universitätbonn ROOTS

# Ferientutorien
# Softwaretechnologie 2010

von Eva Stöwe und Jan Nonnen

© 2010 Eva Stöwe, Jan Nonnen

# Disclaimer

- Die Folien
  - stellen lediglich die Basis der jeweiligen Themen dar
  - erheben keinen Anspruch auf Vollständigkeit
- An mehreren Stellen wurden Sachen vereinfacht oder weggelassen.
- Wir raten dringend dazu, auch andere Quellen zur Klausurvorbereitung zu nutzen.

**Klausurrelevant sind die Vorlesungsfolien.**

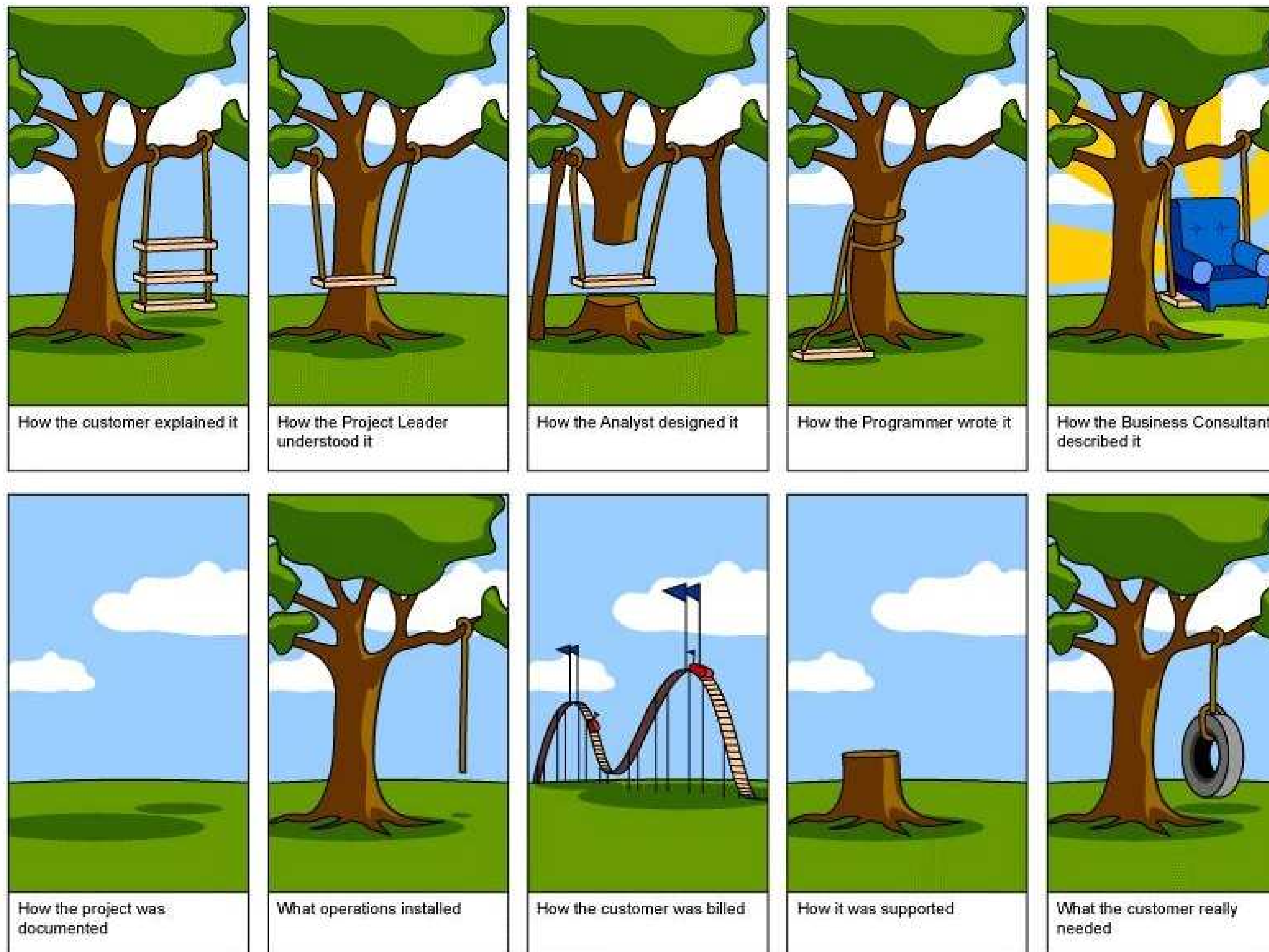
# Prozessmodelle
# Code-Design

von Eva Stöwe und Jan Nonnen

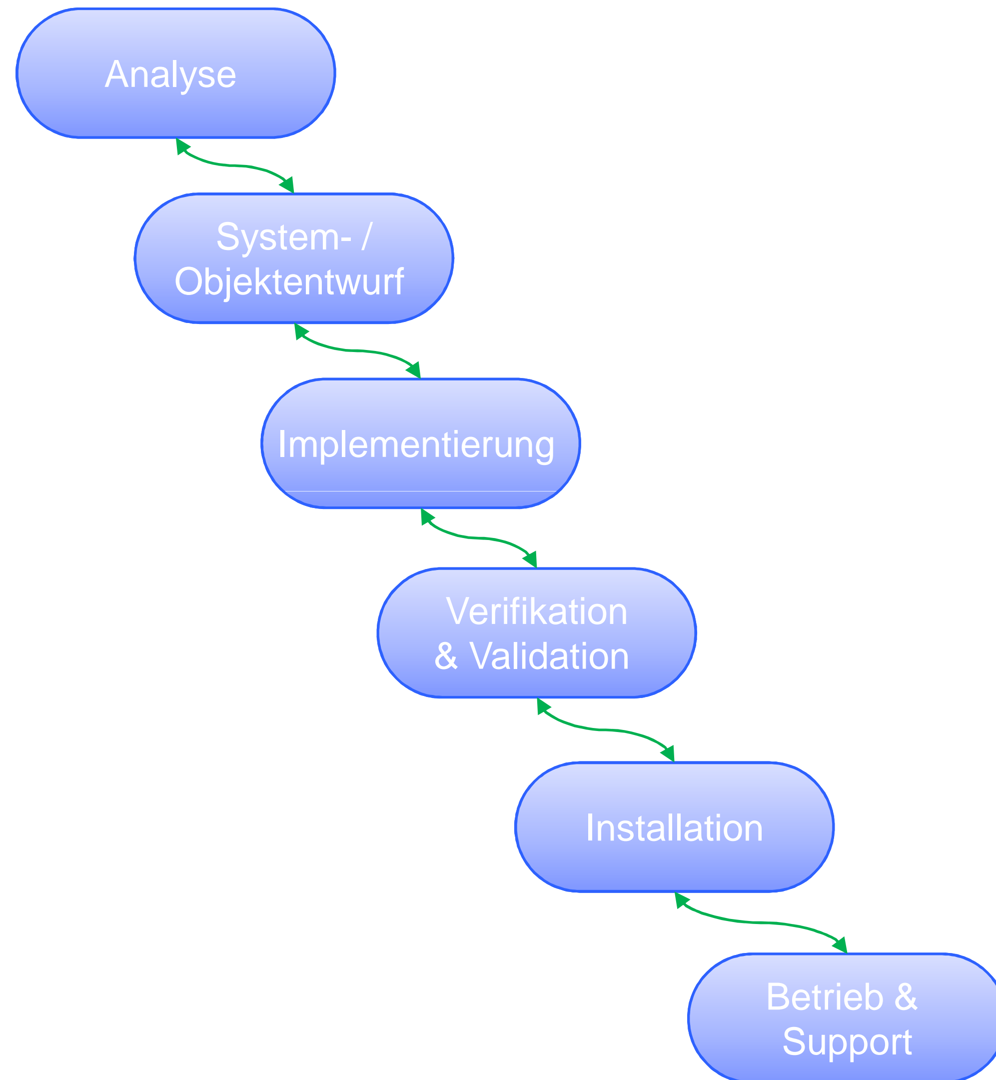
18.03.2010

© 2010 Eva Stöwe, Jan Nonnen

# Wozu, weshalb, warum?



# Prozessmodelle ▸ Wasserfallmodell



# Prozessmodelle ▸ Wasserfallmodell

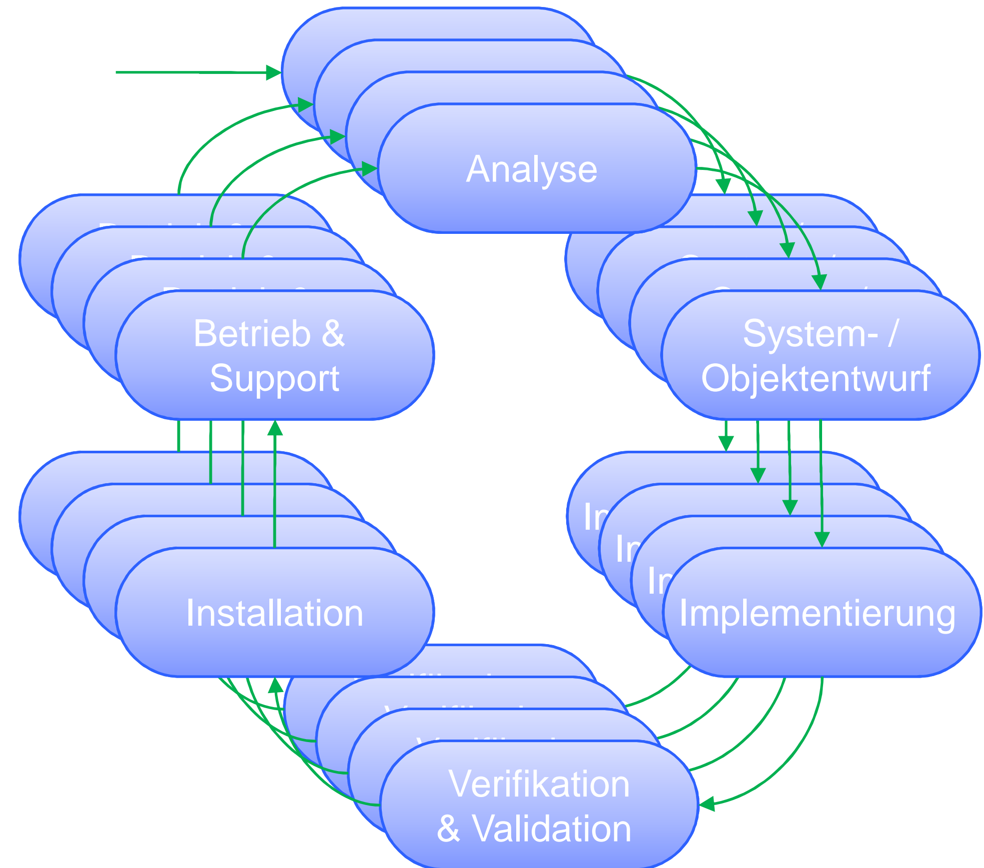
- Aktivitätszentrierter Prozess
  - beschreibt sequentielle Ausführung von Life-Cycle Aktivitäten
    - ⇨ Ergebnis: Dokumente
- Eine Phase entspricht einer Aktivität (z.B. Design)
  - Alle Designaufgaben werden erledigt bevor die nächste Phase anfängt
- Es gibt nur ein Endprodukt und keine Zwischenversionen
- Auftauchende Aufgaben die zu einer anderen Aktivität gehören werden zur späteren Bearbeitung liegen gelassen, bis sie im Prozess bearbeitet werden

# Prozessmodelle ▸ Wasserfallmodell

- Vorteile
  - Entspricht klassischen Ingenieursprozessen
  - Dokumente werden in jeder Phase erstellt
  - Vereinfachte Softwarenentwicklungs-Sicht
  - Präsentiert logische Reihenfolge von Artefakten
  - Manager lieben feste Meilensteine

- Probleme
  - Unflexible Einteilung des Projekts in klar abgegrenzte Phasen
  - Späte Auslieferung versteckt viele Risiken
    - ⇨ (zu) späte System- und Akzeptanz-Tests
  - nur wenn die Anforderungen sehr gut verstanden sind, passt das Prozessmodell
  - Anforderungen müssen komplett festgelegt werden, bevor in den Entwurf eingestiegen wird

# Prozessmodelle ▸ Spiralmodell



# Prozessmodelle ▸ Spiralmodell

- Anforderungen ändern sich kontinuierlich

- **Idee**: Iterativer Prozess
  - Ermöglicht auf häufig ändernde Anforderungen zu reagieren

- Zusätzlich
  - Unterstützt frühe Risiko-Behandlung
  - Ermutigt alle Beteiligten früh zusammenzuarbeiten
  - Aus Fehlern lernen, die in früheren Phasen gemacht wurden
  - Komponentenweise Auslieferung

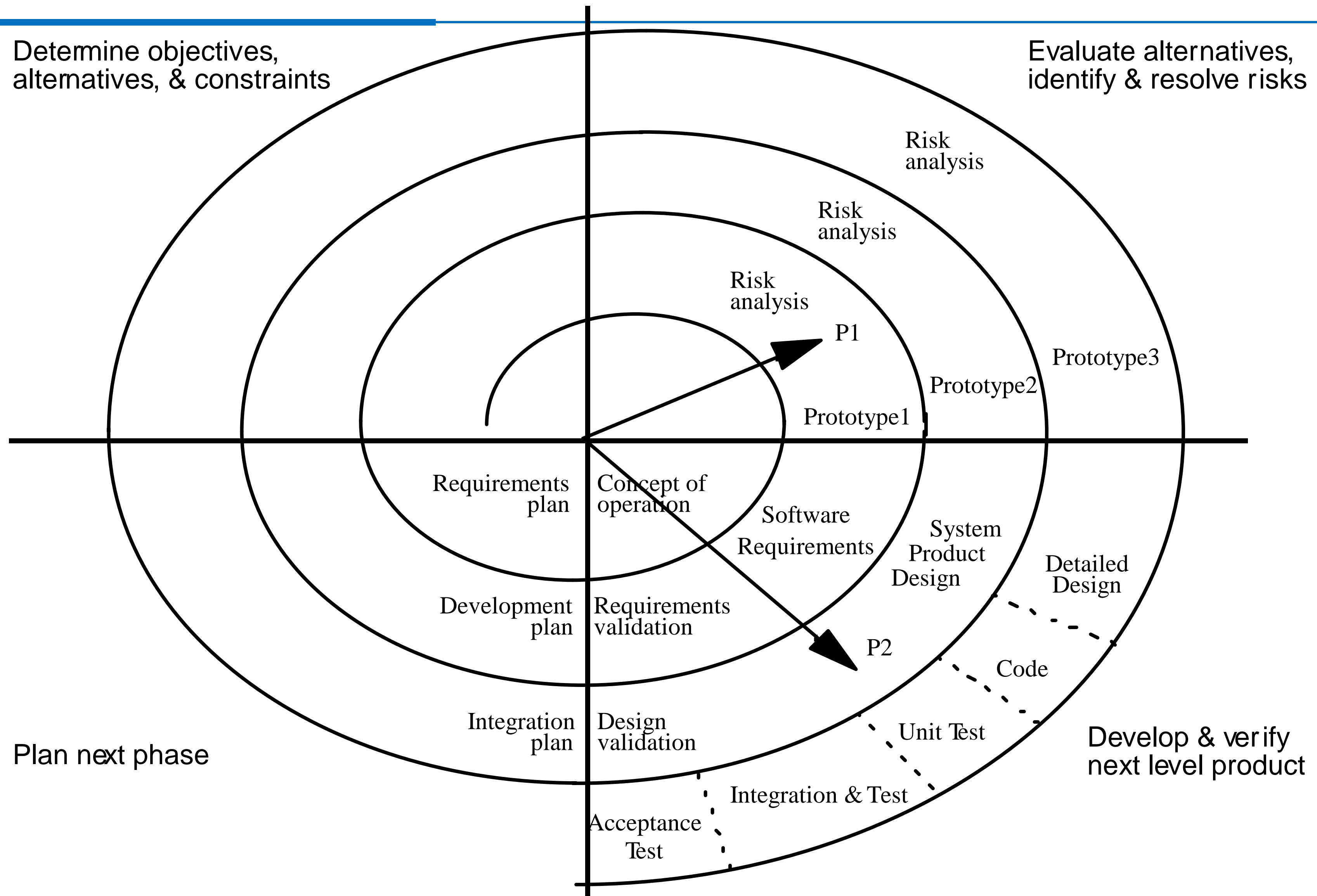
# Prozessmodelle ▸ Spiralmodell

- Risiko-Behandlung:
  - Risiken identifizieren
  - Risiken priorisieren
  - Reihe von Prototypen für die einzelnen Risiken entwickeln und testen
    - ⇨ … in der Reihenfolge fallender Priorität
  - Wasserfallmodell zur Entwicklung jedes Prototyps (“Zyklus”)
  - Wenn ein Risiko erfolgreich beseitigt wurde, wird
    - ⇨ Ergebnis des Zyklus bewertet
    - ⇨ nächste Iteration geplant

- Wenn ein bestimmtes Risiko nicht beseitigt werden kann,

  **Projekt sofort beenden!**

# Spiralmodell



# Prozessmodelle ▸ Spiralmodell

- Arten von Prototypen
  - Illustrativer Prototyp
    - ⇨ „auf einem Bierdeckel“ -> erster Dialog mit Kunden
  - Funktionaler Prototyp
    - ⇨ erste Funktionalitäten werden später nach Risikoreihenfolge ergänzt
  - Explorations-Prototyp ("Hacken")
    - ⇨ Gut um neue Ideen/Techniken auszuprobieren
  - Revolutionärer Prototyp
    - ⇨ Wegwerf-Version um Anforderungen durch „Verwendung“ zu bestimmen
  - Evolutionärer Prototyp
    - ⇨ Basis für die Implementierung des finalen Systems

# Prozessmodelle ▸ Grenzen beider Modelle

- Keines befasst sich mit andauernden Änderungen

**“Das einzig konstante ist die Änderung”**

- Das Wasserfallmodell
  - unterstellt, dass alle Probleme einer Phase
    - ⇨ nach dieser abgeschlossen sind und
    - ⇨ nicht wieder angegangen werden

- Das Spiralmodell
  - kann mit Änderungen zwischen den Phasen umgehen,
  - aber nicht mit Änderungen während einer Phase

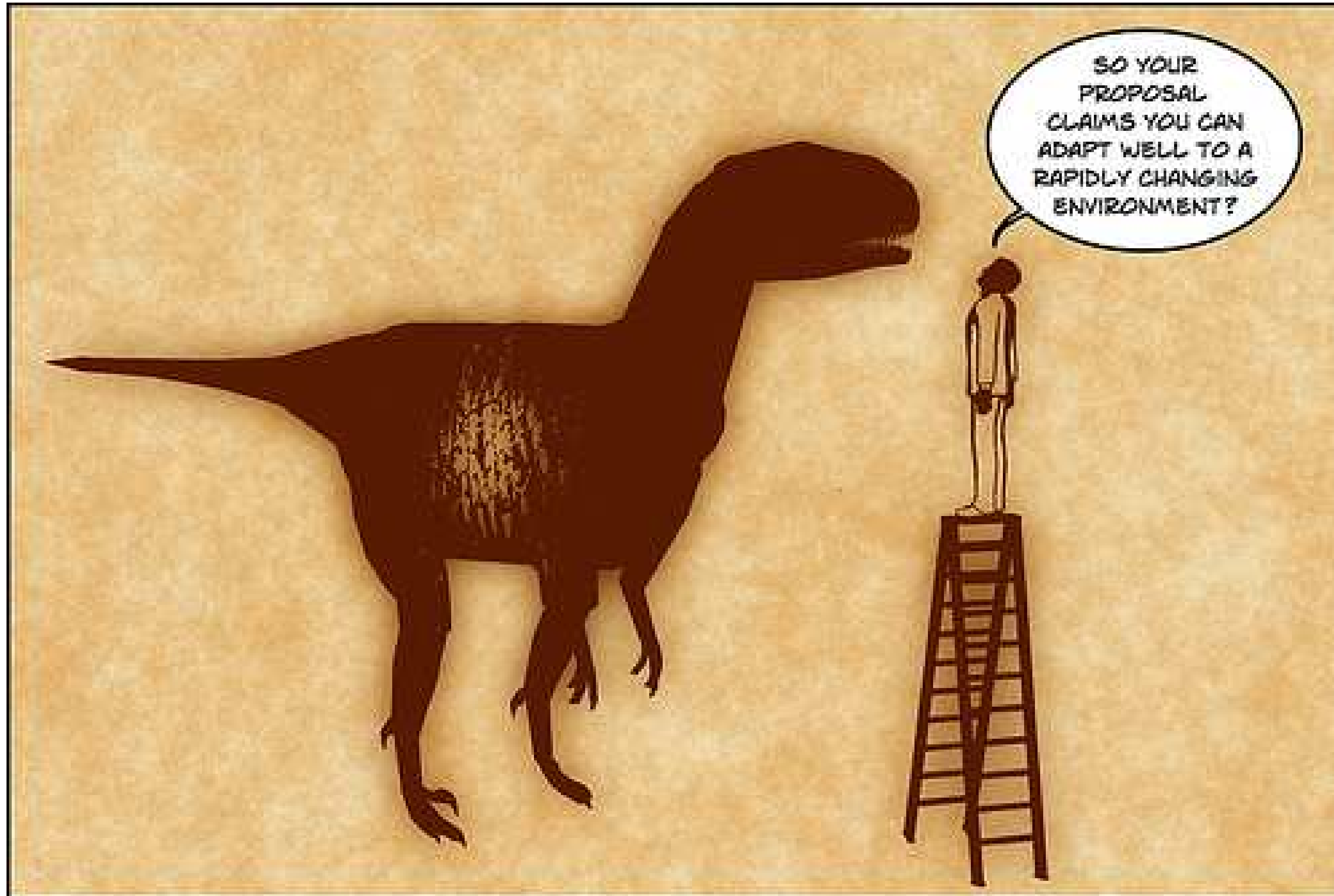
# Issue-Based Development

# Issue-Based ▸ Übersicht

- Ein System wird durch eine Sammlung von Issues beschrieben
  - Issues können von andern Issues abhängen
  - Issues sind offen A_I2 oder geschlossen A_I2
- Geschlossene Issues
  - haben eine Lösung
  - können wieder geöffnet werden (Iteration!)
  - sind die Basis des Systemmodells
- Iterationen umfassen Issues verschiedener Aktivitäten
  - zusammengefasste Issues gehören zu einem „top-level“ Issue
- Jede Iteration produziert einen wohldefinierten neuen Zustand
  - Möglichst einen, der ein funktionierendes Gesamtsystem ergibt
  - ... das für Benutzer einen erkennbaren Mehrwert darstellt.

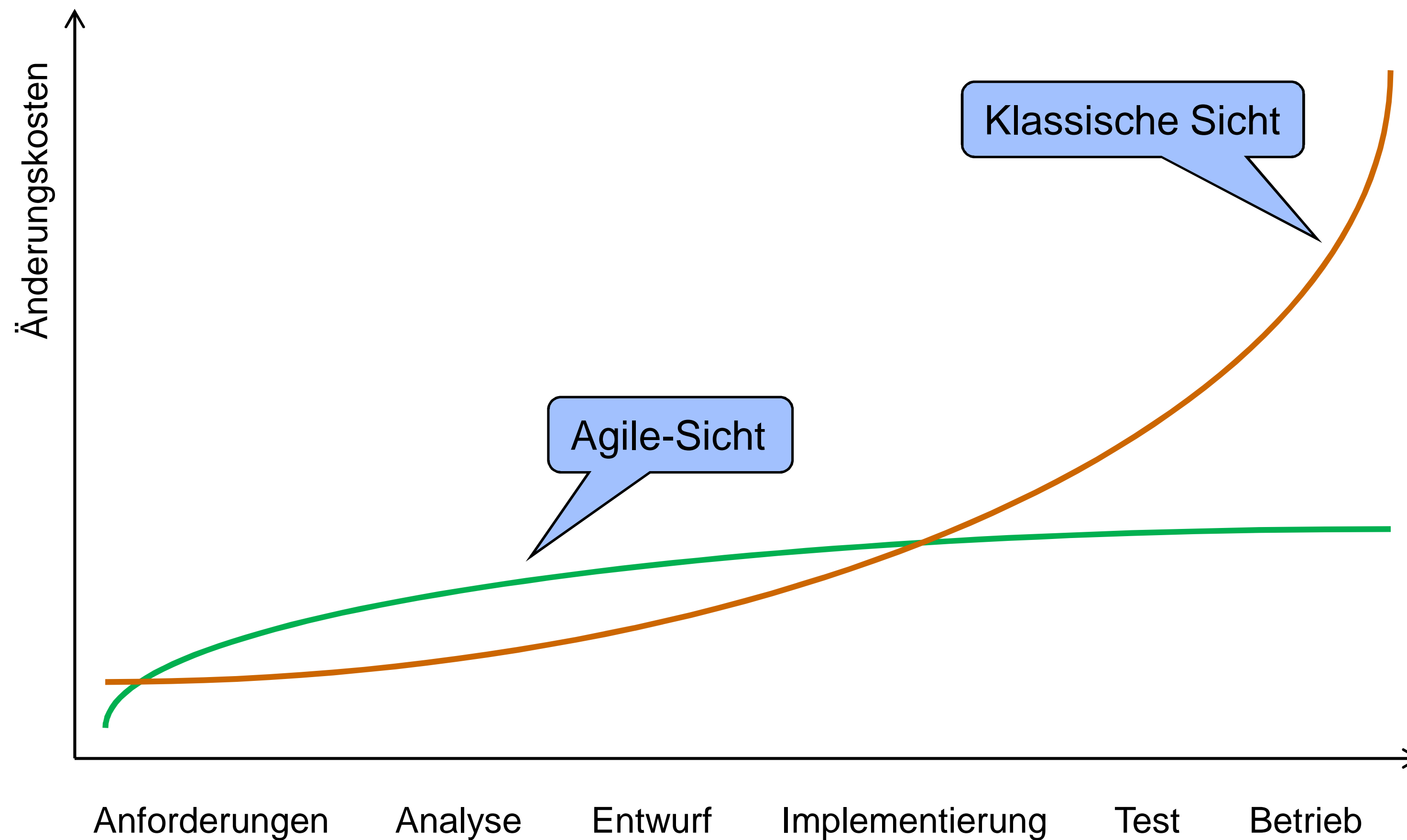
# Issue-Based ▸ Beispiel

# Wechselnde Anforderungen



# Agile Prozesse

# Agile Prozesse ▸ Änderungskosten



# Agile Prozesse ▸ Agile?

- Ziel: Entwicklungsprozesse flexibler und schlanker zu gestalten:
  - Zusammenarbeit mit dem Kunden ↔ Anstelle von Vertragsverhandlungen
  - Auf Änderungen reagieren ↔ Anstatt einem Plan zu folgen
  - Funktionierende Software ↔ Anstelle von vollständiger Dokumentation

- Gegenbewegung zu den oft als schwergewichtig und bürokratisch angesehenen traditionellen Entwicklungsprozessen

- Beispiele:
  - Extreme Programming
  - Crystal
  - Scrum

# Agile Prozesse ▸ Agile Werte

- bilden das Fundament der Agilen Entwicklung und wurden als Agiles Manifest formuliert:
  - Individuen und Interaktionen gelten mehr als Prozesse und Tools.
  - Funktionierende Programme gelten mehr als ausführliche Dokumentation.
  - Die stetige Zusammenarbeit mit dem Kunden steht über Verträgen.
  - Reaktion auf Veränderungen ist wichtiger, als das Befolgen eines festgelegten Plans.

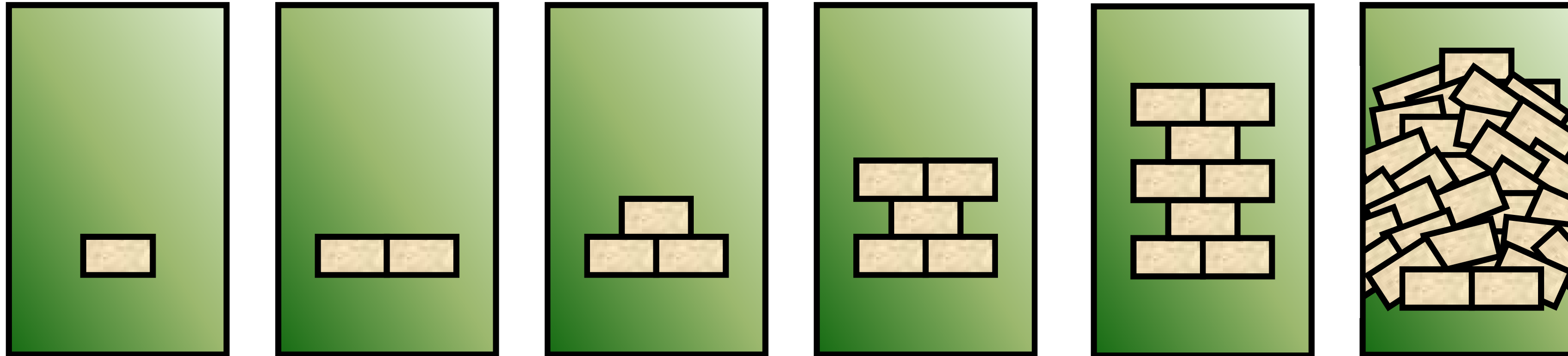
# Agile Prozesse ▸ Agiles Prinzip

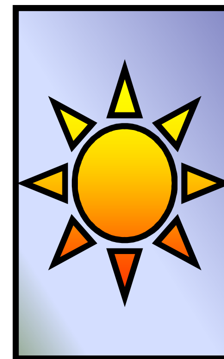
- Leitsätze für die agile Arbeit:
  - Vorhandene Ressourcen mehrfach verwenden (DRY-Prinzip)
  - Einfach (KISS-Prinzip)
  - Zweckmäßig
  - Kundennah
  - Gemeinsamer Code-Besitz (Collective Code Ownership)
  - Übergang zwischen Prinzipien und Methoden ist fließend

# Agile Prozesse ▸ eXtreme Programming

- Kunde ist ein Teil des Teams
  - permanentes Kundenfeedback
- Pair Programming
  - permanente Code Reviews!
- Permanentes testen!
  - Modultests sowie Funktionstests
- Kontinuierliche Integration
  - So oft wie möglich
- Refactoring
  - permanentes (Re)Design
- ziehe einfache Lösungen vor
  - Refaktoriere später, wenn nötig
- „Planning Game“
  - Sehr kurze Iterationen

# Agile Prozesse ▸ Planning Game





# Agile Prozesse ▸ Story Cards

# Agile Prozesse ▸ Einflussfaktoren

- Kosten
- Qualität
- Zeit
- Funktions-Umfang

**Grund-Zusammenhang**

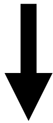
Durch Festlegung von drei beliebigen Faktoren ist der Vierte mit festgelegt !

**Konsequenz**

Der Kunde kann höchstens drei Faktoren nach seinem Wunsch bestimmen.
Die Entwickler müssen ihm den Einfluss auf den vierten Faktor erläutern!

# Agile Prozesse ▸ Faktoren: Folgerung

- Kosten → Resourcen bedarfsgerecht einsetzen
- Zeit → keine Einflussmöglichkeit, da extern bestimmt
- Qualität → hohe interne Qualität erreichen

bestimmen 

- Umfang → minimalen Funktions-Umfang anstreben

# Agile Prozesse ▸ Rollen

- Kunde
  - Spezifiziert und priorisiert Stories
- Coach (Betreuer)
  - erklärt den Prozess dem Management
- Teamleiter
  - Kontrolliert den Prozess
  - Vermittelt die Kommunikation mit dem Kunden
  - Erklärt dem Kunden die Folgen seiner Anforderungen
  - Leitet die Diskussionen des Teams

© 2010 Eva Stöwe, Jan Nonnen

# Agile Prozesse ▸ Rollen

- Programmierer
  - Schätzungen der Zeit / Schwierigkeit und des Risikos der Stories
  - Unterteilung der Stories in Tasks
  - Schätzung und Priorisierung der Task
  - Implementiert Tasks

- Technischer Experte („Consultant")
  - Ist Experte auf einem für das Projekt wichtigen Feld

- XP Mentor
  - Hat Erfahrung in XP Techniken und Praktiken
  - führt das Team durch den XP Prozess

# OO-Design

von Eva Stöwe und Jan Nonnen

© 2010 Eva Stöwe, Jan Nonnen

# Prinzip ▸ DRY

- **Don‘t Repeat Yourself (DRY) Prinzip**
  - Prinzip besagt Redundanz zu vermeiden oder zumindest zu reduzieren
  - Redundante Informationen sind schwierig zu pflegen (Konsistenz)

# Prinzip ▸ Abhängigkeit zu inhärenten Typen

- Inhärente Typen
  - Eine Typ A ist für einen Typ B inhärent, wenn B nicht ohne A beschrieben werden kann
- Prinzip:
  - Sei nur von inhärenten Typen abhängig
  - Depend Only on Inherent Types  (DO-IT Prinzip)

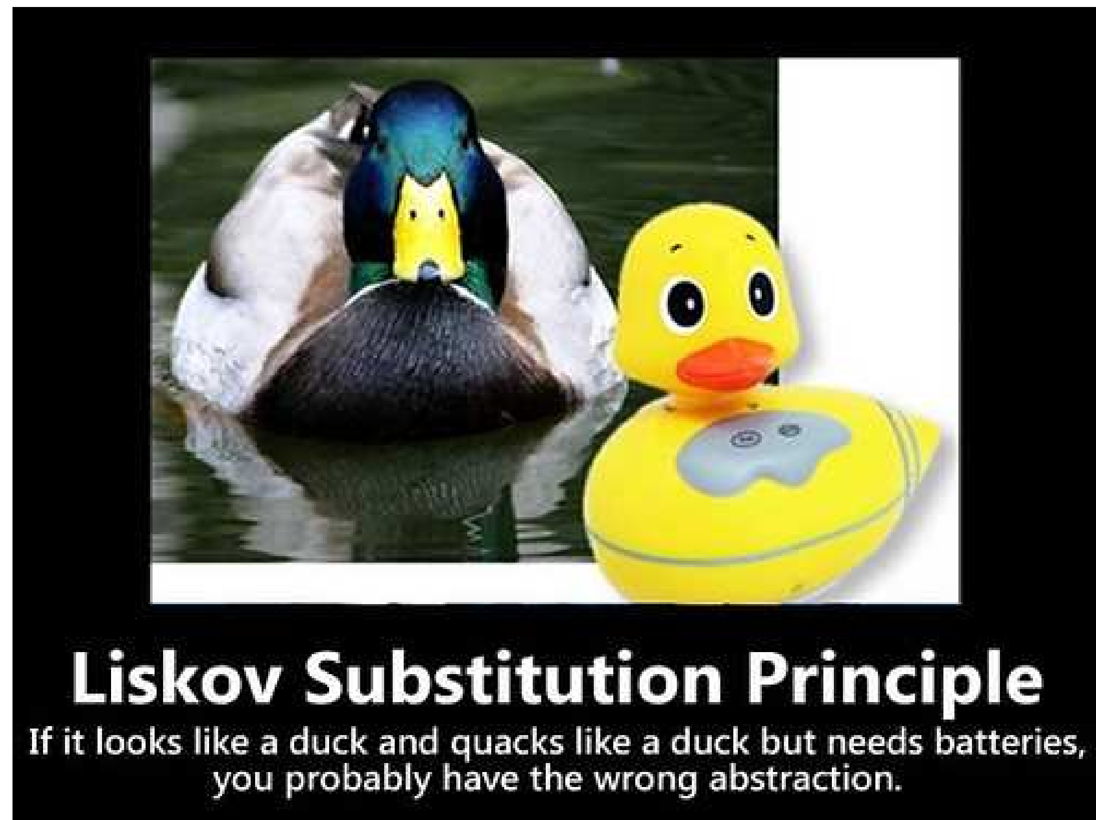
# Prinzip ▸ Einheitliche Eigenschaften (UP)

- Prinzip:
  - Eigenschaften einer Klasse müssen für alle Instanzen einheitliche Bedeutung haben
  - Uniform Properties (UP)

# Prinzip ▸ Liskov‘sche Ersetzbarkeit (LSP)

- Prinzip
  - Jede Methode die eine Oberklasse  braucht, muss auch mit jeder Unterklasse funktionieren

- Ausprägungen:
  - Design by Contract
  - Behaviour Protocols



# Code Design ▸ Law of Demeter

- Ziel:
  - Verberge Interna eines Subsystems
  - Verberge Interna einer Klasse
  - Vermeide transitive Abhängigkeiten
    - ⇨ Führe eine Operation nicht auf dem Ergebnis einer anderen aus.
  - Zielkonflikt
    - ⇨ Vermeidung transitiver Abhängigkeiten vs. „schlanke“ Schnittstellen.

**Law of Demeter („Talk only to your friends!“)**

- Klasse sollte nur von „Freunden“ (= Typen der eigenen Felder, Methoden- und Ergebnisparameter) abhängig sein.
- Insbesondere sollte sie nicht Zugriffsketten nutzen, die Abhängigkeiten von den Ergebnistypen von Methoden der Freunde erzeugen. Beispiel:
  - Nicht: param.m().mx().my()….;
  - Sondern: param.mxy(); wobei die Methode mxy() den transitiven Zugriff kapselt.

# Prinzip ▸ Kohäsion & Kopplung

- Kohäsion = Maß der Abhängigkeiten innerhalb der Kapselungsgrenzen
  - Hier Kapselungsgrenzen: Typen
  - Starke Kohäsion: Die Methoden in einem Typ haben ähnliche Aufgaben und sind untereinander verknüpft

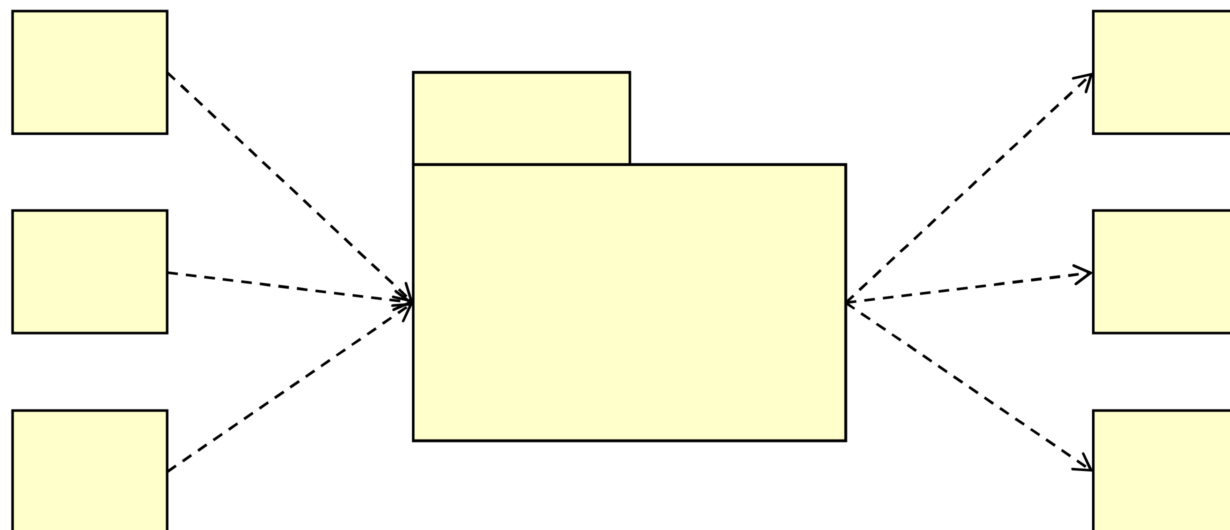
- Kopplung = Maß der Abhängigkeiten zwischen den Kapselungsgrenzen
  - Hier Kapselungsgrenzen: Typen
  - Starker Kopplung: Modifikation eines Typs hat gravierende Auswirkungen auf die anderen

- Prinzip:
  - **maximale Kohäsion**
  - **minimale Kopplung**

© 2010 Eva Stöwe, Jan Nonnen

# Metriken ▸ Abhängigkeiten

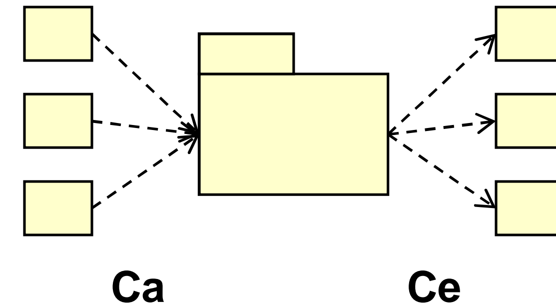
- Afferent Coupling (Ca)
  - Eingehende Abhängigkeiten anderer Komponenten
- Efferent Coupling (Ce)
  - Ausgehende Abhängigkeiten



# Metriken ▸ Stabilität und Abstraktheit

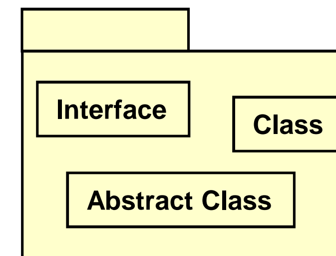
- Stabilität:

$$S = \frac{\#Ca}{\#Ca + \#Ce}$$

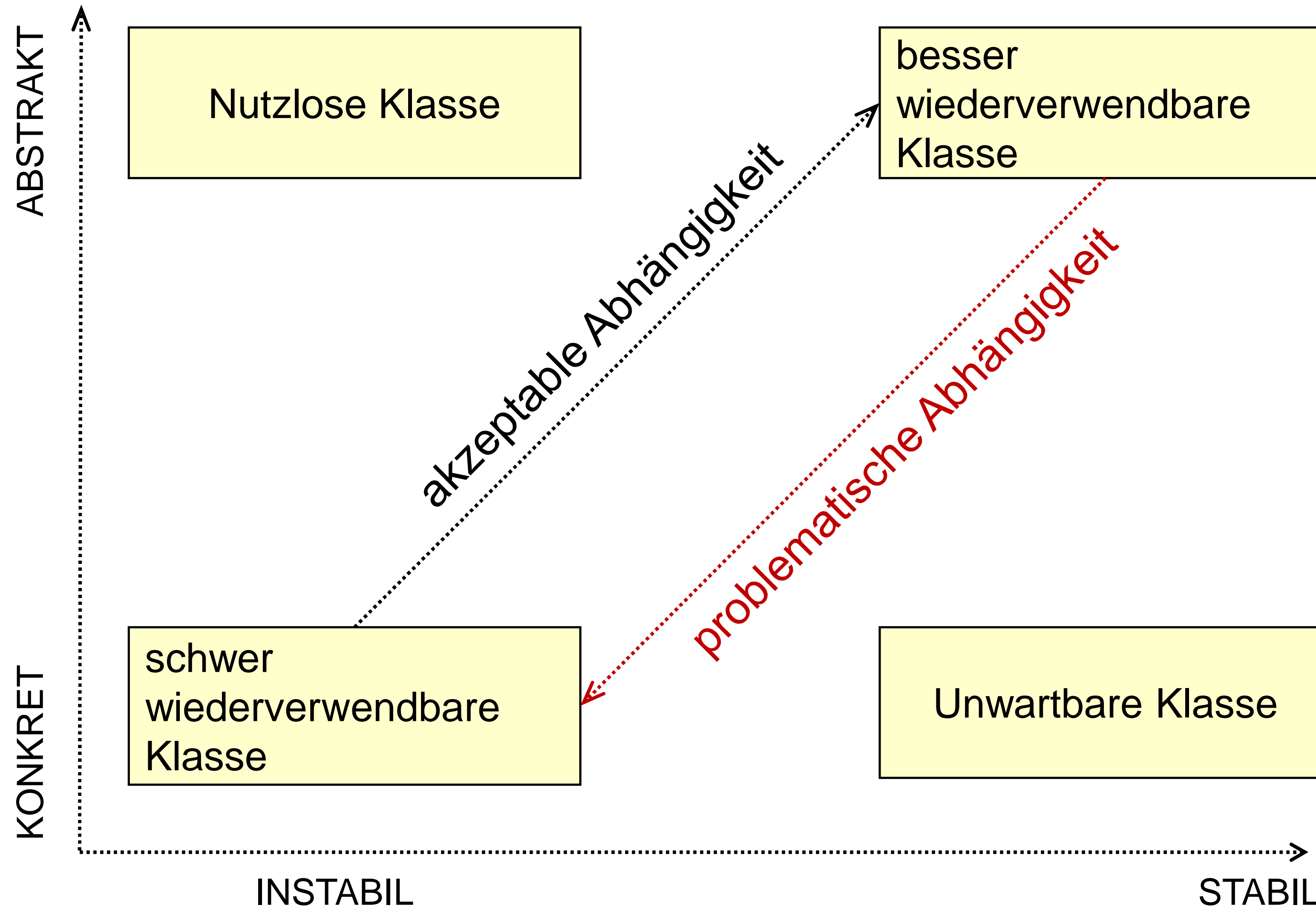



- Abstraktheit:

$$A = \frac{\#I + \#A}{\#I + \#A + \#C}$$



# Code Design ▸ Robert C. Martin Diagramm



# Code Design ▸ Entwicklungs – Prinzipien

- **Stable Dependencies Principle (SDP)**
  - Abhängigkeiten nur zu Stabilerem, nicht zu Instabilerem
- **Dependency Inversion Principle (DIP)**
  - Abhängigkeiten nur zu Abstrakterem, nicht zu Konkreterem
- **Stable Abstractions Principle (SAP)**
  - Abstraktion und Stabilität eines Paketes sollten zueinander proportional sein. Maximal stabile Pakete sollten maximal abstrakt sein. Instabile Pakete sollten konkret sein.
- **Acyclic Dependencies Principle (ADP)**
  - Der Abhängigkeitsgraph veröffentlichter Komponenten muss azyklisch sein

# Literaturempfehlungen

- Vorlesungsfolien SWT 2009/2010 Kapitel 8,13,14
- Mike Cohn

  **Succeeding with Agile: Software Development Using Scrum**
- Kent Beck, Martin Fowler

  **Planning Extreme Programming (XP)**
- Robert C. Martin
  - Article on Stability and Abstractness: http://www.objectmentor.com/resources/articles/stability.pdf
  - Article on OO Design Quality Metrics: http://www.objectmentor.com/resources/articles/oodmetrc.pdf