

## Teil VIII

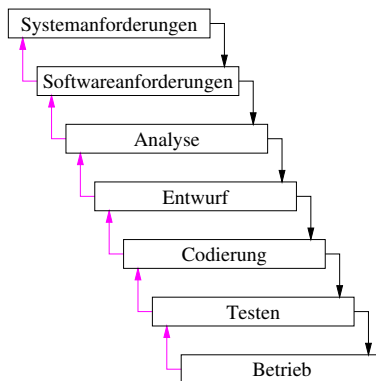
# Prozessmodelle

## 8.1 Einführung

Prozessmodelle legen fest:

- Elemente und Reihenfolge des Arbeitsablaufs
- Definition der Teilprodukte
- Fertigstellungskriterien
- notwendige Mitarbeiterqualifikationen
- Verantwortlichkeiten und Kompetenzen
- anzuwendende Standards, Richtlinien, Methoden, Werkzeuge

## 8.2 Wasserfall-Modell



(Royce'70)

- weit verbreiteter 'Klassiker'
- nicht mehr Stand der Technik wegen teurer Fehlerkorrektur
- dokumentengetrieben
- ursprünglich ohne Rückkopplung

## 8.3 Der vereinheitlichte Software-Entwicklungsprozess

- Engl.: (Rational) Unified Process
- Vorläufer:
  - Ericsson-Ansatz (Jacobson) seit 1967
  - Objectory (Jacobson) seit 1988
- Grundprinzipien:
  - Use-Case-getrieben
  - Architektur-zentriert
  - iterativ und inkrementell
- anpassbar
- Werkzeugunterstützung notwendig für konsistente, aktuelle Modelle

## Grundprinzipien des Unified Process

### Use-Case-getrieben:

- Was soll das System für jeden Benutzer tun?
- Analyse, Entwurf und Implementierung und Testen von Use-Cases getrieben

### Architektur-zentriert:

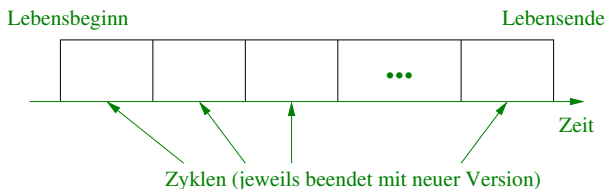
- Architektur parallel zu Use-Cases entwickelt (gegenseitige Abhängigkeit)

### iterativ und inkrementell:

- Risiken zuerst beseitigen
- Kontrolle nach jeder Iteration
- bei Scheitern einer Iteration: nur letzte Erweiterung betroffen
- überschaubare Schritte beschleunigen Projekt
- erlaubt sukzessives, einfacheres Bestimmen der Anforderungen

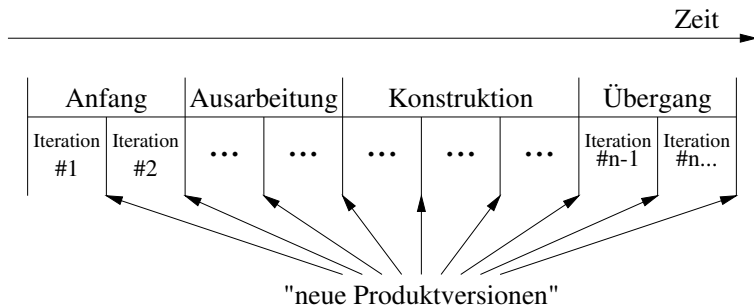
## Aufbau des Unified Process

- Folge von Zyklen
- nach jedem Zyklus: fertiges Produkt-Release aus Code, Handbüchern, UML-Modellen, Testfällen

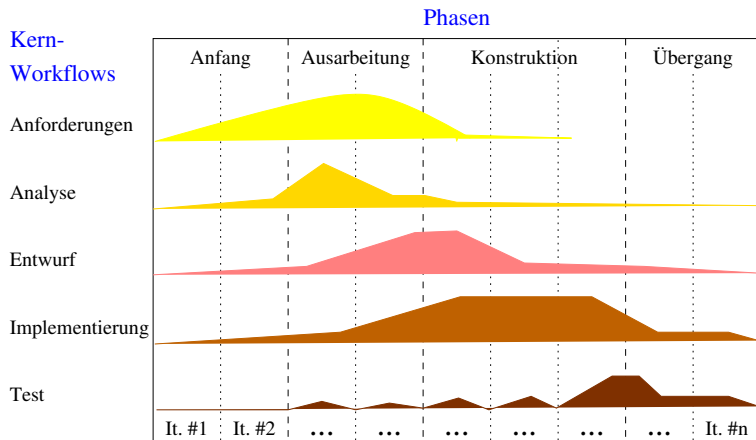


- Zyklus besteht aus 4 **Phasen**:  
Anfang, Ausarbeitung, Konstruktion, Übergang  
(**Inception, Elaboration, Construction, Transition**)
- jede Phase besteht aus Iterationen

# Zyklus



# Haupt-Workflows und Phasen



- jede Phase endet mit Meilenstein
- jede Phase bearbeitet alle Workflows (verschieden intensiv)

## Anfangsphase (inception phase)

- Ziel: grobe Vision des Produkts
- Was soll das System für jeden Benutzer tun? wichtigste Use-Cases
- Wie könnte eine passende Architektur aussehen? (vorläufig)
- Projektplan und Kosten?
- wichtigste Risiken identifizieren (Prioritäten)
- Ausarbeitungsphase planen

## Ausarbeitungsphase (elaboration phase)

- die meisten Use-Cases werden im Detail spezifiziert
- Architektur wird entworfen
- Realisierbarkeit der Architektur durch versuchsweise Teilimplementierung belegen
- kritischste Use-Cases werden realisiert
- Ergebnis: **Basis-Architektur**
- Aktivitäten und Ressourceneinsatz für Restprojekt werden geplant
- Sind Use-Cases und Architektur stabil?
- Werden die Risiken beherrscht?

## Konstruktionsphase

- System wird implementiert
- höchster Ressourcenverbrauch
- ggf. kleinere Architekturanpassungen
- Ziel: System (mit ggf. noch wenigen Fehlern) fertig für Pilotkunden

## Übergangsphase (transition phase)

- $\beta$ -Version wird an Pilotkunden ausgeliefert
- entdeckte Unzulänglichkeiten werden beseitigt  
(oder Behebung auf nächstes Release verschoben)
- Ausbildung der Kunden
- Hotline

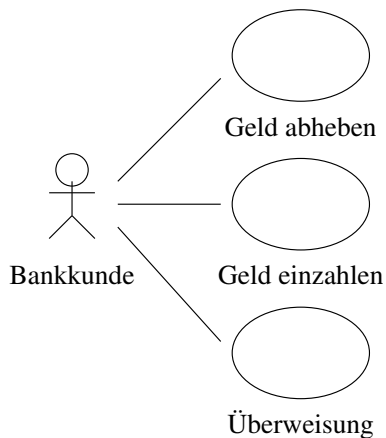
## Modelle des Unified Process

- Use-Case-Modell
- Analysemodell
- Entwurfsmodell (design model)
- Einsatzmodell (deployment model)
- Implementierungsmodell
- Testmodell
- basierend auf UML
- Modelle bieten konsistente Sichten
  - Diagramme eines Modells stehen in Beziehung (**trace dependency**) zu Diagrammen des übergeordneten Modells
  - **Rückverfolgungsmöglichkeit** einer Verfeinerung erleichtert Verständnis

## 8.3.1 UP ist Use-Case-getrieben

- jeder Arbeitsschritt wird von Use-Cases getrieben
- Vorteil: systematische, intuitive Einbeziehung funktionaler Anforderungen

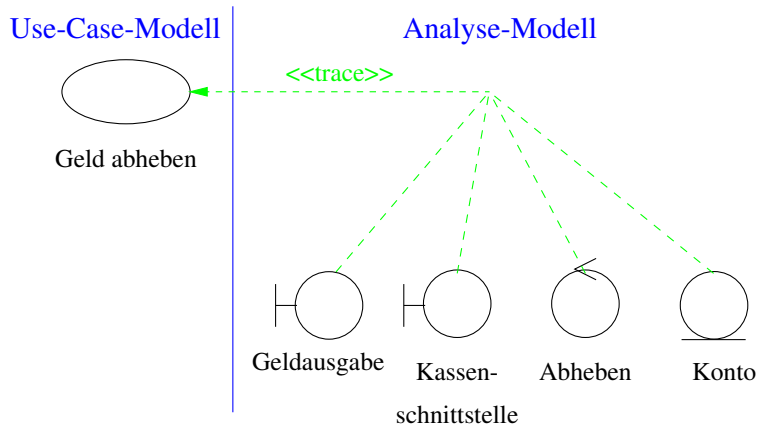
## Beispiel: Use-Case-Diagramm für Bankautomat



## Vom Use-Case-Diagramm zum Analysemodell

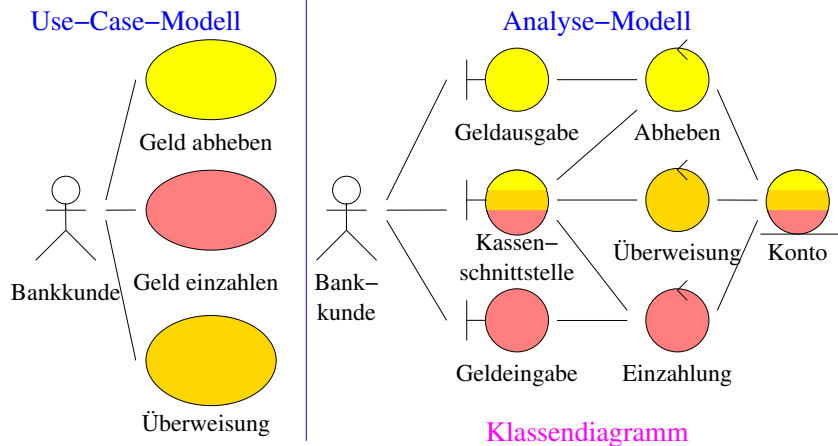
- Ableitung der (Analyse-)Klassen aus Use-Case-Beschreibungen
- **Stereotype:**
  - **Kontrollklasse:**
    - zur Koordination und Kontrolle anderer Objekte
    - für Transaktionen
    - eine pro Use-Case
  - **Grenzklassen**
    - zur Interaktion mit Akteuren
  - **Entitätsklassen**
    - langlebige, persistente Informationen

## Beispiel: Bankautomat



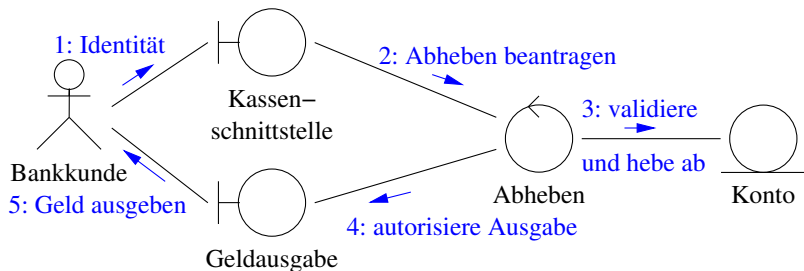
Analyseklassen zur Realisierung des Use-Cases "Geld abheben"

## Beispiel: Bankautomat



farbliche Zuordnung der Analyseklassen zu Use-Cases

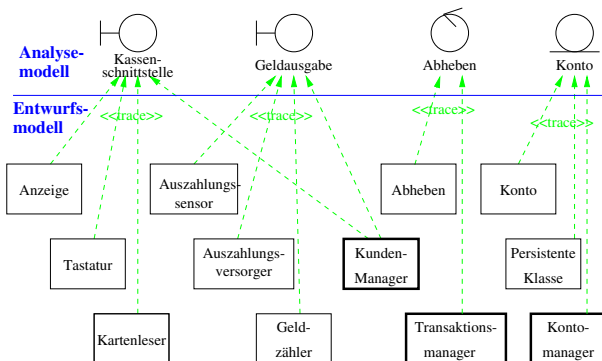
## Kommunikationsdiagramm: Zusammenspiel der Klassen



- alternativ: Sequenzdiagramm
- Nachrichten entsprechen Methodenaufrufen (vgl. Klassendiagramm)
- aus allen Rollen einer Klasse ergibt sich ihre **Zuständigkeit**

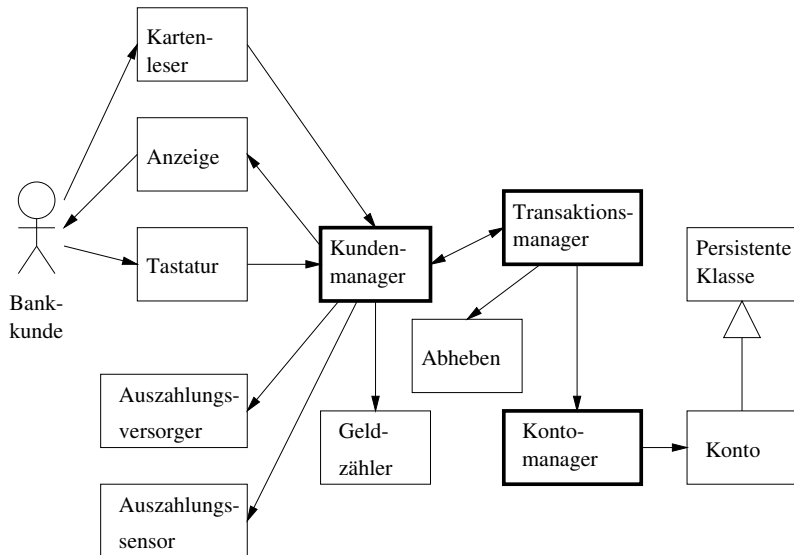
## Vom Analyse- zum Entwurfsmodell

- Verfeinerung unter Einbeziehung von Middleware (z.B. CORBA), GUI-System, DBMS, Legacy-Systemen
- statt konzeptioneller nun physische Klassen

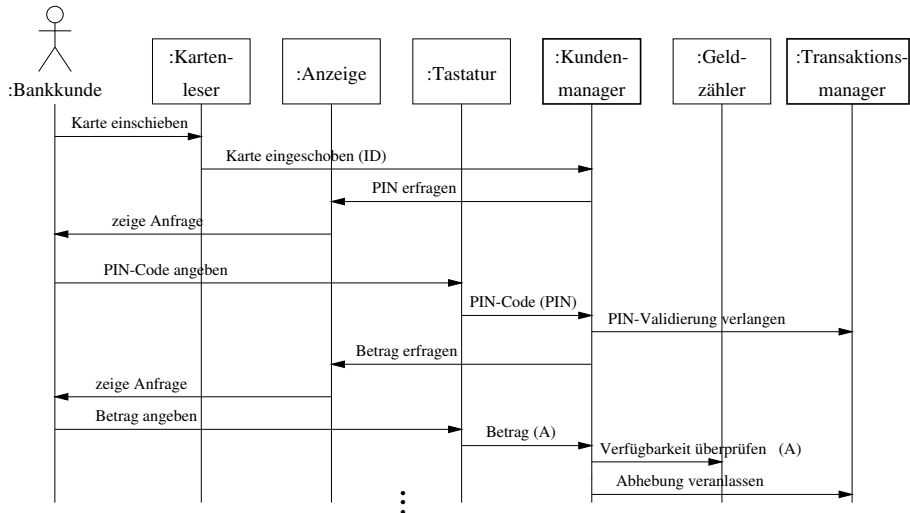


- oft  $\geq$  eine Entwurfsklasse pro Analyse-Klasse; Struktur bleibt

## Klassendiagramm im Entwurfsmodell

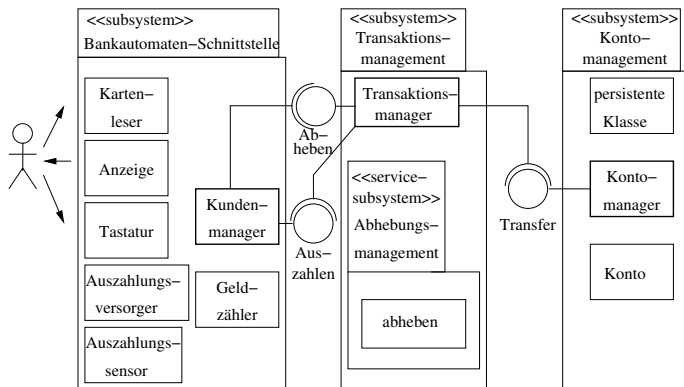


## Sequenzdiagramm im Entwurfsmodell



Zusammenspiel der Entwurfsklassen

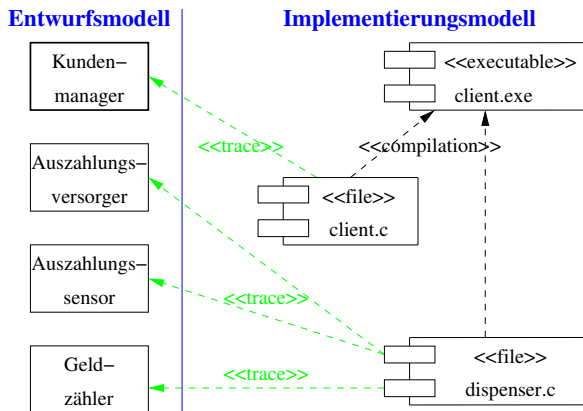
## Subsysteme



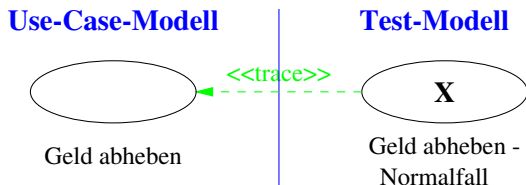
- semantisch zusammengeh. Klassen zu Subsystemen zusammengefasst
- Vorgehen bottom-up oder top-down
- Deployment Model bestimmt Zuordnung von Komponenten zu Rechenknoten (bei Einzelrechner trivial)

## Vom Entwurfsmodell zum Implementierungsmodell

- alle Klassen werden in Programmiersprache implementiert
- Implementierungsmodell umfasst ausführbare Komponenten, Datenbanken, Quellcode-Dateien, ...



## Testen der Use-Cases



- Black-Box-Testfälle nach Fertigstellung des Use-Case-Modells erzeugen

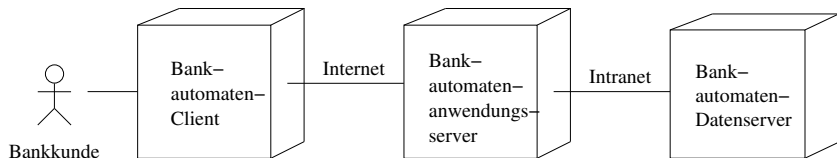
## 8.3.2 UP ist Architektur-zentriert

- statt allein von Use-Cases getriebener Software-Entwicklung: zunächst grober, **anwendungsunabhängiger Architekturentwurf**, u.a.
  - Architekturmuster (z.B. Client-Server, Broker, 3 Ebenen)
  - Schichten
  - Middleware
  - Legacy-Systeme
  - DBMS
  - GUI-System
- zugeschnitten auf **Anwendungsgebiet**
- hauptsächlich in Ausarbeitungsphase

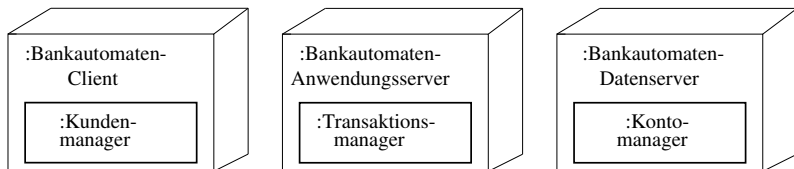
## Anwendungsspezifischer Teil der Software-Entwicklung

- ausgehend von **zentralen Use-Cases** (z.B. 10 %)
- Anpassung der anwendungsunabhängigen Architektur an Anwendung
- schließlich: (einfachere) Realisierung der verbliebenen Use-Cases auf erhaltener, stabiler Architektur
- Use-Cases von Anforderungen und Architektur beeinflusst
- ggf. mit Kunden verhandeln über preiswertere Realisierung Architektur-angepasster Use-Cases
- Grundarchitektur bleibt über gesamte Lebenszeit des Systems

## Deployment-Modell der Grundarchitektur



### Zuordnung der aktiven Klassen zu Rechenknoten:



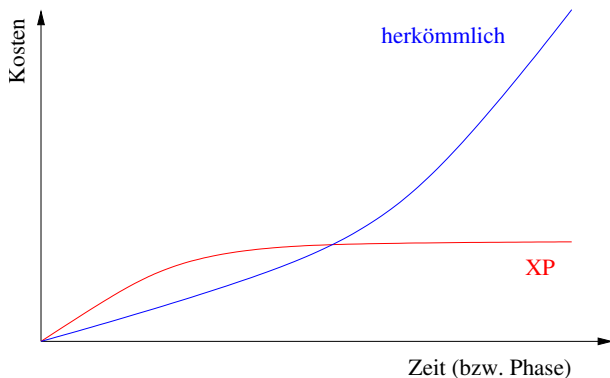
## 8.4 Extreme Programming

- z.Z. vielbeachtetes, leichtgewichtiges Prozessmodell
- **agil** (wie auch z.B. Scrum, Crystal, ...)
- **Code-getrieben**
- **iterativ**
- geeignet für kleinere und mittlere Projekte mit  $\leq 15$  **Entwicklern**
- **Ziel:** leichte Berücksichtigung sich wandelnder Anforderungen

## Grundgedanken

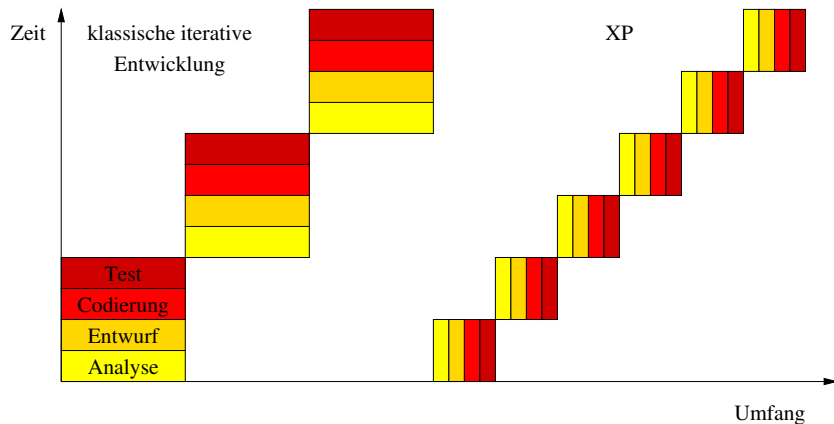
- **Kunden** stark **eingebunden** (→ Feedback)
- Erstellung der **Testfällen vor der Implementierung**
- automatisierte Tests (bzgl. Funktion und Performance)
- **möglichst einfacher Entwurf**
  - nur für das vorliegende (Teil-)Problem
  - keine Wiederverwendbarkeit angestrebt
  - keine Berücksichtigung zukünftiger Erweiterungsmöglichkeiten
- wichtigste Funktionen zuerst (→ 20:80-Regel)
- Code  $\hat{=}$  konkretisierte Gedanken → neue Ideen

## Kosten der Fehlerbehebung



- empirisch unzureichend belegt

## Vorgehensmodell

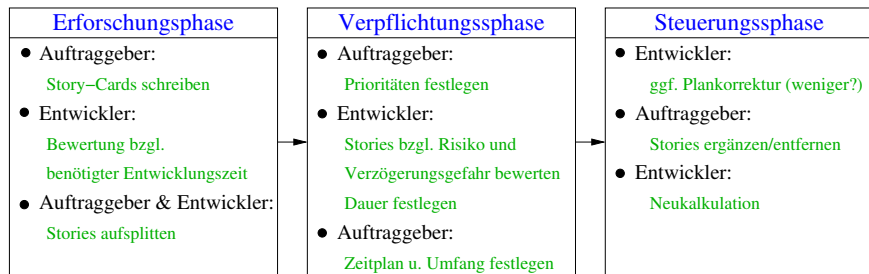


- Phasen simultan

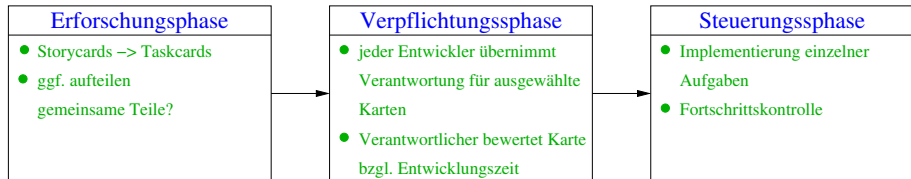
## Planung

- pro Iteration 1-4 Wochen
- nur nächste Iteration detailliert planen (spätere nur grob)
- Planungsphase
  - 1) Planspiel zwischen Kunden und Entwicklern
  - 2) Planung einer Iteration

# Planspiel



## Planung einer Iteration



- durch Entwickler, ohne Kunden

## Implementierung einer Aufgabe

### Entwickler

- übernimmt Aufgabe
- wählt Partner
- erstellt Testfälle
- implementiert Aufgabe
- führt Tests durch

## Entwurf

- einfachste mögliche Problemlösung
  - keine unnötige Komplexität, keine Wiederverwendbarkeit
- geringste Menge an Klassen und Methoden
- **Refactoring**: bei Hinzufügen neuer Funktionalität:  
System anpassen und, wo möglich, ständig vereinfachen  
(→ Tests, Mut)
- Literatur: M. Fowler: Refactoring, Addison-Wesley, 1999.
- Verständlichkeit durch geeignete Metaphern fördern

## Tests

- Testfälle vor Implementierung erstellt
- sichern Fortschritt trotz Änderungen
- Unit-Tests: durch Entwickler erstellt
- Funktionstests: durch Kunden (zusammen mit erfahrenem Entwickler)

## Implementierung

- **gemeinsamer Codebesitz:** jeder darf stets alles ändern  
(→ Tests verhindern Rückschritt)
- **kontinuierliche Integration** (Vor.: Integration/Build/Test-Tool)
- jederzeit lauffähiges System
- kurze Entwicklungszyklen: neue Features direkt Kunden zeigen  
(→ Feedback)

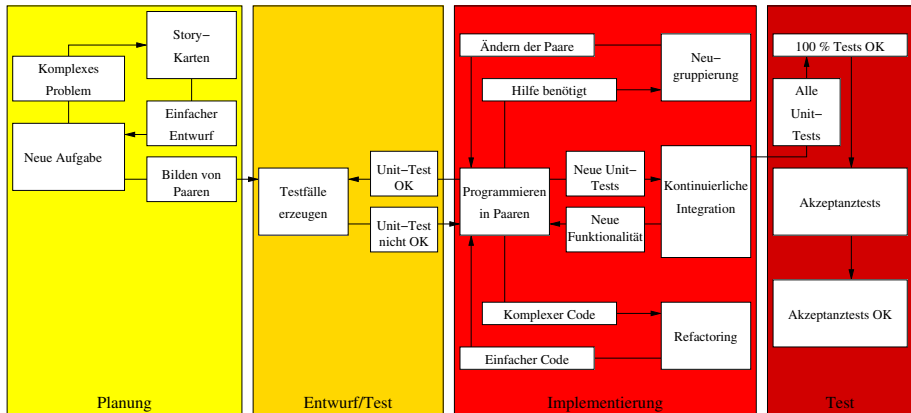
## Programmieren in Paaren

- je zwei Entwickler pro Bildschirm
- simultan: Analyse, Entwurf, Implementierung, Test
- gegenseitige Qualitätskontrolle
- lernen voneinander
- 15% mehr Zeitaufwand, 15% weniger Fehler
- Mehraufwand durch bessere Qualität und Wegfall einer separaten Qualitätssicherung kompensiert
- funktioniert auch unter Stress

## Organisatorische Rahmenbedingungen

- 40-Stunden-Woche: Überstunden nur kurzfristig sinnvoll, sonst Leistungsminderung
- räumliche Nähe der Teammitglieder
- Software-Entwicklungsumgebung geeignet für inkrementelle Entwicklung
- Konfigurations- und Versionsmanagement
- OO-Sprache
- ständiges Testen und Integrieren möglich
- Kunden ständig/regelmäßig verfügbar

# Überblick: Extreme Programming



## 8.4 Weitere Prozessmodelle

- **V-Modell:**
  - bei Behörden
  - zu den Phasen Definition bis Implementierung jeweils eigene Testphasen
- **Spiralmodell:** iterativ
- Prototyp-Modell
- Evolutionäres Modell
- Nebenläufiges Modell
- ...
- Details s. Balzert!