

Bachelor of Science (BSc) in Informatik

Modul Advanced Software Engineering 1 (ASE1)

Software Architektur

Entwurf von Softwarearchitekturen

Institut für Angewandte Informationstechnologie (InIT)

Walter Eich (eicw) / Matthias Bachmann (bacn)

<https://www.zhaw.ch/de/engineering/institute-zentren/init/>

Agenda

- 3.1 Einbettung in den Lehrplan - Lernziele
- 3.2 Überblick über das Vorgehen
- 3.3 Entwurfsprinzipien
- 3.4 Architekturzentrierte Entwicklungsansätze
- 3.5 Techniken für einen guten Entwurf
- 3.6 Architekturmuster
- 3.7 Entwurfsmuster

3.1. Einbettung in den Lehrplan - Lernziele

- Das **Vorgehen und die Heuristiken*** für den Architekturentwurf beherrschen
- Die **Herleitung der Softwarearchitektur** aus Anforderungen und Randbedingungen durch hierarchische (De-) Komposition beherrschen
- **Techniken** für eine guten **Entwurf** beschreiben können
- **Architekturmuster** erklären können
- Ursachen von **schlechter Kopplung und Kohäsion** verstehen

*Heuristik = mit begrenztem Wissen und wenig Zeit zu einer guten Lösung kommen

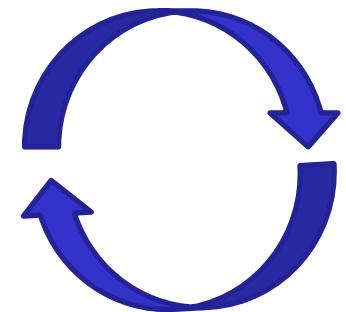
3.2. Überblick über das Vorgehen

- Das **Vorgehen** beim Architekturentwurf ist geprägt durch:

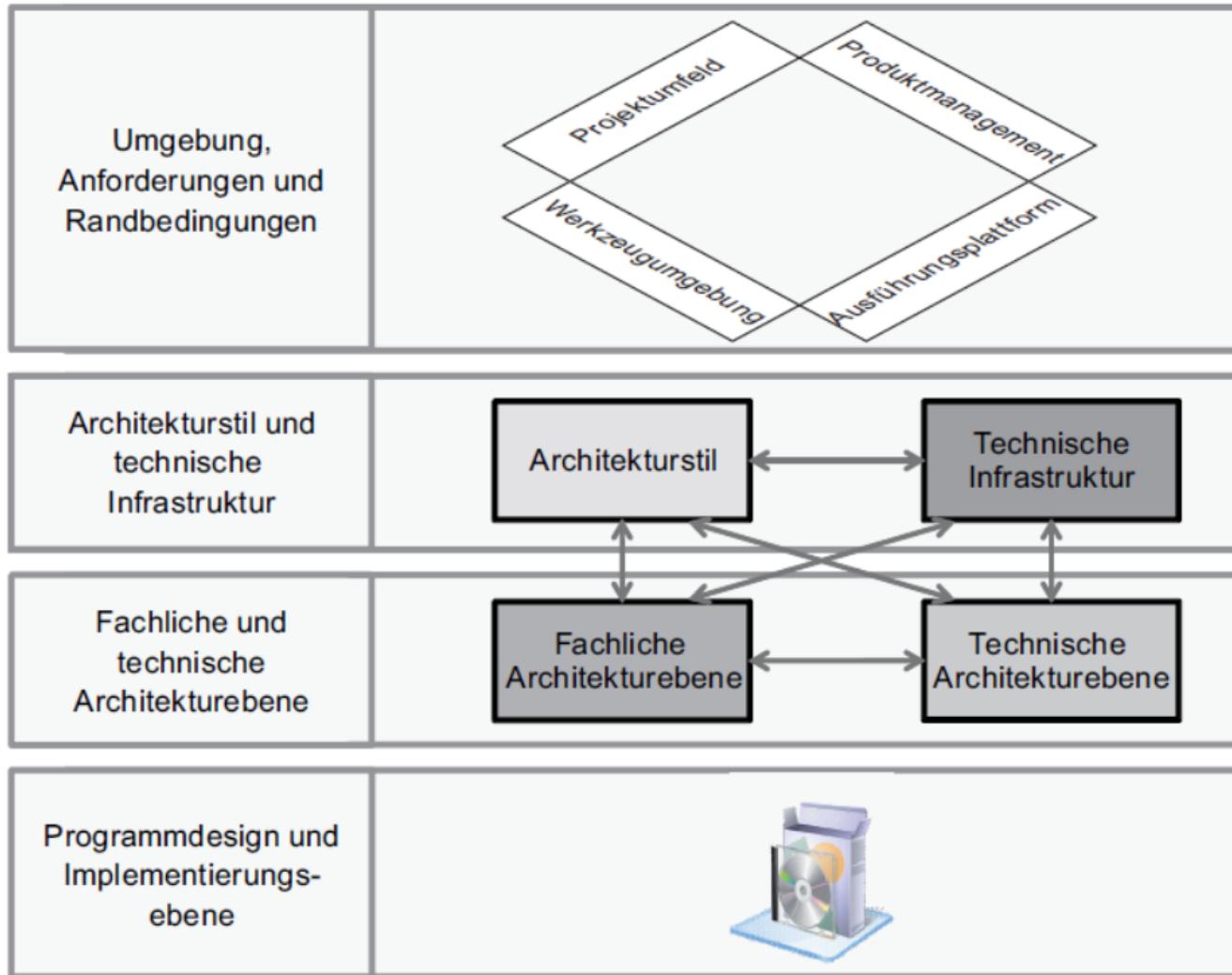
- Wechsel der vier **Abstraktionsstufen** mittels Top-down oder Bottom-up Vorgehen (nächste Folie) .



- Iteratives Wechselspiel zwischen den Tätigkeiten (übernächste Folie)



Wiederholung – 4 Abstraktionsstufen

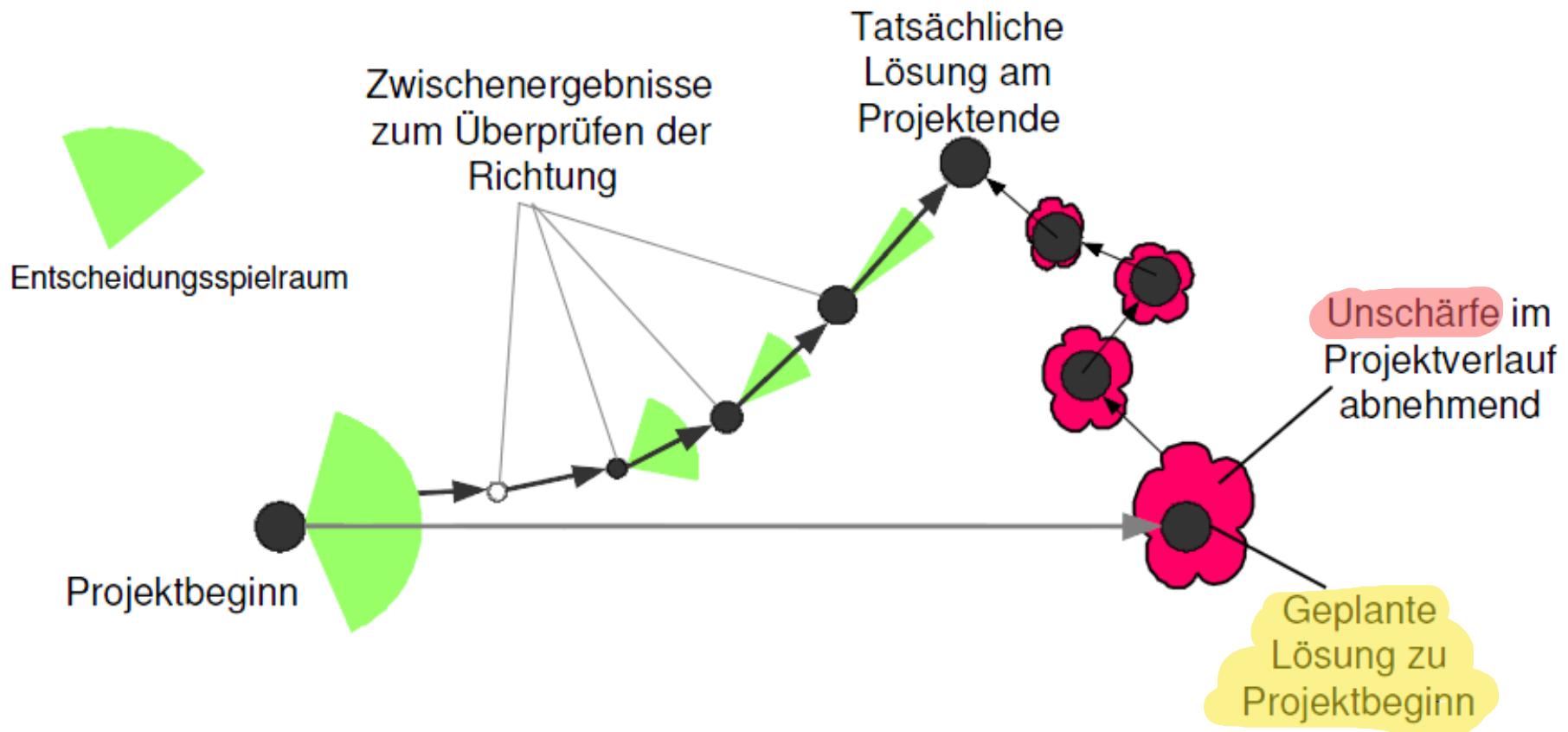


Wiederholung – 4 Tätigkeiten des Entwurfsprozesses

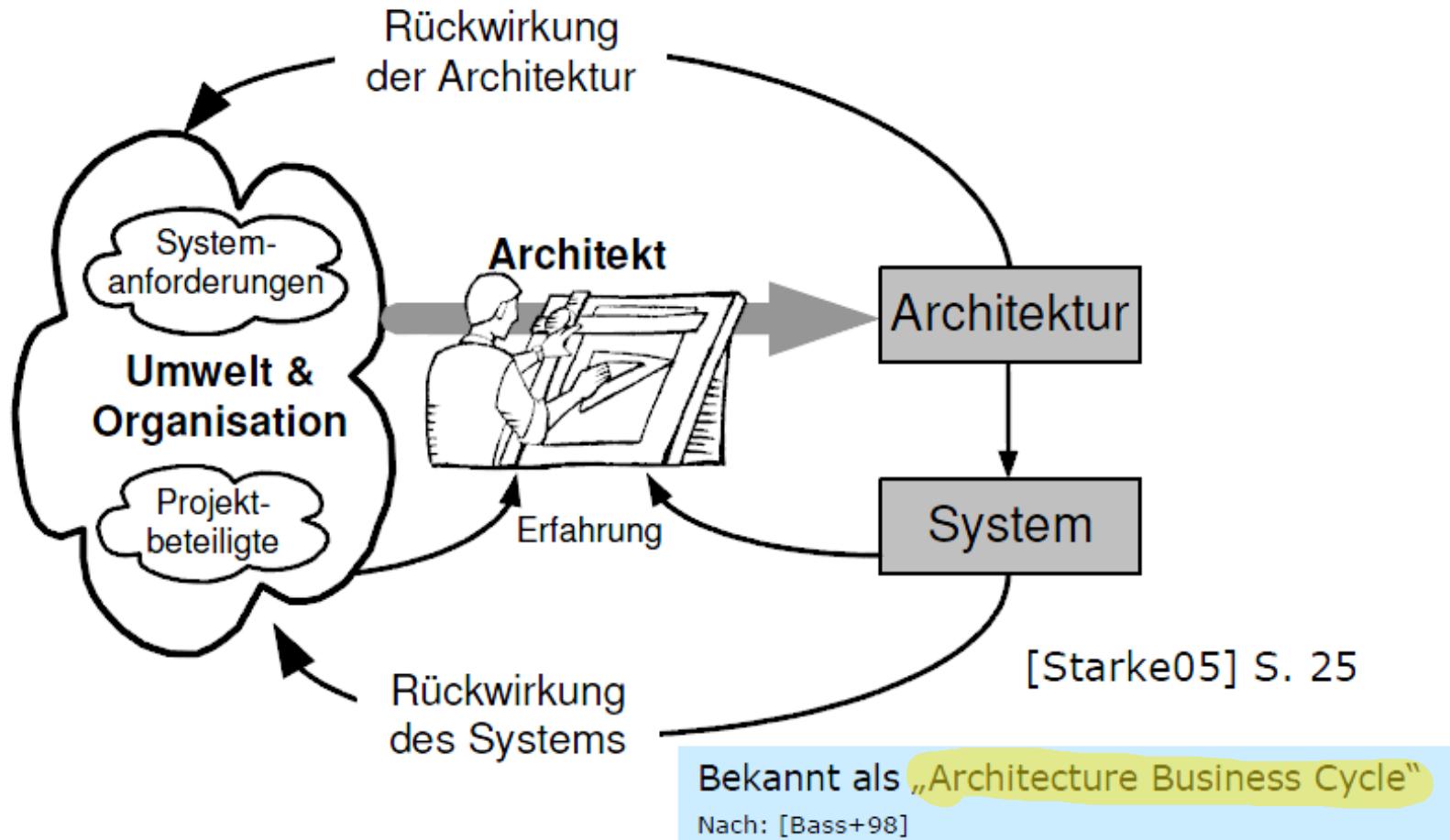


- Praxis: Tätigkeiten werden oft quasi gleichzeitig in beliebiger Abfolge durchgeführt

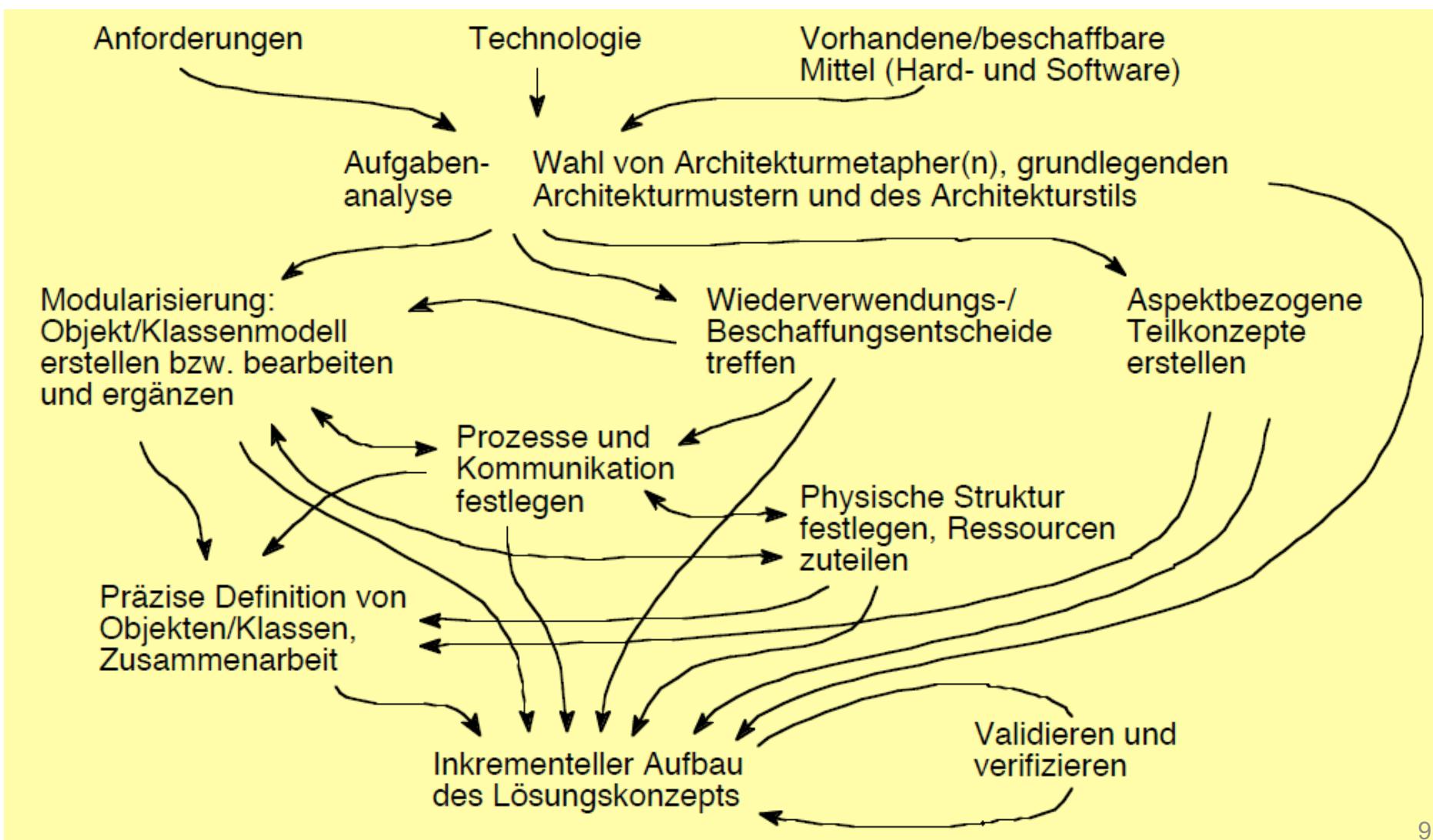
Wie entstehen Architekturen ?



Wie entstehen Architekturen ?



Wie entstehen Architekturen?



Wie entstehen Architekturen ?

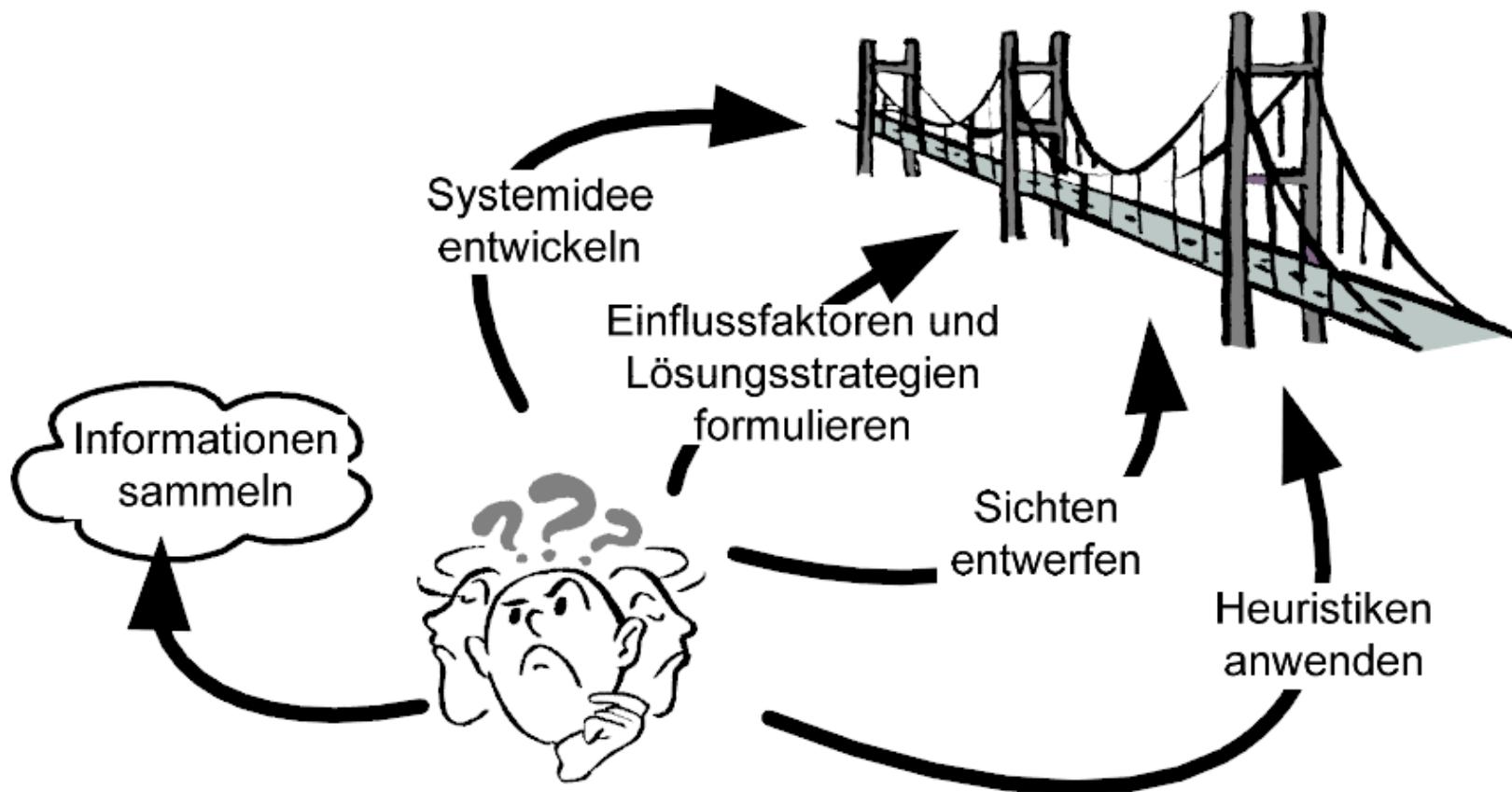
■ Anforderungen und Randbedingungen analysieren

- ▶ **Grundlage:** Anforderungsdokument
- ▶ Welche Anforderungen haben einen **Einfluss auf die Architektur** bzw. auf die **Architekturentscheidungen** (funktional, nicht funktional)
- ▶ **Randbedingungen:** Wo sind wir frei – wo nicht?
(organisatorisch, politisch, technisch und betriebsbedingt)



gibt es für die Bewertung von **Lösungsvarianten Kriterien** (Anforderungen) aus dem Anforderungsdokument?

Würde die **Änderung von Anforderungen** oder Randbedingungen zu einer besseren oder einfacheren Architektur führen?



Wie entstehen Architekturen?

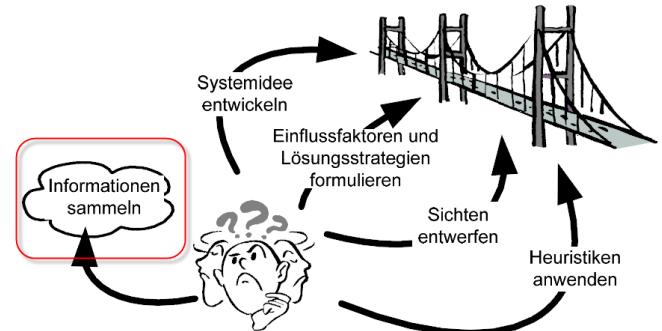
- Wer hat ähnliche **Aufgaben** gelöst?

- ▶ Erfolgreich oder auch nicht!

- **Lösungsideen** sammeln

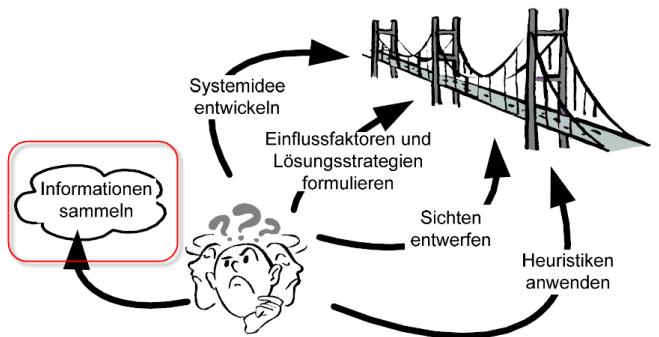
- Gute Ausgangspunkte:

- ▶ Einzusetzendes Framework
 - ▶ Muster (Patterns)
 - ▶ Ihre eigene Organisation
 - ▶ Andere Unternehmen Ihrer Branche
 - ▶ Ihre eigene Erfahrung



Informationen sammeln

- Nur wenige Dinge (in der IT-Branche) sind wirklich innovativ.
- Meist haben die *Produzenten* nur schlecht recherchiert.
 - ▶ Auch als *Not-Invented-Here* Syndrom bekannt (NIH-Syndrom)
 - ▶ Organisationen misstrauen fremden Lösungen



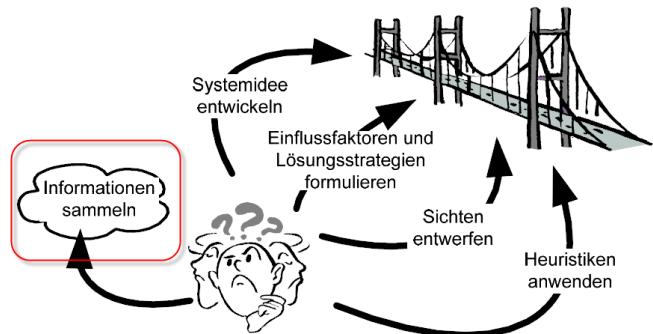
Design durch Routine oder Innovation?

■ Design-durch-Routine

- ▶ soweit wie möglich **Vorlagen** anderer **übernehmen**
- ▶ nur notwendige Teile neu entwerfen
- ▶ Im Zweifel weglassen oder Kompromisse eingehen

■ Design-durch-Innovation

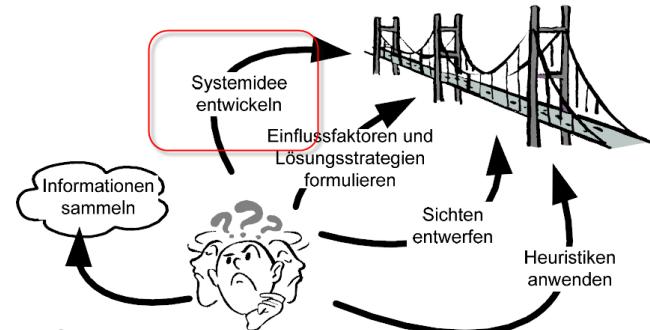
- ▶ Wir können alles besser!
- ▶ **Schlechtes Argumente:** Vorlagen (Frameworks, Anwendungen) sind zu teuer
- ▶ -> Hohe Risiken!



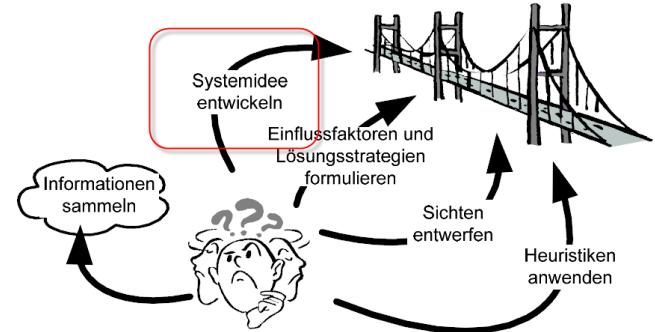
Systemidee entwickeln

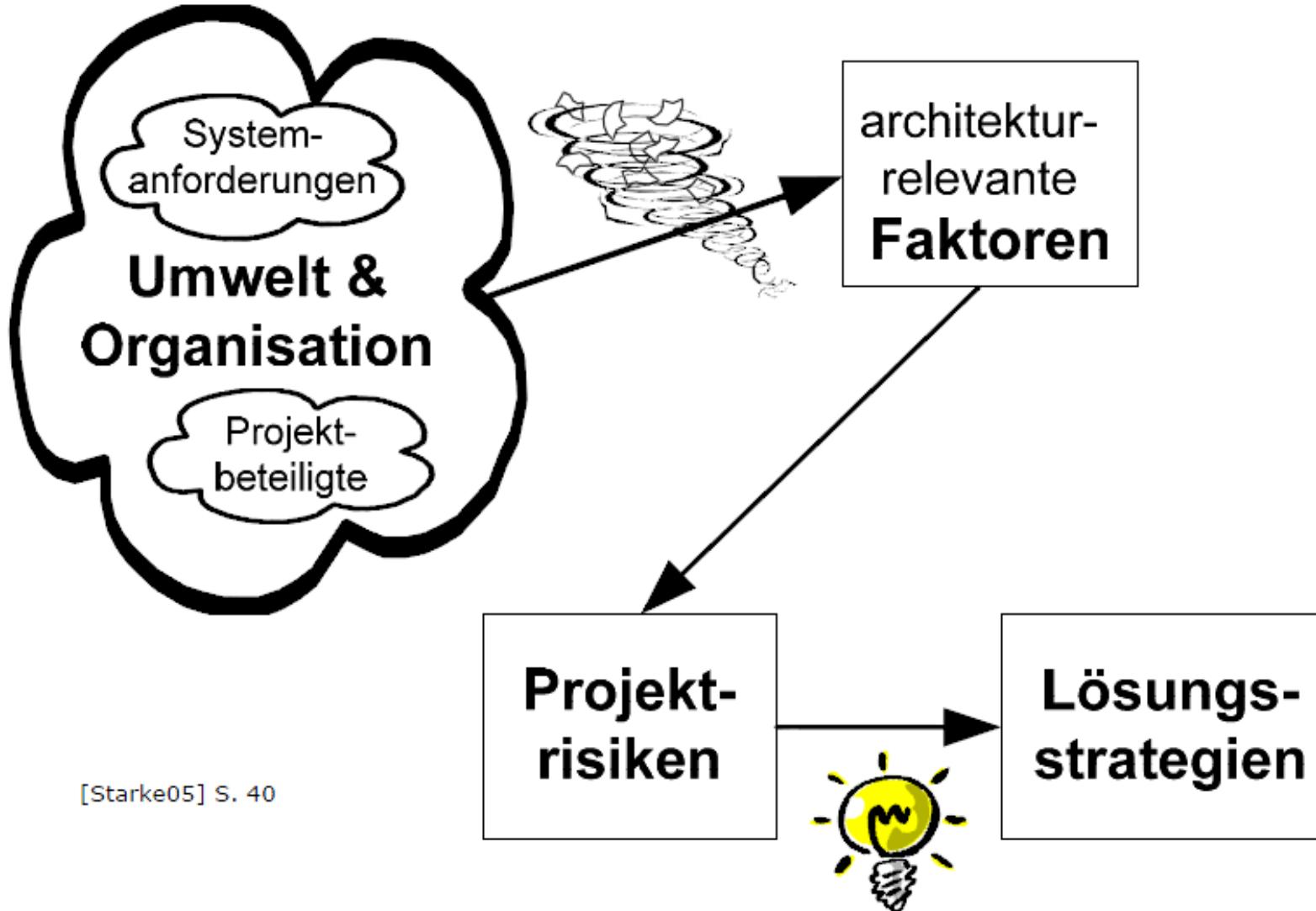
- Stellen Sie einige (kluge aber einfache) **Fragen (aus den Anforderungen):**

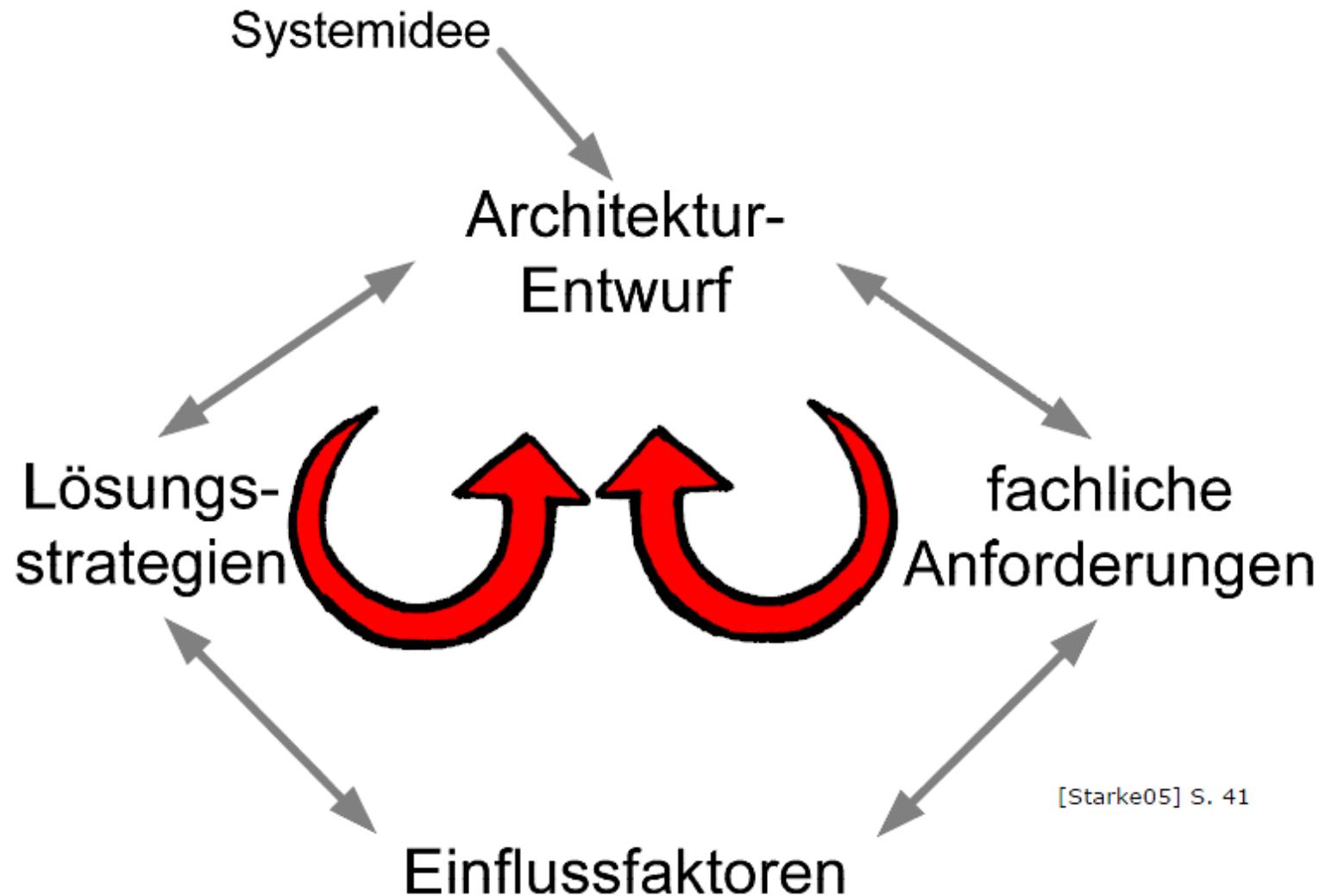
- ▶ Was ist die **Hauptaufgabe** des Systems?
- ▶ Was sind die **wichtigsten Aspekte der Fachdomäne?**
- ▶ Wie wird das System **genutzt?**
- ▶ Von wem wird das System genutzt?
- ▶ Welche Art **Benutzeroberfläche** hat das System?
- ▶ Welche **Schnittstellen** gibt es zu anderen Systemen?
- ▶ Wie werden **Daten verwaltet?** Welche Art von Datenzugriffen ist notwendig?
- ▶ Wie wird der **Ablauf** innerhalb des Systems gesteuert?



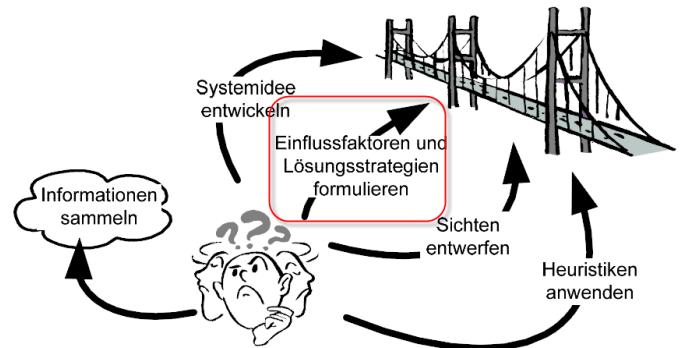
- Idee noch vage?
 - ▶ Potenzielle Risiken
- Systemidee mit Management abstimmen
 - ▶ Denn: Sie ist Grundlage weiterer Entscheidungen!
- Systemidee an alle Stakeholder kommunizieren
 - ▶ Insbesondere an Entwicklungsteam!







- Welche Arten von **Einflussfaktoren** gibt es?
 - ▶ Organisatorische Faktoren
 - ▶ Auch **politische** Faktoren
 - ▶ **Technische** Faktoren
 - ▶ System- oder Produktfaktoren
- Technik allein genügt nicht
- **Einflussfaktoren** sind flexibel
- Welche Faktoren sind relevant?
 - ▶ Alle, die Ziele der Kunden notwendig sind!



- Beispiel einer **Schach-Engine** auf der Basis des **arc42** Frameworks von Stefan Zörner (oose)
- **Iteratives Vorgehen** über 3 Iterationen
 - ▶ Iteration 1: Der Durchstich
 - ▶ Iteration 2: Das Bauerndiplom
 - ▶ Iteration 3: Der Taktikfuchs

“Softwarearchitektur ist die Menge der Entwurfsentscheidungen, die, wenn falsch getroffen, Dein Projekt zum Scheitern bringen kann.”*

(Eoin Woods)



- <https://www.dokchess.de/>

Zentrale Einflussfaktoren auf Entscheidungen

Randbedingungen

- Implementierung mit JavaScript, node.js
- Anbindung an frei verfügbares Frontend

Qualitätsziele (priorisiert)

- Einfachheit
- Erweiterbarkeit
- Effizienz (Spiel- und Antwortverhalten)

Softwarearchitektur := wichtige Entscheidungen

Risiken

- Anbindung an das Frontend?
- Aufwand der Implementierung?
- Spielstärke mit vertretbarer Antwortzeit erreichbar?

1. Iteration („Durchstich“), Steckbrief



Ziel:

Engine interagiert mit menschlichem Spieler über ein graphisches Frontend. Es entwickelt sich eine “Partie” über mehrere Züge.

Zentrale Entscheidungen:

- Darstellung der Spielsituation („Stellung“)
- Auswahl eines graphischen Frontends

Implementierungsaufgaben

- Darstellung des „Brettes“, Felder, Züge, etc.
- Anbindung an das graphische Frontend
- Trivialer Zuggenerator

2. Iteration („Bauerndiplom“), Steckbrief



Ziel:

Die Engine spielt Partien korrekt.

Zentrale Entscheidungen:

- Grundlegende Zerlegung in Subsysteme (Grundriss)
- Festlegung von Abhängigkeiten zwischen diesen

Implementierungsaufgaben

- Spielregeln

3. Iteration („Taktikfuchs“), Steckbrief



Ziel:

Die Engine spielt sinnvolle Partien.

Zentrale Entscheidungen:

- Auswahl der Algorithmen für Stellungsbewertung und Zugauswahl (Architekturentscheidungen?)

Implementierungsaufgaben

- Zugauswahl, Bewertung

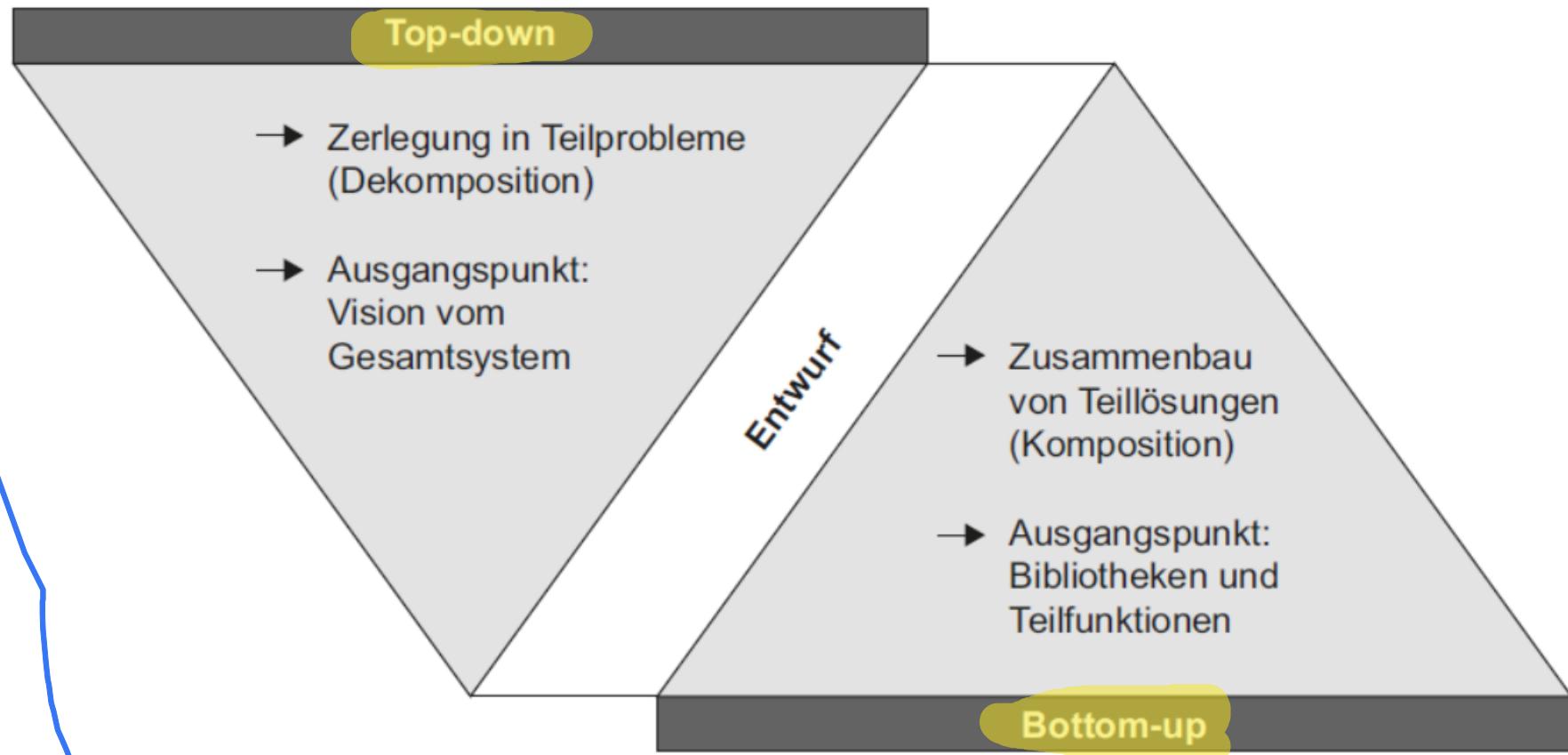
3.3. Entwurfsprinzipien und Heuristiken*

- Top-down und Bottom-up
- Hierarchische (De-)Komposition
- Schmale Schnittstellen und Information Hiding
- Regelmässiges Refactoring und Redesign



* Heuristik = mit begrenztem Wissen und wenig Zeit zu einer guten Lösung kommen

3.3.1. Top-down und Bottom-up



Vor- und Nachteile

■ Top-down

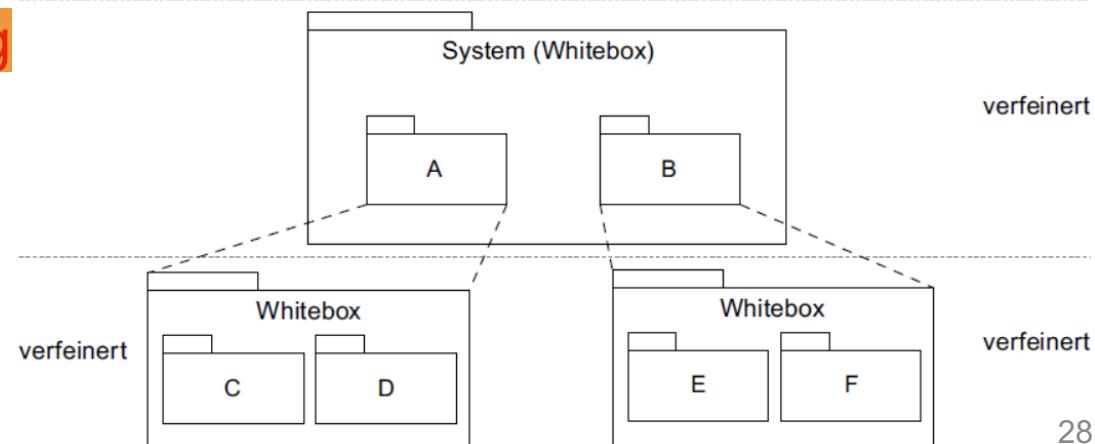
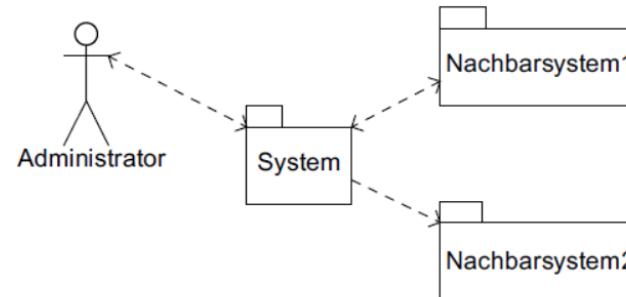
Vorteile	Nachteile
Gutes Problemverständnis	Kritische Integration am Ende
Maschinen- und sprachunabhängig	Übersehen von existierenden (Teil-)Lösungen
Kein Sich-Verlieren in Details	Gravierende Änderungen bei spät erkannten Problemen
Saubere Schnittstellen/Konsistenz	Spätes Feedback, ob der Entwurf das Richtige macht
Entwurf noch im Produkt erkennbar	

■ Bottom-up

Vorteile	Nachteile
Hoher Wiederverwendungsgrad	Potenziell werden nicht alle Teile benötigt
Hohe Funktionssicherheit durch inkrementellen Test	Orientierung an technischen Gegebenheiten statt an Benutzeranforderungen
Schrittweise Integration	Gefahr der frühzeitigen Optimierung
Beginn mit vermuteten Teilproblemen	Gefahr des »Wildwuchses«

3.3.2. Hierarchische (De-)Komposition

- **Divida et impera: Teile und (be-)herrsche**
 - ▶ Zerlegung der Aufgabe in immer **kleinere Teilaufgaben**, bis die Teilaufgabe eine beherrschbare Grösse erreicht hat.
 - ▶ **Ähnlichkeiten zum Top-Down Entwurf:** Zerlegung einer Komponente in Teilkomponenten (Bausteinsicht)
- **Prinzip: Kapselung**
 - ▶ **Kopplung**
 - ▶ **Kohäsion**
- **Prinzip: Wiederverwendung**
- **Prinzip: So einfach wie möglich**
- **Prinzip: Trennung von Verantwortlichkeiten (Separation of Concerns)**



3.3.3. Schmale Schnittstellen / Geheimnisprinzip

- **Geheimnisprinzip (information hiding)** – Kriterium zur Gliederung eines Gebildes in Komponenten, so dass
 - ▶ jede Komponente eine **Leistung** (oder eine Gruppe logisch eng zusammenhängender Leistungen) **vollständig erbringt**,
 - ▶ **ausserhalb** der Komponente nur bekannt ist, was die Komponente leistet,
 - ▶ nach aussen **verborgen** wird, **wie** sie ihre Leistungen erbringt.
 - ▶ Fundamentales **Prinzip zur Beherrschung** komplexer Systeme
- Auch im täglichen Leben fortwährend benutzt
- Liefert gute Modularisierungen

- Modulare Bauweise ermöglicht Produktvielfalt bei geringen Kosten

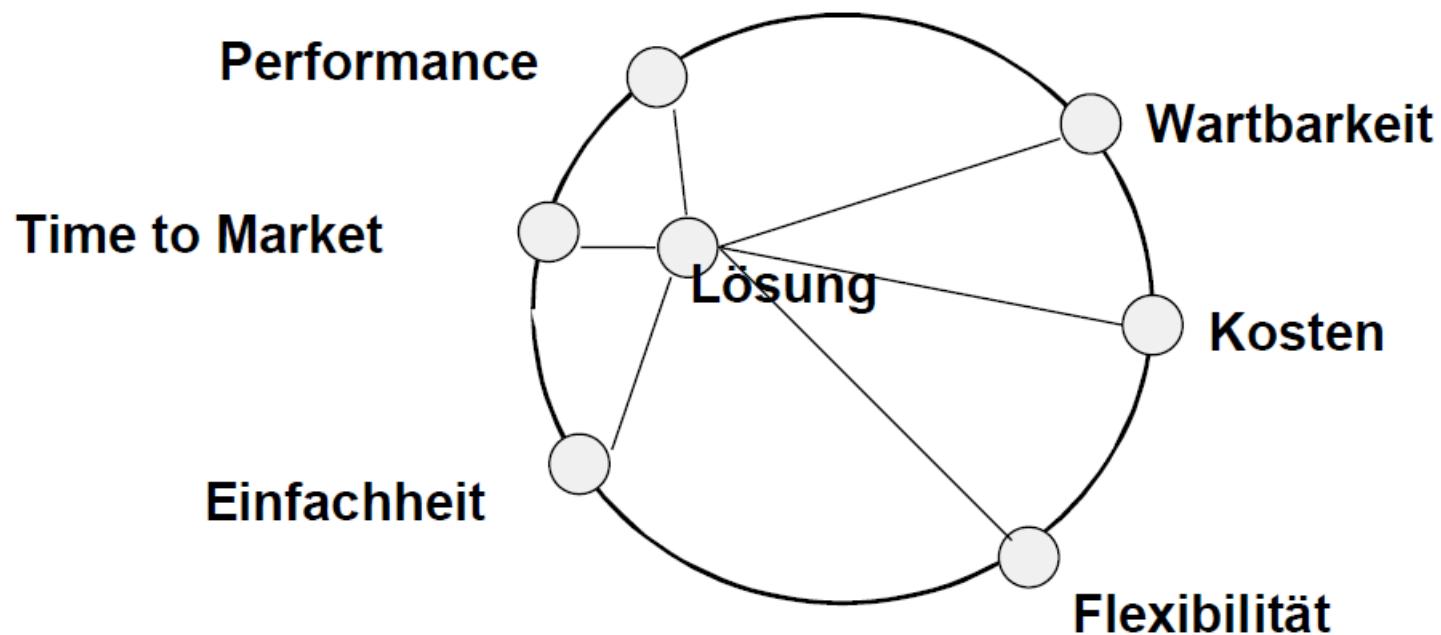


3.3.4. Regelmässiges Refactoring und Redesign

- Fakt: Software Architektur **degeneriert**
- Gründe
 - ▶ Zeitdruck
 - ▶ Unvollständige Anforderungen
 - ▶ Iteratives Vorgehen
 - ▶ Neue Mitarbeiter
- Regelmässiges Refactoring und Redesign ist notwendig
 - ▶ Kostet Zeit und Geld!!!
 - ▶ Muss jedoch regelmässig durchgeführt werden

3.4. Architekturzentrierte Entwicklungsansätze

- Domain Driven Design
- Model Driven Architecture
- Referenzarchitekturen

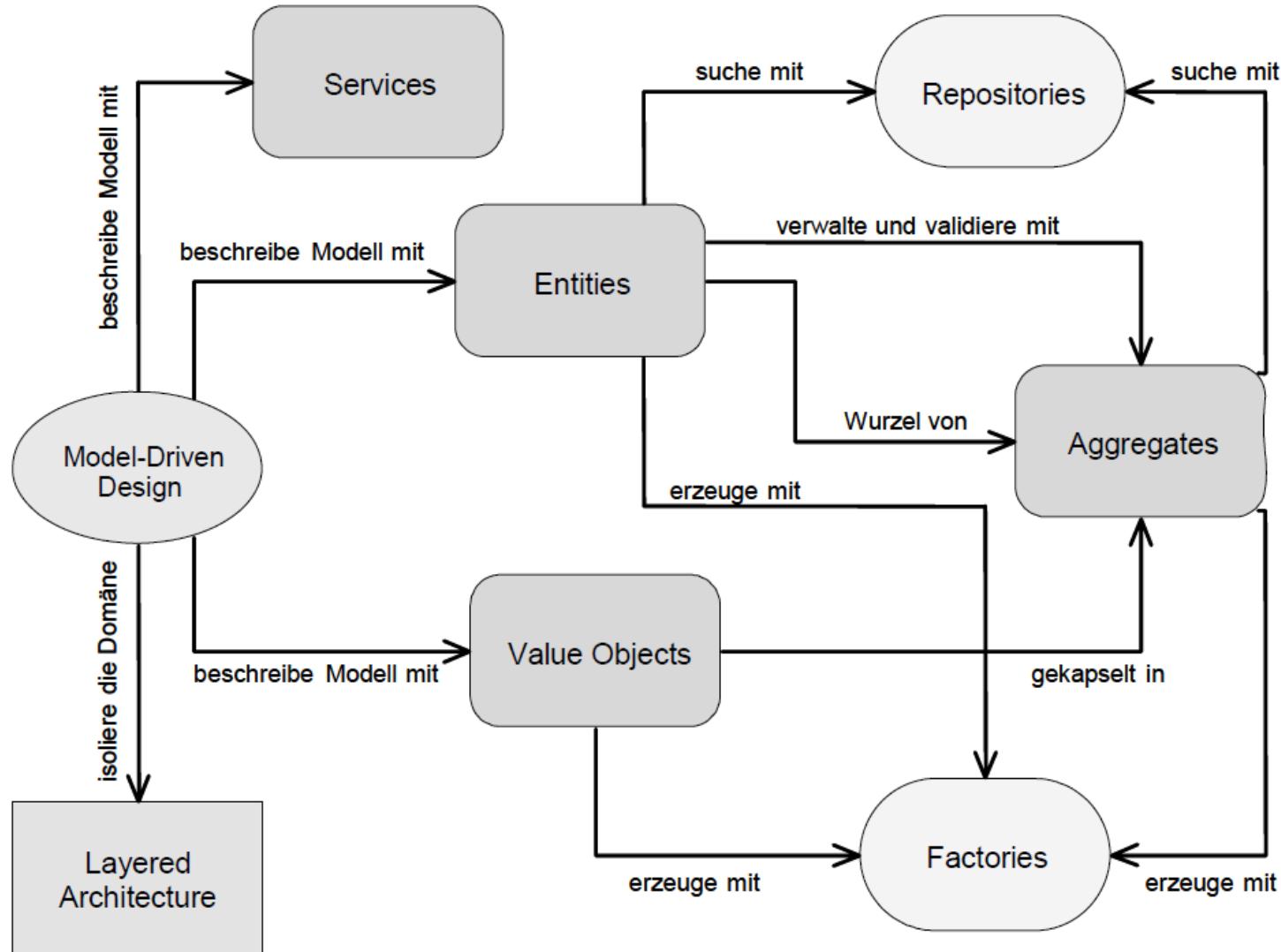


3.4.1. Domain Driven Design (Eric Evans)

- Domain Driven Design ist ein **Kommunikationsmittel!**
- Die Grundidee basiert auf **klassischem OO-Design** und **versucht, die Sprache der jeweiligen Fachdomäne objektorientiert** abzubilden, um den **Bruch** zwischen fachlicher und technischer Beschreibung/Umsetzung möglichst gering zu halten.
- Eine Minimierung dieses Bruchs bedeutet:
 - ▶ dass die **Kommunikation** zwischen Domain-Experten und Entwicklern wesentlich **vereinfacht** wird;
 - ▶ dass die **Anforderungen** an die Software besser spezifiziert werden können;
 - ▶ dass die Software für nachfolgende Entwicklergenerationen **verständlicher** und deshalb auch **wartbarer** ist.

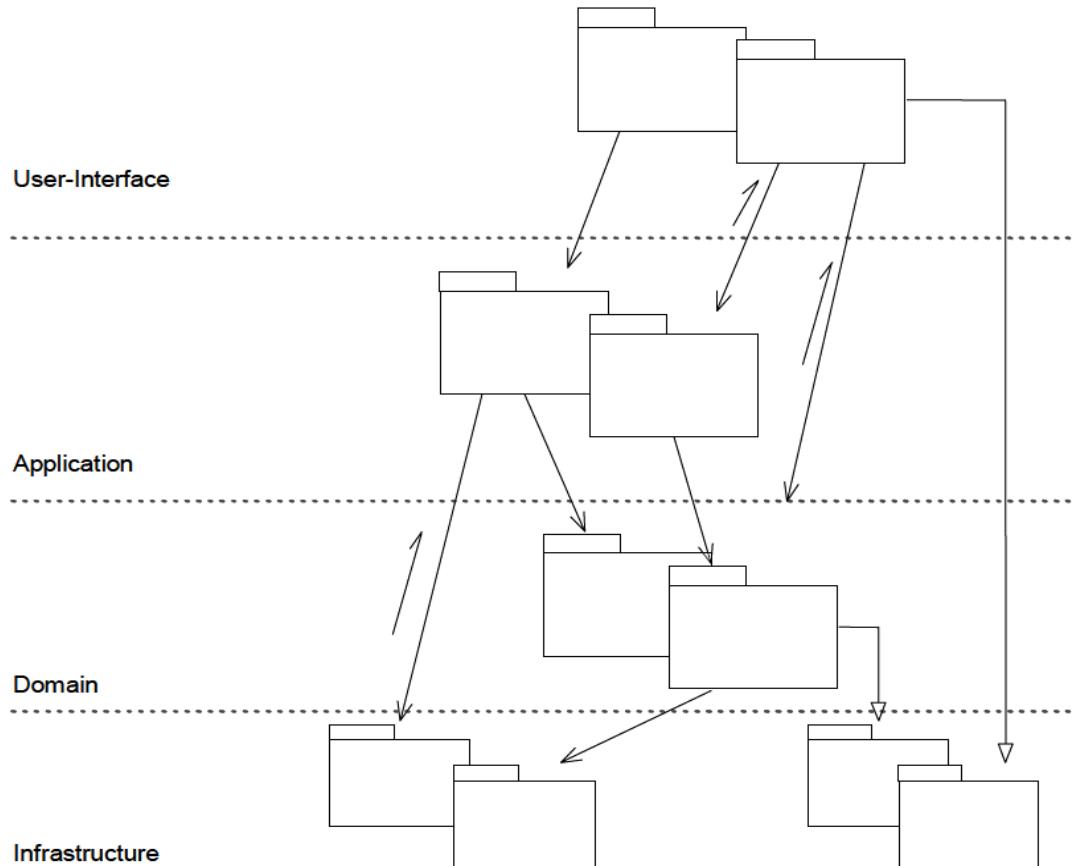
- Die eigentlichen **Geschäftsprozesse** und die Geschäftslogik stellen in Softwaresystemen oft nur einen kleinen Ausschnitt des Gesamten dar.
- Grosse Teile des Codes **behandeln die technische Infrastruktur** und stellen Hilfsfunktionalität zur Verfügung, enthalten jedoch keinerlei Geschäftslogik.
- Kommen neue Entwickler hinzu oder muss das System erweitert werden, besteht das Problem, **die eigentliche Geschäftslogik** enthaltenden Codeteile aus dem Ganzen herauszusuchen.
- Aus diesen Gründen ist die **primäre Forderung des Domain Driven Design die Isolierung der Geschäftslogik** und der diese kapselnden Objekte von den Modellteilen, die unterstützende technische Funktion haben.

Navigationskarte für Domain Driven Design



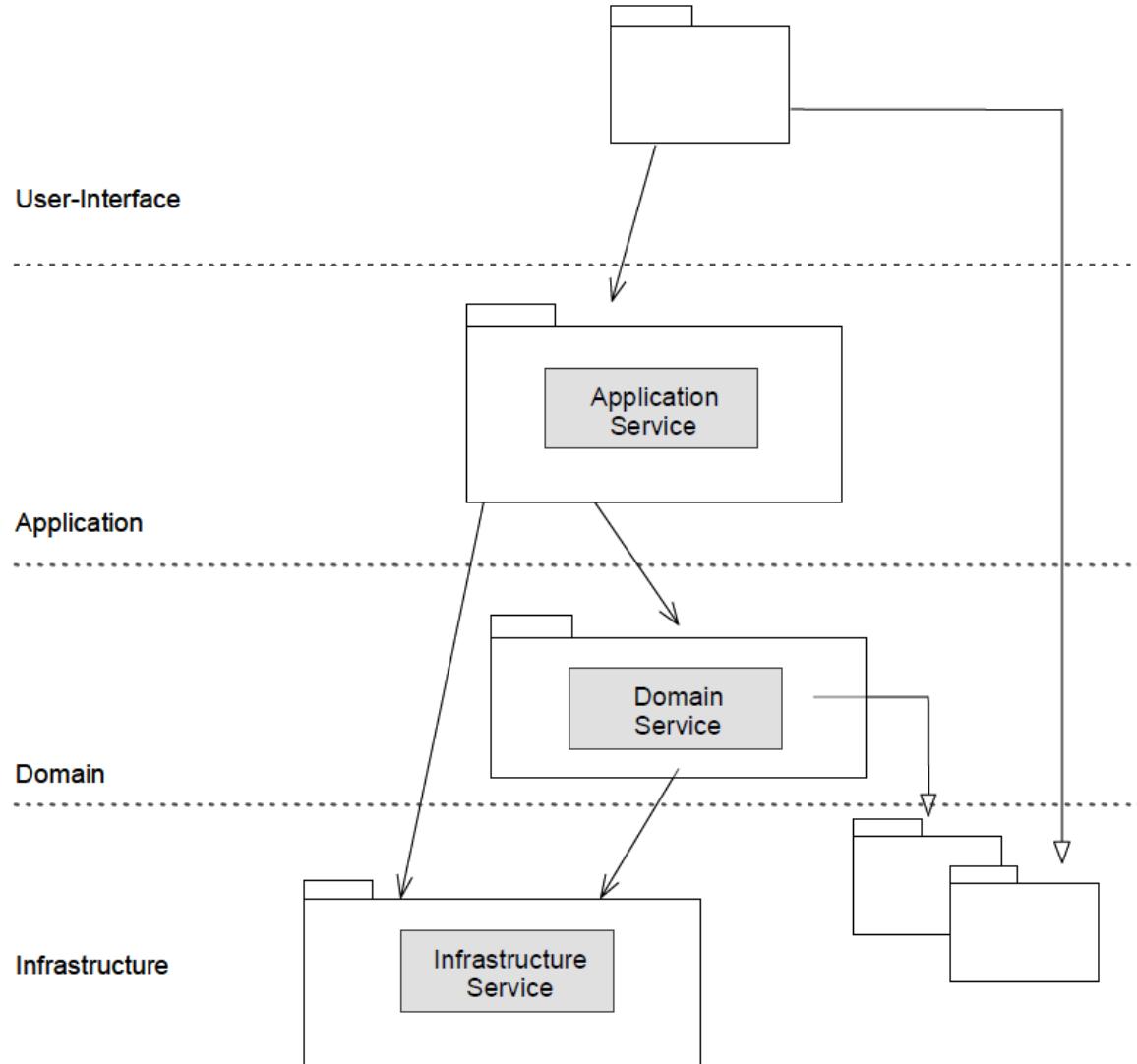
- ▶ **Entities:** Objekte mit eigener Identität werden durch **Entities** repräsentiert.
- ▶ **Value Objects:** Objekte, die über keine eigene Identität verfügen und den Status anderer Objekte beschreiben, werden durch **Value Objects** repräsentiert.
- ▶ **Services:** Objekte, die dynamische Domänenaspekte beschreiben, werden durch **Services** repräsentiert. Services führen die Funktionalität auf Anforderung des Clients aus.
- ▶ **Aggregates:** Entitäten können andere Entitäten referenzieren und aggregieren. Sie können beliebig komplexe Datentypen aggregieren.
- ▶ **Factory:** Komplexe Objekte und Objekt-Hierarchien (Aggregate) werden durch **Factories** neu erstellt oder **wiederhergestellt**
- ▶ **Repository:** Ein **Repository** verwaltet ein Objekt von der Mitte bis Ende dessen Lifecycles. Es ist verantwortlich für das **Auffinden und Laden** von persistenten Objekten

- Die **Gesamtarchitektur** gliedert sich in unterschiedliche **Layer** mit unterschiedlichem **Verantwortungsbereich**. Je nach Applikation oder Gesamtsystem können weitere Layer hinzukommen.



User Interface (Presentation Layer)	Informationsanzeige für den Benutzer, Interpretation der Benutzerkommandos
Application Layer (Larman Contoller) (Realisierung UC)	Schlanker Layer ohne Geschäftslogik. Beschreibt bzw. koordiniert die Geschäftsprozesse und delegiert die eigentliche Arbeit an Kollaborationen von Domänen-Objekten im darunter liegenden Layer.
Domain Layer (Model Layer)	Repräsentiert die Konzepte der jeweiligen Problemdomäne. Der aktuelle Zustand der Objekte der Problemdomäne wird hier verwaltet und genutzt. Die Speicherung wird an die Infrastruktur delegiert. Evans: „The Domain Layer is Where the Model Lives.“
Infrastructure Layer (Technical Services)	Stellt allgemeine, technische Services zur Unterstützung der darüber liegenden Layer zur Verfügung: z.B. – Messaging, Persistence

Services (4)



3.4.2 Model Driven Architecture

- Begriffe
 - ▶ MDA – Model Driven Architecture (OMG)
 - ▶ MDSD – Model Driven Software Development
- Generierung von Software Komponenten aus Modellen
- Im Zentrum steht dabei ein Modell
 - ▶ Textuell oder grafisch erstellt
- Transformation in Code
- Vorteile:
 - ▶ Grössere Entwicklungseffizienz
 - ▶ Fachexperten werden besser integriert.
 - ▶ Software ist leichter änderbar.
 - ▶ Verbesserte Umsetzung der Softwarearchitektur
 - ▶ Die Fachlogik lässt sich relativ einfach auf andere Plattformen portieren.

MDA - Model – Code Ausprägungen

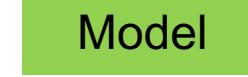
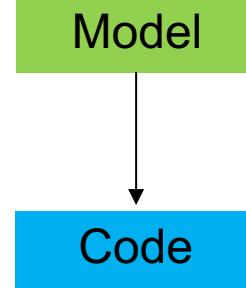
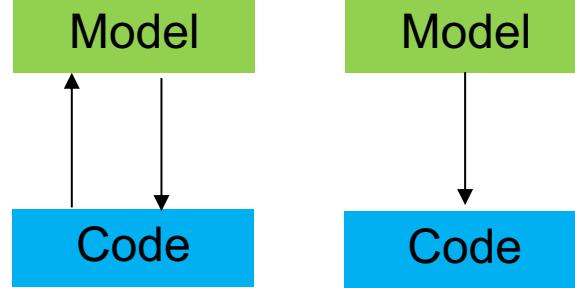
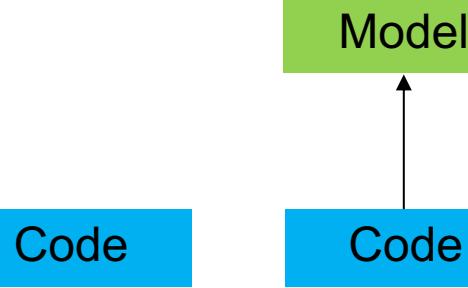
Code
only

Code
Visualization

Roundtrip
Engineering

Model
centric

Model
only



What's the
model?

The code
is the model

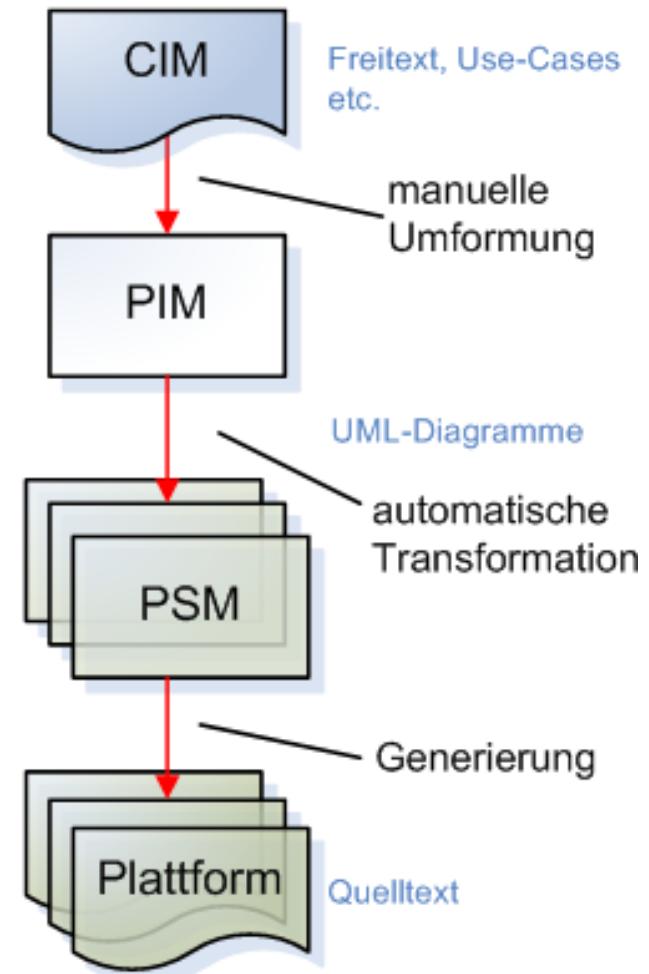
Code and
model coexist

the model
is the code

let's do
some design

- MDA: Model centric

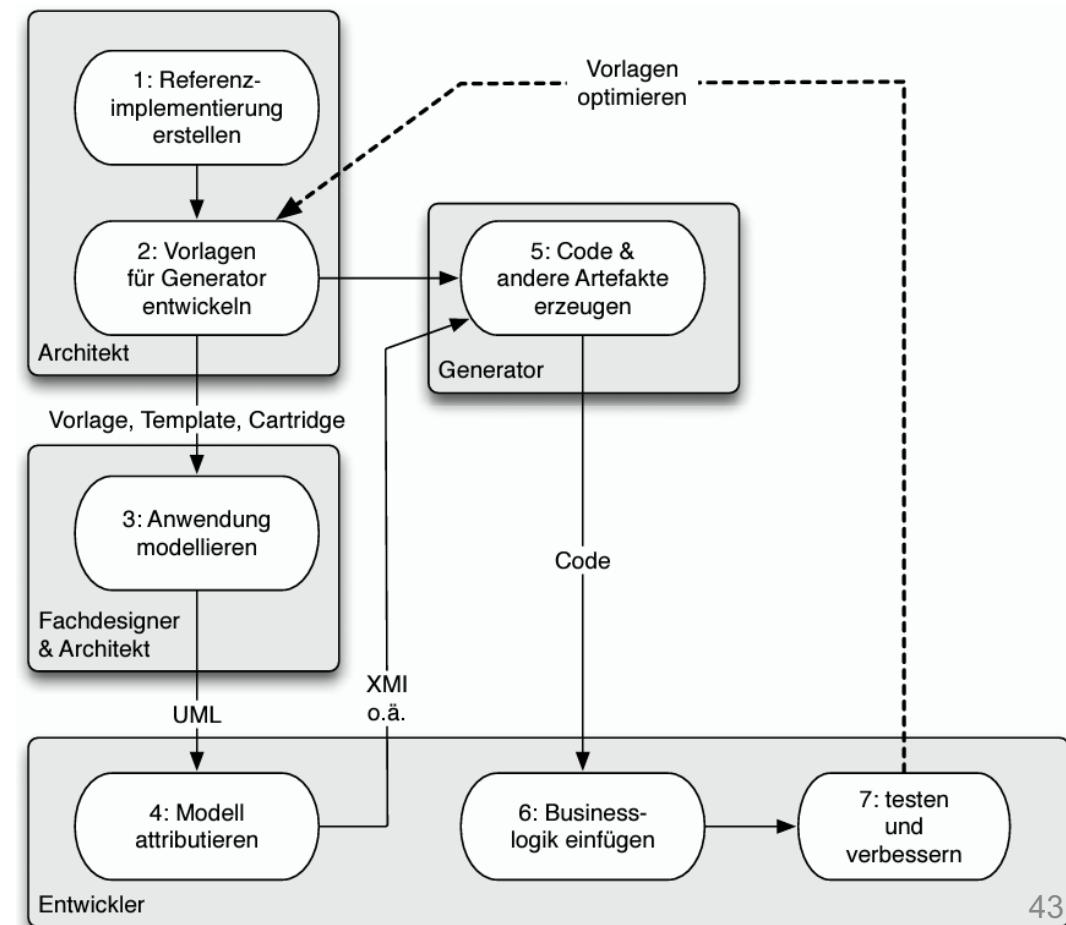
- **CIM**: Computation Independent Model
- **PIM**: Plattform Independent Model
 - ▶ plattformunabhängiges Modell für Geschäftsprozesse -> Output z.B UML
- **PSM**: Plattform Specific Model
 - ▶ plattformabhängiges Modell für Architektur, Services, etc mit **DSL** = Domain Specific Language
- **Plattform**
 - ▶ in Codemodell, die Zielplattform



3.4.3 Referenzarchitekturen - Generative Erzeugung

- Einzelne Systemteile können über einen (template-basierten) generativen Ansatz erstellt werden

- ▶ Beispiel: Eclipse EMF + JET (Java Emitter Templates), XSLT
- ▶ API basierte Generatoren: z.B. PDF
- ▶ **jHipster** DSL erzeugt Sprint-Boot BE und Angular oder React FE

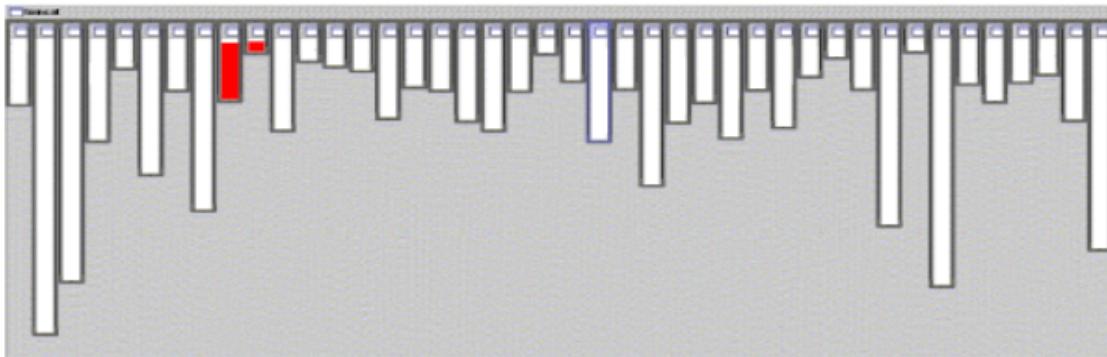


<https://start.jhipster.tech/jdl-studio/>

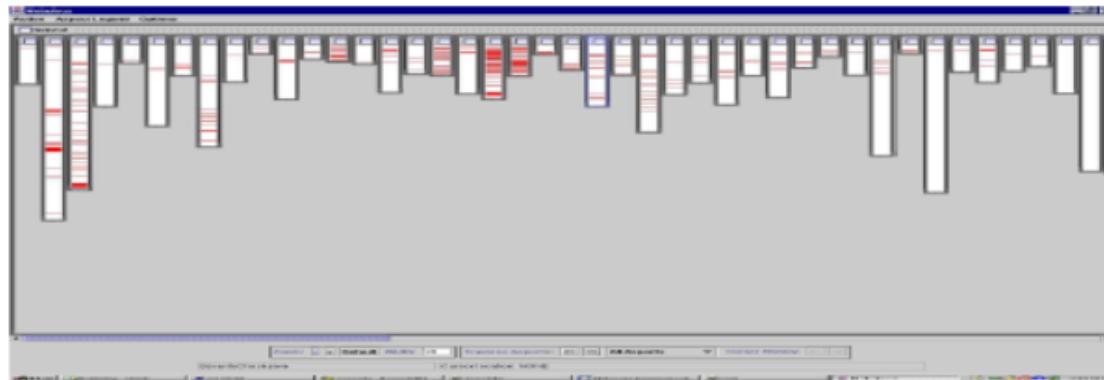
<https://www.jhipster.tech/jhipster-uml/>

3.4.3.2 Aspektorientierung (1)

- **Code von Kernanforderungen (Domänenobjekte)**
 - ▶ Gute Separation der Interessen!



- Log-Meldungen aber sind über den gesamten Code verteilt:



- ▶ Frage: Kann man das Logging besser lokalisieren?

Aspektorientierung (2) : Definition Concern (Anliegen, Gesichtspunkt)

- Definition: **Concern**
 - ▶ Spezifisches Anliegen oder Gesichtspunkt, welcher in einem Software-System behandelt werden muss, um die übergreifenden Systemziele zu erreichen
- Definition: **Core Concerns**
 - ▶ Realisieren die **Kernfunktionalität eines Systems**
- Definition: **Crosscutting Concerns**
 - ▶ Realisieren diejenigen Funktionen eines Systems, welche die Core Concerns oder andere Crosscutting Concerns **quer schneiden**, z.B. **Logging**

Nichtfachliche Concerns:

- Logging
- Performance
- Autorisierung
- Sicherheit

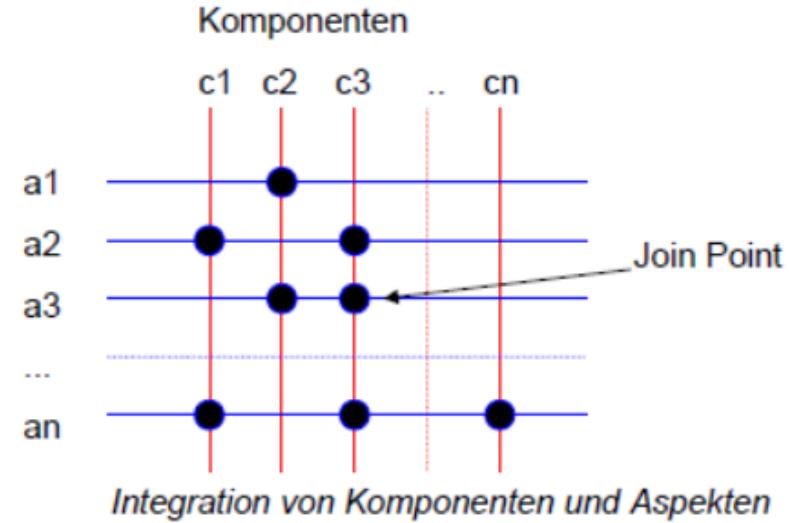
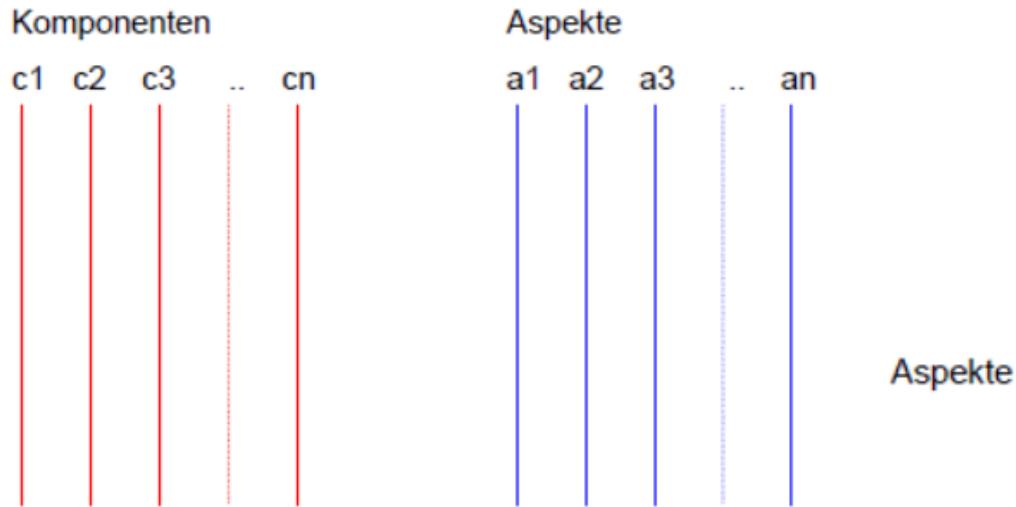


Fachliche Concerns:

- Einzahlung
- Auszahlung
- Überweisung
- Bonitätsprüfung

Aspektorientierung (4)

- Grundidee der aspektorientierten Programmierung ist es, die Separation of Concerns von Komponenten und *crosscutting* Aspekten zu meistern



- **Join Points**

- ▶ Ein *Join Point* ist ein wohldefinierter Punkt im Programmfluss.

- **Pointcuts**

- ▶ Ein *Pointcut* besteht aus bestimmten *Join Points* (und ggf. Werten an diesen join points).
 - ▶ In *Pointcuts* werden mehrere Join Points durch boolesche Operationen oder durch *Wildcards* zusammengefasst.

- Advices**

- ▶ Ein *Advice* ist Code, der ausgeführt wird, wenn ein *Pointcut* erreicht wird.
 - ▶ Zum Beispiel: Ein *After Advice* (*after*) wird *nach* dem Methodenaufruf des *Join Points* ausgeführt.
 - ▶ Weitere Arten: *before*, *around*

Aspektorientierung (6) - Advice und Pointcut

Pointcut Ausdruck

- Gibt den Ort des Pointcuts an
- Beispiel: alle Methoden die mit get beginnen

Wiederverwendung von PointCuts oder direkte Angabe

Advice Ausdruck

Advice Arten

- @Before
- @ AfterReturning
- @ AfterThrowing
- @ Finally
- @ Around

```
public class LoggingAspect {
```

```
    @Pointcut("execution(* get*(..))")  
    public void allGetters(){  
    }
```

```
    @Before("allGetters())")  
    public void loggingAdvice(JoinPoint joinPoint)  
  
    ...  
}
```

```
public class LoggingAspect {
```

```
    @Before("execution(* get*(..))")  
    public void loggingAdvice(JoinPoint joinPoint)  
  
    ...  
}
```

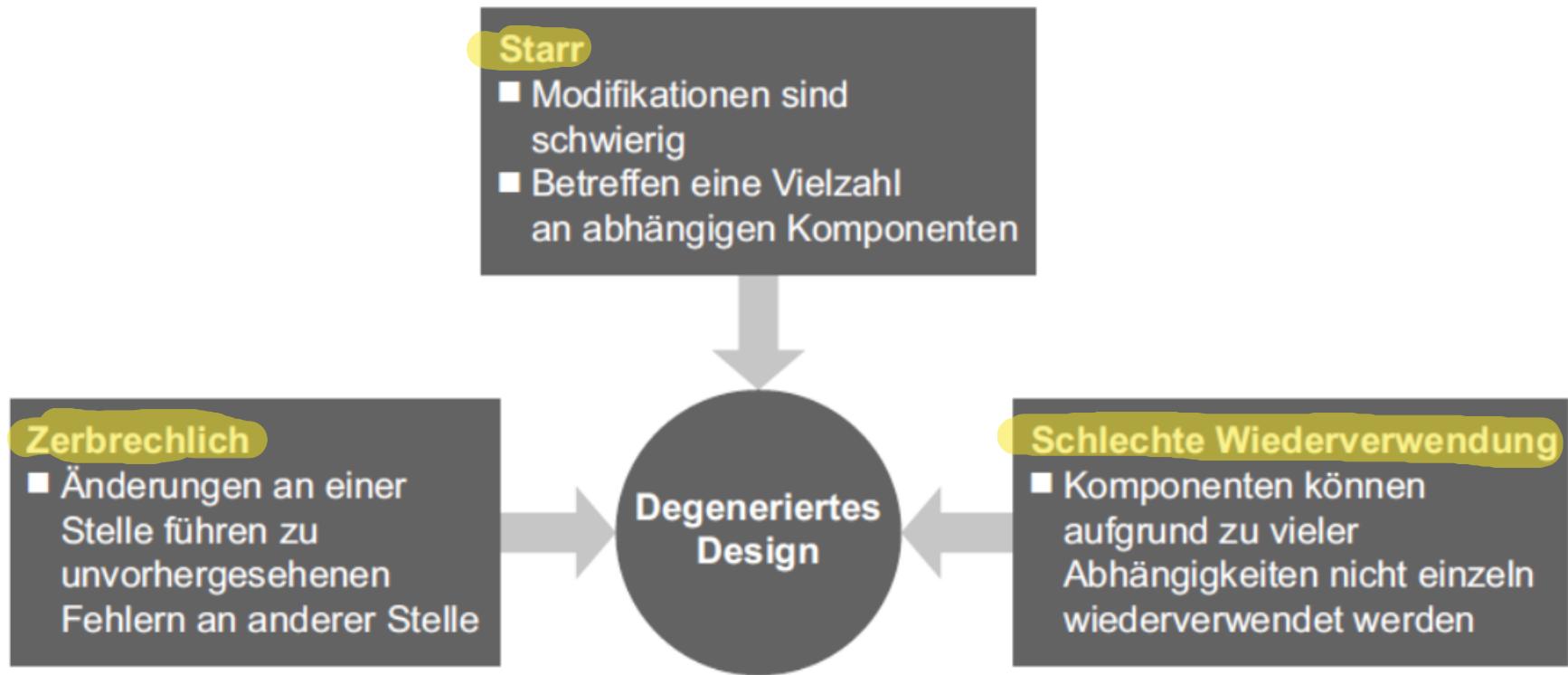
- Deutliche Verbesserung gegenüber der OOP
- Codereduktion
- Reduzierung der Wartungskosten
- Reduzierung der Entwicklungskosten
- Verbesserung der Codequalität
- Verbesserung der Softwarearchitektur
- Verbesserung der Performanz möglich

3.5. Techniken für einen guten Entwurf

- Ausgangssituation und Motivation: Degeneriertes Design
- Lose Kopplung
- Hohe Kohäsion
- Solid Prinzipien
 - ▶ Separation of Concern
 - ▶ Offen-Geschlossen-Prinzip
 - ▶ Liskov'sches Substitutionsprinzip
 - ▶ Interface Segregation
 - ▶ Umkehr der Abhängigkeiten (Dependency Injection)
- Zyklische Abhängigkeiten auflösen

3.5.1. Ausgangssituation und Motivation: Degeneriertes Design

■ Symptome von degeneriertem Design



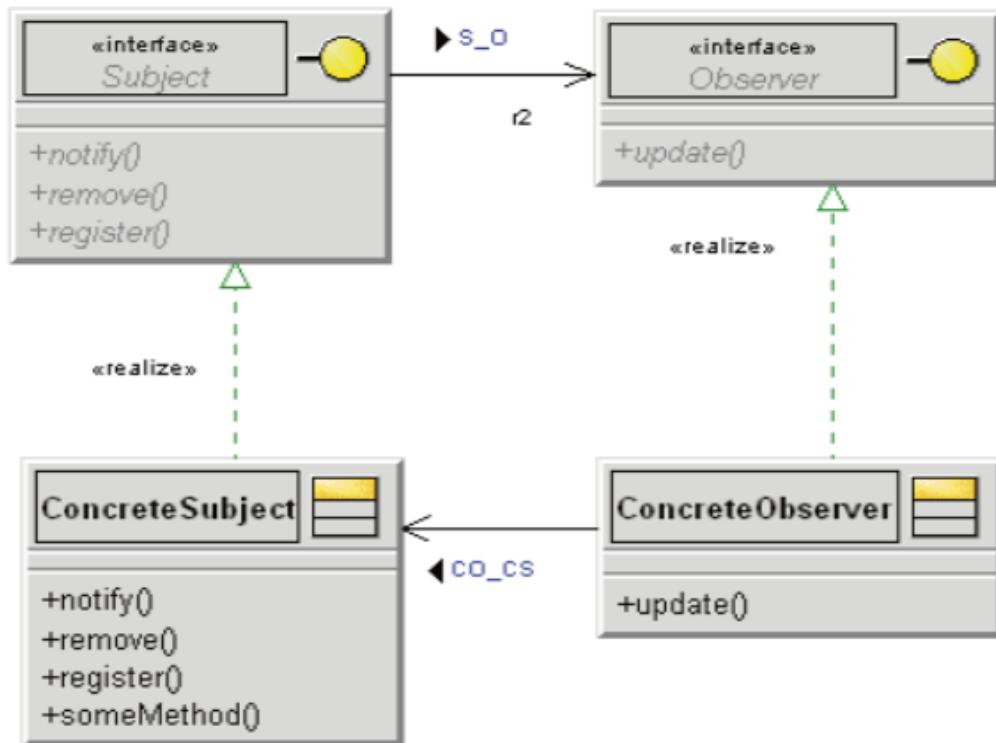
3.5.2. Lose Kopplung

- Die **Beziehung zwischen den Bausteinen** sowie die **Stärke** dieser Beziehung und der daraus resultierenden Abhängigkeiten wird **Kopplung** genannt
- **Kopplungsarten**
 - ▶ **Aufruf:** Eine Klasse **benutzt** eine andere Klasse indem sie eine Methode der **Klasse** aufruft.
 - ▶ **Erzeugung:** Eine andere Art der Kopplung besteht, wenn ein Baustein einen anderen Baustein **erzeugt**.
 - ▶ **Daten:** Eine weniger starke Kopplung liegt vor, wenn die Klassen über eine globale **Datenstruktur** oder nur über **Methodenparameter** kommunizieren.
 - ▶ **Ausführungsart:** Eine Kopplung über **Hardware** besteht, wenn Bausteine in der gleichen Laufzeitumgebung ablaufen müssen.
 - ▶ **Zeit:** Wenn die zeitliche Abfolge von Bausteinaktivitäten eine Rolle spielt, liegt eine Kopplung über die **Zeit** vor.
 - ▶ **Vererbung:** Der Grad der Kopplung ergibt sich aus der Menge der **geerbten** Eigenschaften.

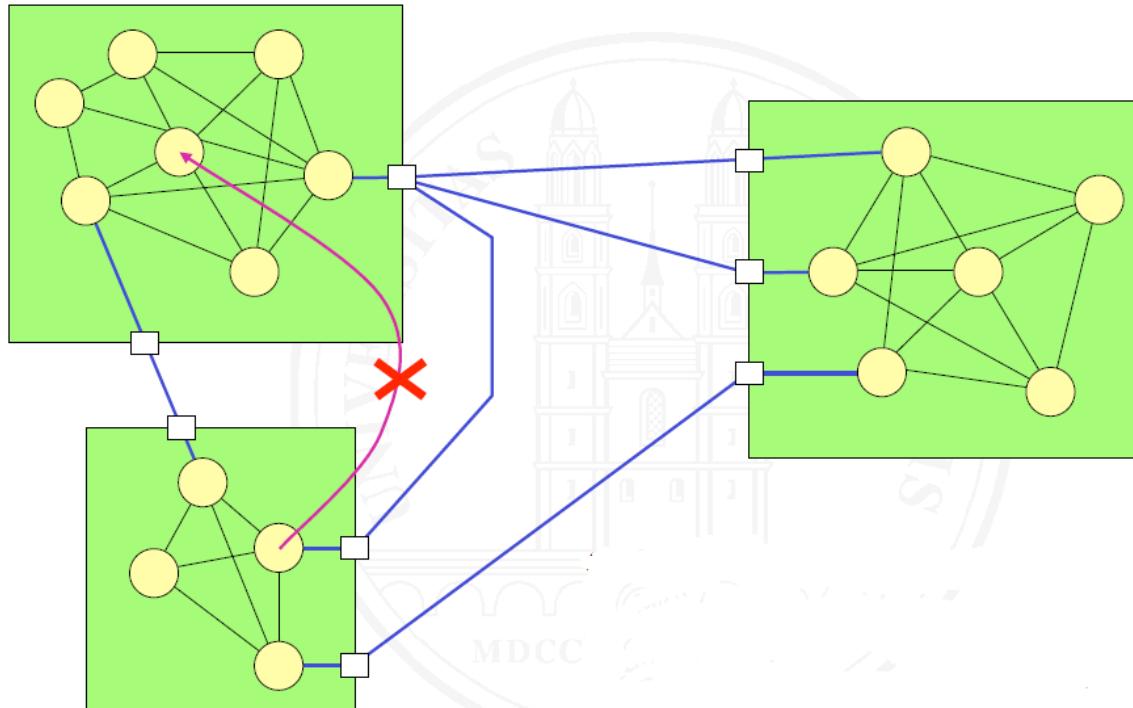
Beispiel für lose Kopplung

■ Observer Pattern

- Das **Subjekt** weiss über seine **Observer** nur, dass diese das **Interface Observer** implementieren. Zwischen Observer und Subjekt gibt es keine feste Bindung.
- Observer können **jederzeit registriert oder entfernt** werden.



- **Modulintern** starker Zusammenhang
- **Modulextern** schwache Kopplung
- **Kommunikation** nur über Schnittstellen
- **Modulinneres** von aussen nicht sichtbar



- Zwei charakteristische **Masse**: **Kohäsion und Kopplung**
- **Kohäsion** (cohesion) – Ein Mass für die Stärke des inneren Zusammenhangs eines Moduls.
 - ▶ Je höher die Kohäsion, desto besser die **Modularisierung**
 - ▶ schlecht: zufällig
 - ▶ gut: funktional, objektbezogen
- **Kopplung** (coupling) – Ein Mass für die Abhängigkeit zwischen zwei Modulen.
 - ▶ Je geringer die wechselseitige **Kopplung** zwischen den Modulen, desto besser die Modularisierung
 - ▶ schlecht: Globale Kopplung
 - ▶ gut: Datenkopplung

3.5.3. Hohe Kohäsion

- Die Kohäsion wird auch Zusammenhangskraft genannt.
- Begriff **cohaerere** -> zusammenhängend
- **Geringe Kopplung** führt dazu, dass die Bausteine **im Inneren stärker zusammenhängend** entworfen werden.
- Eine kohärente Klasse löst **ein einziges Problem** und besitzt eine spezifische Menge an stark zusammenhängenden Funktionen.

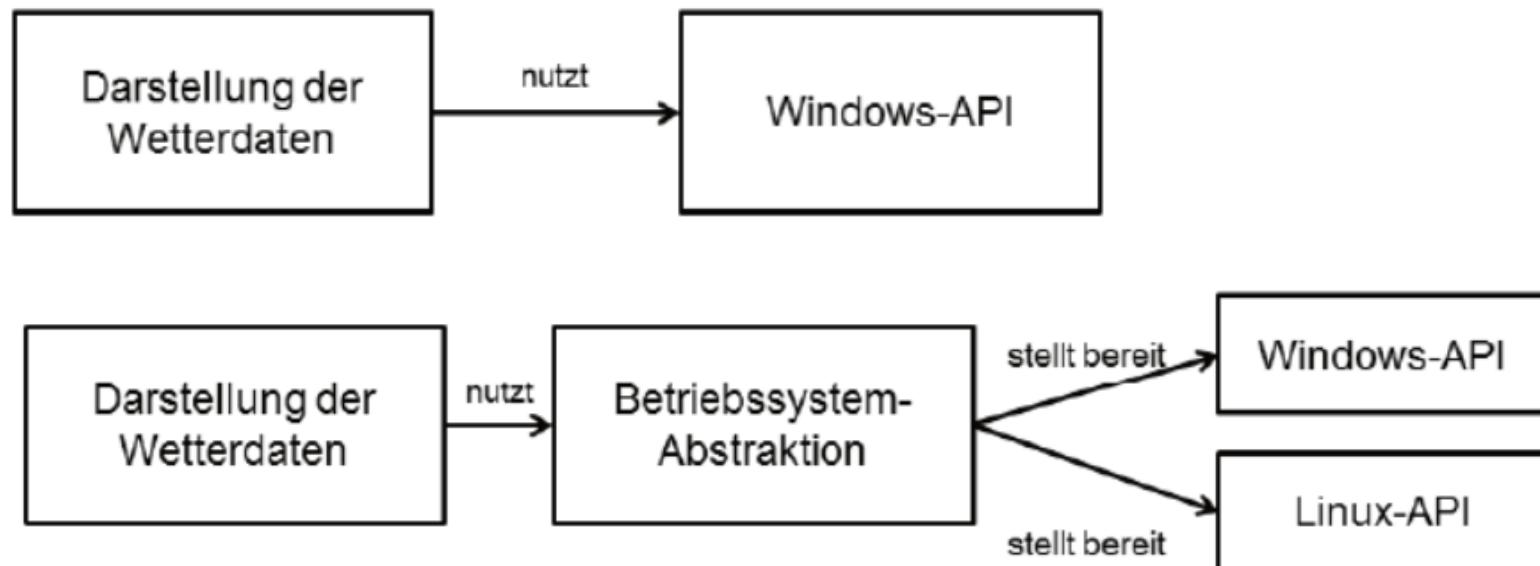
3.5.4. Offen-Geschlossen-Prinzip

- Das Offen-Geschlossen-Prinzip wurde 1988 von Bertrand Meyer definiert und lautet:

- ▶ Softwarebausteine sollen offen für Erweiterung sein, aber geschlossen für Änderungen.
- ▶ Geschlossen heisst, dass das Modul **risikolos verwendet** werden kann, da sich seine **Schnittstelle** nicht mehr ändert.
- ▶ Offen hingegen bedeutet, dass das **Modul problemlos erweitert** (zum Beispiel durch Vererbung) werden kann.

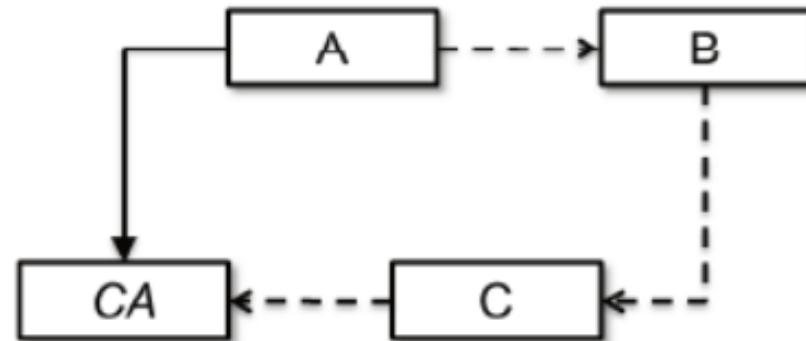
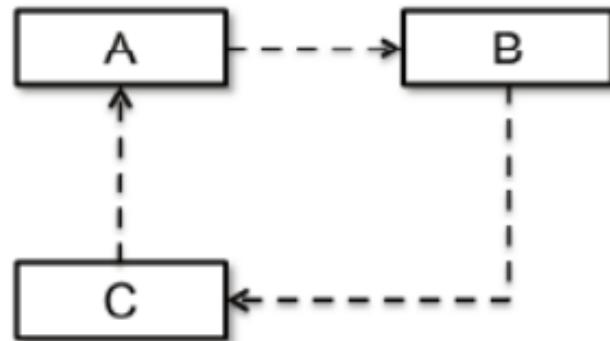
3.5.5. Umkehr der Abhangigkeiten (Dependency Inversion)

- **Dependency Inversion:** keine direkten Abhangigkeiten, sondern nur Abhangigkeiten von Abstraktionen
 - ▶ Austauschbarkeit von Bausteinen wird erleichtert
 - ▶ Direkte Abhangigkeiten zwischen Klassen zum Beispiel durch Factory-Methoden entkoppeln



3.5.7. Zyklische Abhangigkeiten auflosen

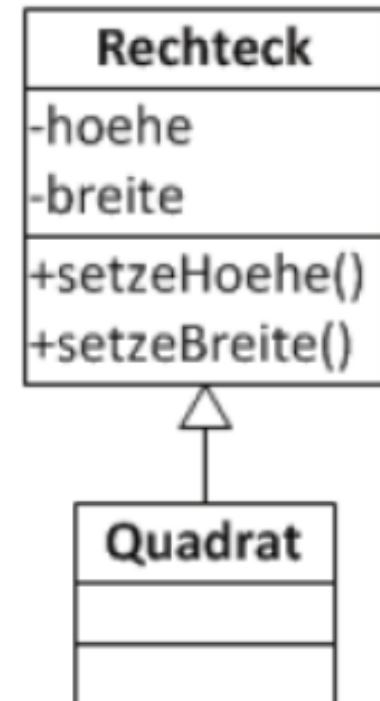
- Zyklische Abhangigkeiten erschweren die Wartbarkeit und die Änderbarkeit von Systemen und verhindern eine getrennte Wiederverwendung.



- Vorgehen:
 - ▶ Trennen Sie aus A diejenigen Teile als Abstraktion CA heraus, die von C genutzt werden.
 - ▶ Die Auflösung der zyklischen Abhangigkeit erfolgt durch eine Vererbungsbeziehung von A zur Abstraktion CA.

3.5.8. Liskov'sches Substitutionsprinzip

- Dieses Prinzip besagt, dass eine Basisklasse immer durch ihre abgeleiteten Klassen (Unterklassen) ersetzbar sein soll.
- In diesem Fall soll sich die Unterklasse genauso verhalten wie ihre Oberklasse.
- Beispiel:
 - ▶ Das Quadrat verhält sich in Bezug auf die Seitenlänge nicht wie das Rechteck
 - ▶ Höhe und Breite sind beim Quadrat identisch



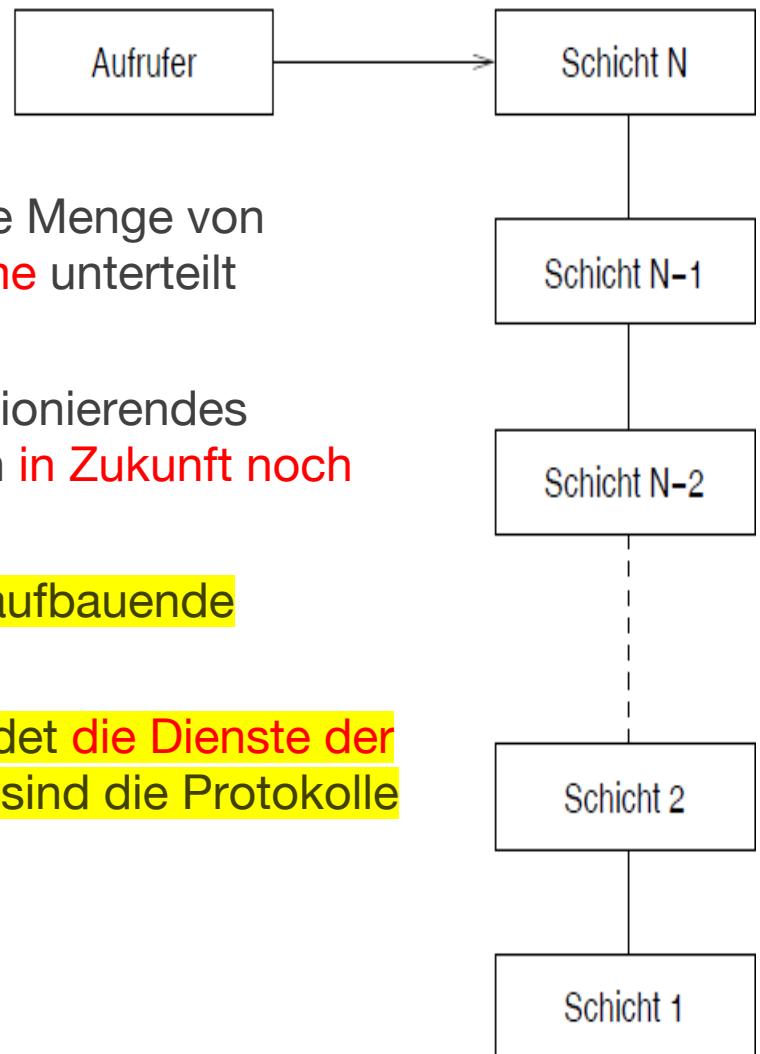
- Einteilung in:

- ▶ **Adaptierbare Systeme:** Muster dieser Kategorie unterstützen die Erweiterung von Anwendungen und ihre Anpassung an sich weiterentwickelnde Technologien und sich ändernde funktionale Anforderungen.
- ▶ **Interaktive Systeme:** Muster interaktiver Systeme unterstützen die Strukturierung von interaktiven Softwaresystemen.
- ▶ **Vom Chaos zur Struktur:** Muster dieser Kategorie dienen dazu, ein Durcheinander von Komponenten und Objekten zu vermeiden. Insbesondere unterstützen sie bei der sinnvollen Zerlegung einer übergeordneten Aufgabe des Systems in kooperierende Teilaufgaben.
- ▶ **Verteilte Systeme:** Muster dieser Kategorie machen Aussagen zu bewährten Formen der Arbeitsteilung und darüber, nach welchem Vorgehen Subsysteme miteinander kommunizieren können.

3.6.1 Schichtenarchitektur

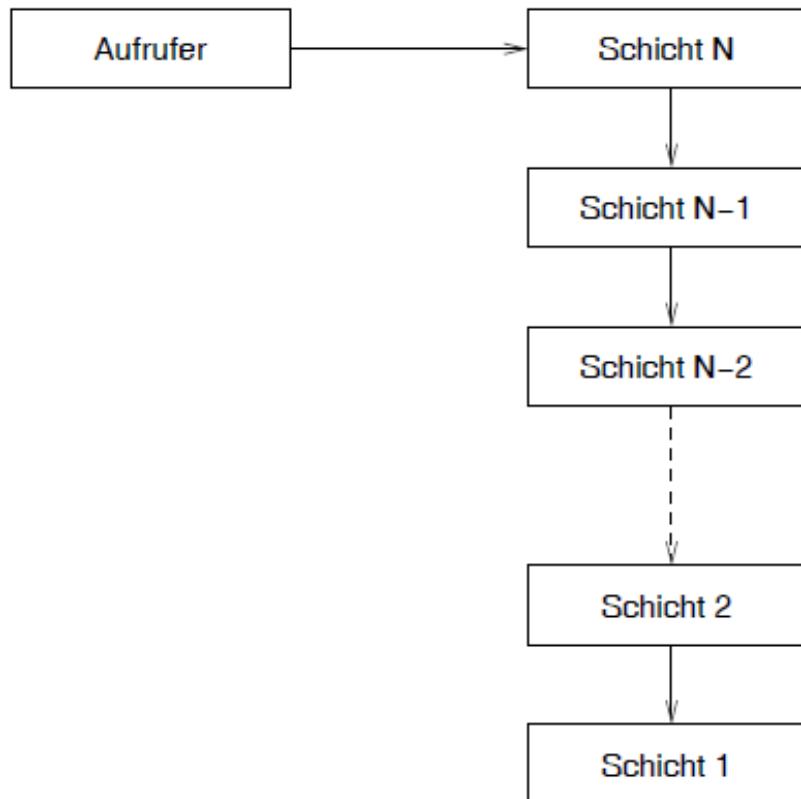
- Diese Muster dienen der Partitionierung von Systemen.

- ▶ Charakteristisch sind eine unüberschaubare Menge von Anforderungen, die **in geeignete Subsysteme** unterteilt werden müssen.
- ▶ Das Ziel des Entwurfs ist nicht nur ein funktionierendes System, sondern ein System, welches auch **in Zukunft noch wartbar** bleibt.
- ▶ Das System wird realisiert als aufeinander aufbauende **Schichten**.
- ▶ Jede Schicht bietet **Dienste an** und verwendet **die Dienste der untergeordneten** Schicht. Die Konnektoren sind die Protokolle der einzelnen Schichten.

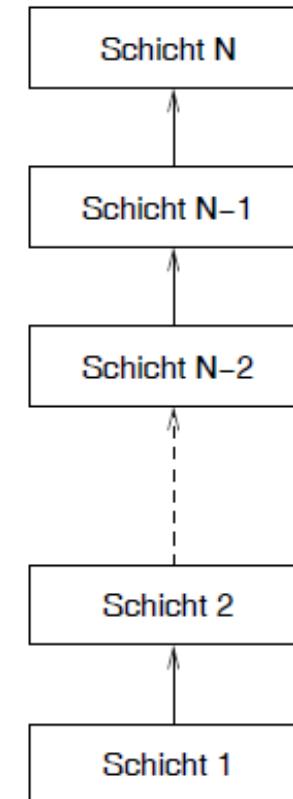


Schichtenarchitektur - Aufrufszenarien

Szenarien für Aufträge höherer Schichten über Aufruf

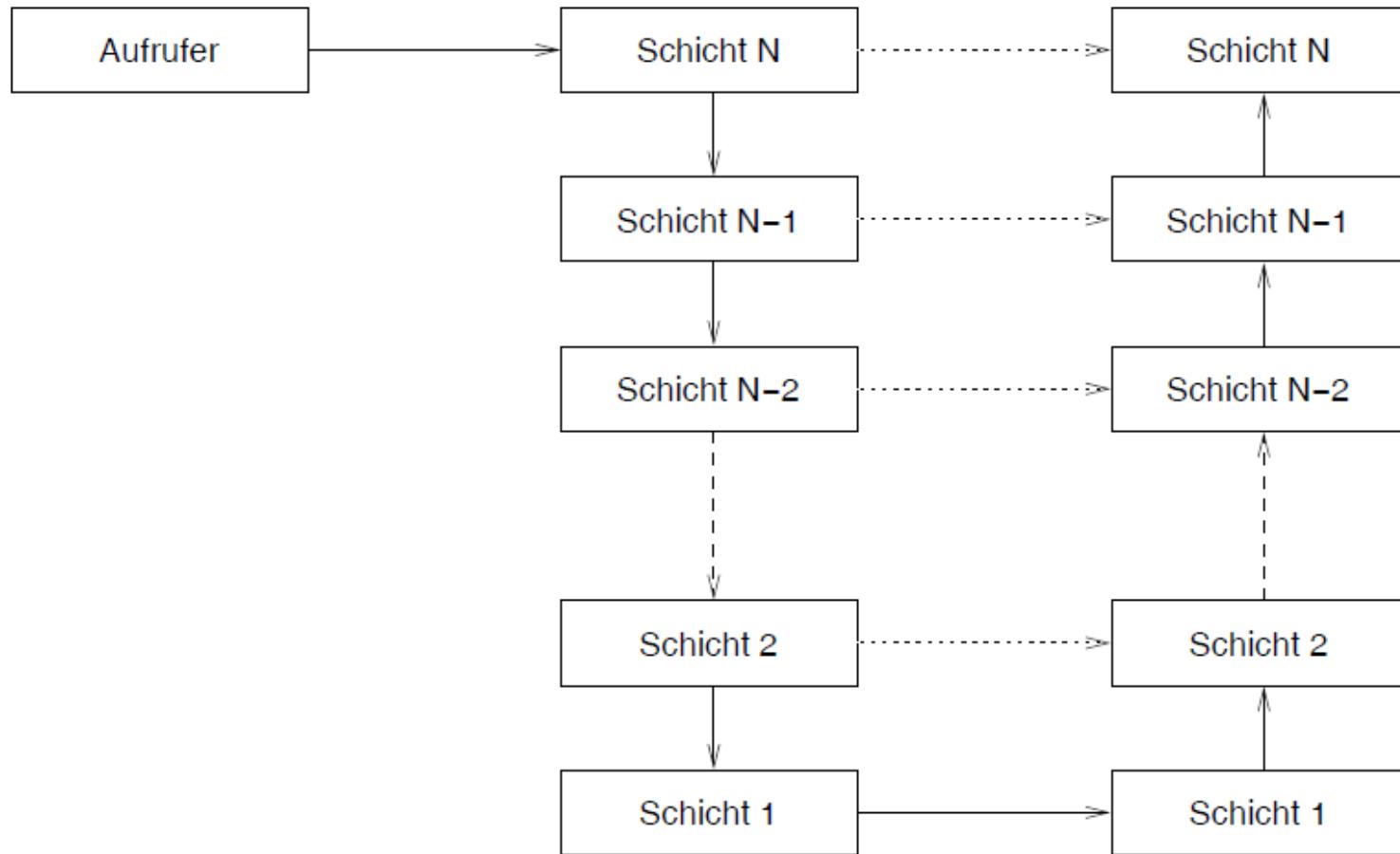


untere Schicht benachrichtigt obere Schicht über Ereignis



Beispiel: Aufruf zweier geschichteter Systeme

- Szenario für Interaktion zweier geschichteter Systeme (OSI)



- **Schichtenarchitekturen** sind konzeptionell unabhängig von objektorientierten Ansätzen
- Umsetzung ist auch in anderen Programmierparadigmen möglich
- Unterscheidung nach Grad der Kopplung
 - ▶ Kopplung zwischen Komponenten
 - ▶ Kopplung zwischen Schichten
- Interaktionen:
 - ▶ White-Box Interaction
 - ▶ Grey Box Interaction
 - ▶ Blackbox Interaction

Schichtenarchitektur Vergleich

Desktop vs Web - Mobile

Presentation: Desktop-App

Aufruf
Systemoperationen

Events
(Listeners, Observables)

Domain-Controller
bzw. Services
(Fowler)

Domain Model

Technical Services
Persistence, Logging

Presentation: Mobile/Web-App

Backend Services im Frontend
http-client, observable streams, JSON/XML mapping

Aufruf
Endpoints,
DTO's

Events
(Web-Sockets)

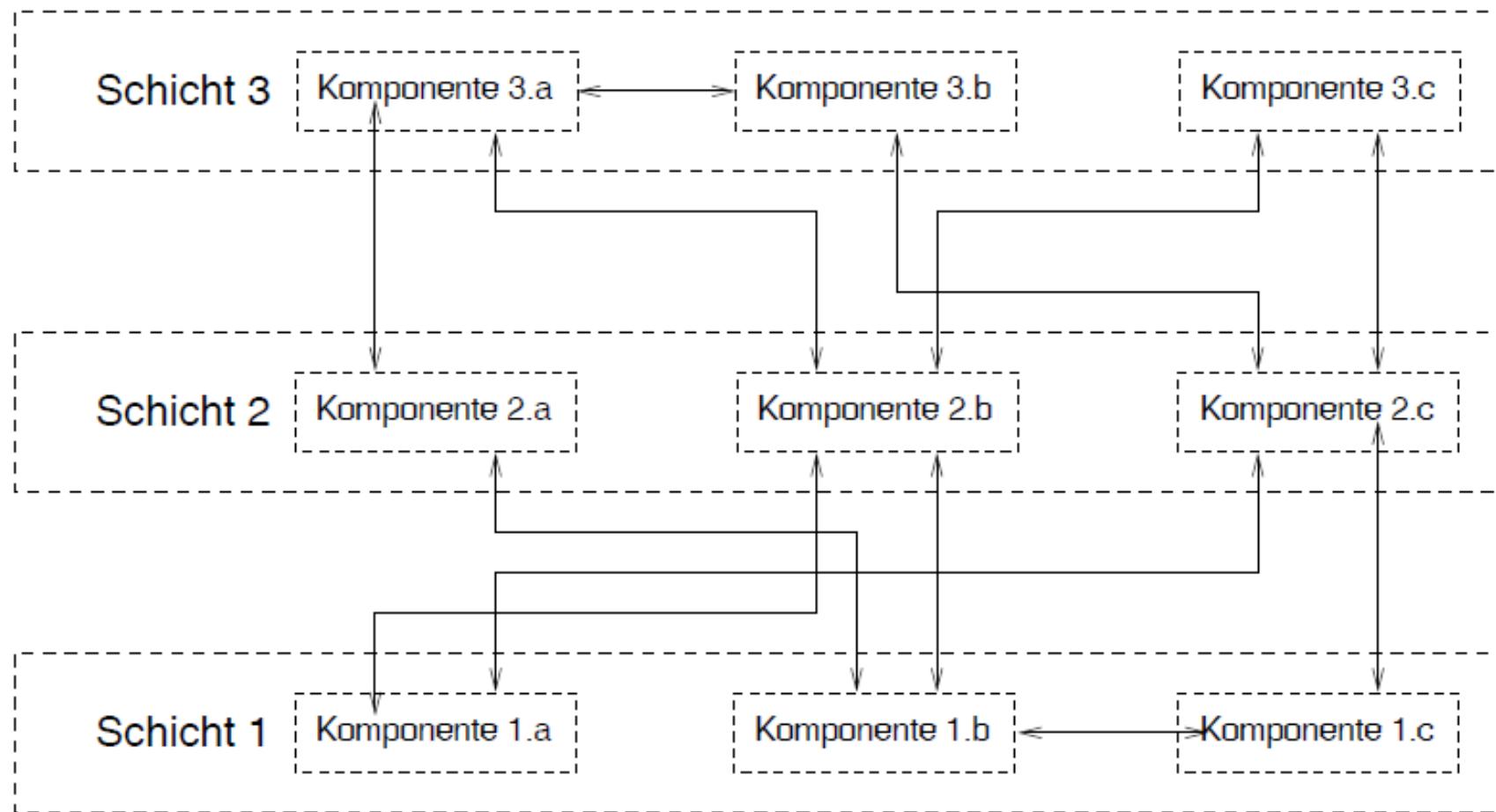
REST Controller: JSON/XML mapping

Domain-Controller
bzw. Services
(Fowler)

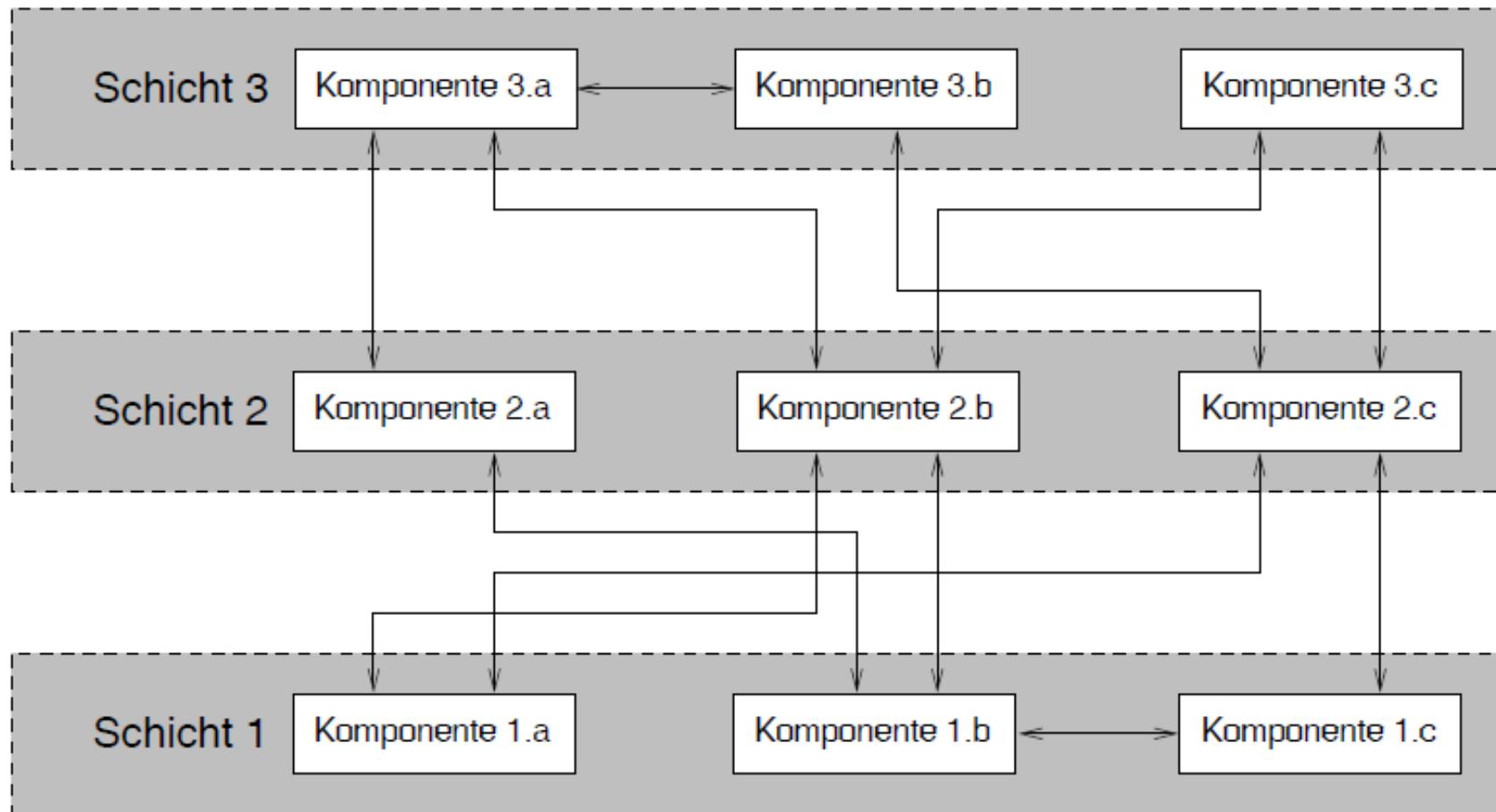
Domain Model

Technical Services
Persistence, Logging

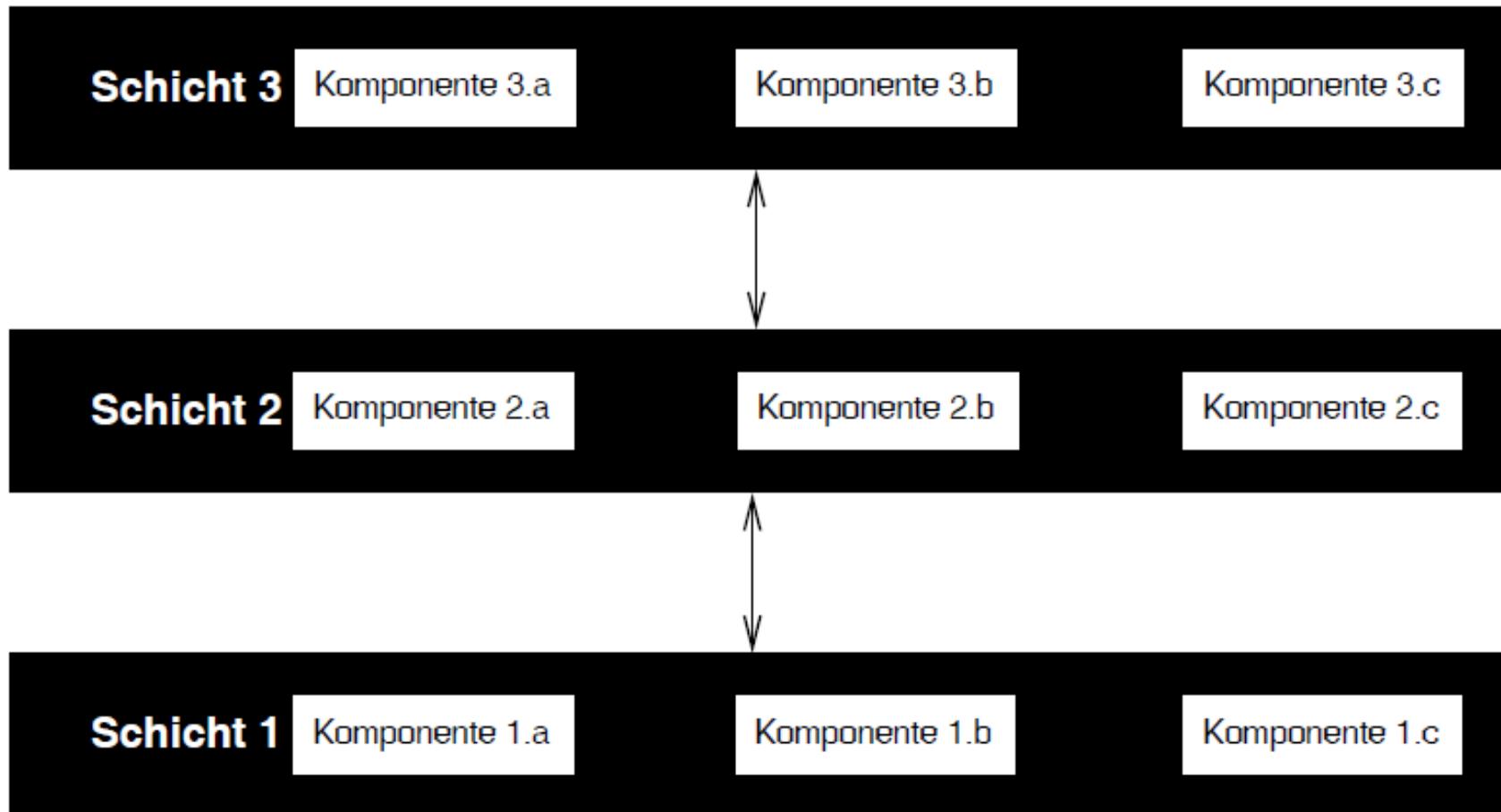
- Komponenten sehen Interna der Komponenten anderer Schichten



- Komponenten rufen Dienste direkt bei Komponenten anderer Schichten auf



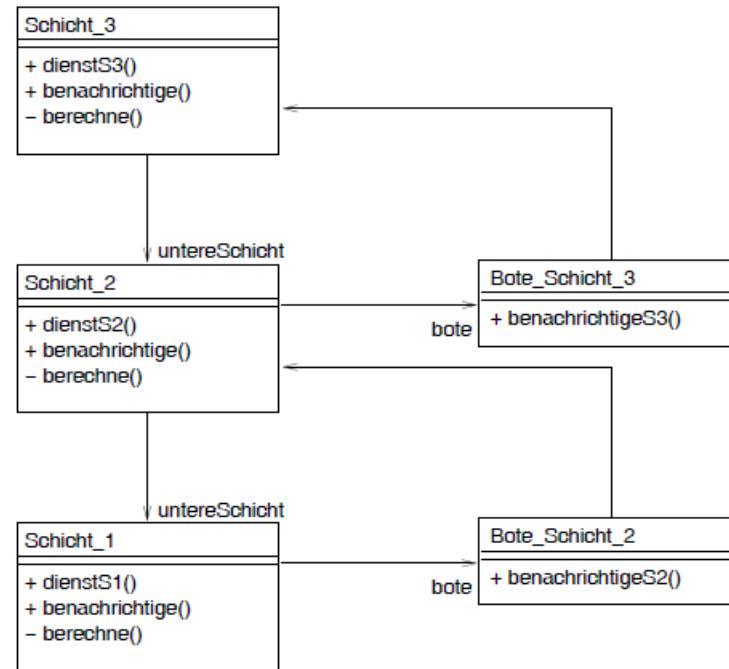
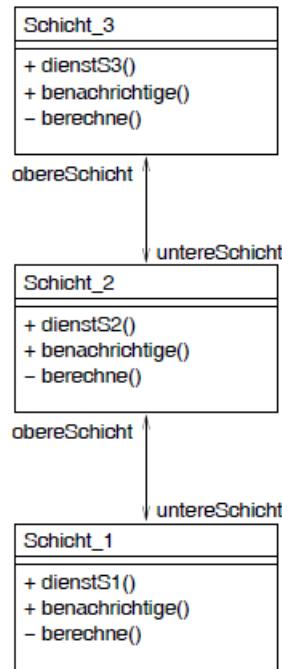
- Komponenten interagieren nicht direkt miteinander



- nicht nur Komponenten der Schichten von Komponenten anderer Schichten entkoppeln (black-box)
- auch einzelne Schichten weitgehend voneinander entkoppeln:
 - ▶ lokale Behandlung von Änderungen
 - ▶ erhöhte Austauschbarkeit
 - ▶ einfacheres Testen

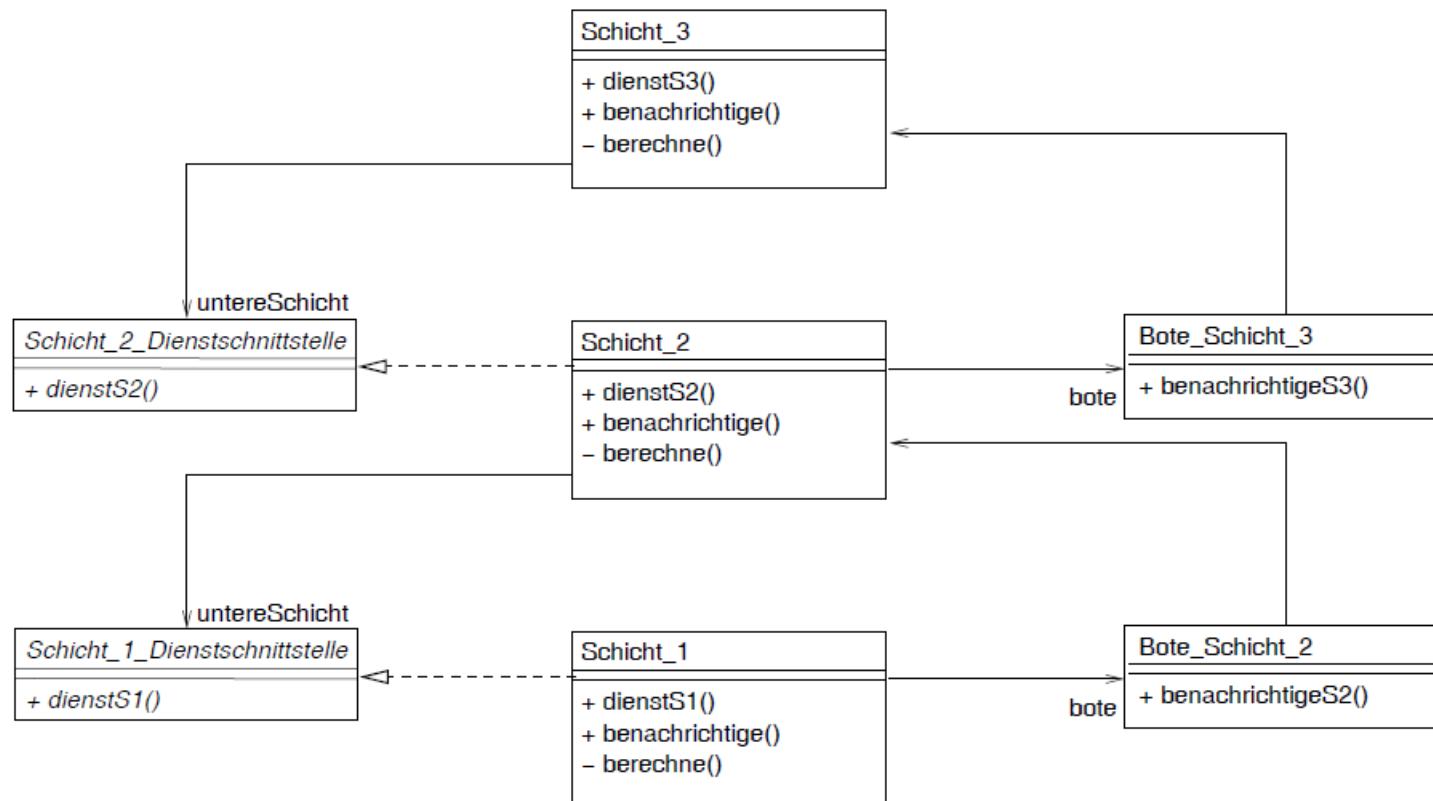
Entkopplung der Schichten – **Callbacks**

- **Vorher:** Schicht kennt obere und untere Schicht und ruft direkt Dienste auf



- **Nachher:** untere Schicht hat Referenz für Verschickung von Benachrichtigungen – callbacks (Observable, Listeners, Events)

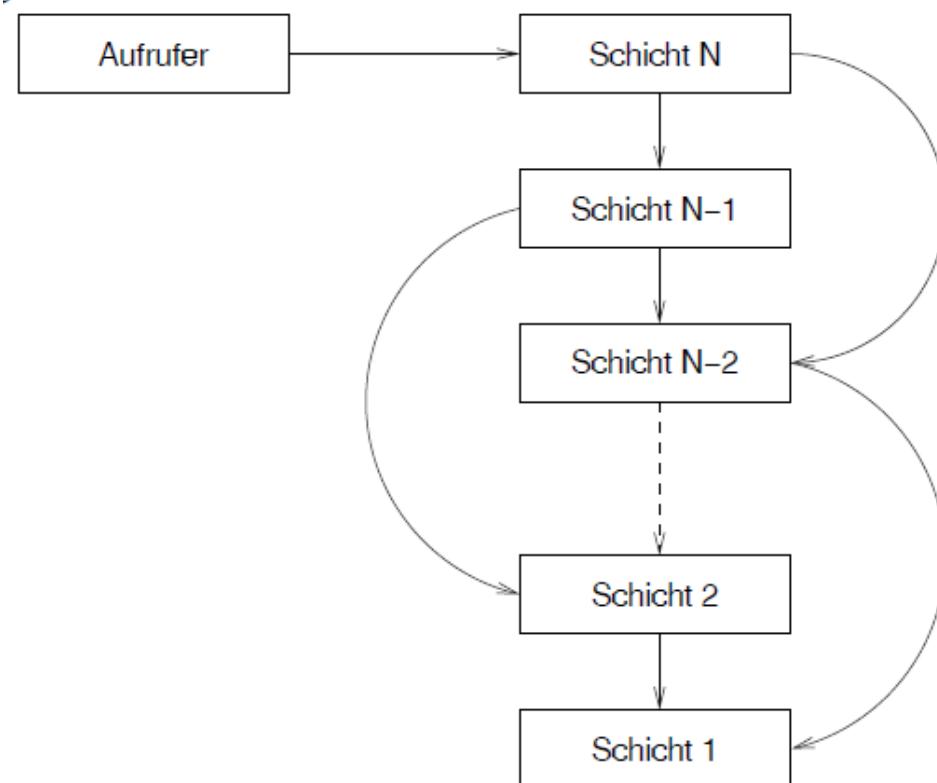
- **Vorher:** obere Schicht kennt untere Schicht und ruft direkt Dienste auf.



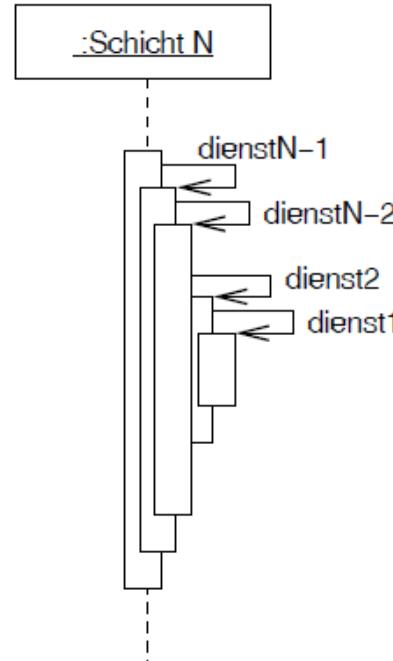
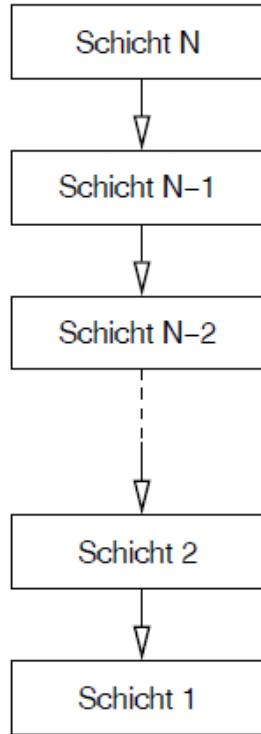
- **Nachher:** gegen Schnittstelle programmiert, Implementierung austauschbar

Varianten: Gelockerte Schichtung

- Schichten interagieren auch mit weiter entfernten Schichten
 - ▶ **Vorteil:** Performance-Steigerung
 - ▶ **Nachteil:** Wartbarkeit, Änderbarkeit geringer



Varianten: Schichtung durch Vererbung



- **Vorteil:** Modifikation von Diensten möglich auf höherer Ebene
- **Nachteil:** Wartbarkeit, Änderbarkeit geringer

3.6.2. Modularisierung

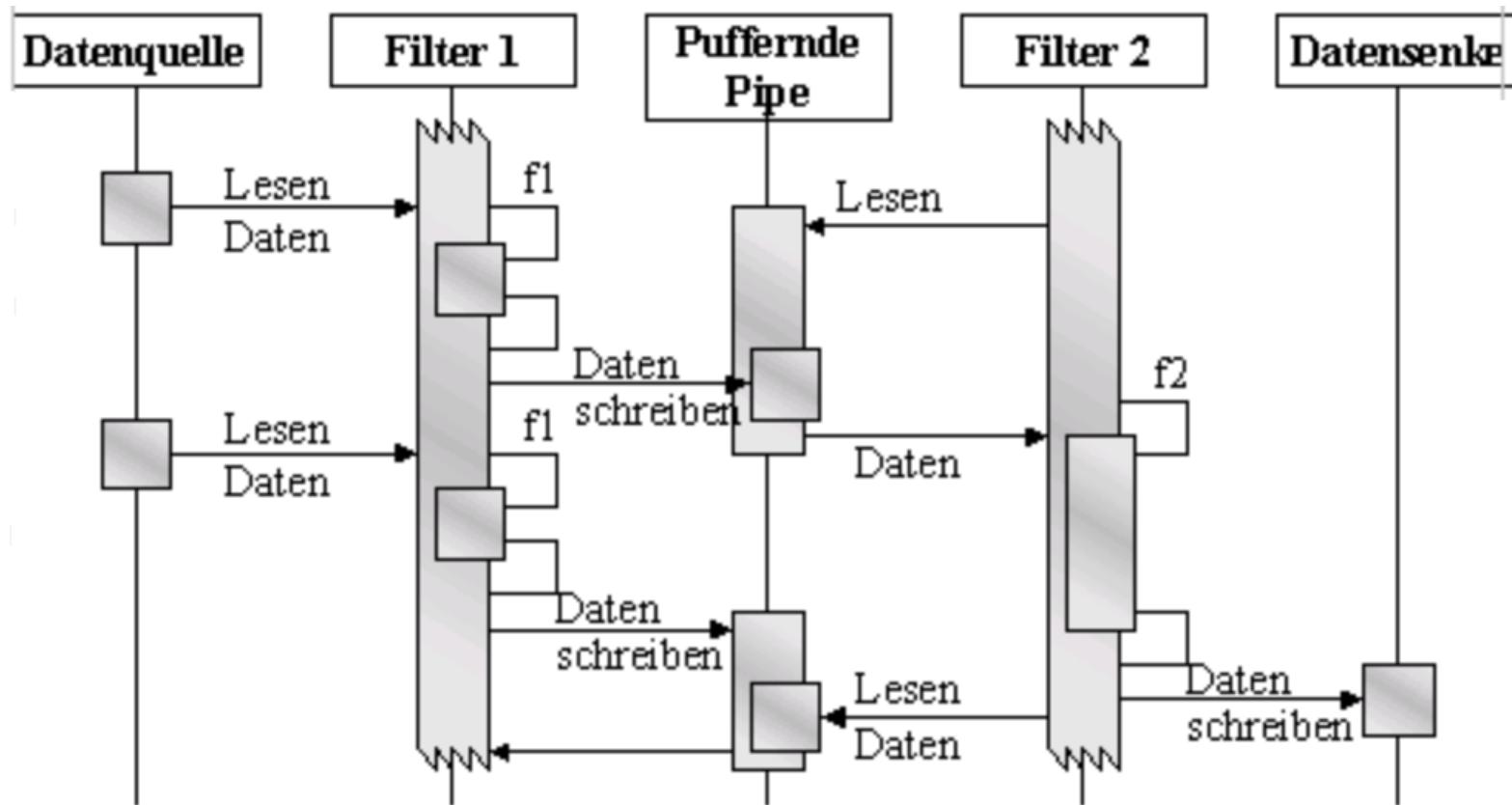
- **Modularisierung:** Zerlegung eines Softwaresystems in Subsysteme und Komponenten. Daraus ergibt sich eine logische Struktur
- **Modul** (module). Ein benannter, klar abgegrenzter Teil eines Systems. Ist ein Behälter für Funktionalität und Verantwortungsbereiche
- **Beispiele für Module** (auf verschiedenen Stufen)
 - ▶ Komponente
 - ▶ abstrakter Datentyp
 - ▶ Klasse
 - ▶ Prozedur/Methode
- **Abgrenzung** zur Schichtenarchitektur
 - ▶ Es lassen sich auch vertikale Schichten bilden (getrennte Aufgabenbereiche)

3.6.3. Pipes und Filter

- Durch **Pipes and Filters** kann die Aufgabe eines Systems als Kombination von einzelnen sequenziellen Teilschritten (**Filter**) implementiert werden.
- Ein **Datenstrom** wird dabei durch die **Pipeline** zwischen den einzelnen Filtern und wird **Schritt für Schritt** weiterverarbeitet.
- Jeder **Filter** übernimmt dabei **genau eine Aufgabe**, die er unabhängig von den anderen Filtern umsetzt (die Filter wissen nicht, dass andere Filter existieren).
- Die Ausgabedaten des einen Filters (**Datenquelle**) sind dabei gleichzeitig die Eingabedaten des folgenden Filters (**Datensenke**).

Beispiel: Pipeline mit Puffer

- Es gibt **zwei oder mehr** aktive **Filterelemente**. In diesem Beispiel existiert eine Pipe zwischen Filter1 und Filter2. Sie verfügt über einen Puffer.



Microsoft PowerShell: Echte Objekte in der Pipeline

```
Gwmi Win32_Service | where {$_ .State -eq "Running"} |  
select Name, StartMode
```

(Name und Startmodus aller laufenden Dienste ausgeben)

```
du | sort -rn | head -5
```

(Die fünf grössten Dateien ausgeben)

```
cat LocalSettings.php | grep "^ *#[\r\n]" | wc -l
```

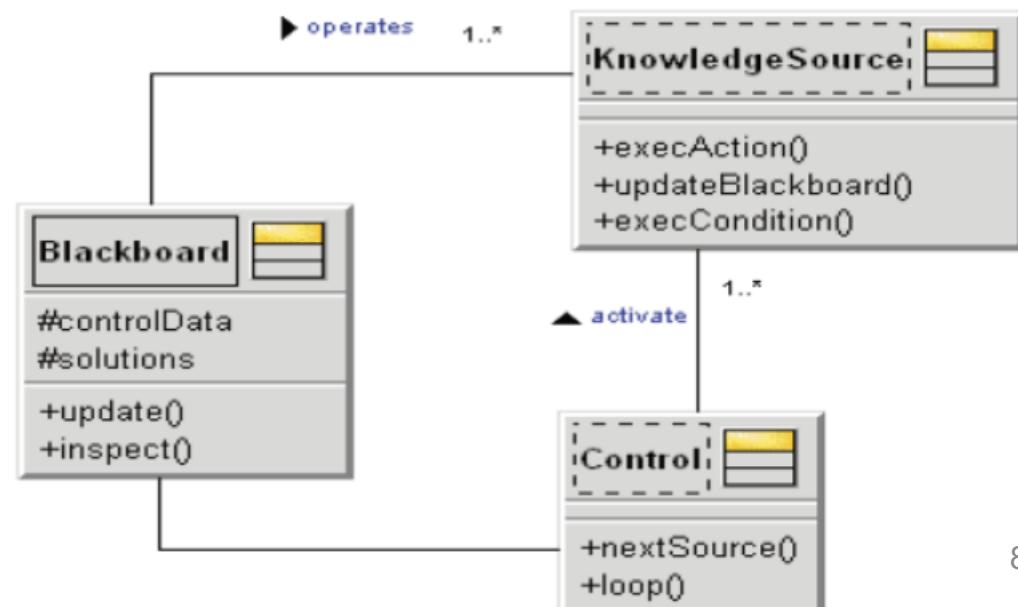
(Anzahl Kommentarzeilen in PHP-Datei ausgeben)

- Compiler
 - ▶ Schrittweise Verarbeiten: Weiterreichen des Ergebnisses nach jedem Verarbeitungsschritt (Lexer, Parser, Codegenerator)
- Digitale Signalverarbeitung mit Filtern
 - ▶ Bilderfassung, Farbkorrektur, Bildeffekte: Daten werden jeweils weitergegeben

3.6.4. Blackboard

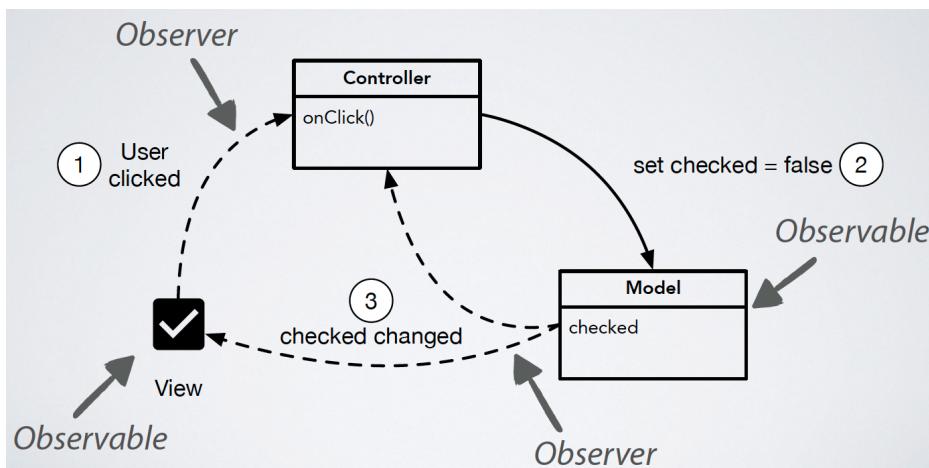
■ Blackboard

- ▶ Im Bereich **Robotersteuerung und Mustererkennung** (Bild, Ton, Sprache, Schrift, Systemüberwachung) wird **aufgrund der nichtdeterministischen Problemlösung die Blackboard Architektur häufig verwendet.**
- ▶ **Ein schwarzes Brett dient als zentrale Datenstruktur für Agenten** die vorhandenes Wissen verarbeiten bzw. Neues hinzufügen. Ein Controller prüft welcher **Agent** bzw. **KnowledgeSources** eingesetzt wird.
- ▶ Die **Kontrollkomponente** beobachtet das Blackboard und steuert bei Bedarf die **Ausführung** der KnowledgeSources.



Model View Controller (1)

- Trennung von Zustand, Präsentation und Logik
 - ▶ Bessere Testbarkeit
 - ▶ Viele unterschiedlich Interpretationen
- View: Was der Benutzer sieht
 - ▶ z.B. Checkbox
- Model: Anzuzeigende Daten
 - ▶ checked =true/false



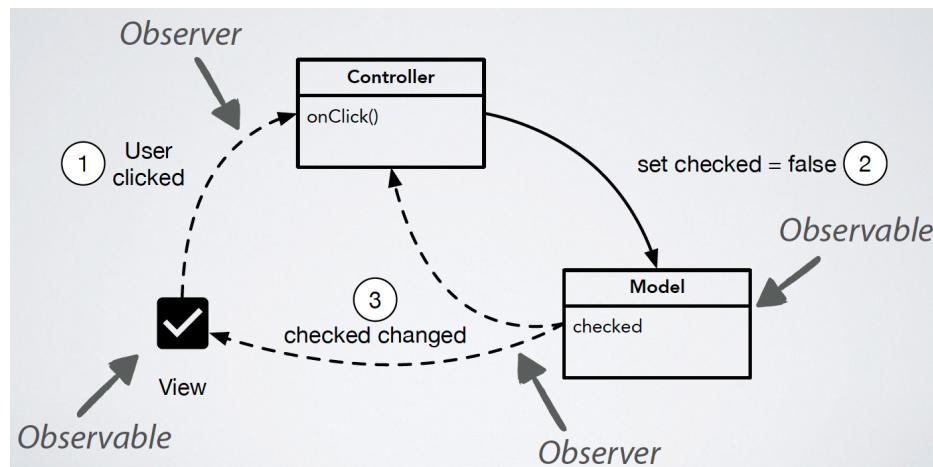
Model View Controller (2)

■ Controller

- ▶ Behandelt Benutzer-Ereignisse
- ▶ Koordiniert Aktionen mit anderen Controllern in der Anwendung
- ▶ Enthält die Logik für die Veränderung des Models (über DomainController)

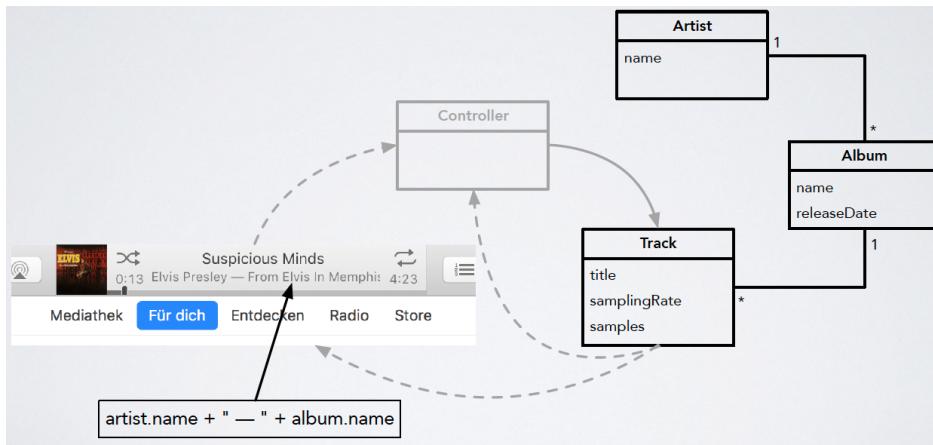
■ Observer-Pattern ist integraler Teil von MVC

- ▶ Das Model (Observable) informiert die View (Observer) über Änderungen
- ▶ Der Controller sollte nicht direkt mit der View kommunizieren, sondern indirekt über die Aktualisierung des Model.



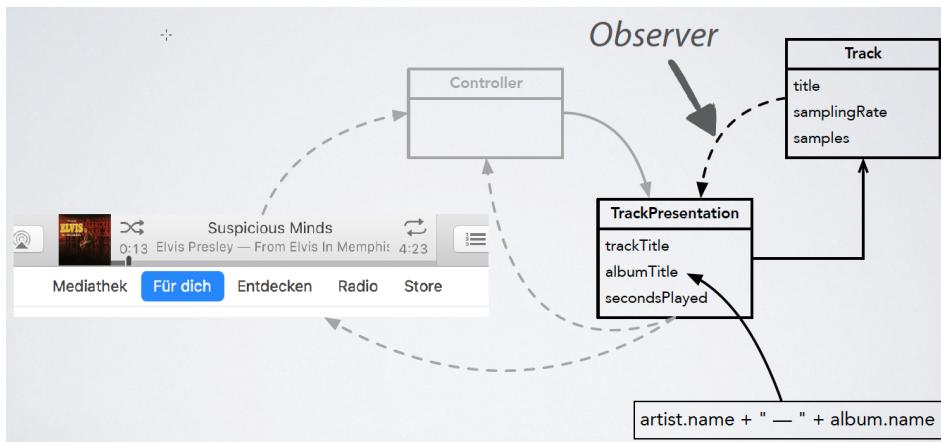
Das Modell

- Model => **Domain Model**
 - ▶ Ein Model pro Anwendung
- Wird schnell unübersichtlich
 - ▶ Wo sollen berechnete Werte gespeichert werden?
 - ▶ Wo der Zustand des UI's?
 - ▶ Wo I18n?
- Diese UI-Daten ruinieren das Domainmodel
 - ▶ Das Domainmodel soll nur fachliche Daten enthalten



Das Presentation Model

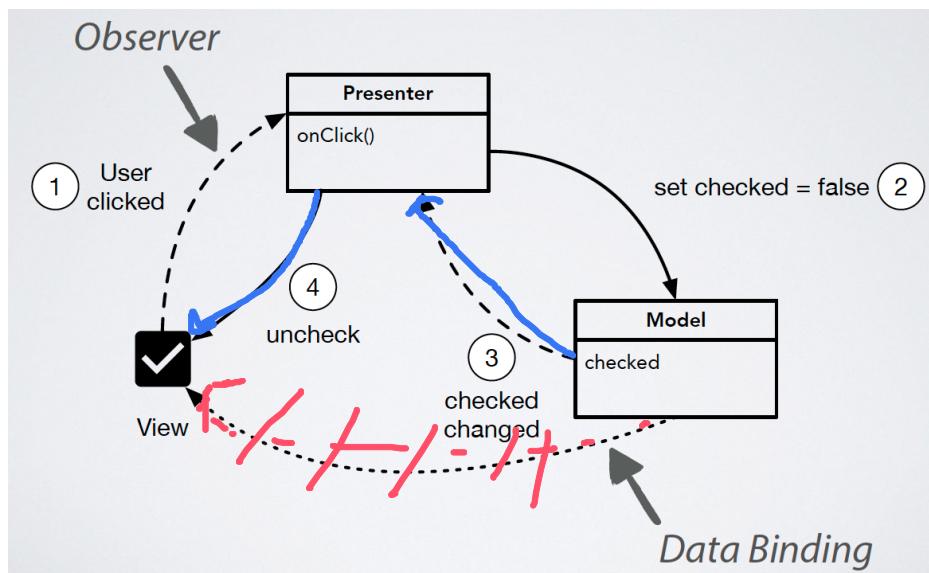
- Das Presentation-Model löst das Problem
 - ▶ Wird vor dem Domain-Model positioniert
- Enthält alle darzustellenden Daten plus den Zustand des UI's
 - ▶ Minimiert die Logik des Views
 - ▶ Hält das Domain-Model sauber



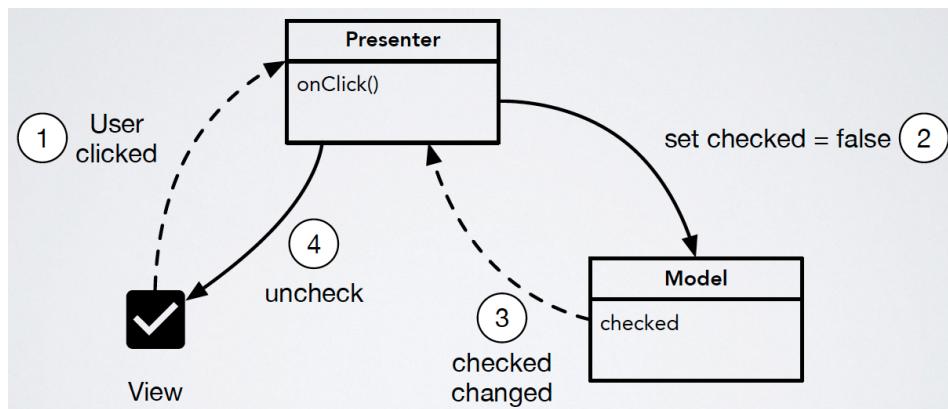
- MVP ist die ursprüngliche Bezeichnung
- Zwei Varianten
 - ▶ Supervising Controller
 - ▶ Passive View

Supervising Controller

- Unterschied MVC zu Supervising Controller
 - ▶ View wird aktualisiert mittels DataBinding anstelle von Observer
 - ▶ Presenter manipuliert die View direkt
- Verwendung eines Presentation Models ist empfohlen
- Erhöht die Testbarkeit, weil sich die Logik nicht mehr in der View befindet

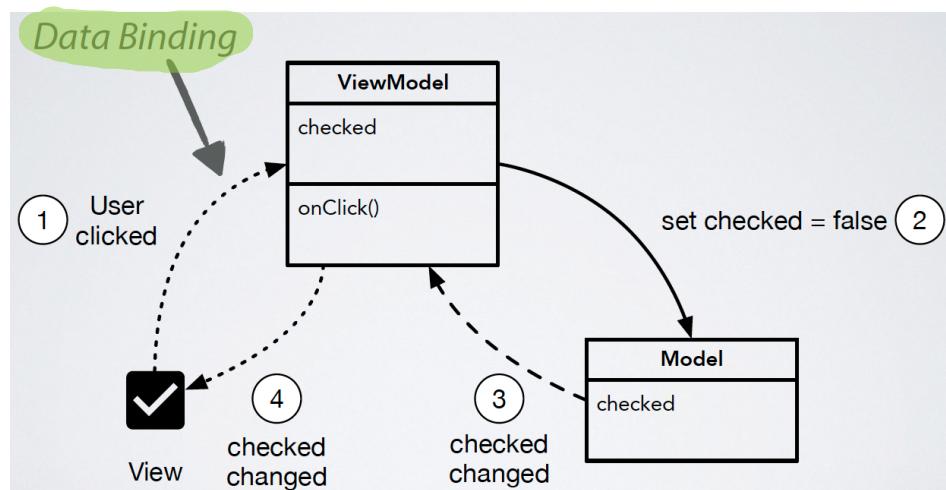


- Im Vergleich zu Supervising Controller
 - ▶ Die View enthält keine Logik
 - ▶ Kann sich selber nicht aktualisieren
 - ▶ Der Presenter enthält alle Logik für die Aktualisierung des Views
- Verwendung eines Presentation Models ist empfohlen
- Erhöht die Testbarkeit, weil sich die Logik nicht mehr in der View befindet



Model View ViewModel (MVVM)

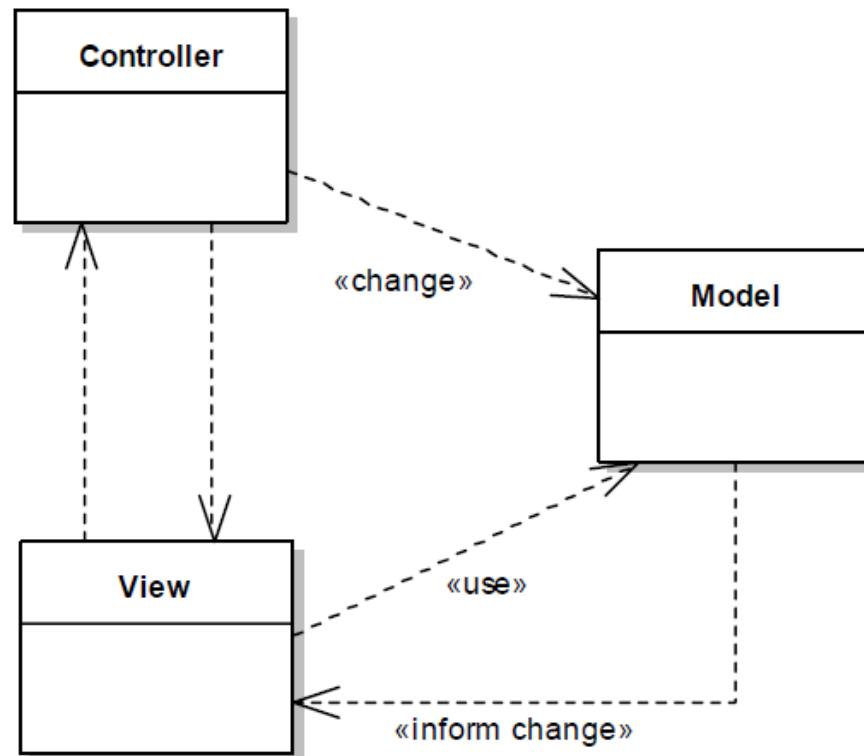
- Eng verwandt mit dem Supervising Controller
- ViewModel
 - ▶ Ersetzt den Supervising Controller
 - ▶ Enthält ein Presentation Model
 - ▶ View kommuniziert mit dem ViewModel durch DataBinding



```
private void initBindings() {  
    label.textProperty().bind(studentViewModel.labelTextProperty());  
    input.textProperty().bindBidirectional(studentViewModel.inputTextProperty());  
    button.disableProperty().bind(studentViewModel.buttonDisabledProperty());  
}
```

3.6.5. Model View Controller

- Model View Controller trennt die User-Interface-Logik in drei Rollen auf.
- Beim MVC handelt es sich um das wahrscheinlich bekannteste aller Patterns



- Was ist denn eigentlich SOA?
 - ▶ Ein neues Paradigma?
 - ▶ Eine technische Architektur?
 - ▶ Ein organisatorischer Ansatz?
 - ▶ Eine alternative Unternehmensarchitektur?
 - ▶ Alter Wein in neuen Schläuchen?

Serviceorientierte Architektur

- Der Management-Consultant:

- ▶ “Ein Paradigma zur besseren **Ausrichtung von IT und Business** mit dem Ziel, **Geschäftsprozesse** zu flexibilisieren.”

- Der (IT-)Enterprise-Architekt:

- ▶ “Ein **Architekturansatz** zur Effizienzsteigerung der IT durch einen Wechsel von einer **Anwendungs- zu einer Servicelandschaft**.”

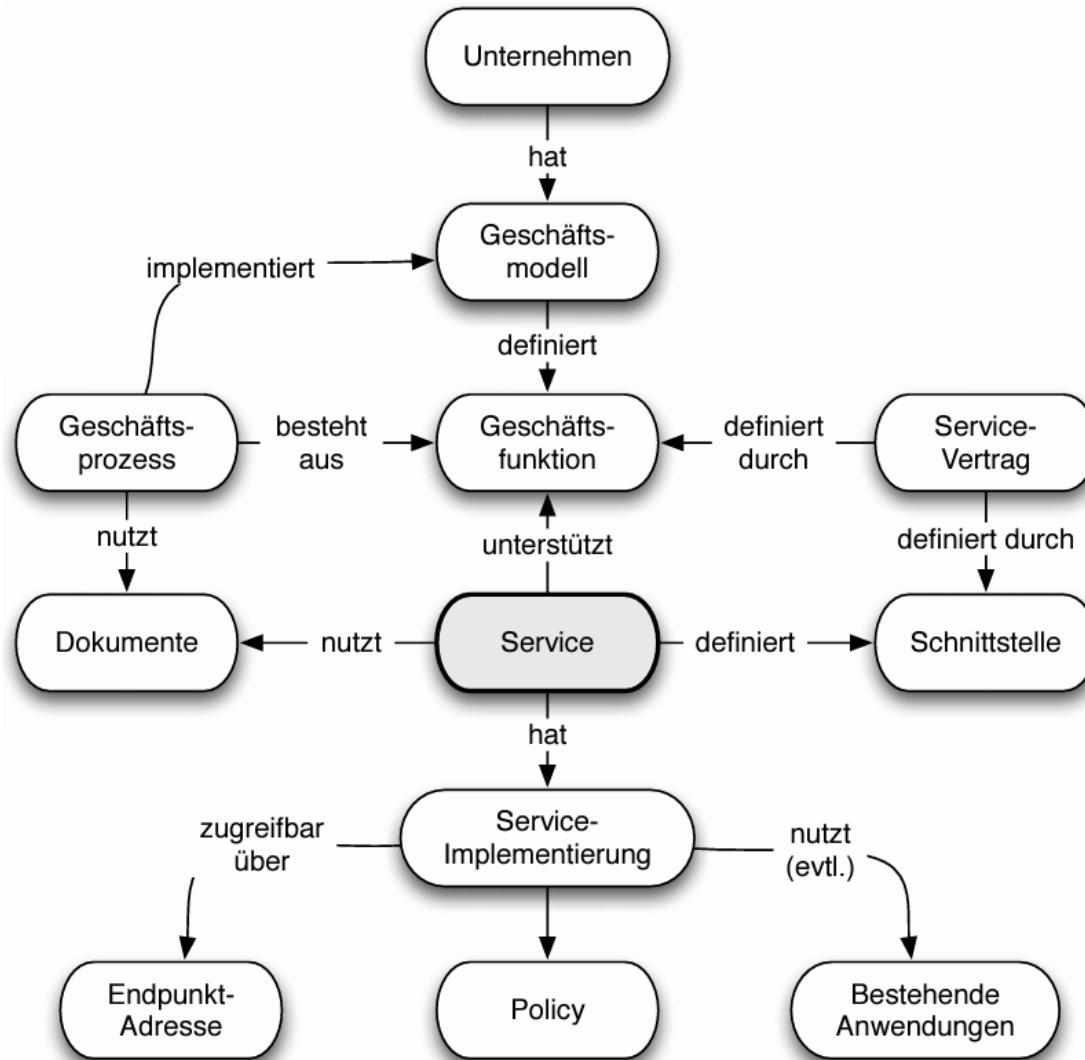
- Der Produktanbieter:

- ▶ “Ein **technischer Architekturansatz**, bei dem eine intelligente **Mediationsinfrastruktur** (ein ESB – Enterprise Service Bus) zwischen Kommunikationspartnern vermittelt.”

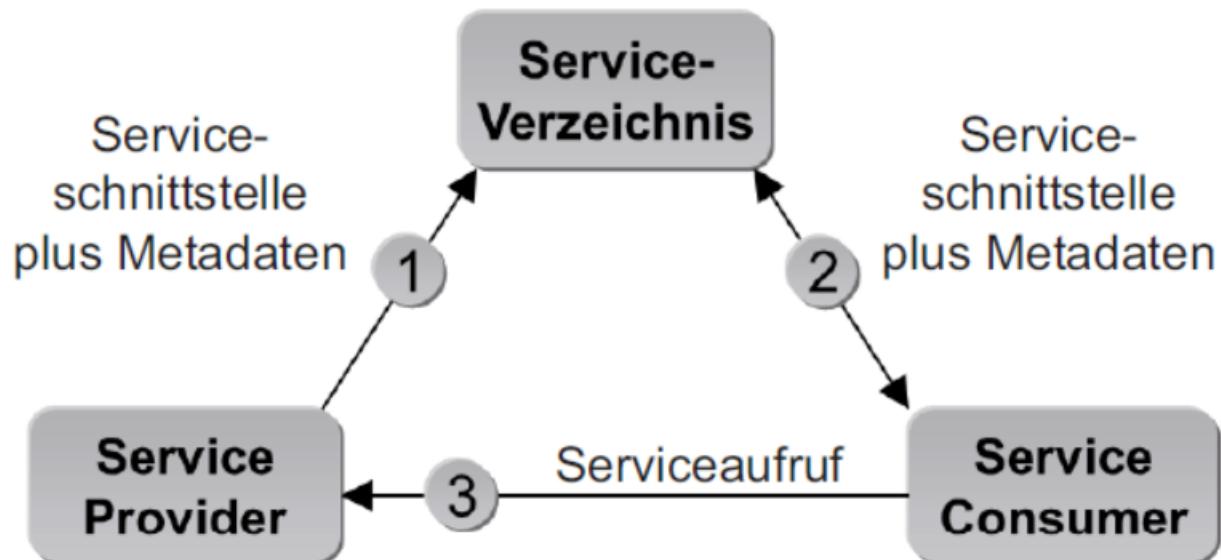
- Der Technologe:

- ▶ “Eine technische Architektur mit einer klaren Trennung von **Schnittstelle und Implementierung** und formalen, in einer zentralen Registry registrierten Schnittstellenbeschreibungen.”
 - ▶ “Eine standardisierte technische Architektur, basierend auf XML, SOAP, WSDL, UDDI, und **div. WS-*-Standards**.”

Was ist SOA

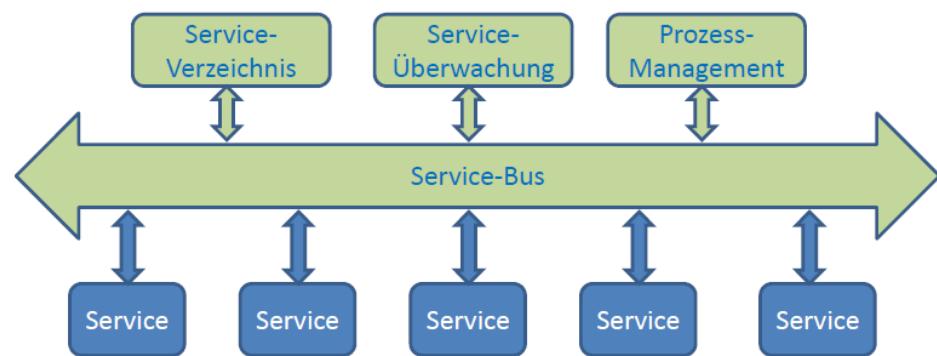


- SOA definiert drei Rollen: **Dienstanbieter, Verzeichnisdienst, Dienstnutzer**
- Zusätzlich optional: Serviceüberwacher und Prozessmanager



Das Servicedreieck: Consumer, Provider, Verzeichnis

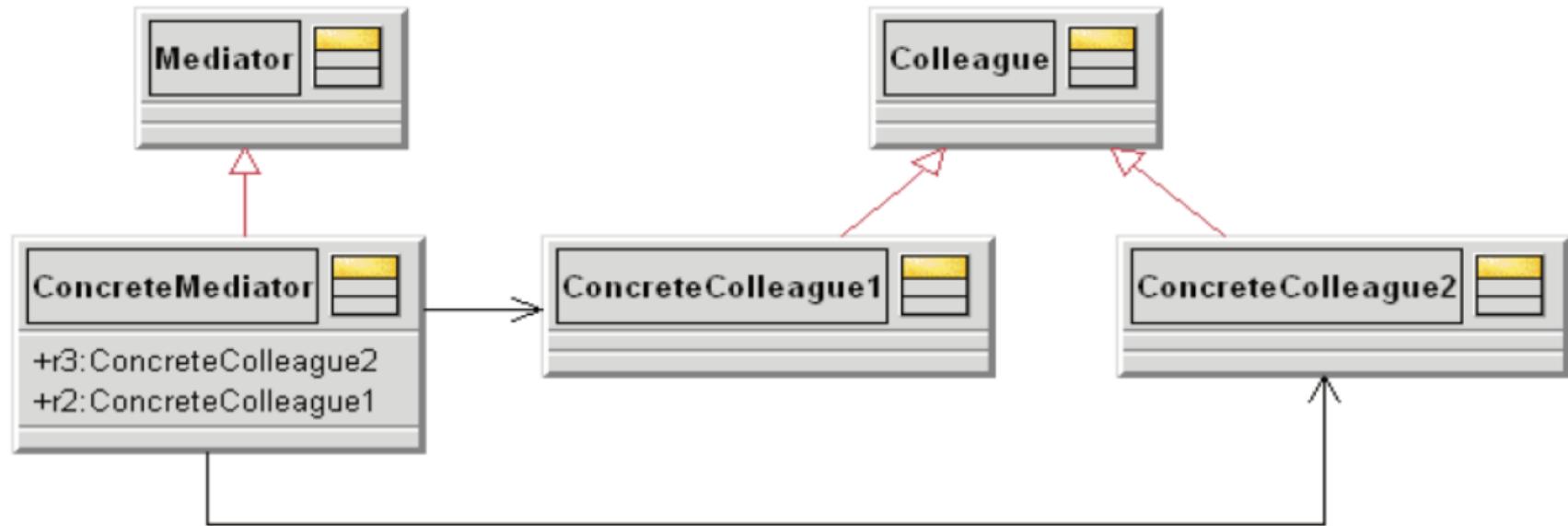
- Nachrichtendienst (Service-Bus)
 - ▶ *Flaschenpostversand*
 - ▶ Versand und sichere Zustellung von Nachrichten in einem Standardformat
- Serviceverzeichnis
 - ▶ Registrieren, Suchen, Binden
- Service-Überwachung:
 - ▶ Kompatibilität neuer Services
 - ▶ Einhaltung des Level of Service (Verfügbarkeit und Performance)
- Prozess-Management zur Orchestrierung (optional)
 - ▶ Auch: **Workflow-Engine** → B. Camunda
 - ▶ Abstrakte Implementierung der Geschäftsprozesse aus Services
 - ▶ Zumeist BPEL-Implementierung



3.6.8. Mediator

- **Der Mediator ist ein Vermittler**

- ▶ Steuert das kooperative Verhalten von Objekten
- ▶ Die Objekte kommunizieren über den Mediator
- ▶ Die teilnehmenden Komponenten nennt man Colleagues
- ▶ Die Colleagues kennen den Mediator. Sie sind als Interaktionspartner registriert



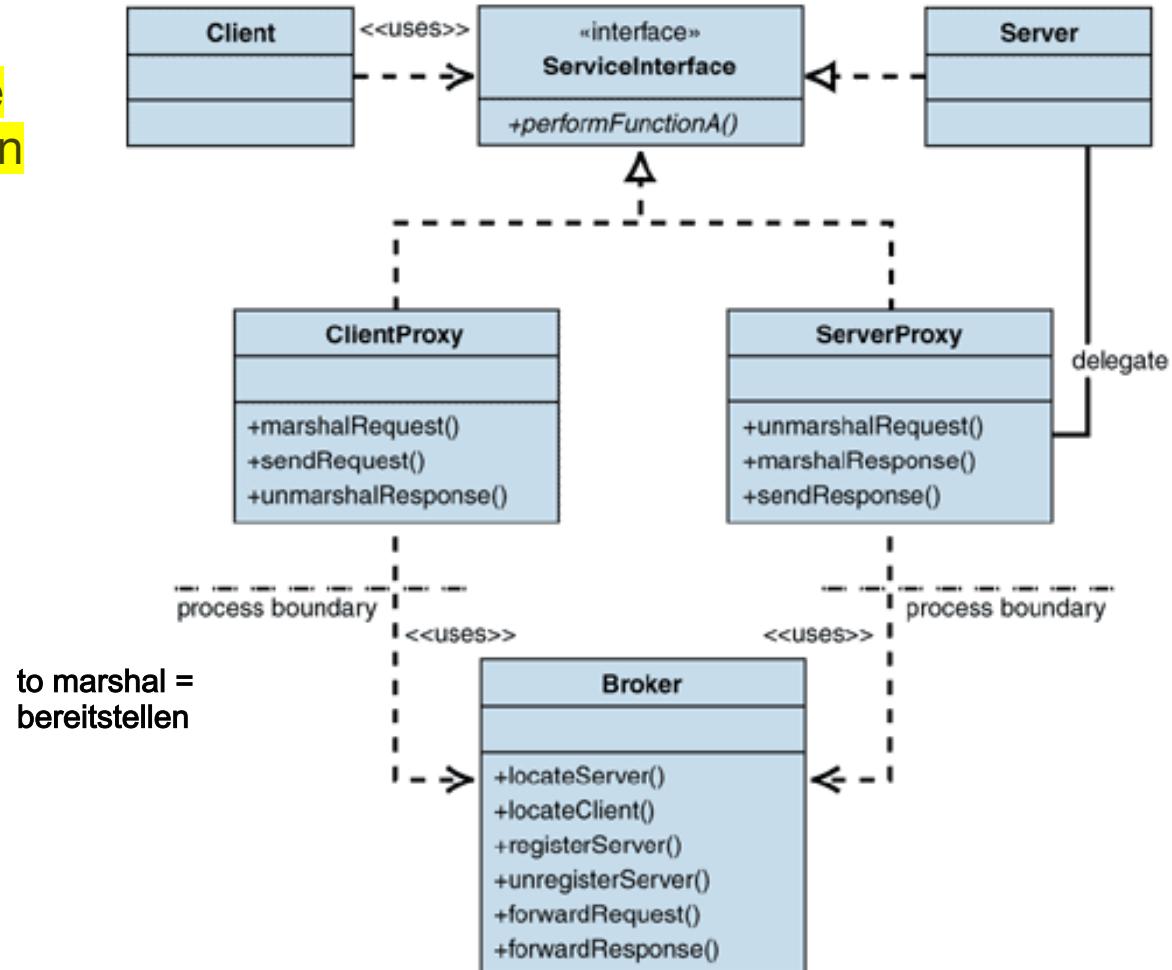
3.6.9 Broker

- Einsatz in verteilten Systemen

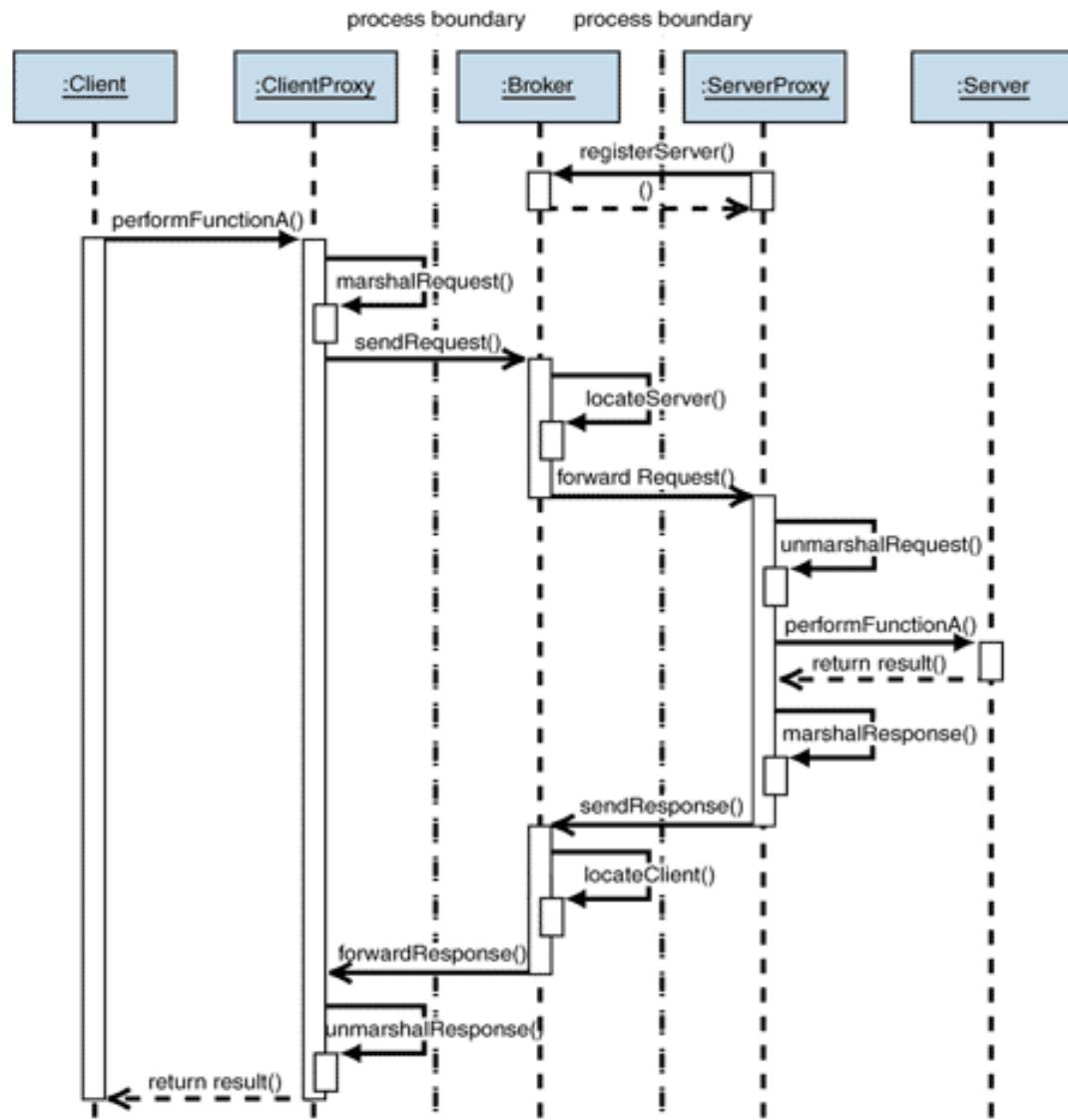
- ▶ Vermittlungsstelle für die Kommunikation zwischen Client und Server

- ▶ Für jeden Service muss auf dem Server ein Service-Interface implementiert werden

- ▶ Beim Zugriff der Clients auf einen bestimmten Service, kennt der Broker den Server, welcher den Service anbietet.



- Server registriert sich beim Broker
- Broker leitet die Anfrage an der entsprechenden Server



- **Creational Pattern**

- ▶ Abstract Factory
- ▶ Builder
- ▶ Factory Method
- ▶ Prototype
- ▶ Singleton

- **Structural Pattern**

- ▶ **Adapter (3.7.1.)**
- ▶ Composite
- ▶ **Facade (3.7.5.)**
- ▶ **Bridge (3.7.6.)**
- ▶ **Decorator (3.7.3.)**
- ▶ Flightweih
- ▶ **Proxy (3.7.4.)**

- **Behavioral Pattern**

- ▶ Command
- ▶ Iterator
- ▶ Mediator
- ▶ Memento
- ▶ **Observer (3.7.2.)**
- ▶ **State (3.7.7.)**
- ▶ Strategy
- ▶ Template Method

- **ORM Patterns**

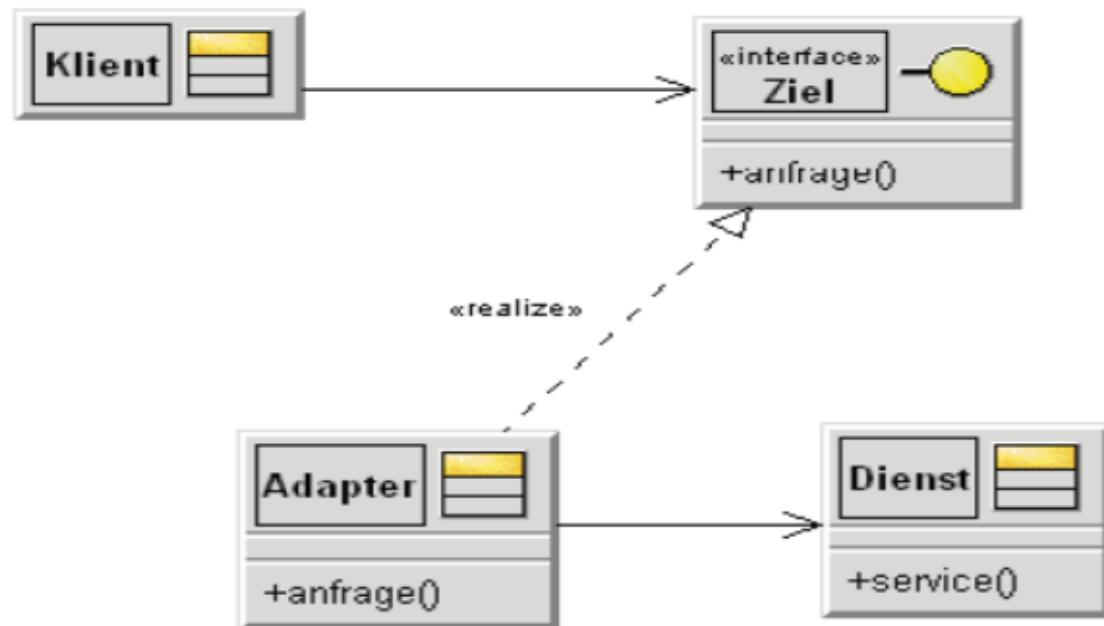
- ▶ Ca 17 Stück

- **Weitere Patterns**

- ▶ **Dependency Injection (3.7.8.)**
- ▶ Ca 17 Stück

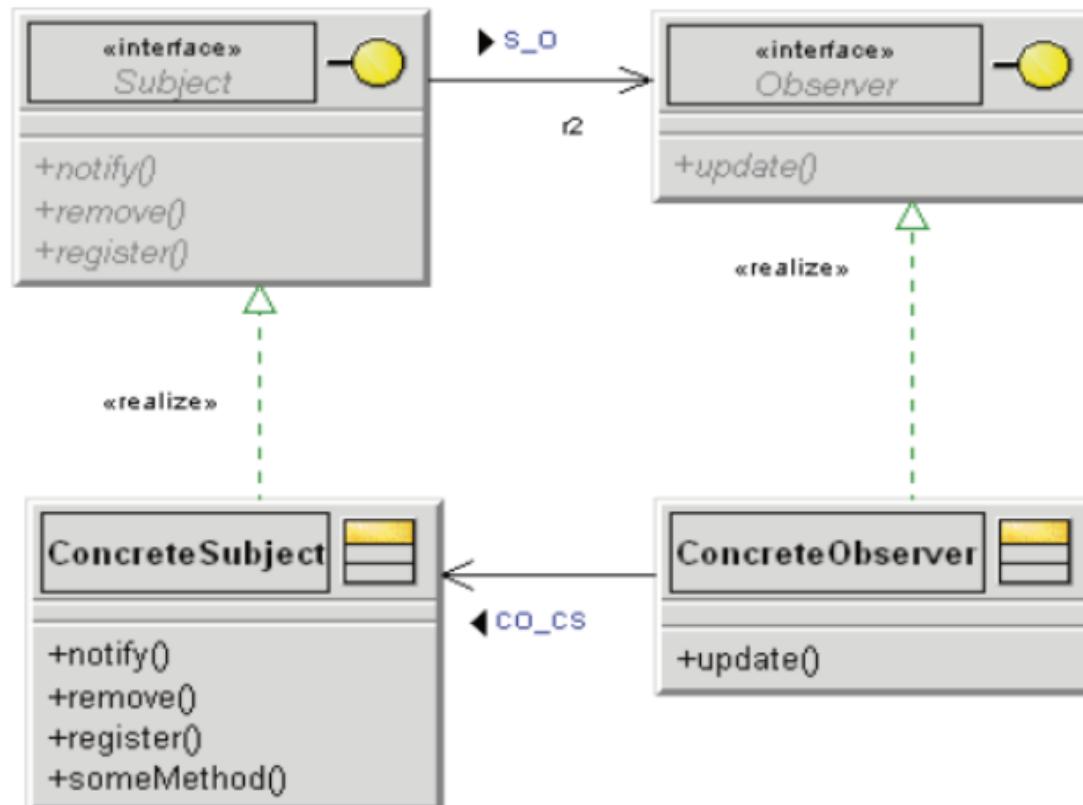
3.7.1. Adapter

- Adapter auch Wrapper passt eine inkompatible Schnittstelle an



3.7.2. Observer

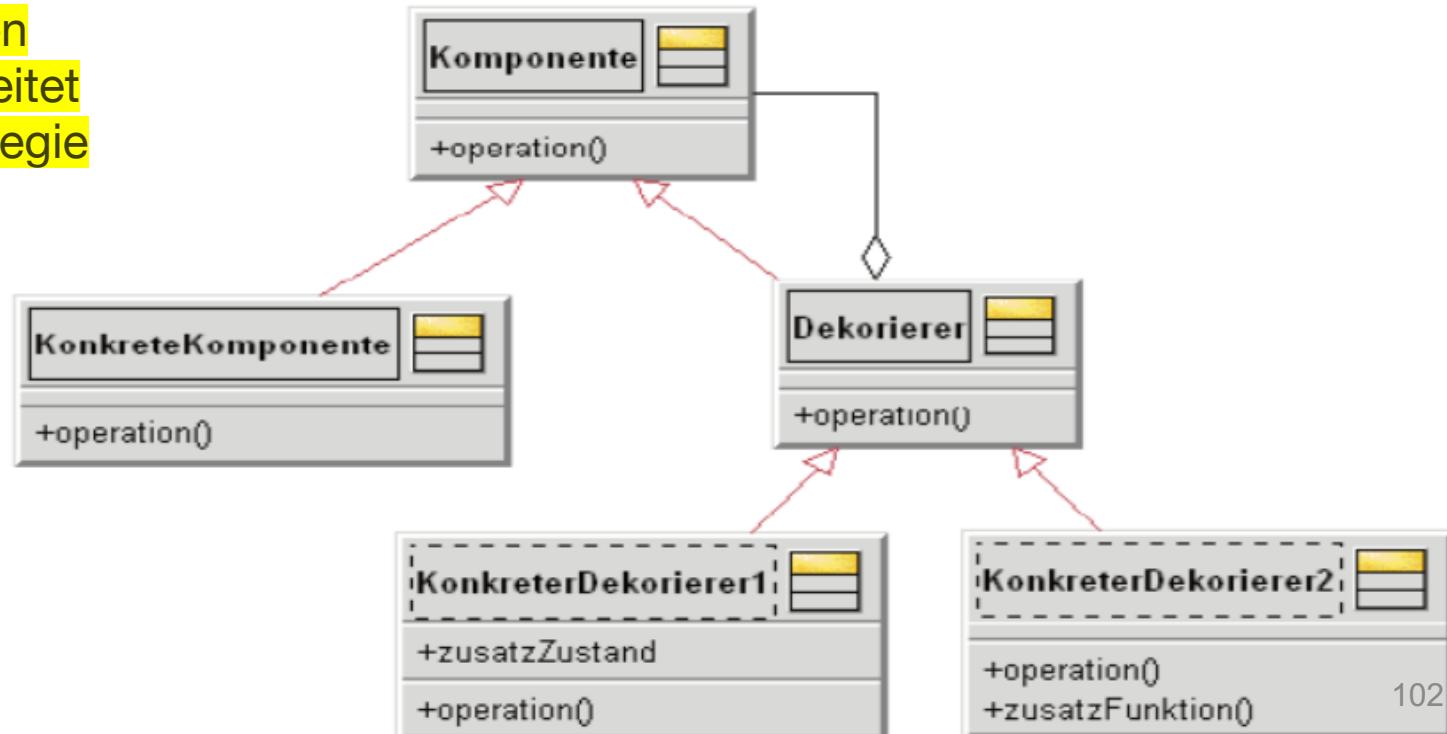
- Eine Komponenten kann andere Komponenten benachrichtigen ohne zu wissen wer die anderen Komponenten sind bzw wie viele es sind.
- Beobachter implementieren das I/F Observer



4.7.3. Decorator

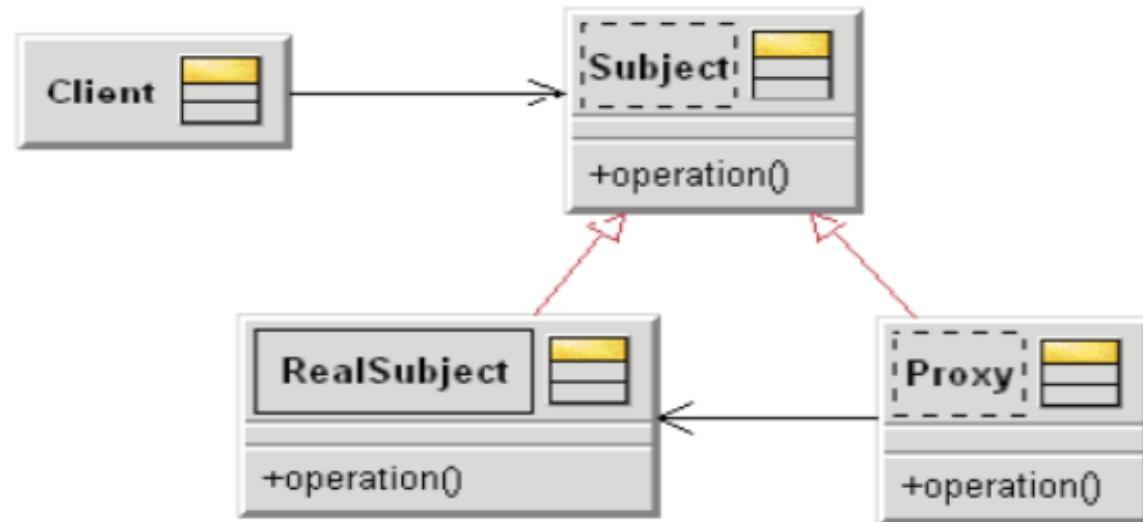
- Ein Decorator oder Dekorierer:

- ▶ fügt einer Komponente dynamisch und transparent neue Funktionalität hinzu, ohne die Komponenten selbst zu erweitern.
- ▶ Eine Instanz eines Dekorierers wird vor die zu dekorierende Klasse geschaltet und hat dieselbe Schnittstelle wie die zu dekorierende Klasse.
- ▶ Aufrufe werden nun weitergeleitet oder in Eigenregie verarbeitet.



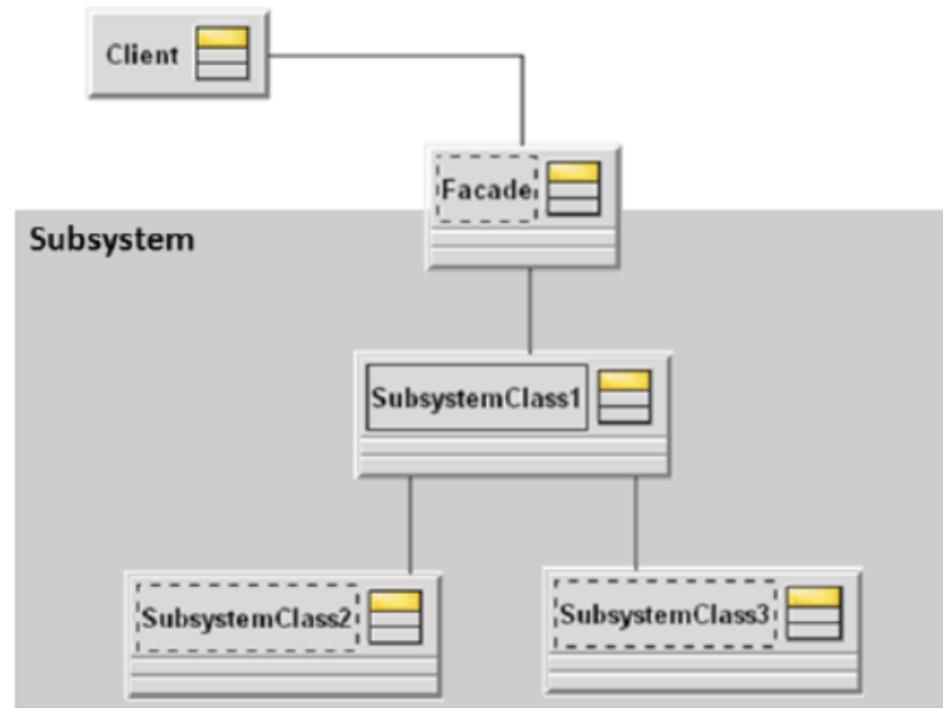
3.7.4. Proxy

- Ein Client muss auf die Operationen einer Instanz einer bestimmten Klasse zugreifen.
 - ▶ Direkter Zugriff ist nicht möglich (bei verteilten Systemen)
 - ▶ Der Client kommuniziert mit einem Stellvertreter (engl. Proxy) statt mit einer Instanz der eigentlichen Klasse.
 - ▶ Der Proxy bietet dieselbe Schnittstelle an wie die Instanzen der Klasse



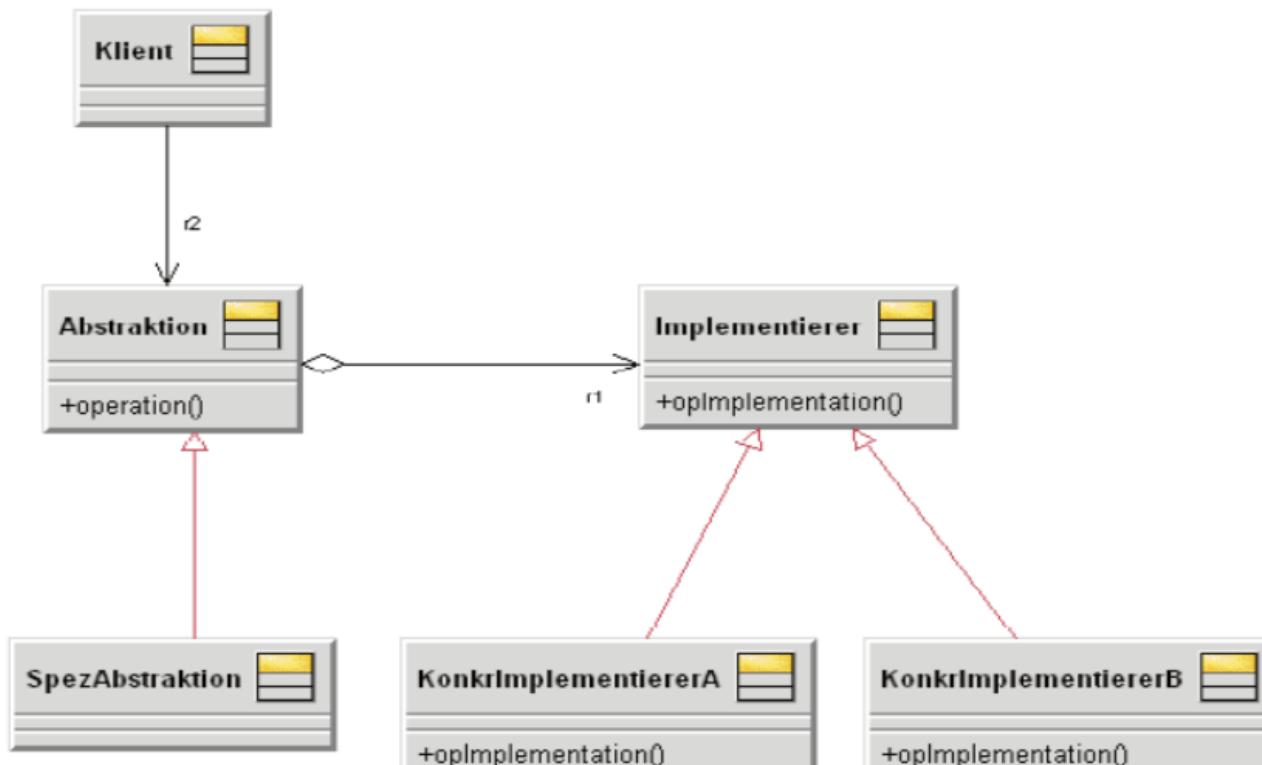
3.7.5. Fassade

- Eine Fassade stellt eine weitere Möglichkeit dar, **Abhängigkeiten** zwischen unterschiedlichen Systemkomponenten zu **verringern**.
 - ▶ Mithilfe einer Fassade werden die internen Komponenten eines Subsystems nach aussen unsichtbar.
 - ▶ Sie stellt eine vereinfachte Schnittstelle zu einem komplexen Teilsystem dar.



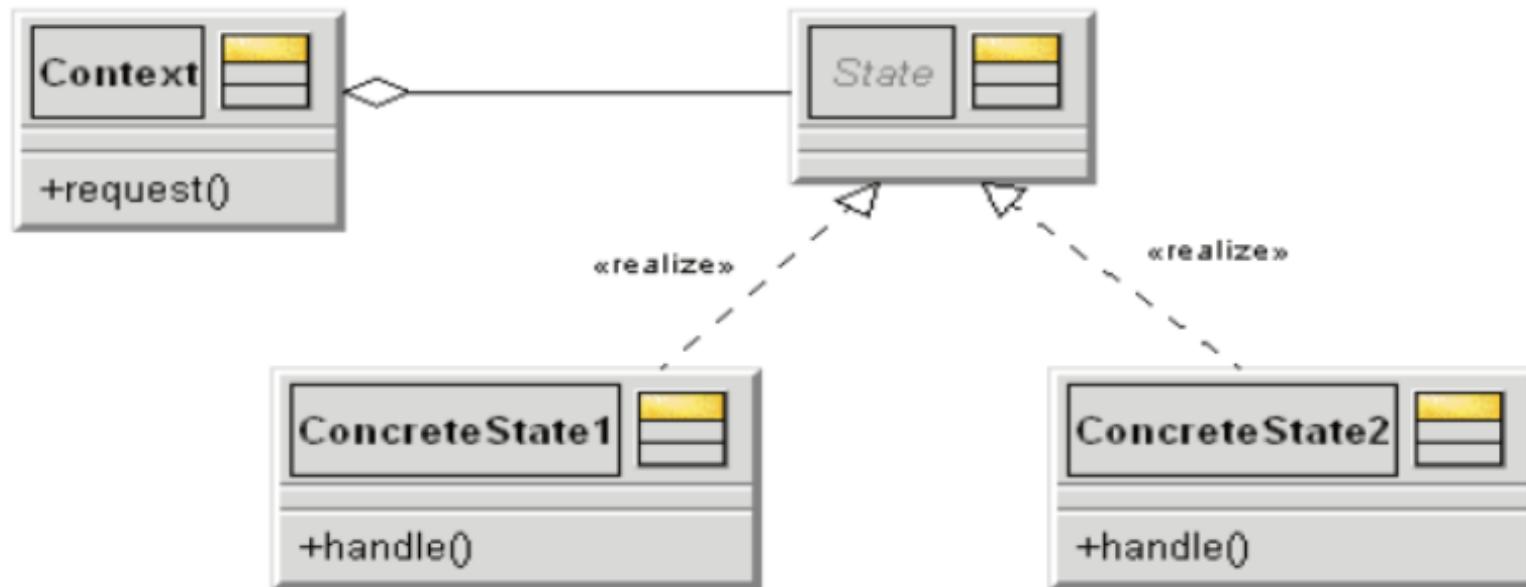
3.7.6. Brücke - Bridge

- Das Muster dient zur **Trennung der Implementierung** von ihrer **Abstraktion (Schnittstelle)**, wodurch beide unabhängig voneinander verändert werden können.
 - ▶ Abstrakte Klassen und die Implementierungen werden in zwei verschiedenen Hierarchien verwaltet



3.7.7. State

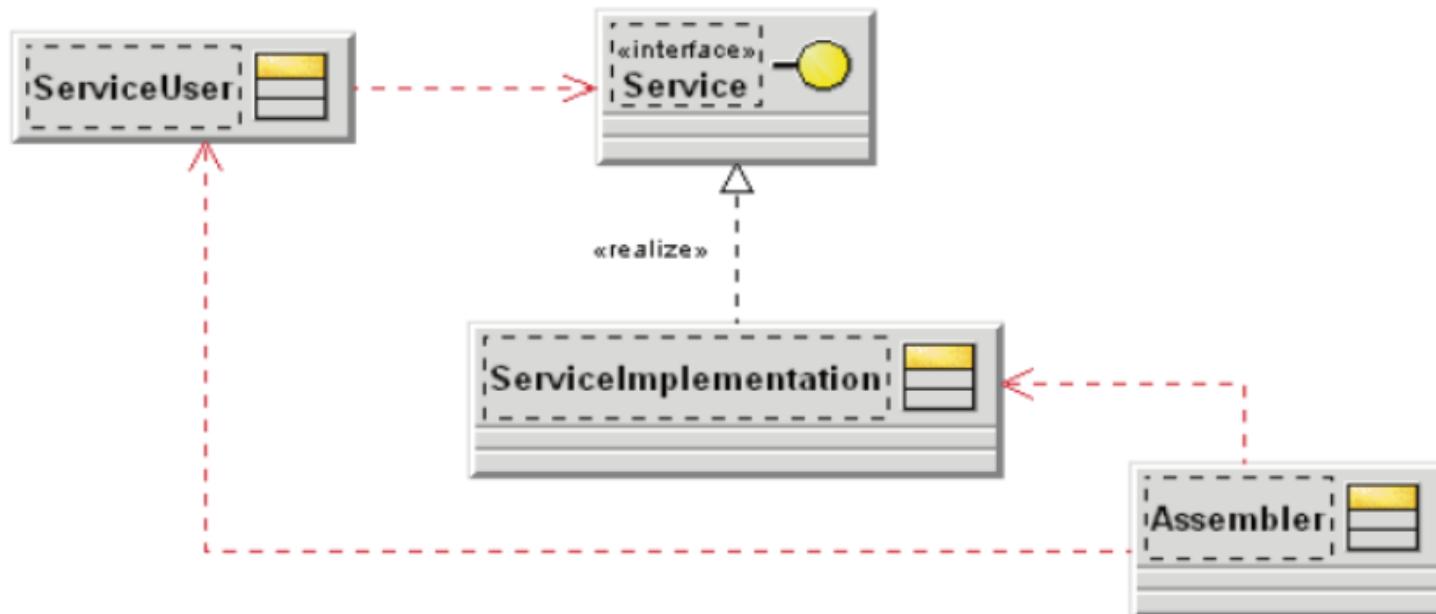
- Kapselung unterschiedlicher, zustandsabhängiger Verhaltensweisen eines Objekts.
 - ▶ das Verhalten eines Objekts ist abhängig von seinem Zustand
 - ▶ Vermeiden von Implementierung in grossen Switch-Anweisungen
 - ▶ jeder Fall der Switch-Anweisung wird in einer eigenen Klasse implementiert



3.7.8. DI – Dependency Injection

- Im objektorientierten Entwurf besteht bei der Nutzung von Schnittstellen oft das Problem, zur Laufzeit für eine abstrakte Schnittstelle eine konkrete Instanz zu beschaffen.

- ▶ Wer verwaltet den Lebenszyklus der genutzten Instanzen?
- ▶ Wer bestimmt, welche konkrete Klasse zur Laufzeit letzten Endes instanziert werden soll?



- Dieses Muster stellt zu diesem Zweck einen eigenständigen Baustein bereit, den **Assembler**.
 - ▶ Er entscheidet zur Laufzeit über die eben geschilderten Fragen.
 - ▶ Der Assembler übergibt den abhängigen Objekten die Referenzen auf konkrete Instanzen.
 - ▶ Er kann als eine Art »Universalfabrik« angesehen werden.
- Bekannte **Java-Implementierungen** für Dependency Injection sind:
 - ▶ Contexts and Dependency Injection (JSR-299)
 - ▶ Spring Framework
 - ▶ Google Guice

Fragen