priorisieren (Kunde)

ID 1: Der Benutzer kann nach Büchern suchen.

4

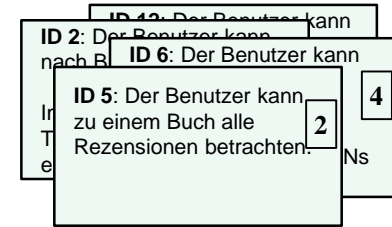
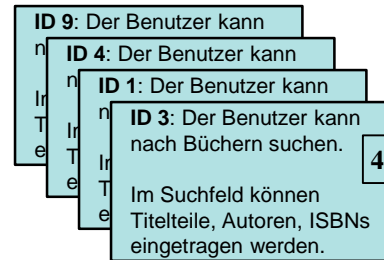
Im Suchfeld können Titelteile, Autoren, ISBNs eingetragen werden.

- **Schätzung durch Erfahrung (ähnliche Karten)**
- **Muss nicht in Stunden sein**
 - Abstraktes Maß vermeidet Fehleinschätzung von Brutto- und Netto-Stunden
 - Z.B. Gummibärchen
- **Nachfragen an Kunden (interaktive Anforderungskklärung)**
- **Große Karten aufteilen**
 - Schätzung wird sonst unzuverlässig
- **Oder „Spike“ machen**
 - Prototyp mit festem Aufwand



Planning Game / Planungssitzung

1. Anforderungen auf Story Cards sammeln (zusammen)
2. Aufwand schätzen (Entwickler)
3. Story Cards für Iteration auswählen und priorisieren (Kunde)



- **Kunde wählt die wichtigsten Story Cards für die nächste Iteration aus**
- **Priorisierung: die Karten, die für den Kunden am meisten Wert schaffen, werden am wichtigsten priorisiert**
 - Das ist eine Verpflichtung an den Kunden!
- **Limit: bisherige Produktivität des Teams**
- **Wichtig: Jede Iteration hat die gleiche Dauer**

Beispiele: Story Card

Storycard

Titel Vom Studenten zu Arbeit springen		Nr 5
Stand/Autor 1	Prio H	Aufwand 2
Abgenommen (Kunde)	Ausführendes Paar	Eingecheckt

Ablauf (Use-Case/Schritte)

aus der Studentenübersicht sind
die zug. Arbeiten erreichbar


- Titel der Arbeit mit anzeigen

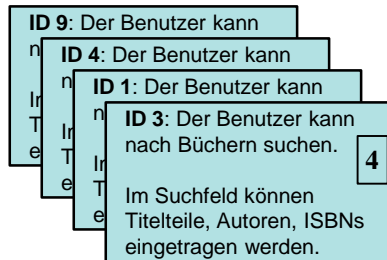
Storycard

Titel Status-Panel für Status „noch Endvortrag“		Nr 47
Stand/Autor I. Eric Krauss	Prio A	Aufwand 1
Abgenommen (Kunde) KSt 12.06.07 17:34	Ausführendes Paar	Eingecheckt 40

Ablauf (Use-Case/Schritte)

AFTER-PRESENTATION

- 1 * - Exzellente ExcelList Task
- 2 * - Arbeit beurteilen Grade Task
- 3 * - ~~PDF die Arbeit~~ Metadaten der Arbeit im PDF überprüfen Check Data Task
- 4 * - PDF online stellen  ~~PDF~~ ohne Pub. Publish Thesis Task
- 5 * - Weiterleitung Beurteilung an Zweitprüfer R Forward Evaluation Task
- 6 * - Beurteilung Zweitprüfer erhalten Receive Second Evaluation Task
- 7 * - Unterlagen abhelfen File Documents Task
- 8 * - Student Note münden Announce Evaluation Task



- Und dann?
- Entwicklerpaar nimmt sich oberste Story Card vom Stapel und fängt an, diese umzusetzen
- Kunde beschreibt Kriterien für Akzeptanztests auf der Rückseite
- Karten bilden Grundlage für Gespräche mit dem Kunden, um Details festzulegen
- Zum Schluss: Karten behalten oder wegwerfen?

Planning Game / Planungssitzung

- 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
- Abhilfe: Gold Cards
 - Ein „Joker“ für einen lange angespannt arbeitenden Mitarbeiter
 - Muss mal (z.B. ein, zwei Tage lang) an einer neuen technischen Frage knabern
 - Darf rumprobieren und Grundlagen legen
 - In der Schätzung gehen Gold Cards *automatisch* ein, sofern regelmäßig durchgeführt

Acceptance Testing

- Jede Story Card muss am Ende vom On-Site Customer abgenommen werden
 - Zumindest informell durch On-Site Customer
 - Noch besser: Auch formelle Akzeptanztests aufschreiben

Kurze Release-Zyklen

- Iteration dauert 1-2 Wochen
- Feedback einholen
- Neue Planung & Ausrichtung
 - Eventuell neue oder geänderte Story Cards
 - Eventuell neue Aufwandsschätzung
- Fortschritt messen (Story Cards)
 - Kunde sieht Projektfortschritt
 - Man zählt Aufwands-Punkte, die für fertige Cards geschätzt waren
 - Man zählt Punkte, die noch bis Fertigstellung geschätzt werden
- Selbst bei Abbruch ist schon ein Nutzen erzeugt worden
 - Geld für Nutzen (fertige Story Cards) sinnvoll eingesetzt
 - Kein gescheitertes Projekt: irgendwas Nützliches ist da!

Kurze Release-Zyklen

- Release alle 2-3 Monate
- „Release“ = „Inkrement“
 - Erweitert bisheriges Produkt
 - Auslieferung zur *operativen* Nutzung
 - Nicht bloß rumprobieren
- Inkl. Dokumentation, Handbücher, ...
- Feedback von Kunden & Anwendern einholen

Kurze Release-Zyklen – Bedenken

- 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: operative Nutzung im Vertrag fixieren. Kunde muss nutzen

Metapher

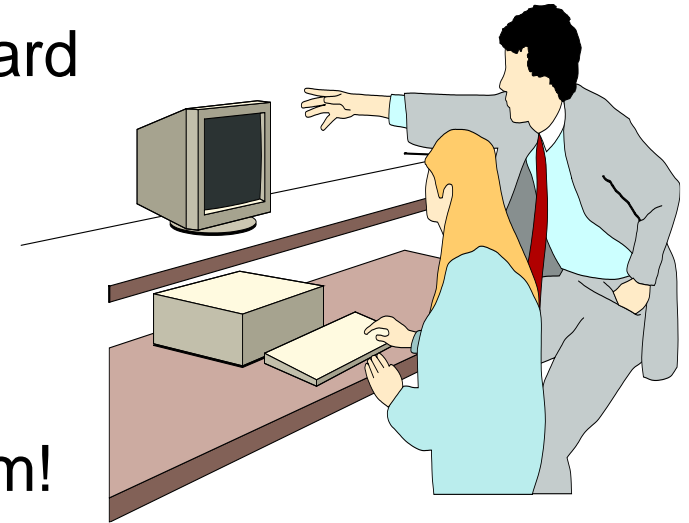
- Bildhafte Beschreibung der Grundidee des Systems
 - Beispiele: „Desktop“, „mobiles Faxgerät“, „Patienten-Assistent“
- Verständlich und verbindlich für alle Beteiligten
 - Schafft gemeinsame Terminologie
 - Zwischen Kunden und Entwicklern
 - Auch zwischen Entwicklern (Code)
 - „Richtschnur, falls gerade keine Klärung möglich“

Metapher – Herausforderungen

- Verstehen alle das gleiche unter der gewählten Metapher?
 - Woher wissen sie, ob das so ist?
- Können alle metaphorisch denken?
 - Manche sind begeistert, andere mögen es gar nicht
 - Auch Experten haben oft nur wenige Metaphern im Repertoire
- Metaphern-Finden erfordert Fantasie und Routine darin
 - Kreative Herausforderung

Pair Programming

- Jedes Paar arbeitet an einer Story Card
 - Oder Task Card
 - Nach dem 4-Augen-Prinzip
- Pair Programming heißt:
Keine Arbeitsteilung, alles gemeinsam!
 - Tastatur geht schnell hin und her (10 Min)
- Auch die Paar-Zusammenstellung wechselt ständig
 - Spezialwissen verteilt sich so im Team („Truck-Factor“)
 - Einheitlicherer Code
 - Mindestens 50% gute Leute sind nötig



Pair Programming – Wirkungen

- Zwang zur Disziplin
 - Keine E-Mails, keine Telefonate!
- Kein Schummeln
 - Wenn einer mal TestFirst „vergisst“, redet der zweite ihm ins Gewissen
- 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 dazu
 - Aber der bessere wird langsamer
- Beifahrer hören sich gegenseitig
 - Erneut Informationsverteilung

40 Stunden-Woche (besser: sustainable pace)

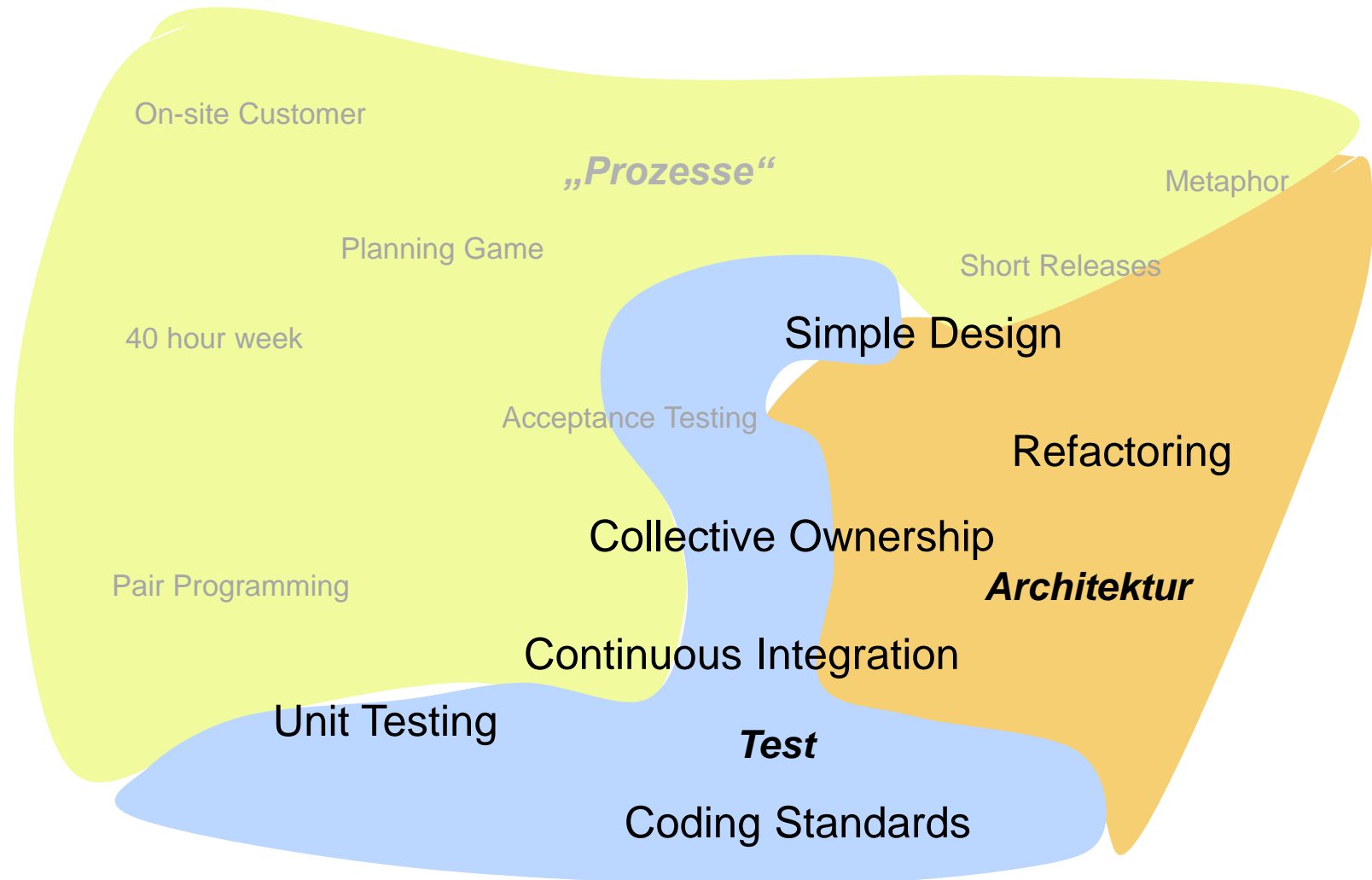
- Grundidee: Es gibt eine Grenze für intensives Arbeiten
 - Mehr Arbeiten bringt weniger
 - Oder: „5 h/Tag Pair Limit“
 - Limit für effektive, effiziente Arbeit
- Überstunden (durch Zeitdruck) führen zu
 - Fehlern im Code
 - Nachlässigkeiten im Prozess
 - Sinkender Motivation
 - Fehlenden Testfälle
 - Schlechter Architektur (Refactoring vernachlässigt)

40 Stunden-Woche – Hinweise

- Überstunden für Entwickler in Deutschland ein Problem?
 - Vielleicht nicht so sehr
 - Oft aber für „Berater“: Externe Unterstützer
 - Haben oft besonders abwegige Zeitvorgaben (Überstunden)
 - Woher? Von Managern, die das normal finden
 - Hinweis: Viele von Ihnen werden Berater werden
- Fazit
 - Heiße Phasen mit Überstunden sind normal in SW-Projekt
 - Dagegen hat auch XP nichts
 - Nur nicht ständig

Praktiken von „eXtreme Programming“

- Ausschnitt: Software



Einfacher Entwurf

- Einfachheit ist Trumpf: Unnötige Mehrarbeit vermeiden
 - Kein „Vorausschauen“, keine Strukturen „für möglichen Ausbau“, ...
 - Dieses Prinzip ist Recht und Verpflichtung!
- Einfacher Entwurf bedeutet
 - Einfach zu implementieren
 - Einfach zu ändern
 - Einfach zu testen
 - Einfach zu verstehen
- Einfacher Entwurf heißt nicht, möglichst wenig in den Entwurf zu investieren
 - Im Gegenteil, man sollte jeden Tag in das Design investieren (Incremental Design)
 - Vor Implementierung kurz Entwurf diskutieren, Ideen skizzieren, auswählen
 - Wichtig: nicht lange aufhalten, nur an gegenwärtigen Anf. orientieren
- Bei Streitigkeiten mehrere Ideen verfolgen
 - Alle Kandidaten kurz ausprobieren
 - Den besten übernehmen, andere verwerfen

**Nicht auf Vorrat
programmieren. YAGNI:
You ain't gonna need it!**

Einfacher Entwurf – konkret

- Kriterien für Einfachen Entwurf
 - Bewährte Heuristik: eliminiere duplizierten Code (Kent Beck)
 - Code und Testfälle sollen alles verständlich machen
 - Z.B. durch gute Bezeichner
 - Keine „Magic-Values“
 - So wenige Klassen wie möglich, so wenige Methoden wie möglich
 - Dennoch eleganter/sauberer Code
- 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
- Oft wirklich hilfreich: Standardarchitektur für Bereich
 - Flexibilität durch tragende Struktur
 - Aber natürlich: Refactorings/Aufräumen nicht zu lange aufschieben

Einfacher Entwurf – Tipps

- Einer der häufigsten Einwände überhaupt:
 - „Wenn man nicht gleich mit einplant, dass ..., tut man sich später schwerer.“
 - Das lernt man seit jeher im Software Engineering
 - Hier braucht man einen XP-Coach, um bei der (agilen) Stange zu bleiben

- Wenn Testen gut ist, dann ist häufig/immer Testen besser
- Idee: Test schreiben, bevor implementiert wird
 1. Test schreiben
 2. Test fehlschlagen lassen
 - Fange ich damit wirklich noch nicht vorhandene Funktionalität ab?
 3. Implementieren, bis Test erfüllt
 - Und das möglichst **einfach**!
 4. Refactoring
- Auf Unit Test Ebene
- Auf Acceptance Test Ebene

Prinzip von TestFirst: ein Dialog

Aufgabe: Java-Methode `len(int)` gibt zurück, wie viele Stellen eine `int`-Zahl hat.

Test beginnt

„`len(5)` soll 1 sein!“

```
assertEquals(1, len(5));
```

JUnit

COMPILER-FEHLER!

Was soll „`len`“ bedeuten?

Programm: Das ist einfach:

```
public int len(int zahl) { return 1; }
```

JUnit: ok. Testfall erfüllt.

Test: Na, warte!

„`len(321)` soll 3 sein!“

```
assertEquals(3, len(321));
```

JUnit: Fehler! 1 statt 3

Programm: auch nicht so schwer ...

```
if zahl < 10 then return 1 else return 3
```

JUnit: ok.

Test: Ich glaub es nicht!

„`len(12345678)` soll 8 sein!“

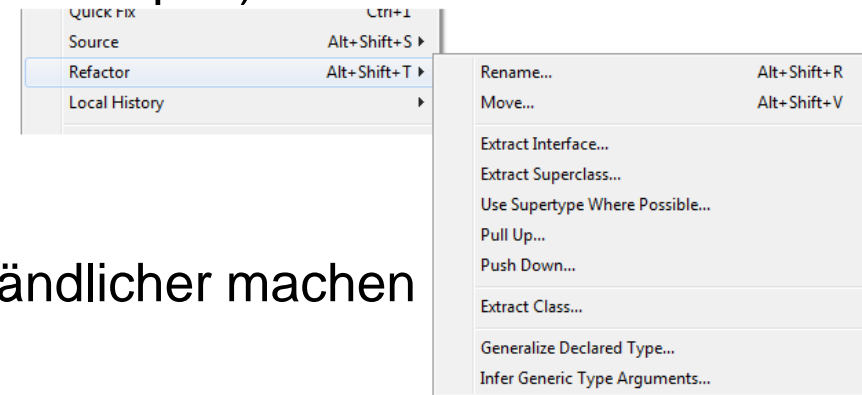
```
assertEquals(8, len(12345678));
```

JUnit: Fehler! 3 statt 8

Programm: ... ok, ich sehe das Muster:
eine Schleife: `for (i=...`

- TestFirst hilft, „*Einfaches Design*“ umzusetzen
 - Tests beschreiben, was alles funktionieren soll
 - Implementierung entlang der Tests fokussiert auf notwendige Funktionalität
 - Abschweifen in nicht benötigte Funktionen unwahrscheinlicher

- Änderung des Codes ohne Änderung des Verhaltens
 - Man macht sich anfangs Gedanken über Architektur
 - ... aber nicht unendlich lange, dann geht's los*
 - Änderungen kommen ohnehin - vorbereitet sein!
- Vorgehen nach definierten Schritten
 - Martin Fowler: Refactoring
 - Heute teilweise automatisiert (z.B. in Eclipse)
 - Umbenennung
 - Herausziehen von Interfaces
- Verbesserung der Architektur
 - Priorität: Code vereinfachen, verständlicher machen
 - Erweiterungen ermöglichen
 - Wildwuchs verhindern, Hilfskonstrukte „zurückschneiden“
- Testfälle sichern konstantes Verhalten



Collective Code Ownership

- Jeder darf alles ändern
- Jedem gehört alles
- Gewöhnungsbedürftig, aber nötig, um schnell Änderungen durchzuführen
 - Storycards laufen nicht entlang von Modulgrenzen!

Programmierstandards

- Dienen dazu, dass jeder den Code lesen und ändern kann
 - Collective Code Ownership
- Hohe Wartbarkeit des Codes
- Gute Lesbarkeit
- Hier (im Labor) gelten die Java Code Conventions
 - <http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html>
 - <http://www.oracle.com/technetwork/java/codeconventions-150003.pdf>

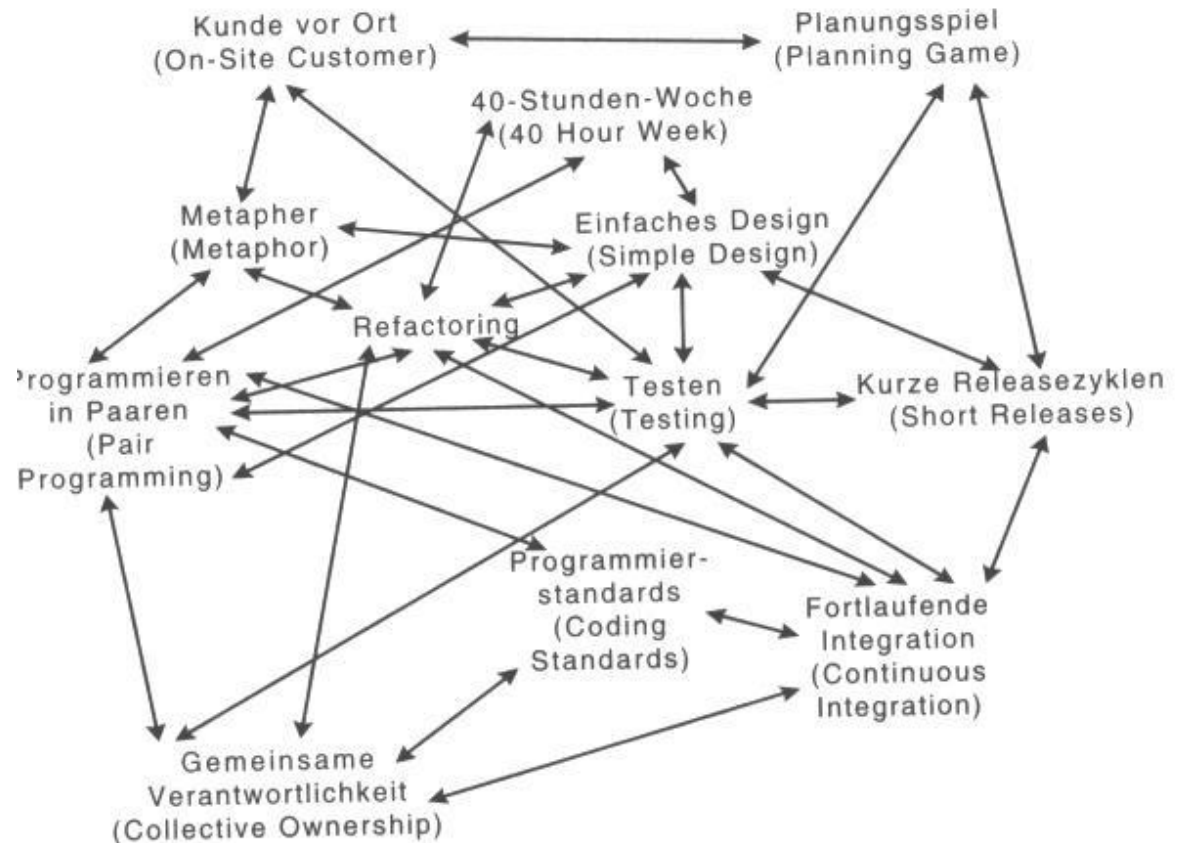
Java Coding Standards

- *Packages* haben Kleinbuchstaben und sind die umgedrehte Domain
 - Hier: de.unihannover.se.xplab15
- *Klassennamen* beginnen mit einem Großbuchstaben. Mehrere Wörter werden durch Binnengroßschreibung konkatentiert
 - Beispiel: BusinessFactory
- *Methoden, Variablen* und *Parameter* fangen mit einem Kleinbuchstaben an. Binnengroßschreibung
 - Beispiel: getName(), firstName
- *Konstanten* werden großgeschrieben. Wörter mit _ getrennt
 - Beispiel: MAX_COUNT
- *Klammern* öffnen am Ende einer Zeile und schließen am Anfang einer neuen

Kontinuierliche Integration

- Kleinschrittige Integration mehrfach am Tag mit optimistischer „Sperre“
 - Spätestens nach jeder Storycard
 - Erfordert kurze Buildzeiten
 - Jeder hat immer aktuellen Code
 - Bei zu großen Abständen gehen die Versionen zu sehr auseinander
- Jederzeit lauffähige, getestete Version
 - Dadurch stets Feedback vom Kunden
 - Basis für Entscheidungen
 - Rückfallposition
 - Refactorings werden kleiner

- Nach Kent Beck



- Ein Bild oder sechs Worte „Alles hängt von allem anderen ab“
- „Mit 80% Techniken nur 20% Nutzen!“ (sagt Beck)