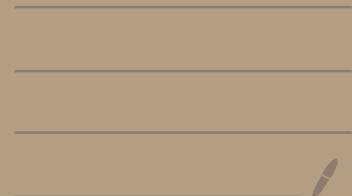


# **SWT 2**

---

**WS 19/20**



# Clean Code

Why?

- effort for changes increases the later it occurs
- ↳ continuous refactoring

## Object Oriented Design

- design strategy for a system made of interacting objects
- objects maintain own local state
- objects provide operations on that state information

## 5 SOLID Principles for good OO design

S<sub>ingle</sub> Responsibility Principle (SRP)

O<sub>pen</sub> Closed Principle (OCP)

L<sub>iskov</sub> Substitution Principle (LSP)

I<sub>nterface</sub> Segregation Principle (ISP)

Dependency Inversion Principle (DIP)

## Single Responsibility Principle

- one responsibility handles one core concern
- bad smell: big class ( $\approx 200\text{LOC}$ , > 15 methods/fields)
- ↳ Refactoring: Extract class

- ⊕
- code easier to understand
  - adding/modifying functionality affects few classes
  - minimized risk of breaking code

Example:

Modem
...
+ dial(pno: String) : void
+ hangup() : void
+ send(c: char) : void
+ receive() : char



«interface»  
Connector

+ dial(pno: String) : void  
+ hangup() : void



«interface»  
Communicator

+ send(c: char) : void  
+ receive() : char

## Command-Query-Separation

- Separate commands (actions) from queries (requests)
- ↳ getter / setter
- ↳ getter cannot modify value!
- commands have side effects on an object's state
- ↳ queries should not change the object's state

```
Example: public class Die {  
    private int faceValue;  
    ...  
    public void roll() {  
        faceValue = ...  
    }  
    public int getFaceValue() {  
        return faceValue  
    }  
}
```

## Open-Closed Principle

- software: open for extension, closed for modification
- ↳ add code, not change old code

```
for (Shape shape : Shapelist)  
    switch (shape.getType()) {  
        case SQUARE: square.draw  
        :  
    }
```

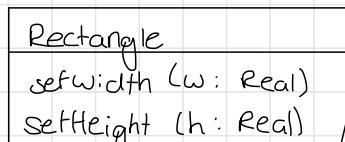


```
for (Shape shape : Shapelist)  
    shape.draw()
```

## Liskov Substitution Principle

- Unterklasse ist Spezialisierung der Oberklasse
- ↳ hat gleiche Funktionalität wie Oberklasse (+ Erweiterungen)
- ↳ Objekte müssen unbemerkt als "Ersatz" der Oberklasse verwendbar sein!

- beim überschreiben einer Methode in der Unterklasse:  
Erlaubt: precondition wird schwächer  
postcondition wird stärker  
! nicht umkehrbar!



void f (Rectangle r) {

r.setWidth(5)

r.setHeight(4);

assert (r.getWidth() \* r.getHeight()  
== 20)

} macht das gleiche

↑  
postcond.

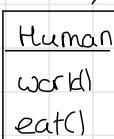
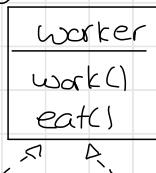
Square breaks it ↴

→ Lösung: setWidth, setHeight nicht als parent Methoden

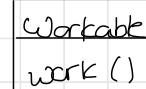
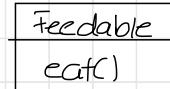
## Interface Segregation Principle

- Interface so klein wie möglich halten
  - High Cohesion: Interface kümmert sich nur um ein Konzept
  - Interface pollution: Interface sollte keine Methoden haben die nur eine Subklasse braucht
- Trenne Interfaces die von versch. clients verwendet werden

Beispiel:



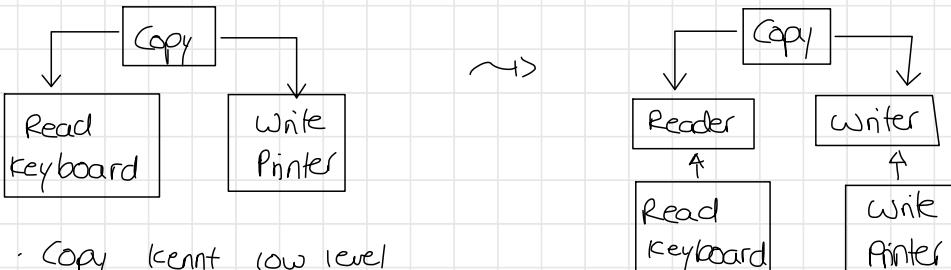
→



sinnlos

## Dependency Inversion Principle

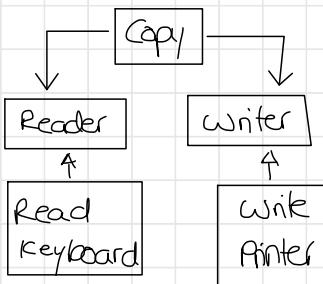
- High level module sollten nicht von low level Modulen abhängig sein
  - ↳ beide sollen von Abstraktionen abh. sein
- ~> Abstraktionen sollten nicht von Details abh. sein und umgekehrt



• Copy kennt low level Module → abh. von

Veränderungen

↳ copy schlecht wiederverwendbar



• Copy kennt nur Abstraktionen  
↳ andere Reader/Writer möglich

## More Principles

### Law of Demeter Don't talk to strangers

• ein Modul sollte nur über die Internas des Objekts das es manipuliert Bescheid wissen

Regel:

Eine Methode in der Klasse C sollte nur Methoden von ... aufrufen

... C

... einem von m erzeugten Objekt

... einem an m übergebenen Objekt

... einem Objekt in einer Instanzvariable von C

Beispiel: class Motor {  
    public void anlassen() {...}  
}

```
class Auto {  
    public Motor motor;  
    ... }
```

```
class Fahrer {  
    public void fahren() {  
        Auto meinAuto = new Auto();  
        meinAuto.starte();  
    }  
}
```

3

```
Class Auto {  
    public Motor motor;  
    public void starten() {  
        motor.anlassen();
```

→ meinAuto.starten()

# Boy Scout Rule

"Leave the campground cleaner than you found it"

- Code aufräumen / refactorn vorne einchecken  
↳ Teil der def. of alone machen

## Principle of Least Surprise

- Funktion / Klasse sollte erwartbares Verhalten implementieren

Bsp. String dayName = "Monday"  
Day day = DayDate.StringToDay(dayName);

- Output : Day. MONDAY
  - Sollte auch "monday" , "mon" , ... akzeptieren

# Coding Conventions

## Namen

- Projekt-/Team spezifische Standards
- bedeutsame Namen → kein 'if', 'them', 'getThem', ...

## Beispiele:

- Kontexthinweis: Klasse Account → Attribut AccountName statt nur Name
- Typhinweis: nameString, accountList  
→! kann auch Fehlinweise geben, bsp. switch von List zu dict, aber Name bleibt gleich ↗
- Präfixe: IName → I für Interface

## Kommentare

- Gute Kommentare erklären ...
  - ... rechtliche Fragen // Release under MIT licence, ...
  - ... performance Fragen // test cases very slow
  - ... Gedankengänge // It's safe to assume, ...
  - ... Absicht // RegEx matches ...
  - ... Algorithmen // Create 1k threads to provoke race cond.
- Gute Kommentare warnen ...
  - ... vor Konsequenzen // x is not threadsafe
  - ... wenn etwas wichtig // trim is important, because ...
- Gute Kommentare sind informativ
  - todos, open issues, ...
- Schlechte Kommentare: Redundanzen  
↳ noise im code, zB unnötige JavaDoc comments
- Auskommentierter Code ↗

## Formatierung

- optische Klarheit welcher Code zu was gehört

### Vertikaler Abstand $\approx$ ]

- zwischen Konzepten  $\rightarrow$  zusammengehörige Zeilen eng beieinander
- zB nach imports

### Horizontaler Abstand

- Operatoren Präzedenz
- Parameterseparation
- Einrückungen: scope

## Don't Repeat Yourself (DRY)

- Code nicht duplizieren  $\rightarrow$  kein Copy & paste!
- Copy & paste erschwert
  - Maintainability: wo sind überall Kopien die den gleichen Bug haben?
  - Understandability: gleiches Konzept muss mehrmals verstanden werden
  - Evaluability: wo überall müssen Kopien modifiziert werden?

$\rightarrow$  führt zu Fehlern und Inkonsistenzen

## Keep it simple, stupid (KISS)

- alles so einfach wie möglich, aber nicht einfacher

Guter Code ...

- ... kann von jedem verstanden werden
  - ... kann das Problem gut lösen
- Spezialisierung nur falls nötig  $\rightarrow$  zB Enumerable statt Collection oder List falls egal
- $\leadsto$  Code reviews, pair programming helfen

## You Ain't Gonna Need It (YAGNI)

- nur benötigte Features implementieren
- Featurismus ist teuer!
  - ↳ testing, doku, ...

## Single Level of Abstraction (SLA)

- Statements innerhalb einer Funktion sollten gleiches Abstraktionslevel haben
  - ↳ ggf. Statements mit größerem Detail in eigene Methode auslagern
- Funktionen einer Klasse: Abstraktionslevel sollte sich nach unten hin verringern (depth-first)

## Refactoring

- Codestruktur verändern ohne das Verhalten noch außen zu ändern

## Bad Smells:

- lange Methoden
- Codeduplikation
- Feature envy: eine Klasse benutzt viele/oft Methoden einer anderen Klasse
- Data class: eine Klasse hält nur Daten
- God-Klasse: Klasse macht zu viel
- Inappropriate intimacy: Klasse abh. von Implementierungsdetails einer anderen Klasse
- Details einer Methode müssen ständig nachgeschaut werden
  - wie sind Params nochmal?
  - was macht die Methode nochmal?

## Lösungen:

- Lang Methoden: Methoden extrahieren
- Feature envy: Teile von A wollen in B sein?
  - ↪ code block aus A der B's Methoden aufruft extrahieren
  - ↪ entspr. Methode in B erstellen
- Data class: verstecke Daten hinter gettern/settern (nur falls write)
  - ↪ verschiebe Daten an Ort der Funktionalität erlaubt)

## Limitations

- Refactoring kann Performance verschlechtern
- persisten~~ce~~ce layer (zB Datenbanken) schwer zu refactern
- published interfaces?
  - nur von mir benutzt → alles ok
  - ansonsten: interfaces erst gut durchdenken, dann publishen  
ggf. deprecation / mapping auf neue Implementierung



Nur mit Testunterstützung refactorn

## Software - Architektur

- beschreibt wichtigsten Strukturen des Systems
  - "die Design-Entscheidungen die nur schwer rückgängig gemacht werden können"
- ~> Architektur - Design ist kreativer Prozess basierend auf Systemanforderungen

Requirements ~> Software - Architektur ~> Code

## Architektur Design

- früh im Software-Entwicklungsprozess
  - Link zw. Spezifikation und Design
- ↳ Identifizieren der Hauptkomponenten, deren Kommunikation und deren (Hardware) Ressourcen-Nutzung

## Design-Entscheidungen Beispiele

- gibt es nutzbare Referenzarchitekturen?
- Unterteilung in Subsysteme → Module / Komponenten / .. ?
- welche Frameworks?
- wie Dokumentieren?
- wie Legacy Software integrieren?
- was kann von alten / für neue Projekte wiederverwendet werden?

## Definition Software - Architektur

- Ergebnis von Design-Entscheidungen
- beinhaltet Systemstruktur: Komponenten, ihre Beziehungen und ihren Umgang mit der Laufumgebung

Zeitpunkt in der Entwicklung:

- requirement-oriented: früh im Prozess
- code-oriented: unklare Dinge auf später verschieben

↳ Softw.-Arch. ist nicht implizite interne Systemstruktur

↳ muss mit geeigneten Sprachen dokumentiert werden

### Unterscheidung Struktur $\hookrightarrow$ Architektur

- Arch. = explizite Entscheidungen
- Struktur = Aufbau des Systems

### Vorteile expliziter Architektur

- Stakeholder Kommunikation: Zur einfacheren System-Diskussion
- System Analyse: entspricht System den Anforderungen
- Large-Scale reuse: in anderen Systemen alles oder Kompon. wiederverwenden
- Projektplanung: Kostenabschätzung, mile-stone Organisation, staffing, ...

### Tradeoff Behaviour - Maintainability

- Behaviour: Software erfüllt alle Anforderungen des Stakeholders
- Maintainability: Änderungen sind einfach umzusetzen

100% behaviour, 0% maintainability:

- unbrauchbar wenn Anforderungen sich ändern
- da unveränderbar: Programmierer nutzlos

0% behaviour, 100% maintainability:

- Programmierer kann behaviour erhöhen
- Änderungen leicht umgesetzt
- Programm bleibt nützlich auch bei Änderungen

# Requirement Engineering

- frühe Phase der Entwicklung
- identifizieren der Hauptsystemkomponenten, ihrer Kommunikation und ihren (Hardware) Ressourcen Anforderungen
- versch. Perspektiven → je nach Projekt versch. beachten

## View Points in Palladio

### Structural

- Infos über die statischen Eigenschaften eines Systems
  - Unterscheidung systemspezifisch und systemunabhängig
- Repository view type
- einzige systemunabh. Perspektive
  - zeigt alle Komponenten / Interfaces die in mehreren Systemen wiederverwendet werden können

Assembly view type

- zeigt konkrete Instanzierungen der Komponenten im System und deren Verbindungen

### Behavioural

- Infos über funktionales und extra-funktionales Ausführungsverhalten des Systems

usage model

- Verhalten der Nutzer / anderer Systeme die mit System interagieren

## Deployment

allocation

- Info welche Instanzen welcher Komponenten (aus assembly view type) auf welchen Containern des environment view types laufen

resource environment

- Info über alle Container und deren Verbindungen untereinander

! Logische und deployment Arch. nicht vermischen

↳ logische Arch. zeigt keine externen Ressourcen explizit (z.B. Datenbank)

## Dokumentation

Dokumentieren von

- Einflussfaktoren
  - Entscheidungen + Begründungen
  - Alternativlösungen + warum nicht verwendet
- ]} = Rationale

Rationale warum wurden diese Entscheidungen getroffen

- damit ich / andere später nachvollziehen können
  - ↳ zB gefrorene Annahmen, ...
- um nicht später reverse-engineeren zu müssen was die Idee war

## Einflussfaktoren

### Requirements

- Architektursignifikante Anforderungen
- funktional und
- nicht-funktional (quality req.): stehen oft zueinander im Konflikt
  - Performance
  - Security
  - Safety
  - Availability
  - Maintainability
  - Scalability

↳ müssen ausbalanciert werden

## Organisation

### Conway's Law

- Entwürfe bilden interne Kommunikationsstrukturen des Teams ab → Teamgröße, Erfahrung, Struktur, ...

## Reusability

- gesamte Arch. → Definiere als "Referenz-Architektur" für andere Projekte
- Subsysteme / Komponenten
- Styles / Patterns / Guidelines

## Reuse von Design Prinzipien:

- Separation of concerns
- Single Responsibility principle → one resp. per module / comp.
- Information hiding
- Principle of least knowledge → = law of Demeter
- Don't repeat yourself
- Minimize upfront design  
↳ YAGNI vs. Design for extensibility / Reusability

## Architekturpatterns

Architekturpattern = Lösung zu wiederkehrendem Problem bei dem mehrere Einflüsse auf dem Architekturierei balanciert werden müssen

Architekturstil = Lösungsansätze, unabhängig von Anwendung, ein Stil pro Architektur  
zB OO, modular, ...

Referenz-Architektur : Konzept auf Domänen-Ebene, Komponenten und Subsysteme können bei konkreten Domänen-Instanzen (einzelnen Projekten) verwendet werden

Übergang zw. Arch.-Pattern, Design-Pattern fließend  
zB MVC

## Architekturstile

- Set von Constraints die Systemweit gelten
  - große Systeme haben oft versch. Stile
    - ↳ je nach Anwendungsbereich
  - zB Kommunikation : Service-Oriented Arch. (SOA), Message Bus
  - Deployment: Client/Server, N-Tier, 3-Tier
  - Structure: Component-based, Object-Oriented, layered Arch.
- OO Systeme : gruppiert in Layer
- je mit Subsystemen mit zusammenhängenden Aufgaben
  - hohe Layer rufen niedrige auf
    - ↳ nur Layer direkt darunter
- ! Layer (konzept) ≠ Tiers (physische Trennung auf Servern)

## Layered Architecture

- ⊕ klare Separation of concerns
  - verbessert Modifiability
  - verringert Komplexität
  - vereinfacht testen
  - unabhängige Austauschbarkeit
- ⊖ · benötigt hohe Anzahl Klassen

## User Interface Layer

- präsentiert Daten an den User
- verwaltet Userinteraktion, zB screen flow
  - ↳ akzeptiert UI events
  - ↳ ! UI events unverarbeitet weitergeben an Application layer  
"don't call us, we call you"

## → Separation of Concerns

- "smart UIs" = anti-pattern (vermeiden)
  - ↳ presentation, domain logic trennen
  - ↳ opt.: UI und control logic trennen

UI

- presentation, view
- ↳ GUI

Application

- verarbeitet UI Anfragen
- hält session state
- Vorbereitung Daten für UI

Domain

- verarbeitet Appl. Anfragen
- domain services

Business Infrastructure

- low-level business services die von versch. Systemen verwendet werden
- zB currency converter

Technical Services

- high-level technische Services und frameworks
- zB Persistence, Security

Foundation

- low-level technische Services, utilities, frameworks
- zB Datenstrukturen, DB, math, threads, ..

## Application Layer

- verteilt eingehenden UI requests
  - speichern des session states
  - kontrollieren des workflows
  - implementieren der Systemoperationen
- 
- optional bei kleinen Systemen → UI ruft direkt domain layer auf

## Domain Layer

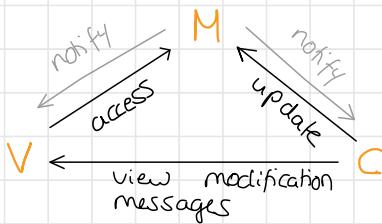
- beinhaltet domain model
- implementiert business logic
  - ↪ OO sehr nützlich, Objekte inspiriert von real-life Objekten
- sehr spezifisch für Anwendung

## Model - View - Controller (MVC)

- Aufteilen der Anwendung in 3 Teile
  - Model = Hauptfunktionalität, Daten → processing
  - View = Anzeigen der Daten → output
  - Controller = user input verarbeiten → Input
- gut bei interaktiven Anwendungen mit flexiblen Interfaces

### Model

- impl. Hauptfunktionalität
- registriert abhängige Views, Controller
- benachrichtigt abh. Komponenten bei Datenänderungen



### View

- erzeugt / init der zugehörigen Controller
- Anzeige der Infos/Daten
- impl. update procedure
- Holt Daten vom Model

### Controller

- verarbeitet user input events
  - ↪ erstellt Services für Model
  - ↪ erstellt display Aufgaben für View
- ggf. impl. update procedure

Daumentregel: ein Controller pro use case

## Controller modelling

- a) ein Controller pro use case  
 ↳ gut bei vielen use cases topkek
- b) ein Controller pro Anwendung / System  
 ↳ gut bei kleinen systemen
- c) Direkter Zugriff auf domain objects  
 ↳ reduziert Parameter-Passing  
 ↳ ! bringt schnell Kontroll-Logik ins domain layer :-)

## Data transfer object

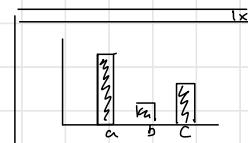
- serialisierbares Objekt
- transferiert Daten zwischen Prozessen / Architekturelementen (layers, ...)
- beinhalten normalerweise nur Attribute + getter / setter

## Observer Pattern

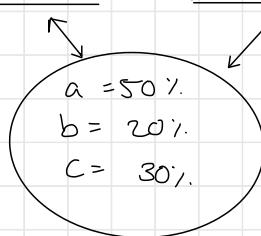
- one-to-many Beziehung zw. Objekten  
 ↳ wenn ein Objekt sich ändert werden die abhängigen Objekte benachrichtigt und automatisch geupdated
- Objekt kann benachrichtigen ohne Annahmen darüber zu machen wen alles es benachrichtigen muss

zB

	a	b	c
x	60	30	20
y	50	30	30
z	80	10	10

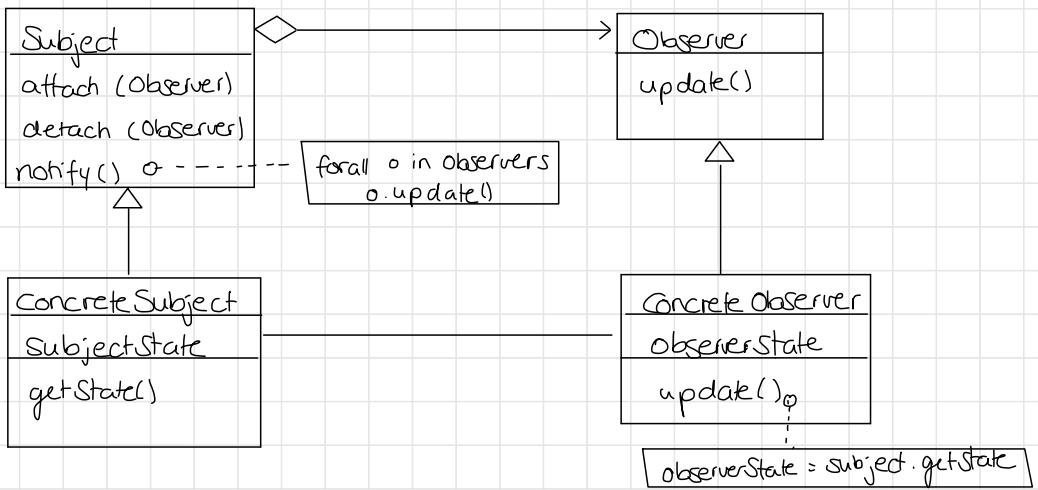


UI layer

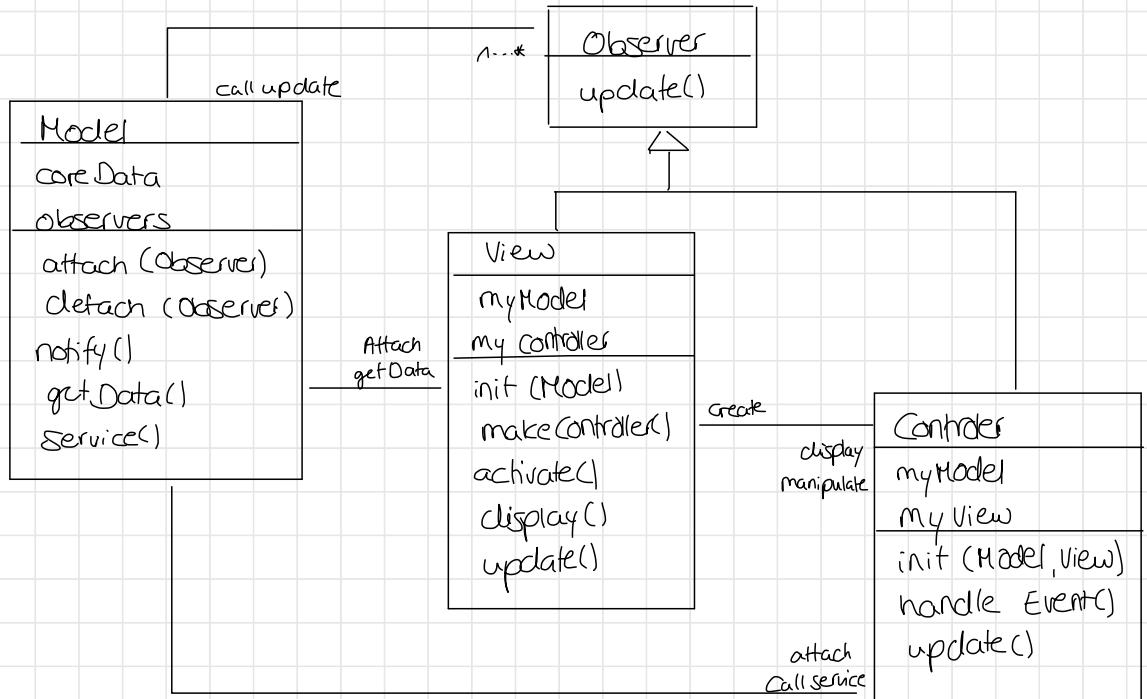


Domain Layer

## Struktur



## Model - View - Controller and Observer



# Clean Architecture

- versch. Layer: separation of concerns
- ↪ Architektur unabh. von Technologie und besser testbar

⚠ Stil ist nur zugeschnitten auf Maintainability

## Abhängigkeiten

- nur in Richtung der Stabilität

zB business rules sollten unabh. davon sein ...  
... wie Datenbankzugriff funktioniert  
... wie die UI gebaut ist

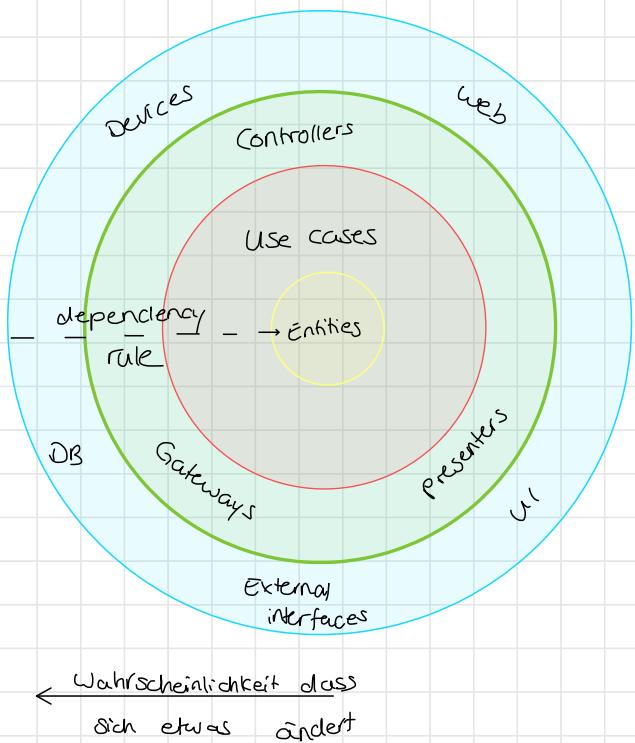
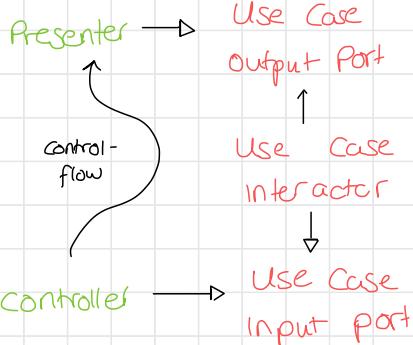
## Clean Architecture Style

Yellow: Enterprise business rules

Red: Application business rules

Green: Interface adapters

Blue: frameworks & divers



- Kreise = versch. Software gebiet
- je weiter innen desto higher-level ist Software
- äußere Kreise = Mechanismen
- innere Kreise = policies

### • Dependency Rule:

- code dep. nur nach innen gerichtet, das gilt auch für Funktionen, Klassen, Variablen, Datenformate

⇒ äußere Kreise beeinflussen innere nicht!

## Entities

- business rules
  - Objekte mit Methoden
  - Datenstrukturen mit Funktionen
- individuelle Anwendungsänderungen haben keinen Einfluss auf Entities

## Use Cases

- anwendungsspezifische business rules
- impl. use cases des Systems
  - ↳ Datenfluss von / zu Entities
  - Kollaboration zw. Entities
- Änderungen von DB/UI, ... haben keinen Einfluss auf use cases
- Einfluss nur von Änderungen der Anwendung
  - ↳ Änderung der use cases
  - Code der use cases ändert sich

## Interface adapters

- Datentransfer zw. der Layern
  - set von Adapters
  - konvertiere Daten von Entities/Use cases Format nach DB/web Format

## Grenzen überschreiten

- Controller, Presenters kommunizieren mit use cases
  - ↳ dependency inversion principle (siehe SOLID)

## Vorteile

- Unabhängigkeit von Frameworks
  - bei gleichzeitigem Nutzen der Vorteile von Frameworks
- System testbarer
- UI unabhängig
- Datenbank unabhängig
- Unabhängigkeit von externen agencies

## Beispiele

- Hexagonal Architecture
  - = Ports and Adapters
- Onion Architecture
- Boundary / control / Entity BCE
  - Variation von MVC

# Components

≈ funktionaler Baustein für Software

## Definition Komponente

- Einheit der Komposition, kann von anderen verbaut, wieder verwendet werden
- vertraglich spezifizierte, kontextabhängige Schnittstellen
- kann unabh. verwendet werden, auch von Dritten
- kann verwendet werden ohne internas zu verstehen

↳ System kann aus Komponenten zusammen gebaut werden

## Object ≠ Component

Vererbung vs. Black-Box Prinzip

```
class A {
    public T m {
        ... x(); ...
    }
    public T x { ... }
```

```
class B extends A {
    public T x() {
        // redefinition
        ...
    }
}
```

```
B b = new B();
b.m();
```

// B der A nicht kennt weiß nicht, dass  
// er durch Änderung von x auch m  
// geändert hat

→ Vererbung mit Überladen: Code der Oberklasse muss verstanden sein  
≠ Black-Box Prinzip

## mimic Components in Java

- 1) Fassade → Problem: Komp. muss Abh. nach außen spezifizieren  
↳ nicht mögl. in Java  
→ Lösung: dependency inversion
- 2) Packages (unschöner)

## Component Models

- Komponenten sind feature-orientiert
- liegen im Application-, Domain-Layer
- Definiert ...
  - ... was Komponente ist
  - ... wie Komponente Service bereitstellt
  - ... wie Komponenten verbunden / kombiniert sind
  - ... wie Komponenten kommunizieren
  - ... wo Komponenten liegen

## Technische Umsetzung von Komponenten

- viele Ansätze, zB CORBA, CCM, Java Beans, ...
- Problem: meistens objekt-orientiert statt komponenten-basiert  
↳ oft sind 'Komponenten' dann nicht kombinierbar ☹

## OSGi: Open Service Gateway Initiative

- Bundles als Komponenten
  - JAR files mit public Interface (via manifest)

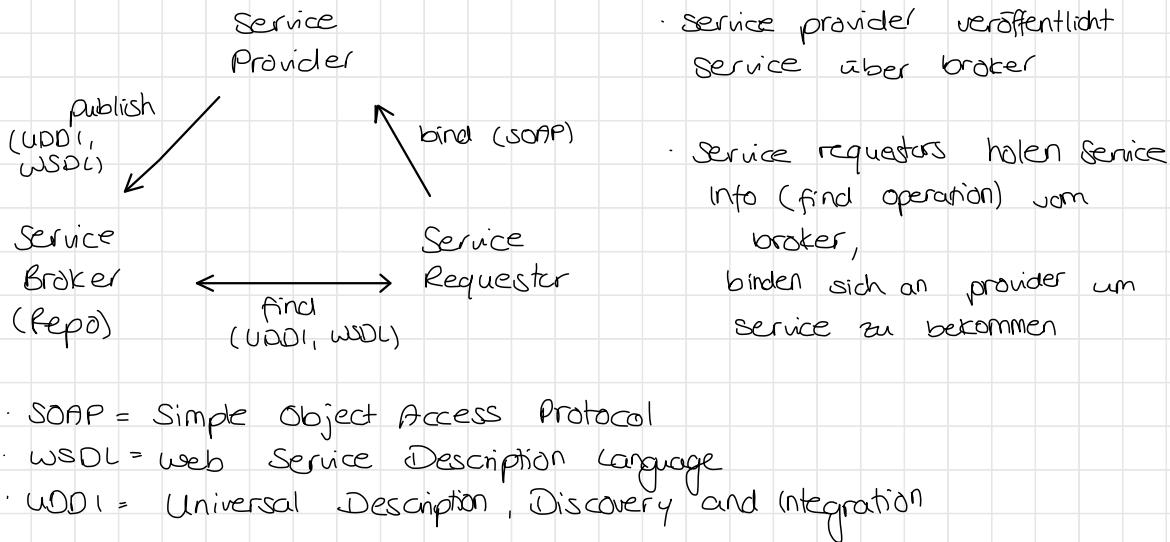
## Web Services

- self-contained: Funktionalität, Attribute im public Interface spezifiziert, Implementierung versteckt
- self-describing: maschinen-lesbare Beschreibung des Interfaces
- modular: wiederverwendbar, kombinierbar für higher level Funktionalität
- published: registrierbar in "yellow-pages" damit einfach zu finden für andere Anwendungen
- located: fixe, globale Location, identifizierbar über URL
- invoked: über standard Protokoll abrufbar

↳ web services = deployed components

Untersch. web services für user gebrauchsfertig  
Components müssen vom dev noch verbaut werden  
↳ keine direkte Nutzung vom user

# Service Oriented Archikture



## Component Models

- SOFA: protocol checking, compositionality
- ROBOCOP: consumer electronics, Analyse nicht-funktionaler Anforderungen
- Kobra: support for Software-Produktlinien
- Palladio

## Palladio: component Model (PCM)

- domain specific modeling language (DSL)
- frühe Performance vorhersagen für Software Architekturen

Performance Metriken aus dem Modell auswerten mittels

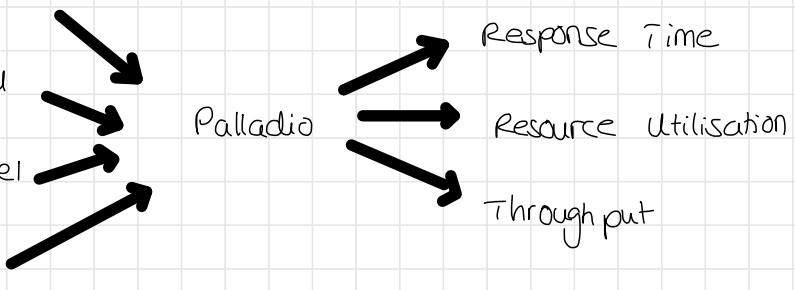
- Simulation
- Analyse techniken

Component Model

Composition Model

Deployment Model

Usage Model



## Performance

Beinflusst durch

- Implementierung
- Nutzerverhalten
- Plattform auf der deployed (HW, OS, ...)
- externe Dienste (zB cloud services)

↪ alle explizit definiert im PCM

## Context Changes

Ändere Kontext in der Simulation

- Allocation context: Änderung des Execution Systems (HW, middleware, ...)
  - ↪ sizing / scalability / relocation
- Usage context: Art der Nutzung
  - ↪ Anzahl User, ...
- Assembly context: Austauschen / Umändern des Zusammenspiels versch. Komponenten

# Submodels

Component  
Developer

Software  
Architect

System  
Deployer

Domain  
Expert



Components

System Design

Deployment

Usage

## Component Developer

PCM tasks:

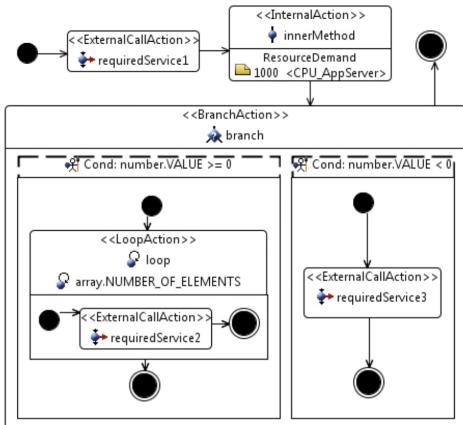
- spezifizierte Komponenten + Interfaces
- spezifizierte Datentypen
- bauen kombinierter Komponenten
- parametrisierte service effect specifications (SEFF) erstellen
- Modellierungs-, Implementierungs artefakte in Repos speichern

General tasks:

- Komponenten implementieren, testen, maintainen

Verhaltensspezifikation: Service effect specification (SEFF)

- beschreibt nach außen sichtbares Verhalten des Services einer Komponente
- Abstraktion des internen Verhaltens
- beschreibt Beziehung zu required/provided Komponentenseiten
- parametrisiert mit Variablen



## Komponenten Performance

- Dev soll schnell verwendbare und wieder-verwendbare komp. entwickeln ohne Kontext zu kennen
- Performance abhängig vom Kontext

## Unabhängigkeit von externen Ressourcen

- modelliert externe Service calls explizit als SEFF
- nur Interface gebundene
  - ↳ tatsächliche calls vom assembly abhängig

→ Kombiniere SEFFs zu Verhalten des ges. Systems

## Parametrisierung

### Resource Environment

- internes Verhalten spezifiziert Ressourcenbedarf der Komponente
- beschrieben als abstrakte Einheiten  $10 < \text{cpu} >$ ,  $5 < \text{hdd} >$

### Nutzungsverhalten

- Ressourcenbedarf abh. vom Nutzungsverhalten
- Abstraktion von Nutzerdaten
  - ↳ performance relevante Infos (Filegröße,...)

## Kombiniert Komponenten

Komponente + Komponente = Komponente

↳ Zusammenbauen von composite components aus basic components

## Software Architect

- spezifiziert system Architektur aus ex. Komp. und Interfaces
- spezifiziert neue Komp., Interfaces
  - die von Comp. dev. entwickelt werden sollen -> Delegiert
- nutzt Architekturstile, -patterns
- analysiert Architektspezifikation, trifft Designentscheidungen
- Performance Vorhersage basierend auf Architektur
- leitet Entwicklungsprozess

## System deployer

PCM tasks:

- modelliert Laufumgebung (Ressourcen: OS, hardware, ...)
- modelliert Ressourcenallokierung der Komponenten

General tasks:

- Aufsetzen der Laufumgebung (HW konfig., SW installieren, ...)
- Komponenten deployen auf Ressourcen
- System maintainen

→ Abstrahierte Ressourcen jetzt spezifiziert: timing values bestimmen

## Domain Expert

- kennt Einsatzkontext
- ↳ spezifiziert Nutzerverhalten
  - Anzahl user
  - Anfragen der User ans System
  - Art der Inputparameter

## Usage model

- modelliert Userverhalten
- ähnlich SEFF

## Resolve Dependencies

- Nachdem Modell gesamthaft gebaut, also
  - Component Dev: Components, SEFFs
  - Software Arch.: Assembly
  - System Deployer: Ressource env
  - Domain expert: Usage

↳ alles in Simulator werfen

Ergebnis: Diagramm

x-Achse: Response time

y-Achse: Wahrscheinlichkeit

# Enterprise Applications

- > Software für Unternehmen, insbesondere
    - Unterstützung der Unternehmensprozesse
    - Businesstransaktionen und Daten
  - hauptsächliches Thema: anzeigen, manipulieren, speichern von Daten
- ⚠ ≠ Business Architecture (= Businesslogik und Infrastruktur organ.)

Beispiele:

- Online shops
- Patientenakten Management System
- Shipping tracking
- Leasing system

Gegenbeispiele:

- OS
- telekomm. Systeme
- plant controllers

## Eigenschaften

- Persistente Daten
  - langlebiger als Software / Hardware
  - existierende Daten müssen integriert werden
- große Datenmengen → effizienter Zugriff nötig
- zeitgleicher Zugriff auf Daten → Fehlerquelle (Inkonsistenzen,...)
  - hunderte Nutzer gleichzeitig
- versch. user interface screens (für erfahrene & unerfahrene Nutzer)
- Interfaces zu anderen Systemen
  - ↳ muss in Softwarelandschaft integriert werden
- Konzeptuelle Dissonanz (versch. Leute verstehen Begriffe anders)
- Business rules/process muss nicht zwingend logisch sein
  - ↳ rules, processes komplex aber oft unveränderbar
  - ↳ logic muss sich anpassen

## Beispielsysteme

### 1) web Shop

- viele Nutzer
  - einfache, gut definierte Business logic
- Issues:** scalability, performance

### 2) Leasing management system

- komplizierte business logic
- wenige gleichzeitige Nutzer

**Issues:** complexity, maintainability

### 3) Expense Tracking for small company

- wenige Nutzer
- einfache business logic
- schnelle, billige Entwicklung nötig
- muss erweiterbar sein

**Issues:** time-to-market, extensibility

## EA Layers

PRESSENTATION  
(Front-End)

- Interaktion Software ↔ User
- Anzeige der Infos
- Eingaben interpretieren

DOMAIN  
(Middle, Business)

- eigentliche Anwendung
- Datenmanipulation

DATA SOURCE

- Kommunikation mit anderen Systemen, zB Datenbank

## Pattern Families

- Pattern Family = versch. Patterns die gleiches Problem lösen
  - Pattern je spezifisch für Probleme eines Layers
  - Interaktion mit anderen Patterns (zB anderer Layer)
  - Patternauswahl abhängig von
    - Systemanforderungen
    - Vr- / Nachteile eines Patterns
    - Keine isolierte Entscheidung → von anderen Patterns abhängig
- ! Pattern ≠ Lösung → Pattern ist Ansatz

## Domain Layer: Domain Logic Patterns

- Domain logic: Repräsentation der Business logic (Geschäftslogik)
- Mögliche Challenges:
  - hohe Komplexität der business logic
  - muss änderbar sein
  - muss verbindbar sein mit Präsentation, Datenquelle(n)

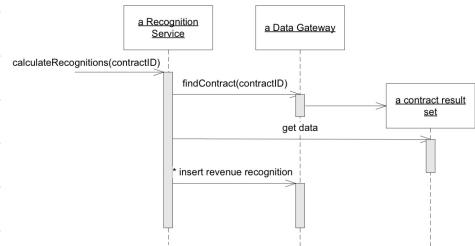
## Transaction Script

≈ shell script das alles nacheinander ausführt

- eine Prozedur pro TransaktionsTyp
- Subroutinen für gleiches Verhalten

- (+) · 1 Skript pro Use Case → leicht austauschbar/erweiterbar  
· einfache Prozeduren  
· einfache Verknüpfung zur Datenquelle  
· Transaktionsgrenzen leicht festzustellen

- (-) · skaliert schlecht bei komplexer logik  
· häufige Codeduplikation

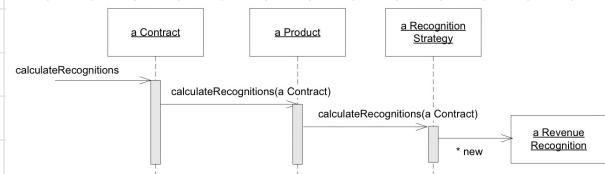


## Domain model

- OO - Ansatz
- Sortierung nach Konzepten + deren Logik
- Objekte arbeiten zusammen

+ komplexe domain logic gut organisiert

- mapping auf Datenquelle schwierig  
 ~ OO-Objekte (Vererbung,...) auf relationale DB?

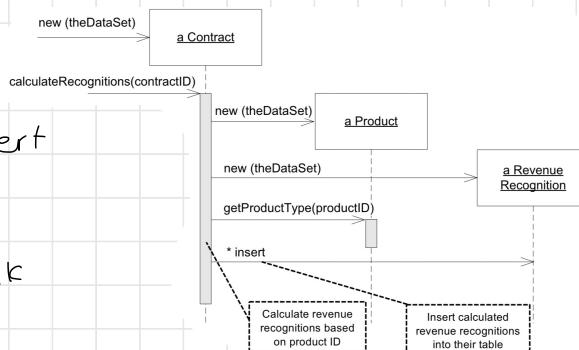


## Table module

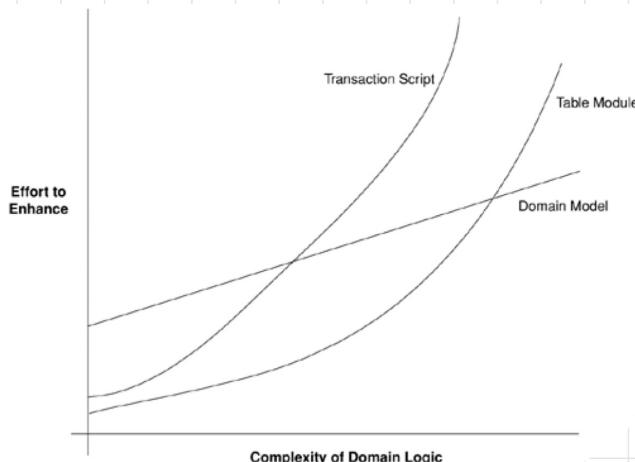
- eine Klasse pro Tabelle (oder View / Query)
- Objekte haben keine Identität

+ Mapping auf Daten offensichtlich  
 · Logik nach Konzepten separiert

- keine Objektinstanzen  
 ↳ schlecht bei komplexer Logik



Übersicht: wann welches Pattern? (vereinfacht)



- web shop: transaction script
- leasing sys.: domain model
- expense tracking: all possible

## Data Source layer: Architectural patterns

Domain Logic Patterns: wie organisiere ich Geschäftslogik?  
JETZT: wie Zugriff auf relationale DB modellieren?

- Architektur auf Data Source Ebene

↳ Ziel: Separiere domain logic und DB Zugriff

### OO vs DB

#### OO:

- Objekte mit Daten + Operationen
- Referenzen
- Aggregation und Vererbung

#### Relationale DB:

- flache Tabellen
- Transaktionen (ACID)
- relationale Algebra

### Record Set

- in-memory Objekt das aussieht wie SQL-Query-Ergebnis  
↳ einfach generiert und manipuliert von allen Teilen des Systems

Record Set = mehrere Tabellen; Tabelle = mehrere Zeilen, Spalten

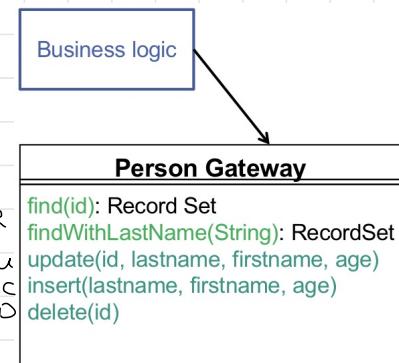
- normalerweise von Frameworks generiert
- wähin mit access code?

### Table data gateway

- Objekt als Gateway zu einer DB Tabelle  
↳ eine Instanz behandelt alle Zeilen einer Tabelle
- einfachste Lösung

- Separierung von SQL statements  
(queries, modifications) und code der Daten verarbeitet

- Sollte CRUD (= Create, read, update, delete)  
implementieren
- ! unnütz für domain model impl.



Gateway = Fassade?

-> Fassade: general use

Gateway: special use

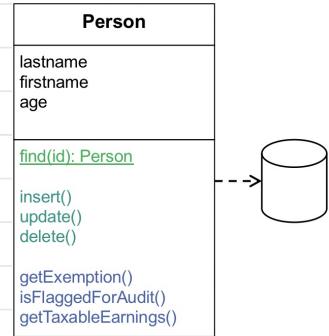
## Active Record

· Objekt repräsentiert eine Zeile, Domain logic in dem Objekt  
↳ OO approach: Daten + Logik an einer Stelle

· jeder active record verantwortlich für modifications, queries

· Domain logic operiert direkt in / auf active records

· Klassen passen gut zu DB Modell  
→ pro Spalte ein Attribut



⊖ Schlechte Separation of concerns

· DB Schema und Active Record müssen isomorph sein  
· nicht gut bei komplexer business logic  
↳ keine direkte Objektrelation, Vererbung

## Row Data Gateway

· Objekt = einzelner DB Eintrag, eine Instanz pro Zeile

· ≈ Active Record, aber

· strikte Teilung: Separation of concerns

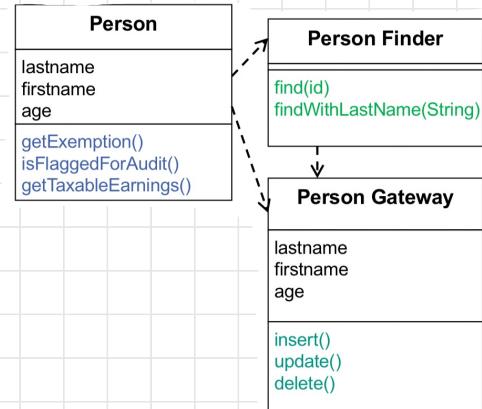
· Domain logic

· Datenmodifikation hinter Interface versteckt

· "finder" Klasse

↳ Attribute ≈ Spalten

· kann automatisiert erzeugt werden

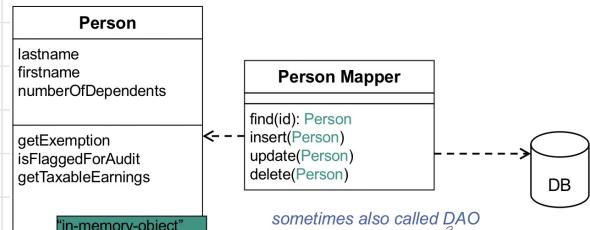


## Identity Map

- = Singleton gekoppelt auf DB-Zugriff
- Objekt wird genau einmal aus DB geladen

## Data Mappers

- Mapperschicht die Daten zw. Objekten und DB transferiert  
↳ mapping zw. ObjektSchema, DBSchema komplex
- Client ruft Mapper auf, der DB Zugriff implementiert
- ein Mapper pro domain Objekt



## Wann was benutzen?

- Transaction Script:
  - Row Data Gateway: explizites Interface, einfacher zu verändern
  - Table Data Gateway: falls Record Set bereits vorhanden
- Domain model:
  - active Record
  - komplexes mapping: Data Mapper
  - Gateways schlecht, da Kopplung zu DB Schema zu stark
- Table module
  - table data gateway (falls Record Set framework)

## Object - Relational Structural Patterns

- Muster die in Gateways verwendet um Objekte auf DB abzubilden
- hauptsächlich für Domain Modell mit Data Mapper

↪ Inheritance Patterns: wie Inheritance mappen?

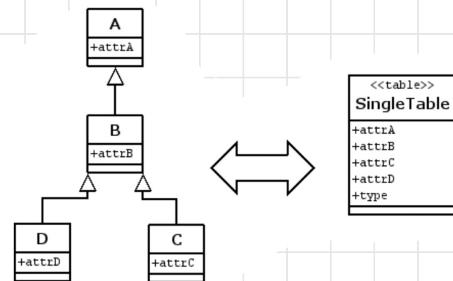
### Single Table Inheritance

Vererbungsbeziehung als eine Tabelle  
↪ extra Feld für Typ

(+) einfaches DB Schema

keine joins nötig

refactoring von Attributzugehörigkeiten von Ober-/Unterklassen ohne DB Änderungen möglich



(-) viele unbenutzte Felder ( $\rightarrow$  C hat kein attrD)

große Tabellen, viele Indizes  $\rightarrow$  Performance loss oder zusätzliche Indextables

nur ein namespace für alle Felder

### Class Table Inheritance

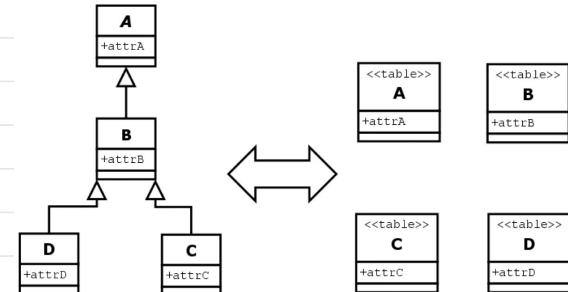
Eine Tabelle pro Klasse

(+) einfaches Mapping

keine unbenutzten Felder

(-) Objekt laden erfordert viele joins (Attributvererbung)  
 $\hookrightarrow$  Performance loss

refactoring der Attribute in Hierarchie erfordert DB Änderung  
Supertypes werden bottleneck



## Concrete table inheritance

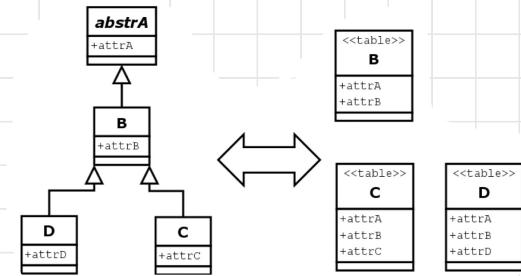
- eine Tabelle pro konkreter Klasse
- Attribut explizit nach unten durch kopieren

(+)

- keine unnötigen Felder
- keine joins nötig

(-)

- refactoring erfordert Schemaänderung
  - Änderungen im Supertyp erfordert Änderungen im Subtyp
  - Query der Superklasse erfordert check / join der Subklassen



## Hilfreiches Tool : Java Persistence API

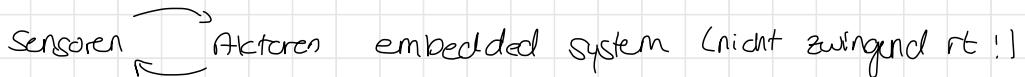
Mapping nicht selbst implementieren

- Annotationen erlauben definieren des Mappings  
  @Inheritance(strategy = InheritanceType.SINGLE\_TABLE)
- ↳ supportet alle obigen strategies

# Real-time Development

## Real-time System

- System beobachtet, regelt Umgebung  $\rightarrow$  Regelsystem  
↪ embedded system
- Zusammenarbeit mit HW:
  - Sensoren: Daten der Umgebung aufnehmen
  - Aktoren: Umgebung verändern



- System muss in spezifizierter Zeit antworten
- Ergebnis der Datenverarbeitung korrekt, falls logisch **UND** zeitlich korrekt
- Soft real-time System: Operationen degradiert falls nicht in Zeit
- Hard real-time System: Operationen incorrect falls nicht in Zeit

## Eigenschaften

- Spezifiziert durch mögliche Stimuli, erwartete Antworten und Zeitconstraints
- periodic stimuli: treten in periodischen Zeitabständen auf
- aperiodic stimuli: Auftreten unvorhersehbar

## Interrupts vs Periodic Processes

### Interrupts:

- Kontrolle abgeben an definierte Speicherstellen mit Behandlungsschema  
 $\hookrightarrow$  ISR = Interrupt Service Routine (ISR)
- weitere Interrupts ausgeschaltet
- nach Interrupt handling: Kontrolle zurück an unterbrochenen Prozess
- ! ISR müssen schnell, kurz, einfach sein

## Periodic processes:

- RT - Systeme enthalten oft viele periodische Prozesse
- PPs haben versch. Perioden, Ausführungszeiten, Deadlines
- periodische Uhr sorgt für Interrupts

## Monitoring, control systems

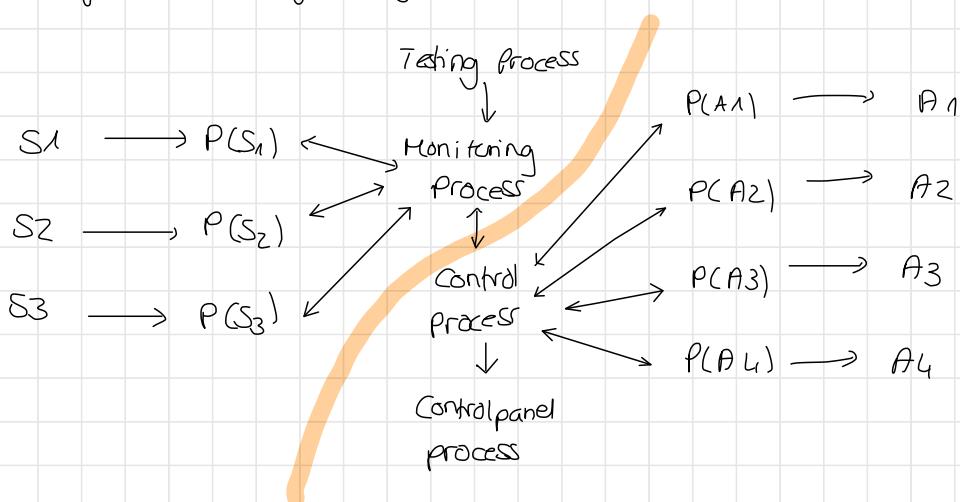
- Arten von RT - Systems

## Steuerungssysteme (= monitoring systems):

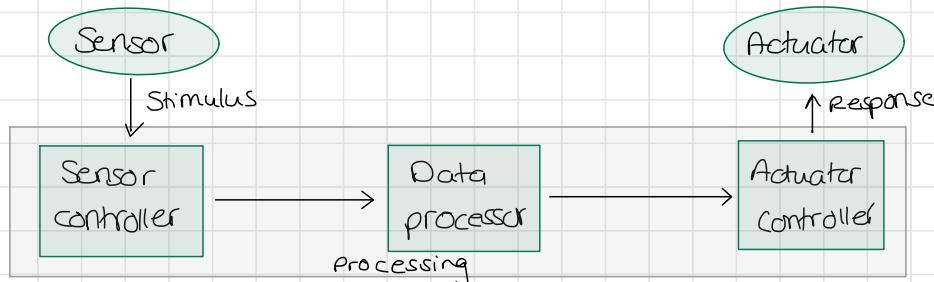
- handeln wenn bestimmte Sensorenwerte gemessen

## Regelsysteme (-control systems):

- regeln kontinuierlich HW Aktoren basierend auf Sensorwerten
- Regelkreis = regeln gleiche Werte die Sensoren messen

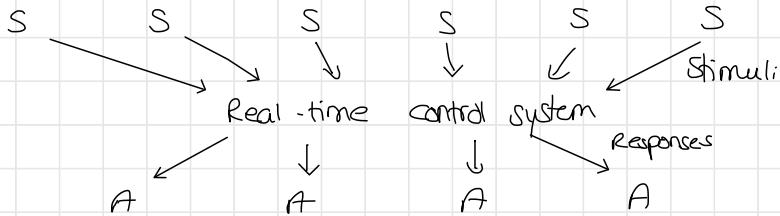


## Schema Regelsystem



## Design

- versch. Zeitconstraints → jedes Sensor/Aktuatorpaar sollte eigenen Prozess bekommen
- Systemarch. muss schnelles switchen unterstützen
- Kooperierende Prozesse, kontrolliert durch Real-time Platform

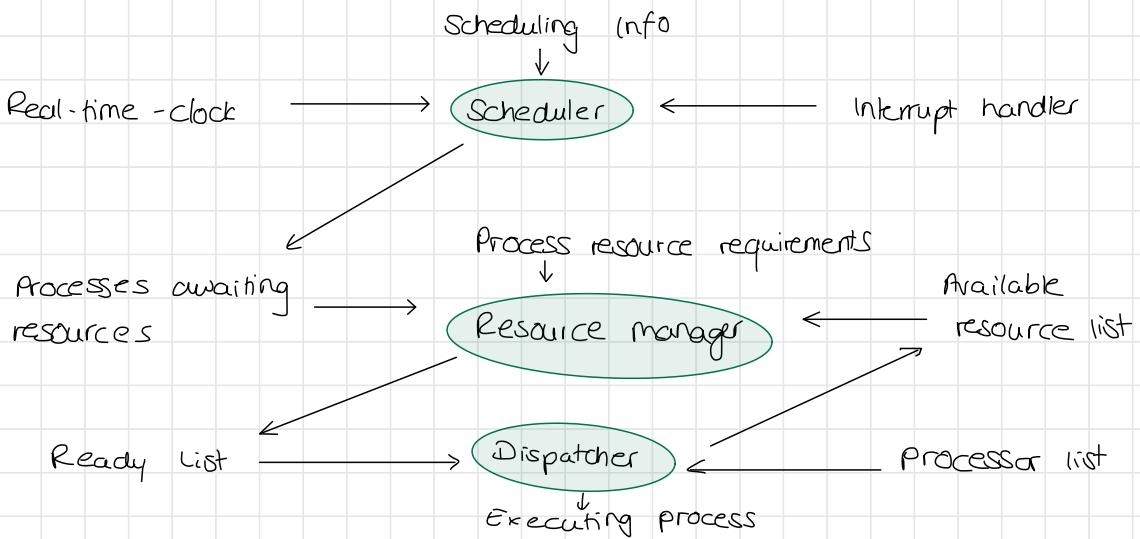


## Real-Time Operating System

- kein FS, UI support, ... nötig → spezielles OS

### Komponenten

- real-time - clock → für scheduling
- interrupt - handler → für aperiodische Service requests
- Scheduler
- resource manager → allokiert Speicher, Prozessoren
- dispatcher → startet Prozessausführung



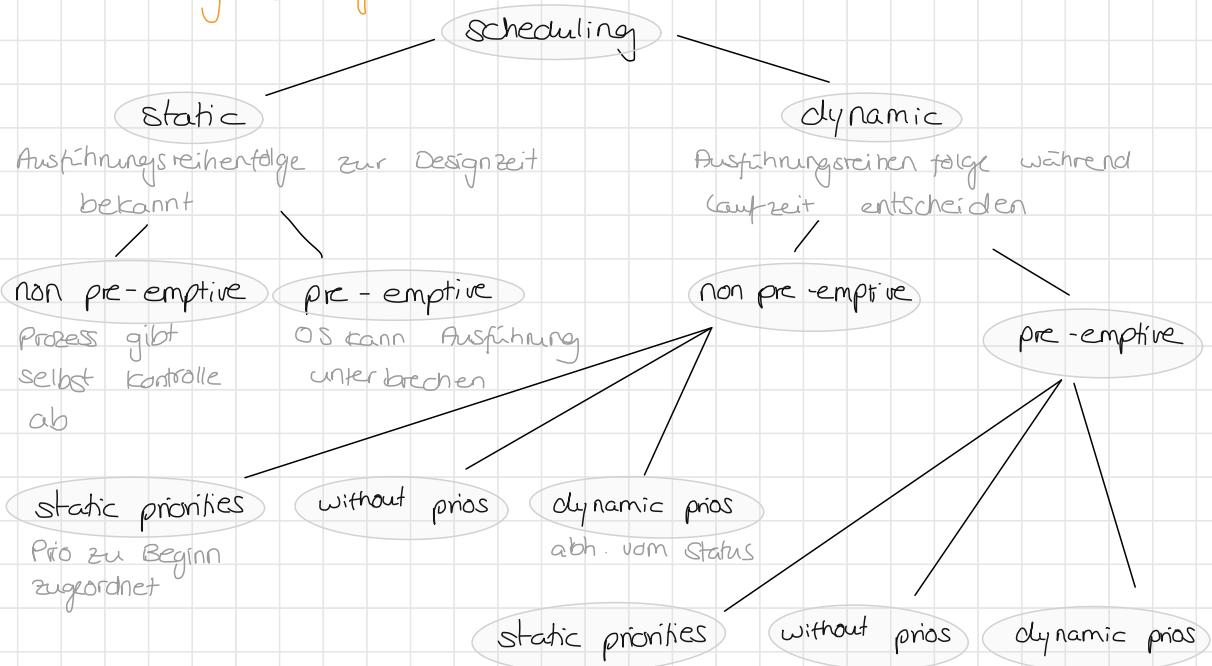
## Process Management

- Manages parallel processes
- Scheduler nutzt real-time-clock und berücksichtigt
  - arrival pattern
  - execution times
  - priority der Prozesse
- mind. 2 priority levels:
  - interrupt level priority  
↳ für Prozesse die sofort laufen müssen
  - clock level priority  
↳ für periodische Prozesse

## Ablauf:

- Scheduler wählt Prozess
- Resource manager allokiert Speicher + Prozessor
- Dispatcher holt Prozess aus der ready list, schiebt ihn auf Prozessor, startet Ausführung

## Scheduling Strategien



## Bsp:

- FIFO: dyn., n pc, without prios
- Fixed prios: dyn., static prios
- Earliest - Deadline - First: dyn., dyn prios
- Least - Laxity - First: beachte zusätzlich zu deadline verbleibende Ausführungszeit (RT-Programm hat worst case time!)  
laxity = deadline - now - remaining ex. time
- Time - slice: dyn., pe, without prios

## RT system Design

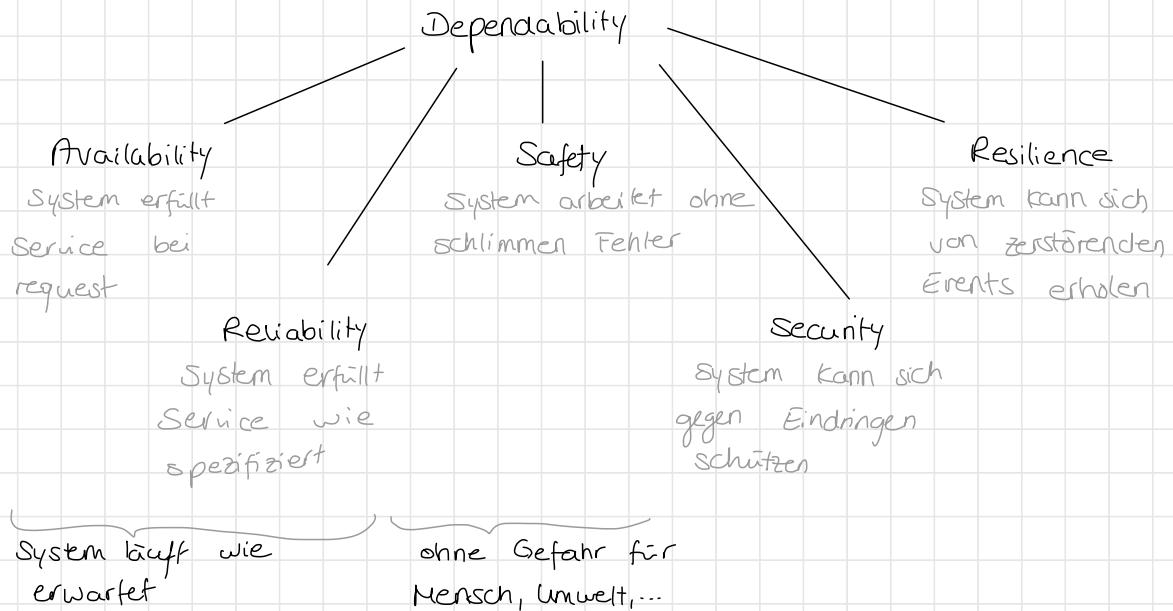
- hardware software co-design
- beachten von Qualitätsanforderungen (Performance, changeability...)
  - spezialisierte HW:  $\oplus$  bessere Performance  
 $\ominus$  langer Entwicklungsprozess, schlechte Changeab.
- Timing constraints prüfen mit ausgiebiger Simulation + Prototyping
  - $\hookrightarrow$  Beweis dass RT schwer!

## Design - Prozess

- Reihenfolge abh. vom zu entwickelnden System
- wähle execution platform: hardware + RTOS
  - bestimme Stimuli + erwartete responses
  - definiere zeitconstraints zu stimuli + response
  - Stimuli + responses zu concurrent prozessen sortieren
  - Algo design für stimuli + responses
    - $\hookrightarrow$  endliche Automaten helfen beim modellieren  
State charts bei komplexen Problemen
  - spezifizierte Daten die ausgetauscht werden
  - Scheduling system designen

# Dimensions of Dependability

## Qualitätsattribute



Reliability: Relation um Kontext, erwartete Antwort, Nutzerprofil zu definieren

Correctness: z.B. Code tut was spezifiziert wurde

correct + reliable: ja, falsche Spezifikation

reliable + !correct: ja, falsche Spezifikation oder korrekte Spezifikation aber user profil aus rel. trigger + Fehler nicht

Safety: keine Gefahr für Mensch, Umwelt

Reliability: Wkt Dauer einer failure-freien OP  
↳ reliab. erhöht durch Redundanz

Safety != Reliability

- Safe systems: dürfen fehlen, sind aber keine Gefahr
- fail-safe state: Systemzustand in dem keine Gefahr besteht  
↳ nicht immer möglich

## Fault - Error - Failure

### Fault:

- Systemdefekt
- Kann zu Failure führen
- = Bug
- Systematic = design fault, Fehler zur Design / build Zeit
- Random hat mal funktioniert, jetzt nicht mehr
  - transient: treten nach einer Weile auf → retry hilft
  - persistent: bleiben bis manuelle Intervention → neue HW, ...

### Error:

- Diskrepanz zw. beabsichtigtem und tatsächlichem Verhalten
- tritt zur Laufzeit auf wenn System unerwarteten Zustand betrifft

### Failure:

- Systemverhalten anders als spezifiziert

## Real - Time Patterns

### Safety + Reliability patterns

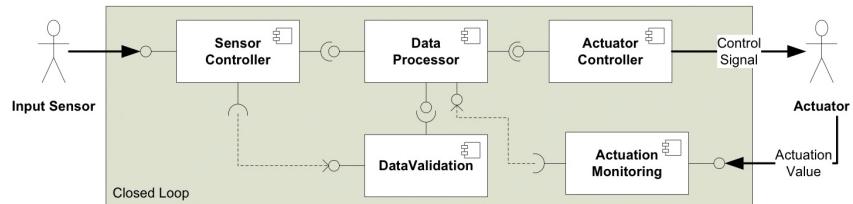
## Channel

- = Pipeline die sequentiell Eingabedaten in Ausgabedaten transformiert
- interne Elemente ≈ Fabrikarbeiter
  - ↪ einfache OP je Element
- viele Channels für bessere Qualität
  - gleiche Channels → Performance
  - viele versch. Channels → fault tolerance → Reliability
  - Channels für fault identification, safety measures → Safety

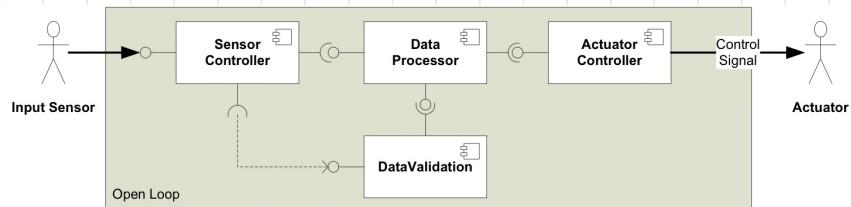
## Protected Single Channel

- keine Redundanzen
- Error detection im channel
- Annahme: es gibt fail-safe Zustand
- kann transient faults handeln (z.B. durch retry)

Closed Loop:

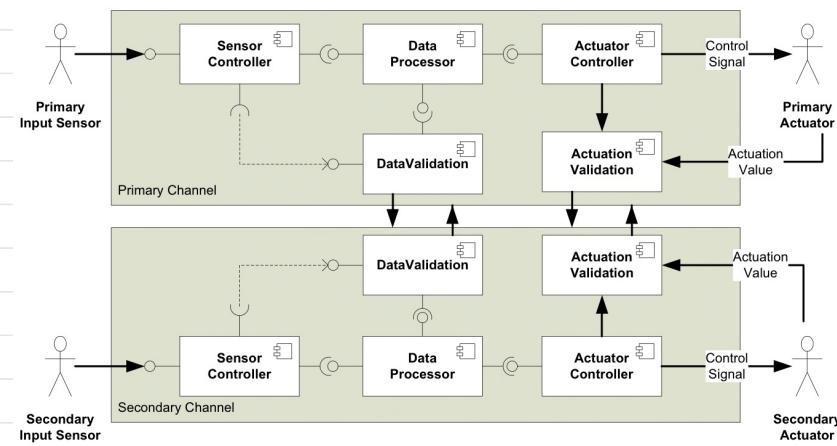


Open Loop:



## Homogenous Redundancy

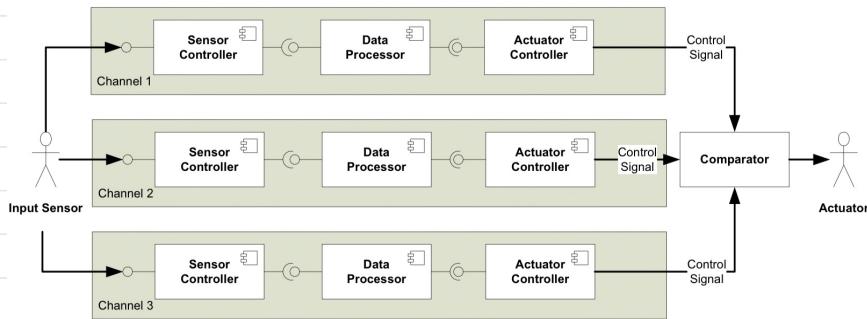
- Schutz gegen random faults
- kein fail-safe Zustand
- System wird doppelt entwickelt (Hw + SW)



## Triple Modular Redundancy

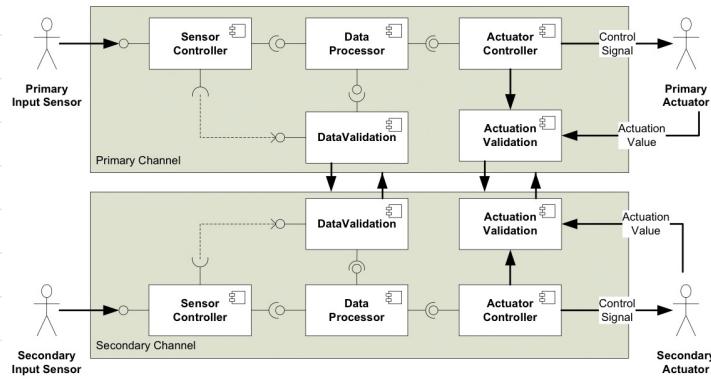
Schutz gegen random faults

Kein fail-safe Zustand



## Heterogeneous Redundancy

- Design + Implementierung unabhängig (versch. Devs,...)
- Schutz gegen random und systematic failures (sofern nicht gleiche Fehler bei beiden Versionen)
- Kein fail-safe möglich

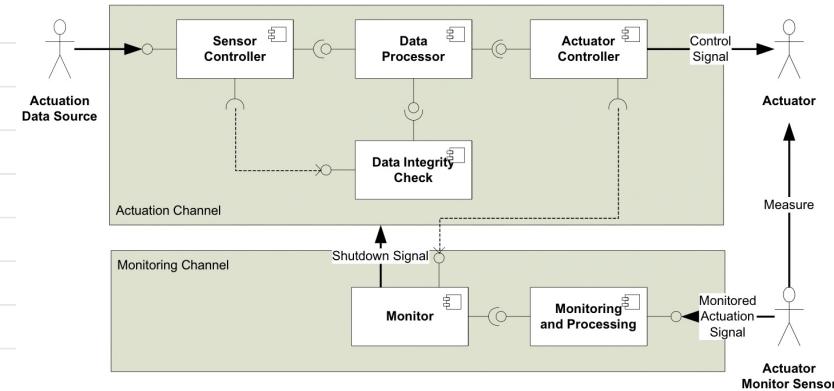


## Redundanz als Lösung?

- Software: von versch. Entwicklern entwickelt  
↳ keine statistische Unabh. → gleiche Spezifikationen, ähnliche häufige Fehler (off-by-one,...)
- Hardware: gleiche physische Einflüsse

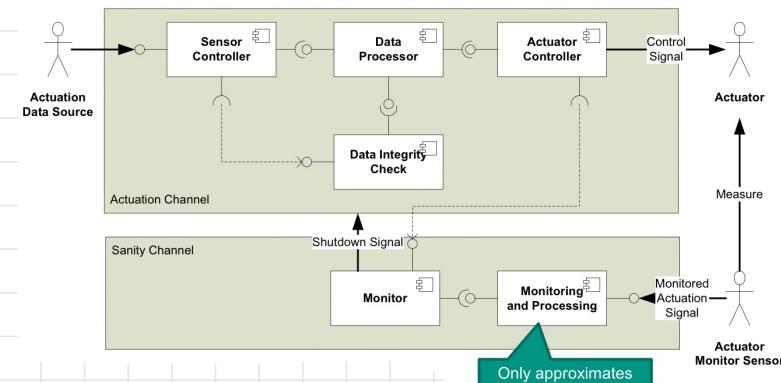
## Monitor - Actuator

- Channel + Actuator werden überwacht
- ↳ Actuator Monitor Sensor = unab. Sensor
- Schutz gegen random, systematic faults
- fail-safe state nötig



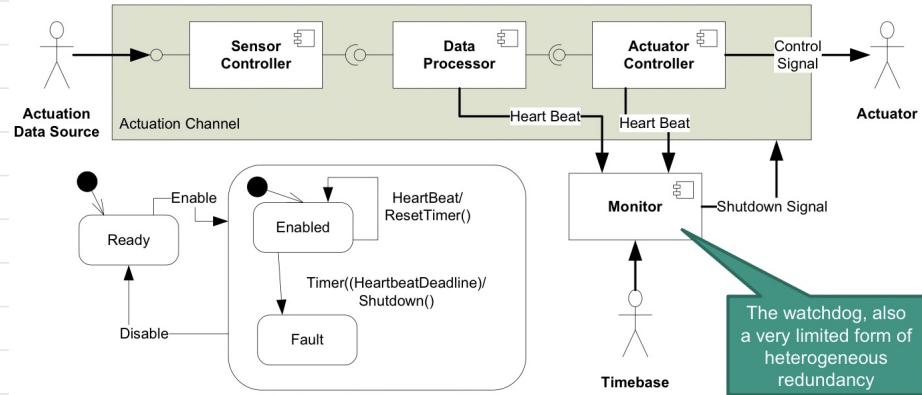
## Sanity Check

- leichtgewichtiger Schutz gegen random, systematic faults
- benötigt fail-Safe-state
- ↳ Monitor- Actuator Pattern, aber nur approx. der Ergebnisse



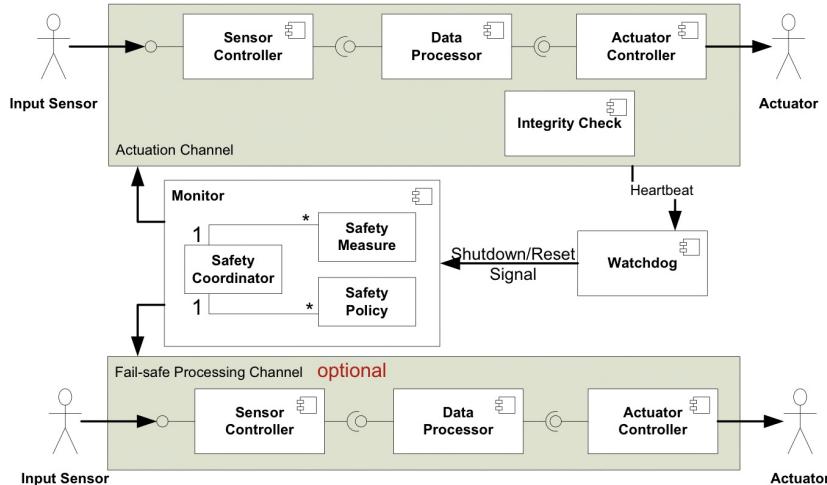
## Watchdog

- Sehr leichtgew. Schutz gegen zeitbasierten faults
- Deadlock Erkennung
- fail-safe-state nötig
- Fehlererkennung auf channel, keine Überwachung des Actuators
- "lebt System noch?"



## Safety Execution

- für komplexe Systeme mit nicht-trivialen Übergängen zum fail-safe-Zustand

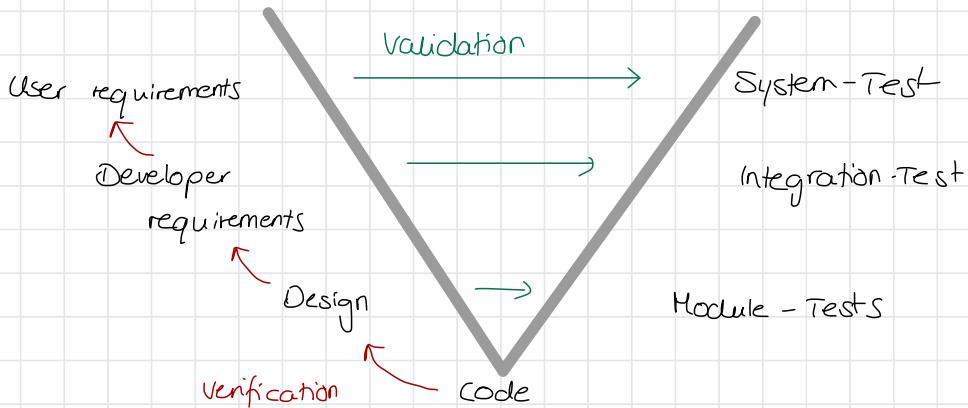


# Reviews

## V-Model

- versch. SW Artefakte und deren Relationen

Business-/System Model



Validation: Fall-basierter Check nach erwartetem Verhalten

Verification: check ob refinement Beziehung stimmt zw. 2 Dokumenten

- req. Specification + Code
- req. Spec. + architecture / design
- arch. / design + code

## Review Process

Review = meeting bei dem SW Artefakt untersucht wird

↳ zur Qualitätsverbesserung, besonders die Korrektheit

• können bis zu 60-90% der Fehler vor dem testing finden

## Gefahren von Reviews

- kein Testing mehr → "Fehler werden schon beim Review gefunden"
- Implementierer frustriert, fühlen sich angegriffen

=> Gefahren für Reviews

## Gefahren für Reviews

- schlechte Vorbereitung
- Obfuscation: Autor schreibt micky Code / Doku die schwer zu verstehen

## Vorteile von Reviews

- Verbesserung von Qualität, Korrektheit
- Verbesserung des Verständnisses des Projekts
  - ↳ Wissen weitergeben
  - ↳ weniger Abhängigkeit von einer Person
- Vermitteln von Style / Erfahrungen an Neulinge
- Bessere Lesbarkeit → Code und Dokumentation

## Review Phasen

- 1) Planung: was wird von welchen Leuten gereviewed?
- 2) Overview: notwendige Dokumente als Review Package
- 3) Preparation: Reviewer lesen Dokumente
- 4) Meeting: ① Diskussion
- 5) Rework: Umsetzung der Diskussionsergebnisse
- 6) Follow-up: wurde Rework durchgeführt?
- 7) Causal Analysis: ≈ Retrospective

## Rollen im Review

- Autor
  - Moderator
  - Reader(s)
  - Recorder
  - Verifier
- } nicht zwingend versch. Personen  
≈ 4 - 5 Leute

Review Arten	Planning	Prep.	Meeting	Correction	Verification
• Inspection	x	x	x	x	x
• Team Review	x	x	x	x	x
• Walkthrough	x		x	x	x
• Pair Programming	x		continuous	x	x
• Peer Deskchecks		x	possibly	x	x
• ad hoc pass around			x	x	x

## Review Anti-Patterns

how not to

### Alcoholic

- Addicted: mal sehen wie weit ich gehen kann (bad habit)  
↳ zB zu spät, unvorbereitet, ...
- Helper
- Punisher
- Ashamed

### now - I - have - got - you

- A findet Fehler von B  
↳ B ist A ausgeliefert

### see - what - you - made - to - me

- A arbeitet an Task
- B unterbricht
- C A macht Fehler

Review: B schlägt Lösung vor, A setzt um → Lösung falsch  
→ A macht B verantwortlich

### Hurried

- A überarbeitet über will / bekommt mehr tasks  
↳ macht mehr Fehler
- B lädt A mehr Arbeit auf

### if - it - were - not - for - you

- A arbeitet an Task, sagt zu B: ohne dich hätte ich schon XY
- zB A coder, B reviewer (der oft rejected)

### Look - now - hard - I - tried

- A merkt dass Projekt failen wird
- A zeigt wie hart er arbeitet (Mails nachts, ...)
- A sagt B dass Fail nicht seine Schuld da er doch so hart gearbeitet habe

## Schlemiel - Tölpatsch

- Tölpatsch A macht Fehler den B beeinflusst ↗ infinite loop
- B findet Fehler, beschwert sich bei A
- Manager C wundert sich warum B so langsam

## Yes-but

- A schlägt improvement vor
- B: "ja-aber"
- B verteidigt sich gegen jede Verbesserung von A

## wouldn't it be nice if

- A schlägt Lösung vor
- B schlägt Alternative / Add-on vor
- A mehr Ahnung von aktuellem Problem als B  
↳ A fällt ins yes-but pattern

## Requirements

- viele Fehler durch schlechtes requirement engineering
- keine Req.
- falsche, missverständliche Req.
- Stakeholders nicht involviert

## Requirement

- Fähigkeit die vom Nutzer verlangt wird um ein Problem zu lösen  
↳ needed
- Fähigkeit die erfüllt werden muss um einen Vertrag, Standard, ... zu erfüllen → wanted

## Beschreibung von Req.

- 1) Adequate → was will der Kunde
- 2) Complete → inkl. Kontext
- 3) Consistent → keine Widersprüche die impl. verhindern
- 4) Understandable → für Stakeholder + Dev
- 5) Unambiguous → kann nicht misverstanden werden
- 6) Verifiable → kann auf Umsetzung getestet werden
- 7) Suitable for the risk

## Arten von Req.

- 1) Funktional
- 2) Qualitäts (nicht-funkt.)
- 3) Constraints

- Spezifikation ist Startpunkt des Entwicklungsprozess
- Acceptance Test ist Endpunkt

# Requirement Engineering Prozess

kooperativer, iterativer, inkrementeller Prozess von

- Req. Elicitation (Gewinnung)
- Req. Dokumentation
- Req. Agreement (Übereinstimmung)
  - ↳ Konflikte lösen, Req. die meisten zufrieden stellen
- Dazwischen
  - Req. Validierung
  - Req. Verwaltung

↳ Ziel: alle Req. rausfinden und im nötigen Detail dokumentieren  
↳ Prozess bis Stakeholder zufrieden

## Stakeholder

- Person / Organisation die Systemreq. (indirekt) beeinflusst
  - Nutzer
  - Kunde
  - Devs
  - Architects
  - Tester
  - System Operator

↳ identifizieren der Stakeholder + deren Req. wichtig!  
→ Fehlende Stakeholder → fehlende Req.

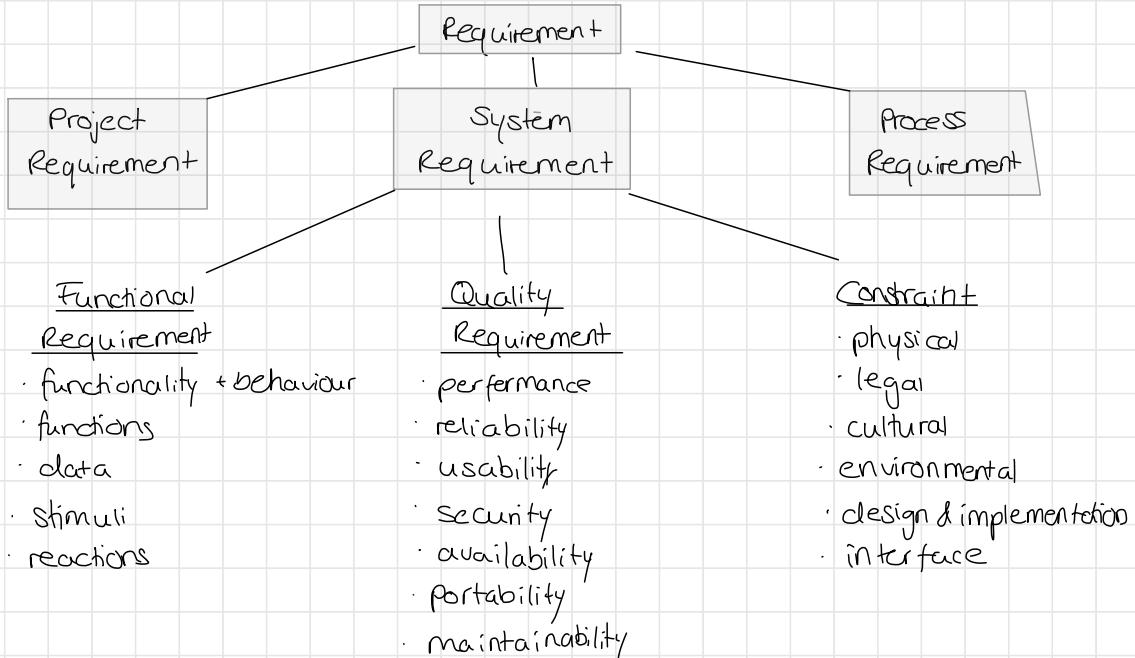
## Requirement Engineer

- übersetzt zw. Nutzern, Entwicklern
- verantwortlich für elicitation, documentation, agreement
- maintains Req. Dokument

## Requirement Elicitation Techniken

- Questioning: Interviews, ...
- Creativity: Brainstorming, Perspektivenwechsel, ...
- Retrospective: Reuse, Competing Systems, ...
- Observation: Field observation, ...
- Supporting actions: Mind Maps, Workshops, Audio/Video recording, ...

# Klassifizierung Req.



Representation

- operational
- qualitative
- quantitative
- declarative

Kind

- functional
- quality
- constraint

Satisfaction

- hard
- soft

Role

- prescriptive
- normative
- assumptive

IC.ind = Klassifizierung

Repres. = wie ausdrücken

Satisfaction = wann ist es erfüllt

Role = prescriptive: beschreibt system-to-be; norm.: Umwelt; assup.: actor

## Requirement Representation

Operational = Spezifikation von Operationen / Daten

↳ verification: review, test, formal verification

Quantitative = Spez. von messbaren Eigenschaften

↳ ver.: messen

Qualitative = Spezifikation von Zielen

↳ ver.: nicht direkt möglich; subjektive Ver.

declarative = Beschreibung einer Anforderung

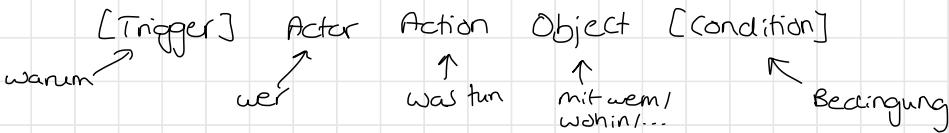
↳ ver.: review

## Vorteile Klassifikation

- besseres Verständnis, weniger unnötige Diskussionen
- gute Validierungsmöglichkeiten

## Requirements Writing

- kurze klare Sätze → pro Satz ein Req.
- aktiv statt passiv (welche Komponente soll Aufgabe erfüllen?)
- "weak" words vermeiden (einfach, schnell, zuverlässig,...)
- ggf. Glossar anlegen für misverstandene Wörter
- Satzbausteine:



- IDs vergeben pro Req. → einfacher zu referenzieren

## Requirement Validation

- Inspections, Reviews, walkthroughs
  - ↳ Fehler manuell finden
- Simulationen
  - ↳ einzelne Systemaspekte simulieren
- Prototyping
- Systemtests
- Model checking → formale Verifikation

## Use Cases

! nur Notation,

ersetzen Req.

nicht

- textuelle Erfassung der Interaktion zweier Partien
- beschreibt Verhalten, Computer System = black box
- hilfreich zur Requirements elicitation

## Goal, Actors, Scopes

- Systemgrenzen identifizieren

### System Scope (Boundary)

- Grenze zw. System und Umwelt
- möglicherweise versch. Grenzen  $\Rightarrow$  einer nach der anderen modell.
- Interface System-Umwelt
  - Informationsein-/ausgabe
  - HW / SW

### System Context

- welche Teile der Umwelt sind relevant für Def. + Verständnis der Requirements
- Modellierung durch Kontextdiagramme

### Context Boundary

- untersch. relevante Umweltinfo und nichtrelevante

$\sim$  Am Anfang Boundaries unklar, werden durch Req. klarer

## Common Scopes

### Business Use Case

#### Design Scope - ges. Unternehmen

- Enterprise Black Box Scope: Enterprise = black box
- Enterprise White Box Scope: Departments + stuff intern explizit

### System Use Case

#### Design Scope = Computer System

- System block box scope: system = black box
- System white box scope: modell. wie Komponenten arbeiten

## Component Use Case

Subsystem / Komponente des Systems

- immer white box

## Spezifizierung Use Case

- Je nach Systemgrenzen, Actor und Zielen versch. Level der Granularität
- Fokus auf Ziel der Interaktion

## Elementary Business Process

Daumnenregel für Computer Anwendungen und user goal use cases  
EBP ist definiert als

- ... task
- ... der von einer Person ausgeführt wird
- ... zu einer Zeit an einem Ort
- ... als Antwort auf ein business event
- ... das messbaren business wert schafft
- ... und die Daten in konsistentem Zustand verlässt

## Heuristiken gute Use Cases

### 1) Boss Test:

Was antwortet man wenn der Chef fragt was man den ganzen Tag gemacht hat

### 2) coffee break test:

Use case fertig wenn man an diesem Punkt eine Kaffeepause machen würde

### 3) size test:

Normalerweise mehr als ein Schritt

## Use Case Goal levels

### 1) High-Level Summary ( $\approx$ business process)

- mehrere user goals + Subfunction use cases

### 2) User Goal (= EBP)

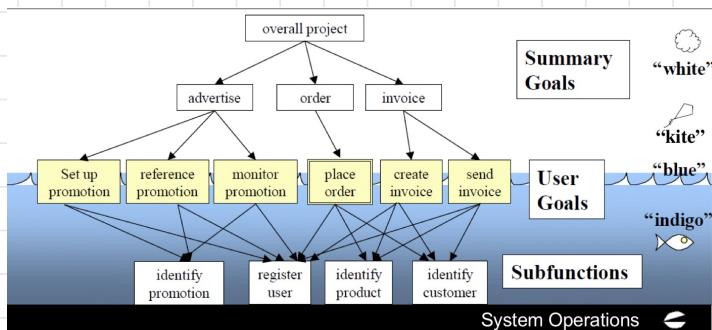
- Beschreibung wie gewünschtes Verhalten erreicht mittels Interaktion mit System

### 3) Sub(function)

- um "Unterziele" auszulagern die nötig um user goal zu erreichen
- normalerweise ohne direkte Wertschöpfung

### 4) Too low (= feature (System operation))

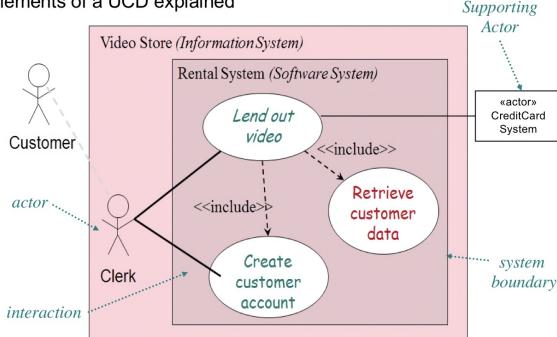
- alles kleiner als Subfunction zu definiert für eigenen Use Case



## Use Case Diagramm

Menge der Use Cases + Beziehungen untereinander

Elements of a UCD explained



<<include>> ~ sub use case "call"

<<extend>> ~ a possible extension, the extended use case is "unaware"

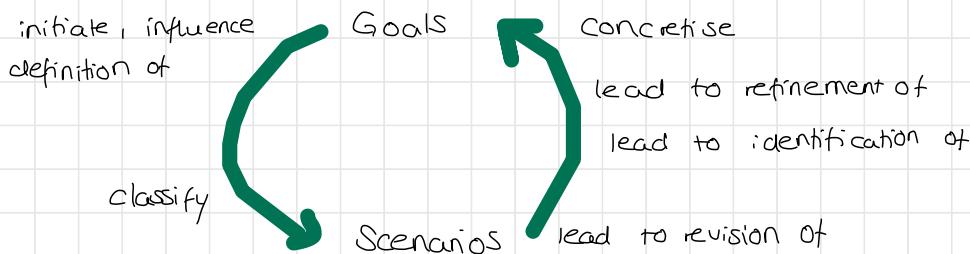
- nur als Überblick verwenden → Text wichtiger
- normal: User Goals modellieren

## Use Case Begriffe

- Stakeholder = Person / Organisation die Einfluss auf Req. hat (interests)
- Actor = Entität mit Verhalten außerhalb des Systems  
zB Nutzer, anderes System
- Primary actor = initiiert Interaktion
- Use case model = Set aller use cases und zugehöriger Diagramme
- Scenario = Sequenz von Aktionen / Interaktionen zw. Actor - System
- Use case = Sammlung zusammengehöriger Erfolg / Failure Szenarien die beschreiben wie Actor System nutzt um User Goal zu erreichen

## Use Cases Finden

- 1) System Boundary definieren
- 2) Primary Actors bestimmen
- 3) Für jeden Actor die User Goals identifizieren
- 4) Use Cases definieren die User Goals erfüllen



## Iteratives Vorgehen

- breadth-first: use cases brainsternen, dann refinen
- ändernde Anforderungen können use cases beeinflussen

## Use Cases Schreiben

- Schritte beschreiben Interaktion, Validierung, interne Änderungen



- keine UI Details
- jeder Schritt = Prozess geht weiter
- Datenbeschreibungen: 3 Level
  - 1: Daten nickname (zB ID)
  - 2: Datenfelder + nickname (zB ID: String)
  - 3: Datentypen + Längen + Validierung (zB ID: String, 13 chars EAN Code)

Templates:

- Fully Dressed
- Casual (nur Text)
- One-Column Table (= Fully Dressed als Tabelle)
- Two-Column Tables (= 1 Spalte Actor, eine Spalte System)
- RUP (= Fully Dressed)

## Fully Dressed

- 1) Preface
  - scope + goal level, primary actors
- 2) Stakeholders + Interest list
- 3) Preconditions
- 4) Post condition (success Guarantees)
  - ↳ was muss erfüllt sein damit success
- 5) main success scenario
  - normal kein if/branching → nur "happy path"
- 6) Extensions (alternative Pfade)
  - alle Pfade beschreiben (success + fail)
  - Conditions + Handling
- 7) Special requirements
  - ↳ nicht funktionale Anforderungen, Qualitätscontr., ... die direkt zu Use Case gehören

## 8) Technology + Data variations

- wie muss etwas gemacht werden
- zB Datenbeschreibungen

## Requirements

- externes Verhalten der Software
- alle Interfaces zw. Software + Umwelt  
WAS statt WIE (System bleibt black-box)

## Analysis

### Requirement Analysis

Investigation of Requirements

## System Sequence Diagrams SSD

- aus Use Cases: wie interagieren System + actors
  - ↳ wer generieren Events die Operationen auslösen
- SSD: Startpunkt für Systemdesign → identifizierte System-Ops
- vereinfachte UML Sequenzdiagramme
- Hauptmerkmal auf Interaktionen über Systemgrenze hinaus
  - ↳ events und deren Reihenfolge

↳ SSDs für main success Scenario + häufige / komplexe Alternativen.

## System Events



## System Boundary

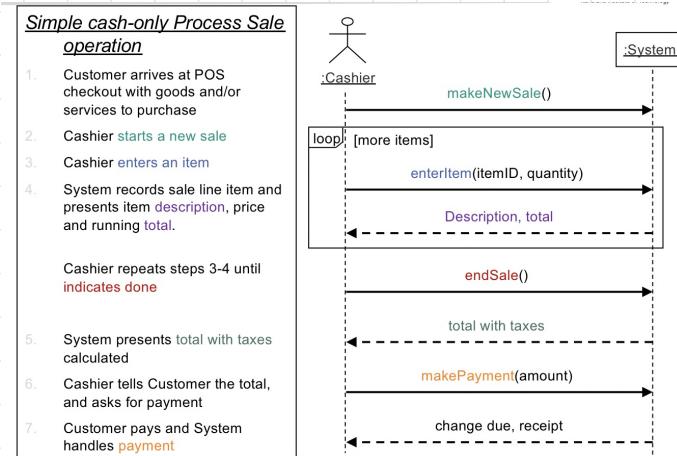
muss klar def. sein um System Events zu identifizieren

- externe Events die direkten Einfluss auf System haben

## System Operations

- stellt System als Black Box bereit → public System Interface
- Namensgebung ≈ Programmieren
  - aussagekräftige Namen
  - Verben verwenden add, enter, end, make, get, delete, ...

## Use Case $\rightsquigarrow$ SSD



## Domain Analysis

- Divide and Conquer strategy
- Structured Analysis: Dimension des Divide = Funktion
- Object-oriented Analysis: Dimension = Dinge / Entitäten (Objekte)

## Operation Contracts

- detaillierte Beschreibung der Systemoperationen
- Analyse der state changes von domain objects  
↳ vor/nach Ausführung der System Op
- Was muss passiert sein damit, ... statt wie wird das erreicht

## Schema

Operation  
(Cross References)

name + params  
optional, use cases in den diese Op auftritt

## Precondition

- Nennenswerte Annahmen über State von System/objekten im Domain Model werden innerhalb der Op nicht getestet und als wahr angenommen

## Postcondition

- State der Objekt im Domain Model nach Ende der Op

## Preconditions

- einfach
- meistens basierend auf Ausführung einer anderen System Op.

## Postconditions

- Zustandsänderungen der Domain Model Objekte bezüglich
  - Instanzerzeugung / -lösung
  - Associationserzeugung / -entfernung
  - Attributänderungen
- sind wahr wenn sys Op fertig
- Formulierung:
  - deklarativ, nicht imperativ (statements, keine Aktionen)
  - Vergangenheit (sind während des Op schon passiert)
  - Metapher: Theater → Vorhang auf / Vorhang zu / Vorhang auf  
= vorher / System Op / nachher
- ! Assoziationen nicht vergessen

## Contracts vs Use Case

- Use case = Basis / main Repository für Requirements
- komplexe State changes, ... schwer in Use Cases zu fassen  
↳ Contracts: Postcond. hilft für klare Formulierung
- Contracts nicht nötig in klaren Kontexten
- Contracts als "best guess" → nicht überspezifizieren  
Contracts als Hilfe zur Verbesserung des Domain Models

# Software Development Process

## Prozessmodell = Vorgehensmodell

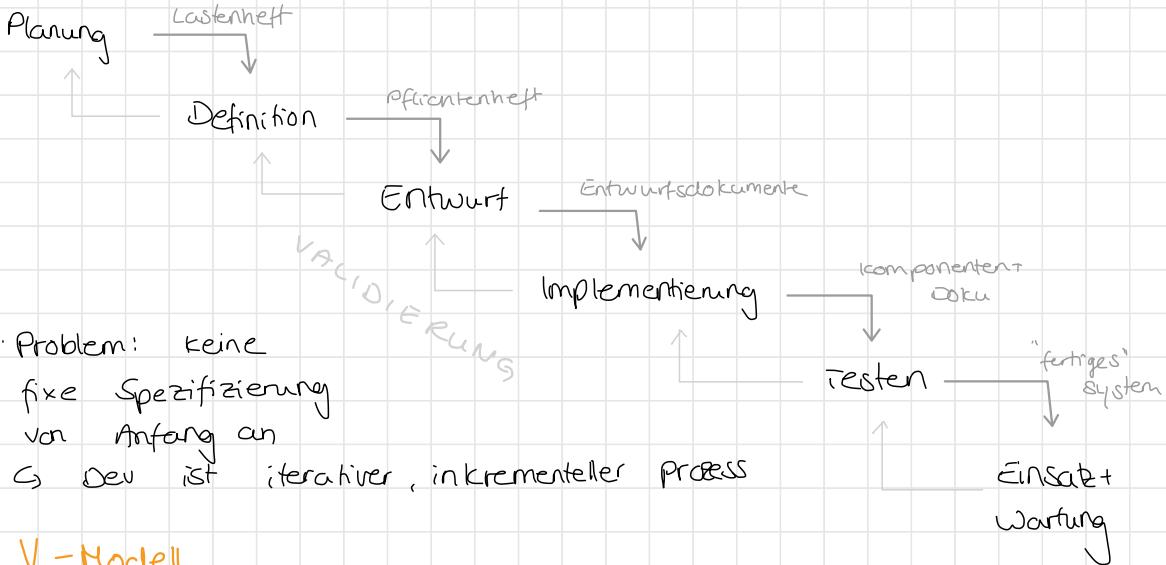
- abstrakte Repräsentation eines Software Dev Prozesses
- Richtlinie für
  - Aktivitäten (wie und wann?)
  - Rollen (wer?)
  - Produkte (Artefakte (was?))
  - optional: Techniken + Tools

- (+) • Strukturiert Entwicklung (divide & conquer)
- macht Entwicklung Skalierbar → parallel Arbeit durchführen
  - Kostenabschätzung, Risikominimierung

Lebenszyklus → Fokus auf Transitions zw. Phasen

## Wasserfallmodell

- sequentielles Modell



• Problem: keine fixe Spezifikierung von Anfang an

↳ Dev ist iterativer, inkrementeller Prozess

## V-Modell

kein richtiges Prozessmodell, eher: wie stehen Aktivitäten in Beziehung

## Spiralmodell

Evolutionsärer Prozess, jede Iteration = gesamter Zyklus von Req.  
Analyse bis Integrationstests

## Inkrementeller Prozess

iterativ = gleiche Artefakte immer wieder überarbeitet

inkrementell = mehrere Schritte / Artefakte - " -

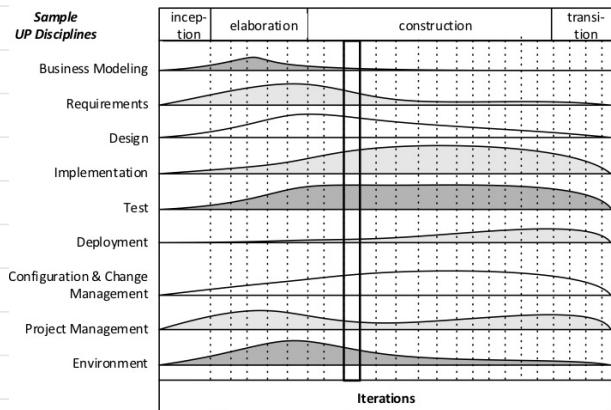
Möglichkeit 1: Wasserfallmodell mehrfach

↪ : Req. Analyse muss nicht immer so ausführlich gemacht werden

## Unified Proces

- 4 abstrakte Phasen  
die bei einem Milestone  
enden

- 9 Disziplinen (6 Entwicklung  
≈ Wasserfall, 3 supporting)



- iterativ, inkrementell
- Risikogaben
- Klientengaben
- Architektur im Vordergrund

## Phasen

### Inception Anfangsphase

- Idee, Nutzen des Systems ermitteln
- große Kosten-, Aufwandsabschätzung → Umsetzbar?
- Entwicklung starten ja oder nein?

## Elaboration Ausarbeitungsphase

- Requirements finden, spezifizieren
- Hauptrisiken ausloten
- Core, risky SW Teil programmieren + testen

## Construction Konstruktionsphase

- iterative Impl. der übrigen (einfacheren, risikoärmeren) Komponenten
- Vorbereitung Deployment

## Transition Übergabe phase

- Beta Tests
- Deployment

↪ Alle Disziplinen in allen Phasen relevant, aber untersch. ausgeprägt

## Disziplinen

### Business Modeling

- verstehen / kommunizieren der Motivation + Umgebung des Systems
- Domain concepts + business Prozesse

### Requirements

- Auswahl, Analyse, Dokumentation, Validierung, Management

### Design

- OO Analyse, Plan wie Reg. in SW umsetzen
- Klassen modellieren

### Implementation, Test, Deployment

### Configuration & Change Management

### Project Management

### Environment

## Rational Unified Process

- spezielle Umsetzung des UP
- Spezifiziert
  - Rollen (wer?)
    - Skills + Verantwortungen
  - Activities/tasks (wie?)
    - Aufgabenpaket die von einer Rolle auszuführen sind
  - Artefakte / Produkte (was?)
    - Resultat eines tasks

## 6 Zentrale best-practices

- 1) Software iterativ entwickeln
- 2) manage Requirements
- 3) komponenten-basierte Architektur verwenden
- 4) Software visuell modellieren
- 5) Software-Qualität verifizieren
- 6) SW Änderungen kontrollieren

## Evaluation Modelle

(+)

- Phasen gut als Strukturierung
- divide & conquer : große Tasks in kleineren Schritten
- Rollen unterstützen Spezialisierung, helfen Fokus und Verantwortung zu definieren
- Überblick über wichtige Aufgaben

(-)

- Rollen können Produktivität senken → nicht jede Rolle im gleich stark gebraucht
- optimum zw. Kosten + Prozessaufwand liegt bei langwierigen Projekten

# Agile Entwicklung

- flexiblerer Prozess, erlaubt Änderungen leichter

# Agiles Manifest

fokussierung zurück von rechts nach links

- Individuals & interactions      =>      processes and tools
  - Working Software                =>      Comprehensive documentation
  - Customer collaboration        =>      Contract negotiation
  - Responding to change          =>      Following a plan

Wichtig!

- Akzeptieren dass man nicht 100% voraus planen kann  
(Ändernde Anforderungen,...)
  - Konstant beobachten:
    - Setzt man um was Kunde will?
    - Ist Produktqualität ausreichend?

## Extreme Programming

- jetzt nur noch ein Artefakt! Code
  - starker Fokus auf Coding

## Values

- Kommunikation
  - Simplicity
  - Feedback
  - Courage

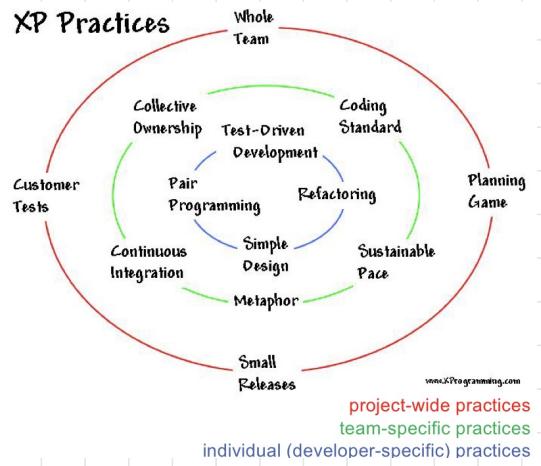
**(-)** ad-hoc Prozess: schlecht

Planbar, nicht Reproduzierbar

↳ schlecht bei großen Projekten

- fehlende Dokumente

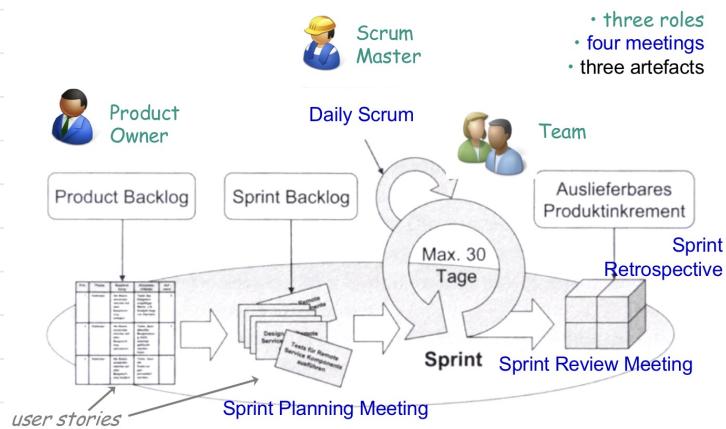
Keine Kunden involviert



## Scrum

- agile management framework
- 3 Rollen
- 4 Ceremonies
- 3 Artefacts

## Prozess



## Meetings

### Sprint Planning

- Team entscheidet welche User Stories mit höchster Prio aus Backlog in Sprint Backlog kommen
- ↳ Planung des nächsten Sprints

### Sprint Review

- Demo der umgesetzten user stories

### Sprint Retrospective

- was kann am Prozess verbessert werden

### Daily Scrum

- Abhängigkeiten feststellen
- Info über aktuellen Stand der Entwicklung teilen
- Arbeitsplan ggf. anpassen
- Probleme identifizieren

## Sprint

- Zeitraum ( $\leq 1$  Monat) in dem User Stories bis zur endgültigen Fertigstellung entwickelt werden
- konstante Zeiträume
- neuer Sprint startet nach Beenden des alten
- während Sprint keine Änderungen die Sprintziele gefährden
- Sprint scope kann angepasst werden (Verhandlung von Team, PO)

## Rollen

- "Pig roles" (PO, Team, Scrum Master) = Arbeiter
- "Chicken roles" (Stakeholder, Manager, Customer,...) = nicht direkt involviert → chicken sagen pigs nicht wie sie etwas machen müssen
- ! Individuals + Interactions  $\gg$  processes + tools

### Product Owner (PO)

- entscheidet über Features und deren Priorisierung
- repräsentiert Kunden
- hält Backlog + Prio aktuell
- nimmt Resultate ab

Probleme: muss genug techn. Verständnis und Kundenverständnis haben

### Scrum Master

- hält Team den Rücken frei
- löst Teamprobleme, sorgt dafür dass Prozess läuft
- Schnittstelle des Teams nach außen
- idealerweise Moderator, Coach, erfahrener Software Dev

### Das Team

- selbstorganisiert, implementiert das Produktincrement
- committet sich auf Sprint-Ziele
- schätzt Entwicklungsleistung der User Stories
- idealerweise kleines Team ( $\leq 7$ ), alle in allem gleich gut

## Artefakte

### Produkt Backlog

- high-level To Do Liste (priorisiert) → User Stories
- keine Subtasks (→ Sprint Backlog)
- Tasks haben story points → repr. Aufwandsabschätzung
- sollte nicht überladen sein ~ haupts. essentielle Features

### User stories

- = Anforderungen
- genaue, klare Beschreibung
- alle bekannten Abhängigkeiten

### Sprint Backlog

- User Stories auf die sich Team committed hat
  - ↳ aufgeteilt in einzelne Subtasks
- pro Subtask  $\approx$  16 Mannstunden

### Wie viele User stories?

- bis 85% Netto kappa des Teams
  - ↳ Abwesenheiten, Feiertage, ... beachten
- 25-30% davon abziehen
  - ↳ Telefonate, Krankheit, Fortbildung, ...

## Projektplanung

3 Planungslevel im Scrum:

1) Release Planning:

- basierend auf Product Backlog + Entwicklungs geschw.

2) Sprint Planning

3) Day Planning

## Skalierung große Projekte

- Brook's Law: neue Leute hinzufügen macht erstmal langsamer
- ↳ Lösung: organisches wachsen
  - am Anfang 1 Team, dann Team splitten und je neue Leute dazu
- Abhängigkeiten zw. den Teams minimieren
- Scrum of Scrums: Repräsentanten jedes Teams treffen sich täglich

## Verteilte Teams

- physisch verteilte Teams  $\rightsquigarrow$  daily scrum schwierig
- Tipps:
  - Scrum Master und Team nicht trennen
  - Teams schrittweise verteilen
  - Subteams von Zeit zu Zeit durchmischen

## Evaluation Agile Prozesse

(+)

- schnelle Anpassung an Änderungen
- disziplinierter Prozess
- Verantwortung aller Teammitglieder gleich

(-)

- unklar wann Zeit für Architekturplanung  $\rightarrow$  flexible Architektur?!
- Skalierung großer Projekte unklar
- wie Reuse planen?
- höherer Druck durch gleiche Verantwortung aller

# Continuous Integration

- Clean, well-defined Build-env ohne Annahmen
  - env = Umgebung auf der alles später laufen soll
  - functional + quality tests
- ↳ kein "aber bei mir läuft's doch"
- CI muss in den Software-Entwicklungs-Prozess eingebunden sein
  - CI erlaubt mehrere Integrationen pro Tag
  - jede Integration automatisch verifiziert durch autom. tests + build
- ↳ schnelles Feedback bei Fehlern

## CI Prozess

1) Mensch:

- Code / Artefakte ändern
- lokal testen
- Änderungen commiten

2) Automatisiert:

- Dependencies erstellen
- Compilieren
- Testen: functional + quality (performance, JUnit, code coverage,...)
- Deployment entities bündeln
- Feedback an Coders geben

3) Mensch:

- kaputte Builds fixen

## Voraussetzungen

- automatisierte SW builds mit Tools / Skripten
- automatisierte Tests (unit + integration)

## Best Practices

- alle relevanten Artefakte in zentralem Repo versionieren
  - Code
  - Tests
- automatisierungs Artefakte
  - Anleitung für Build + Deployment
- duplizierten config code vermeiden
- Build für fail early designen  $\rightarrow$  schnelles Feedback
- ein Build pro deployment - Einheit
- continuous feedback

## Tooling

Jenkins, Travis CI, ...

## CI Evaluation

- keinen negativen Einfluss auf SW Qualität
- höhere Produktivität  $\rightarrow$  effizienteres + effektiveres Mergen von Pull requests
- geringeres Risiko
  - Fehler früh erkannt  $\rightarrow$  früher gefixt
- weniger repetitive Prozesse
  - compiling
  - testing
  - Bundling + Deployment
- bessere Team Kommunikation durch build history

## Continuous Delivery

- automatisiertes Release durch CI
- zusätzliches testing durch
  - Testing Team
  - User: Canary release  $\rightarrow$  J. bekommen neue Version, Rest alte

## Microservices

- einzelne Anwendung als Komposition von kleinen Services
- jede Anwendung hat eigenen Prozess und kommuniziert über leichtgewichtige Mechanismen
- Services
  - ... unabhängig deploybar
  - ... unabhängig skalierbar (bezogen auf Ressourcen, Laufzeit)
  - ... können in versch. Programmiersprachen geschrieben werden
  - ... können von versch Teams getragen werden

## Monolith vs Microservice

Monolith packt alle Funktionalität in einen Prozess  
↳ Skalierung durch Replication des Monolithen

Microservice packt jede Funktionalität in eigenen Service  
↳ Skalierung durch Verteilen der Services und unabh. Replikation

## Microservice Architecture

### Komponentisierung durch Services

- Services impl. erkennbaren Teil der Funktionalität als Gesamtes (also nicht nur GUI)
- Komponenten kommunizieren über zB web Service Requests

### Organisation um business capabilities

- zB Aufteilung nach Teams
- ↳ Conway's Law

## Products not Projects

Produkt als Nutzen für Kunden statt großes SW Projekt als Set von Funktionalitäten

## Smart endpoints dumb pipes

- Endpoints (Services) kümmern sich um Logik, keine Logik zur Übertragungszeit (zB durch Bus ⚡)

## decentralized governance

- Service = universelles Tool zur Lösung eines Problems in versch. Kontexten
- Service contracts : Pattern

### Tolerant Reader:

- unbekannte Objekte ignorieren
- minimale Annahmen nötig um Robustheit zu erhöhen

### consumer-Driven contracts:

- Service implementiert alle consumer-Needs
- Provider verteilt an tatsächlich gebrauchte Services
- Contract: Datenschema, Policies,...

## decentralized data management

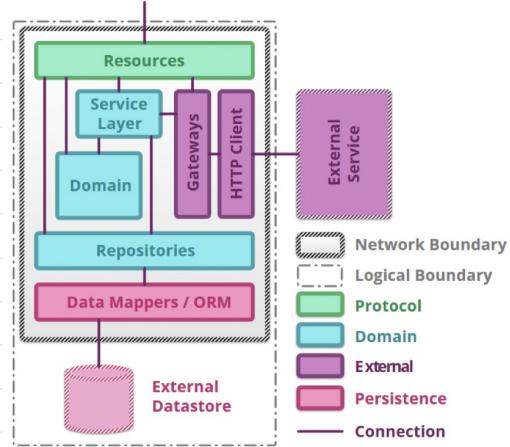
- jeder Service managt eigene DB (Kann auch komplett versch. System sein)

## Infrastructur automation

- durch CI, Continuous Delivery

## design for failure

- Service als Komponente => Anwendung muss mit (zufälligen) Fehlern / Ausfallen der Services klar kommen (detektieren + Service wieder herstellen)
- online monitoring der Anwendung



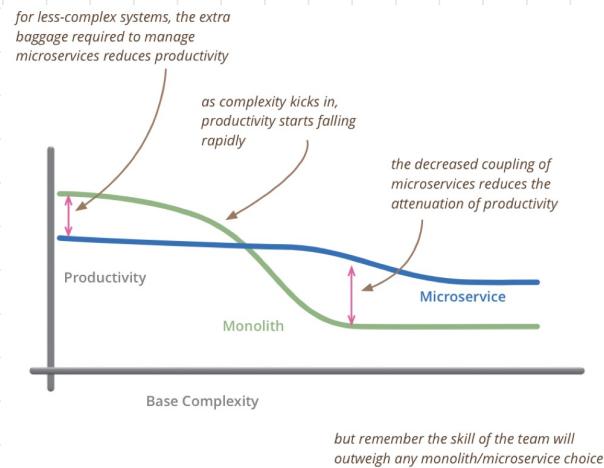
## evolutionary design

- Anwendung so in Services aufteilen, dass Änderungen leicht umzusetzen sind
  - ↳ Aufteilung nach
    - Unabhängigkeit und Möglichkeit der Ersatzung einzelner Services
    - User Funktionalität

## Evaluation

- (+) · modulare Struktur → strikte Modulgrenzen  
↳ gut für große Teams
  - unabh. Deployment
  - technologische Diversität (Sprachen, Frameworks, Datenmanagement)
- 
- (-) · starke Verteilung  
↳ distr. Systems schwerer zu programmieren (Kommunikation,...)
  - eventual consistency: strikte Konsistenz in distr. Systemen schwer / unmöglich umzusetzen
  - operational complexity

~ Microservices gut wenn base complexity (sehr) hoch



## How to?

Option 1: mit Monolith starten, später separieren  
↳ Service Grenzen anfangs unklar

Option 2: mit Microservices starten  
↳ zwingt Modularisierung

# Domain Driven Design

## Application Domain

- Anwendungsbereich mit dem sich System beschäftigt
- SW System kann mehrere haben  
↳ gesamte Business Logic

## Überblick

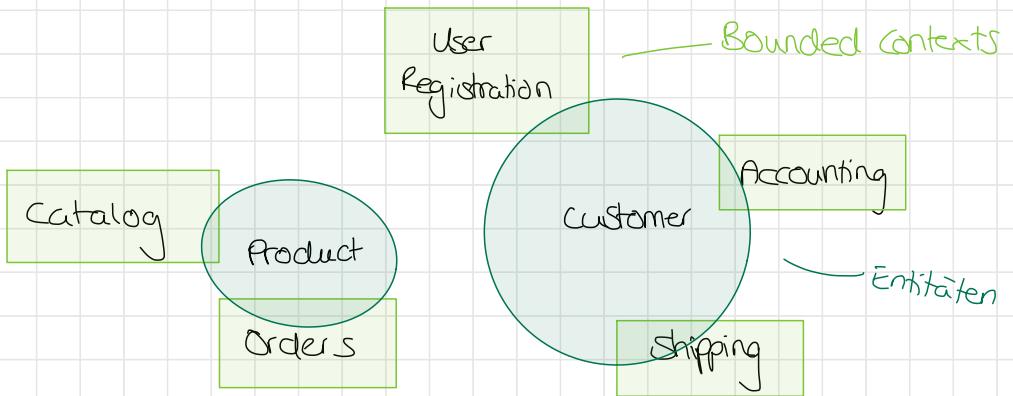
- 1) Bounded context der Application Domains finden
- 2) Für jede Appl. Dom.:
  - a) ubiquitous language definieren
  - b) daraus domain model erstellen
  - c) Modell durch building blocks in SW umsetzen
  - d) Domain app. zusammensetzen → layered architecture
- 3) Domain app. deployen, linken wie in Context Map spezifiziert

## Ubiquitous Language (UL)

- Übersetzung Domain Concepts → Programm konzepte?  
↳ gleiche Sprache für Design + Impl. verwenden
- pro Domain (Kontext) eine UL
  - 1) App. Dom. in bounded contexts aufteilen
  - 2) für jeden bounded context UL definieren
  - 3) Konzepte die zw. bounded contexts geteilt in context map anzeigen

## Bounded Context

- strikte Konsistenz der UL und des Domain models innerhalb
- Fokus auf Issues aus dieser Domain
- Context map:
  - welche Entitäten sind geteilt?
  - welche Domain events werden versendet?



## Domain Driven Design

- Geschäftslogik aufteilen in sinnvolle Strukturen
  - ↳ bounded contexts
- jedes Team bekommt einen Teil
  - ↳ etabliert UI und domain model
- Teams untereinander: Organisation entsprechend context map
- Teams entwickeln unabhängig testbare / deploybare Teile der Anwendung

## Building Blocks

- Patterns um Domain Concepts zu implementieren
- Zuordnung abh. von Domain!

### Entities

- beschreibt Domänen Konzept
- eindeutige ID
- Kontinuität während Lebensdauer

@ Entity

Customer, Product, Order, ...

### Value Objects

- Attribute / Eigenschaften eines Konzepts
- hat keine eigene Identität
- unveränderlich → nur änderbar durch ersetzen

@ Embeddable

Quantity, Monetary Amount, ...

### Services

- Domänen Operationen / Aktionen
- Zustandslos

@ stateless + Singleton Pattern

Customer Management, ...

## Aggregates

- cluster verwandter entities + value objects
- definiert Grenzen um Root entities
- kümmert sich um Invarianten + gleichzeitige Zugriffe
- Zugriff auf Unterelemente nur über root entity

Purchase Order,  
Shopping Cart,  
...

## Factories

- Erzeugen der domain concepts
- verdeckt Objekterzeugungskomplexität
- separiert Erzeugung logik  $\leadsto$  extra domain concept

@Stateless @Service

Customer Factory, ...

## Entities

- erzeugt minimale volatile Entities / Aggregate
- garantiert Invarianten bei Erzeugung

## Value Objects

- Gesamtes Produkt konstruiert (da immutable)
- erfordert gesamte Spezifikation des Value Objects

## Repositories

- Unterstützt querying von domain concepts
- simuliert Collection von Entities / Aggregaten
- normalerweise als persistenter Mechanismus (z.B. DB)

@ Repository

Catalog, ...

## Modules

- bündelt verwandte domain concepts  $\rightarrow$  separiert versch. voneinander
- low coupling zu anderen Modulen
- high cohesion innerhalb
- $\rightarrow$  high-level domain concept

Orders = Shopping cart  
 \* Purchase  
 \* Payment

## Strategic Design

- Core Domain  
Order, Shipping, Accounting
  - wichtiger Aspekt um domain-related Probleme zu lösen
  - alle anderen Domains unterstützen core domains
- Shared Kernel  
Customer, Product
  - domain concepts die zw. versch. domains geteilt
  - maintained / weiterentwickelt von versch. Teams
- Supporting Domain  
Catalog
  - unterstützende Funktionen für core domain
- Generic Subdomain  
User Registration
  - kapselt generische Konzepte die nicht zur core dom. gehören

## Interaktionen zw. Domains

- Shared Kernel
- Consumer / Supplier Dev Team
  - downstream team stellt Funktionalität für upstream team bereit
- Conformist
  - downstream passt sich dom. Model des upstream Teams an
- Anticorruption Layer
  - downstream impl. interface layer für upstream Team
- Open Host Service
  - definiere Protokoll um downstream (3rd party) service zu integrieren
- Published Language
  - formalisiere Protokoll für dom.-spec. Sprache

## Software Reliability

- Wahrscheinlichkeit einer failure-freien Operation eines codes für eine bestimmte Zeit in einer bestimmten Umgebung
- failure-free = Code produziert erwarteten Output
- 1-reliability = Wahrscheinlichkeit eines failures on demand (POF0)
- ROCOF = rate of occurrence of failures = # failures pro qg. Zeit  
MTBF = mean time between failures

### Mission time

- Zeit in der System funktional und eingabebereit sein soll

$$\text{Availability} = \frac{\text{Zeit in der System wirklich erreichbar}}{\text{MTBF} + \text{MTTR}}$$

### SW vs HW Failure

- HW Failure durch Materialdefekte/-ermüdung
- SW Failure durch Bugs → menschliche Fehler beim coden

### Software Testing

#### Functional

alle User functions testen

#### Coverage

= white box testing  
alle Pfade testen

#### Random

Testfälle zufällig ausgewählt aus Eingaberaum

#### Partition

Eingaberaum partitioniert ("strata"), Testfälle jeweils zufällig

## Statistical

formales Zufallsexperiment für randomness  
↳ statistisches Modell für Reliability erzeugen

$P = 1 - R$  = Wahrsch. für failure

↳ für  $n$  runs:  $R^n = (1-p)^n$

↳ Anzahl erwarteter Failures bei  $n$  runs =  $np$

## Security

### Ziele

Main goals: CIA

- Confidentiality : personal data (privacy), company data
- Integrity
- Availability      } of data + system

Supporting goals:

- User AuthN
  - Traceability, auditing
  - Monitoring
  - Anonymity
- Security Probleme verursacht durch intelligente Angreifer

### Angriffsstrategien

- Outside-attacks: durch IT-Tools
  - network sniffers, proxies
  - viruses, trojan horses, ...
- insider attacks
- theft, burglary
- social engineering attacks (mensch. Kontakte: Anrufe, ...)

## Design

- security frühzeitig beachten
  - ↳ in jeder Entwicklungsphase!
- Entwicklungsumgebung auch sichern

## Security Requirements

- Security goals
- misuse cases finden
- Systemkontext beachten (Bspw. im Internet?)
- modelliere Angriffe mit Checkliste:

Identify Assets > Architecture overview > Decompose Application > Identify Threats > Document Threats > Rate Threats > Find Countermeasures

- Formulierung: konkret
  - was
  - vor wem
  - warum
  - wie lange
- Requirements klassifizieren für Risiko und Prioritätsmanagement  
↳ bei Konflikten: trade-off finden

## Design- Prinzipien

- 1) Das schwächste Glied sichern (kann auch social engineering sein!)
- 2) Verteidigung in der Tiefe → mehrere Schichten der Sicherheit
- 3) Fail securely → Exceptions sicher behandeln, keine Systemdetails preisgeben
- 4) Secure by default → alle Sicherheitsoptionen per default
- 5) Principle of least privilege ("need to know")
  - min. access privileges für min. Zeitspanne
- 6) Kerckhoff's principle: no security through obscurity
  - Schutz nicht durch geheimhalten der Protokolle/Verfahren
- 7) minimiere Angriffsfläche → so wenig Code wie möglich
- 8) Privileged core → Code mit Privilegien isolieren

## 9) Input Validierung, Ausgabeverschlüsselung

### 10) Don't mix data and code

## Weiter Idiome

- 1) wenn möglich federated identity management verwenden
- 2) Passwörter: serverseitig validieren, hash + salt verwenden
- 3) encrypted connections für session handling
- 4) Sprachen haben schon Fehler / Sicherheitslücken  
↳ informieren!

## Testing

- Tester müssen sich wie intelligente Angreifer verhalten
- 1) Fokus auf identifizierte Risiken
- 2) Code coverage → Fokus auf nicht abgedeckten Code
- 3) Code + System reviews
- 4) Security evaluation in jeder Phase / Iteration
- 5) versch. Quellen für Testcases verwenden
- 6) Standard Umgebung für testing verwenden
- 7) Sicherheitsexperten als black box + white box - Tester

## Security Issues

### Race Conditions

- Parallelle Threads greifen simultan schreibend auf Ressource zu  
↳ wer gewinnt?

Lösung: atomic operations / synchronize / mutex...

### Buffer overflows

- Buffer voll → Daten werden außerhalb geschrieben  
↳ überschreibt ggf. andere Daten z.B. Rücksprungadressen
- Lösung: checks machen!

## SQL injection

- DB Abfragen als String eingeben  
↳ Lösung: Eingaben validieren

x ') ; DROP TABLE Students;

## Kryptographie

- Protokolle für Komm. zw. mehreren Parteien

### Pseudo-Zufall

- Kryptogr. beruht oft auf Zufall  
↳ falsch gebauter Zufall liefert keine echten Zufallszahlen

### Lösungen:

- 1) KRNG verwenden der Kryptogr. sicher ist  
↳ nächste Zahl nicht  $\in P$  erratbar
- 2) Seed darf nicht zu erraten sein  $\rightarrow$  ausreichend Entropie  
nicht: sec seit Mitternacht

## Cloud Computing

### 5 Charakteristika:

- 1) Elastic Scalability  
↳ elastische Anpassung der benutzten Ressourcen an Nachfrage
  - 2) On-demand Self-Service  
↳ Nutzer kann selbst Services auswählen + konfigurieren
  - 3) Ubiquitous Network Access  
↳ zentralisierte Ressourcen müssen über Netzwerk erreichbar sein
  - 4) Ressource Pooling  
↳ geteilte Ressourcen: weniger Ressourcen als Nutzer
  - 5) Measured Service
- Illusion unendlicher IT-Ressourcen die
- immer erreichbar über Netzwerk
  - on-demand services
  - pay-per-use Prinzip

## Ressources

- computing Power, Storage, Network
- Money, Energy, Real-Estate, Humans
- Software, Data

## Ressource Usage

- Gleichzeitige Nutzung von versch. Kunden (tenants)
- variierende Anforderungen (OS, SW, HW, ...)
- unvorhersehbare Lasten

## ↳ Multi-Tenancy:

- Isolation versch. Aufgaben
- Separation versch. Kunden
- Kunden bekommen Admin-Rechte

Klassisch: Separation of Servers → Physical Resource Sets

Cloud: Separation of Services → Virtual Resource Sets

## Virtualization

- logische Illusion von physischen Ressourcen

## Hypervisor

- Kontrollprogramm
- Partitionierung: mehrere OS auf einem Server
- Isolation: VMs komplett getrennt → keine Seitenefekte

## Paravirtualization

- keine emulierte HW Schicht
- Gast OS nutzt hyper calls → hypervisor interface calls

## Cloud Computing

### 5 charakteristika,

3 Delivery Models (SaaS, PaaS, IaaS),

3 Deployment Models (Private, Public, Hybrid)

## Infrastructure as a Service (IaaS)

HW nach Wunsch → PC, Speicherplatz, OS, ...

zB AWS, Azure

## Platform as a Service PaaS

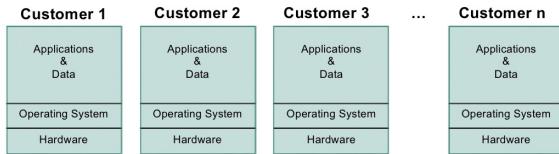
- hauptsächlich für cleins → bietet schon Infrastruktur + manche Software
- zB Google App Engine

## Software as a Service SaaS

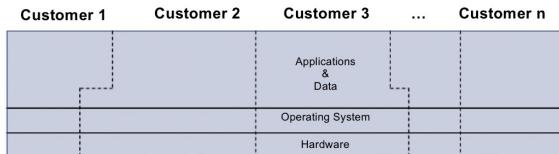
- nutzungsbereite Anwendungen die an zentraler Location gerichtet werden
- erreichbar über kleine Netzwerkschnittstellen (Browser,...)
- one-to-many delivery (single instance, multi-tenant architecture)
- zentralisiertes Feature-updating

Single tenant: Kunden komplett getrennt (HW bis App+Daten)

Multi Tenant: Variable Trennungen per Schicht  
↳ App Konfigurationen als meta-data



single tenant architecture



multi tenant architecture

## Cloud Architecture

- komponentenweise Skalierbar
  - Komponenten lose gekoppelt
  - Parallelisierung
  - Resilience
  - Cost Factor
- } beachten bei Design

## Architekturprinzipien

- Dezentralisierung: Vermeidung von bottlenecks, single points of failure
- Asynchrony: System macht Fortschritt unter allen Umständen
- Autonomy: Komponenten können basierend auf lokalen Infos Entscheidungen treffen
- Local responsibility: jede Komponente selbst für Konsistenz verantwortlich
- Controlled concurrency: Operationen brauchen keine/kaum concurrency Kontrolle
- Failure tolerant: minimale Unterbrechung bei Failure
- Controlled parallelism
- building blocks: kein Single-Service → aufteilen
- Symmetry: Knoten gleich bzgl. Funktionalität
- Simplicity: System so einfach wie möglich

## Model-Driven Development

Essential Complexity = geg. durch Anwendung / Domäne  
↳ abh. vom Projekt

Accidental Complexity = geg. durch schlechte Entwicklungsmethoden  
↳ vermeidbar

## Model

Allgemeine Modelltheorie:

Representation Feature (Abbildungsmerkmal):

- modelliert etwas reales → es gibt ein Original (kann auch ein Modell sein)

Reduction Feature (Verkürzungsmerkmal)

- Modell bildet nicht alle Attribute des Originals ab  
↳ nur relevant erscheinende

## Pragmatic Feature

- Substitutionsfunktion für konkrete Subjekte
- Nutzen des Modells für gewisse Zeitintervalle

- mit Restriktionen zu bestimmten mentalen / tatsächlichen Operationen

## Metamodels

- Modelle die Modellierung beschreiben
- beschreibt Struktur eines Modells
  - Modellierungssprache
  - Verbindungen zw. Elementen
  - constraints / Modellierungeregeln

## Anforderungen an Spezifikation

- Damit Struktur + Regeln eines Modells komplett spezifiziert, muss Metamodell folgendes spezifizieren:
  - Abstract Syntax
  - concrete Syntax
  - Static Semantics
  - Dynamic Semantics

## Abstract Syntax

- beschreibt die Konstrukte aus denen Modelle bestehen
- deren Eigenschaften + Relationen
- Beschreibung unabh. von konkreter Darstellung dieser Konstrukte, etc.

## Concrete Syntax

- beschreibt die Darstellung der Elemente der abstrakten Syntax
- mind. eine konkrete Syntax muss spezifiziert sein, können auch mehrere
- zB CS1: UML, CS2: XML, CS3: Bild

## Static Semantics

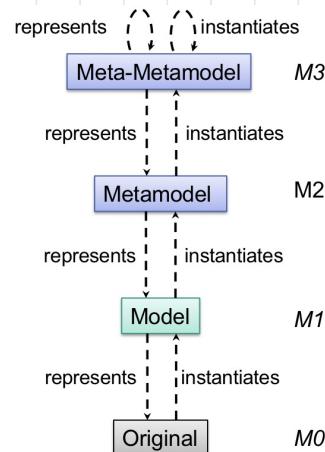
- beschreibt Modellierungeregeln / -einschränkungen die in abstrakter Syntax nur schwer / gar nicht beschreibbar

## Dynamic Semantics

- beschreibt die Bedeutung der Konstrukte
- kann einfach Text sein

## Repräsentation / Instanzierung

- Model repräsentiert Original
- Model vor Original erzeugt : original instanziert Model
- Model als Instanz des Meta-Models
- > typischerweise 4 modelling - layer
- abstraktestes Model oft self-descriptive



## Object Constraint language

- für static Semantics
- logikbasierte Ausdrücke
- keine Control flows, Programmaufrufe, Seiteneffekt
- unterstützt design-by-contract Ansatz
  - invarianten, Post-/Preconditions
  - initiale Werte
  - Ableitungen
  - Body definitions
  - guards

## Model - Driven Development

### Model - Driven engineering

- Kombination aus
  - Domain specific modeling languages : Deklarative Beschreibung der Anwendungsstruktur, -verhalten, Anforderungen
  - Transformation engines + generators

## Model-Driven Software development

- Anwendung von MDE für SW Entwicklung

## Model - Driven Architecture

- Prozess für MDSA

## Model - based

vs.

## Model - driven

- manche Modelle = sekundäre Artefakte
- Models zur Kommunikation + Dokumentation
- Manuelle Analyse + Evaluation

- Modelle = primäre Artefakte
- Modelle als wichtiger Teil des Systems
- Analyse durch Model Transformation
- Models explizit spezifiziert, entwickelt, versioniert, ...

## Ziele MDSO

- bessere Plattformunabhängigkeit und -interoperabilität
- schnellere Entwicklung durch Codegenerierung
- bessere Qualität durch Arbeit mit und Analyse von Models
- Wiederverwendbarkeit in anderen Software Product Lines
- bessere sep. of concerns durch Nutzung versch. Models
- bessere Wartbarkeit durch Vermeidung von Redundanzen
- besserer Einfluss von Domänenwissen

## Model - Driven Architecture

### Computation - Independent Model

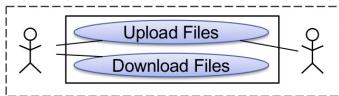
- beschreibt Anforderungen für das System und Umgebung
- Details zu Struktur, Verhalten versteckt oder unbestimmt

### Platform - Independent Model

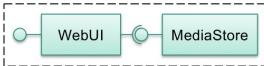
- Fokus auf Ausführung des Systems
- verstecken der Notwendigkeit einer expliziten Plattform
- zeigt den Teil der nicht variabel auf versch. Plattformen

### Platform - Specific Model

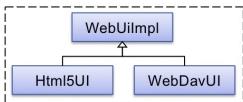
- Kombination PIM + Fokus auf Details einer bestimmten Plattform



**CIM:** Use Case Diagram (UML)



**PIM:** Component Model (PCM)



**PSM:** Class Diagram (UML)



Program code (Java)

## Rollen

- keine Programmierer mehr nötig
- Domain expert implementiert
  - ↳ generiert SW aus Modells mit DSLs
- technology expert: bereitet Umgebung vor
  - modeling platforms
  - transformations
  - meta-models

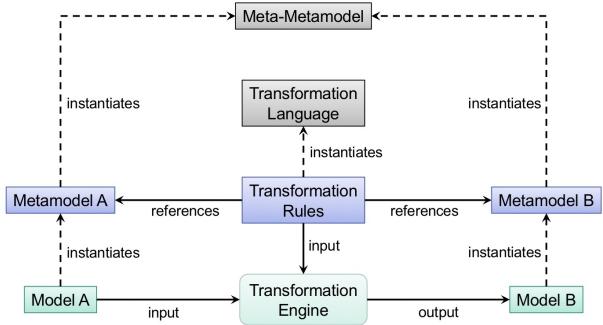
## Model Transformations

### transformation

- automatische Generierung eines target models aus source model entsprechend transform. def.

### transformation definition

- Set von transf. Regeln  $\rightarrow$  beschreiben wie source lang.  $\rightarrow$  target lang.



### transformation rule

- beschreibt wie source konstrukt  $\rightarrow$  target konstrukt transformiert wird

## Transformation languages

### Declarative

- Fokus auf WAS
- Relation zw. source, target model beschreibt was in was transformiert werden muss
- Funktionale / logische Programmierung

### Operational / Imperative

- Fokus auf WIE
- Spezifikation der nötigen Schritte um aus source model target zu machen

## Evaluation MOSSD Gegenstand der Forschung!

(+)

- Kostenreduktion
- geringere time-to-market
- Domänenwissen in Modellen
- höhere SW Qualität
- gut bei accidental complexity

(-)

- sehr hoher Aufwand
- Tools noch nicht sehr ausgereift