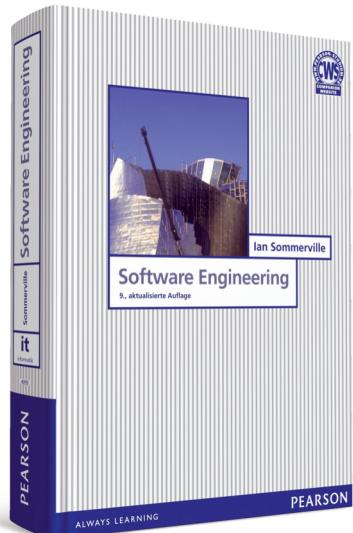


# Teil 2.1:

# Software Architektur

# Literatur

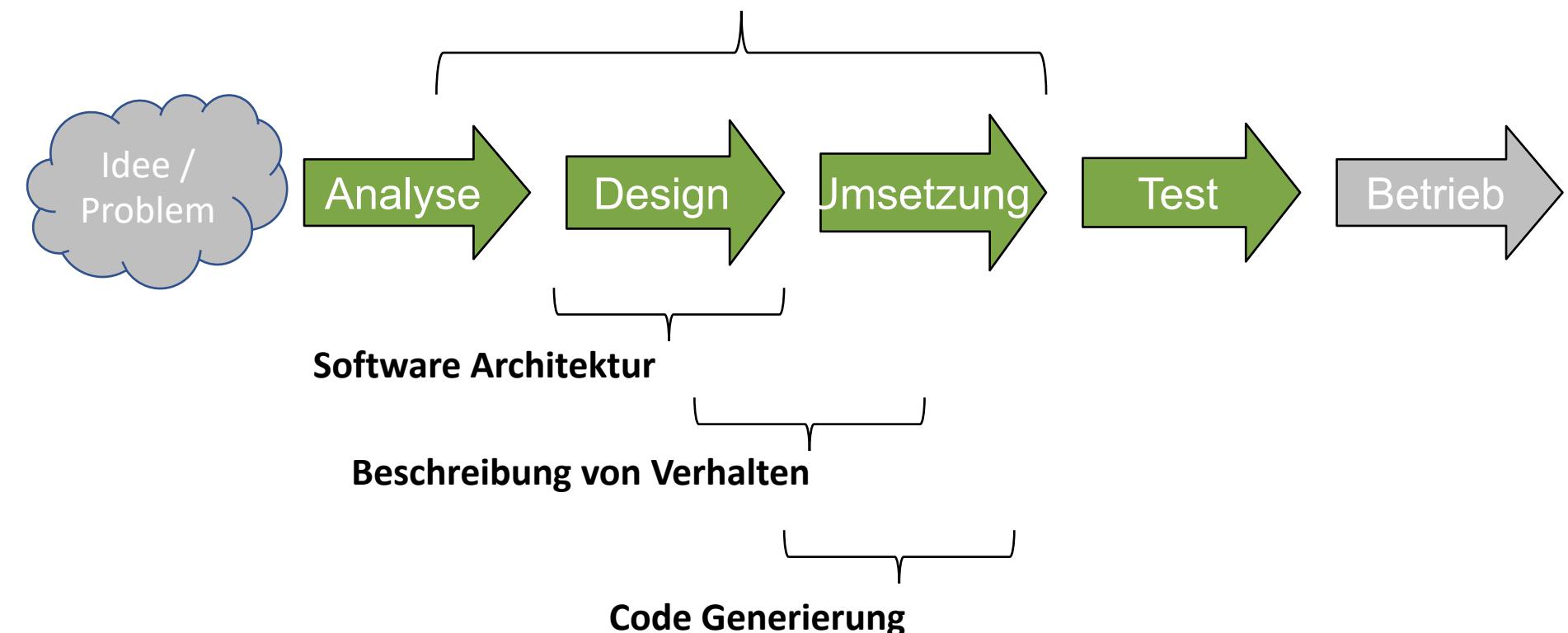


- Ian Sommerville, Software Engineering, 9. überarbeitete Ausgabe, Pearson 2012.
- Stahl und Völter, Modellgetriebene Softwareentwicklung, dpunkt.verlag GmbH, 2005.
- The Unified Modeling Language; Booch, Rumbaugh, Jacobsen; Addison-Wesley; 2001

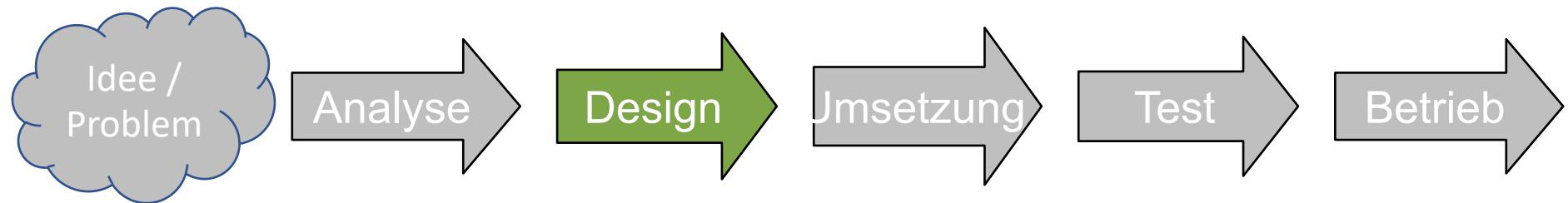


# Einordnung

**Modellgetriebene Softwareentwicklung:**  
Konsistenter Ansatz von Analyse bis Umsetzung



# Softwarearchitekt



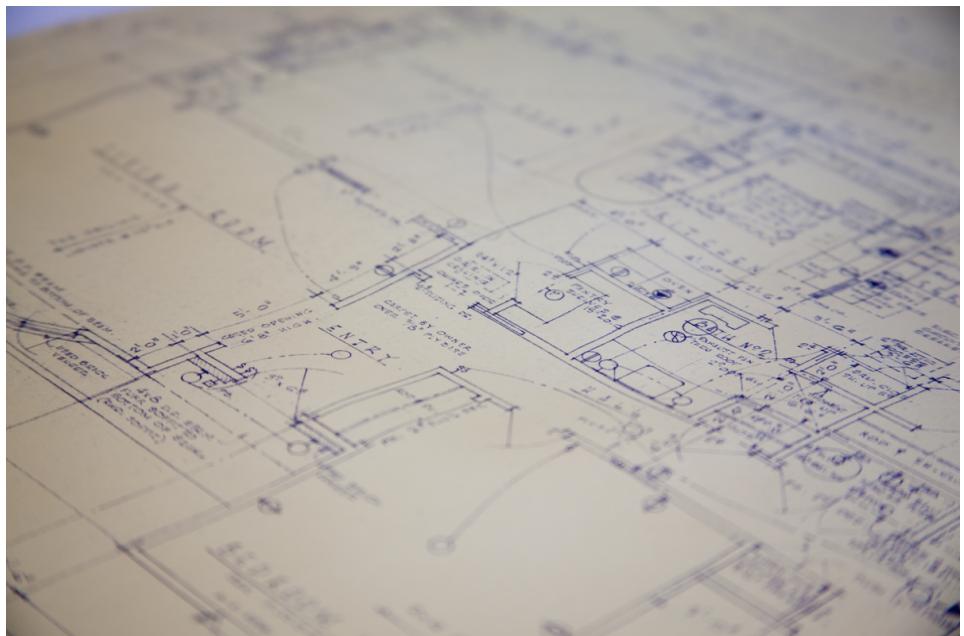
•Aufgabe des Softwarearchitekten?

## Aufgabe des Softwarearchitekten?

- Umsetzbaren Entwurf entwickeln der Anforderungen erfüllt
- Findung, Beschreibung und Begründung der für die Umsetzung gewählten technischen Ansätze
- Balance der Anforderungen erreichen
- Verständlichen Entwurf, der Entwurfsentscheidungen nachvollziehbar macht

# Analogie: Bauplan

- Abstraktion des Systems mit reduzierten Details
- Dokument = Menge der Entwurfsentscheidungen?  
(Medvidovic 2001)
- Kommunikation
- Vertrag
- Rahmenwerk für Entwickler



Quelle: Flickr

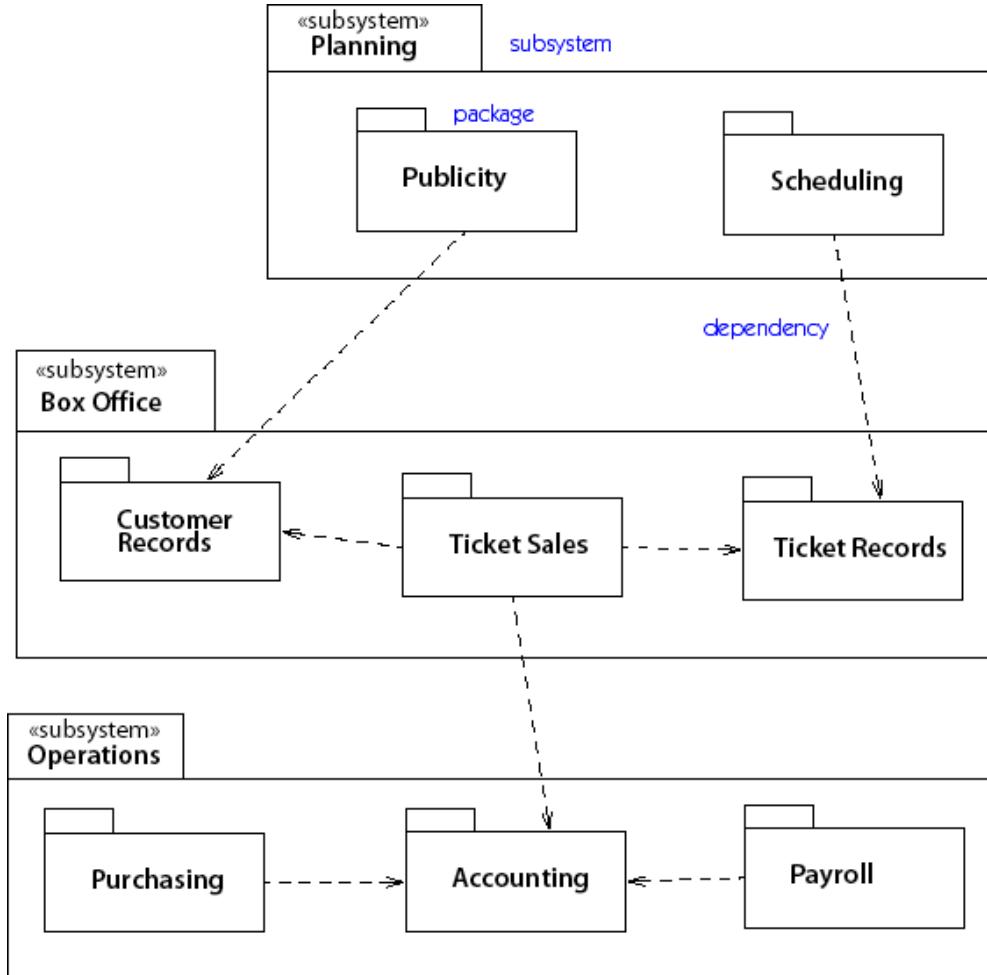
# Bausteine

## Subsysteme

- Ersetzbare **Teile mit** wohldefinierten **Schnittstellen** (Interfaces), welche den Zustand und Verhalten von beinhalteten Komponenten oder Klassen **kapseln**.
- UML Darstellungsmöglichkeiten:
  - Komponenten-Diagramme (*nicht Teil dieser Vorlesung*)
  - Paket-Diagramme
  - Verteilungs-Diagramme

# UML Paket Diagramm

Hier nur:  
 Pakete  
 Hierarchien  
 Abhängigkeiten



Nützlich um Abhängigkeiten von  
 Hauptelementen im System auszudrücken

# Agenda

- Zum Begriff Architektur
- Architekturnmuster
- Qualität von Architektur
- Weitere Muster

# Architekturstile

An **architectural style** defines a **family of systems** in terms of a pattern of structural organization. More specifically, an architectural style defines a vocabulary of **components** and **connector** types, and a set of **constraints** on how they can be combined.

- Shaw and Garlan

# Architekturstile

- Objekt-orientiert
- Client-Server, Object-Broker, Peer to peer
- Pipe and filter
- Ereignis-orientiert – publish/subscribe
- Schichten – drei-/vier-Schicht Architektur
- Repository – blackboard, model/view/controller (MVC)
- Process control

## Beispiel

- Abstrakte Datentypen (Module)

## Wichtige Eigenschaften

- Kapselung (interne Datendarstellung von außen nicht sichtbar)
- Kann Problem in Menge von interagierenden Agenten zerlegen
- Kann multi-threaded oder single-threaded sein

## Nachteil

- Objekte müssen Identität anderer Objekte kennen, um zu interagieren

# Client-Server Architektur

Eine **Client-Server Architektur** verteilt Anwendungslogik und -funktionalität auf eine Menge von Clients und Server Subsysteme, welche u.U. auf verschiedenen Maschinen ausgeführt werden und über Netzwerk kommunizieren.

- Wichtige Eigenschaften
  - Verteilung der Daten einfach
  - Client muss andere Clients nicht kennen
  - Unterteilt ein System in handhabbare Teile
  - Unabhängiger Kontrollfluss
  - Server meistens verantwortlich für Persistenz und Konsistenz von Daten

# Client-Server Architektur

## Nachteil

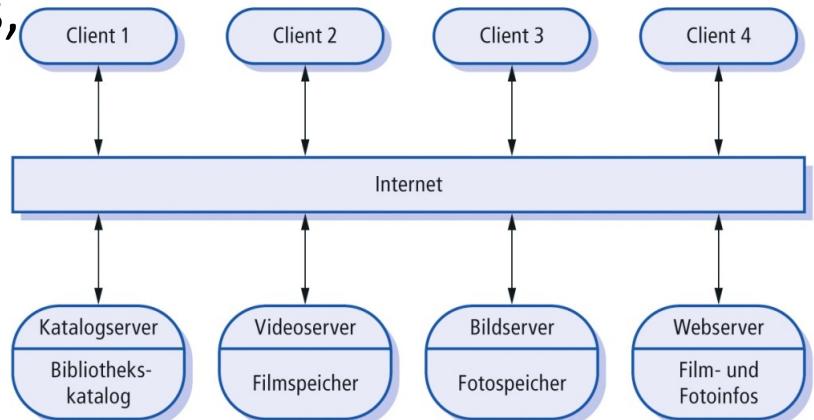
- Client muss Identität des Servers kennen (zentrales Verzeichnis)
- Kein gemeinsames Datenmodell
- Redundantes Management der Server

Client/Server kommunizieren z.B. RPC, SOAP, REST

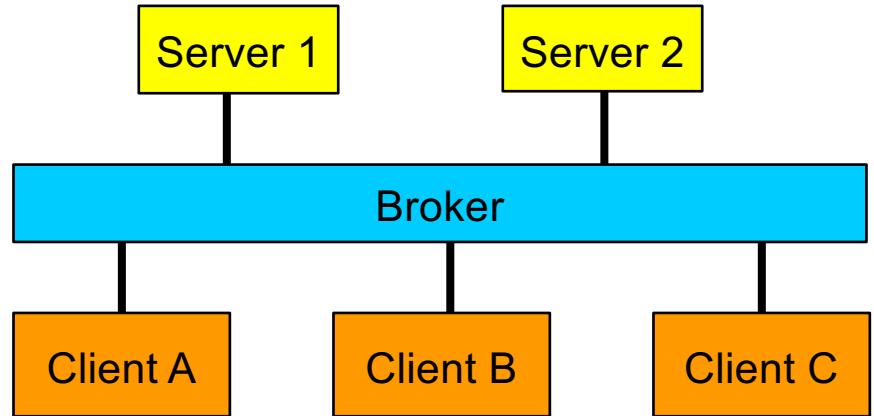
Verteilte Systeme, z.B. Web Services,  
Systeme mit zentraler DB

## Varianten

- Fette Clients mit eigenen Diensten
- Leichte Clients ohne eigene Dienste



# Object Broker



## Wichtige Eigenschaften

- Broker als Indirektion zwischen Client und Server
- Clients müssen Server nicht mehr kennen
- Mehrere Broker und mehrere Server möglich

## Nachteile:

- Broker sind Engpässe und Single-Point-of-Failure
- Verminderte Performanz durch Indirektion

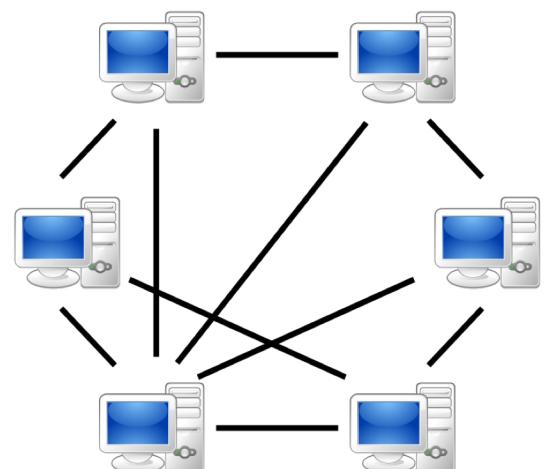
# Peer-to-Peer

## Wichtige Eigenschaften

- Findet Peers durch Server oder Broadcast
- Kommuniziert nachher nur mit Peers
- Reduziert Engpässe und robust gegenüber Peer Ausfällen

## Nachteile

- Server kann Engpass werden
- Peers haben unvollständige Sicht – Synchronisation möglich



Quelle: [wikimedia.org](https://commons.wikimedia.org)

# Pipe-and-Filter

## Beispiele

- Unix-Pipes
- Compiler Kette: Lexer->Parser->Semantische Analyse->  
...
- Signal Prozessoren

## Wichtige Eigenschaften

- Filter müssen nichts über verbundene Elemente wissen
- Filter können parallel implementiert sein
- Systemverhalten = Komposition der Filter
  - spezielle Analysen wie Durchsatz oder Deadlocks möglich



# Ereignis-basierte Architektur

Komponenten in **Ereignis-basierte Architektur** liefern Dienste als Reaktion auf externe Ereignisse von anderen Komponenten.

- Beispiele
  - Debugger (lauschen nach speziellen Breakpoints)
  - DBMS (für Integritätschecks)
  - Graphische Benutzerschnittstellen
- Im **Broadcast Modell** werden Ereignisse an alle Subsysteme gesendet. Jedes Subsystem kann Ereignis behandeln.
- Im **Interrupt-Modell** wird Echtzeit-Interrupt von einem Handler aufgefangen und an andere Komponente weitergeleitet.

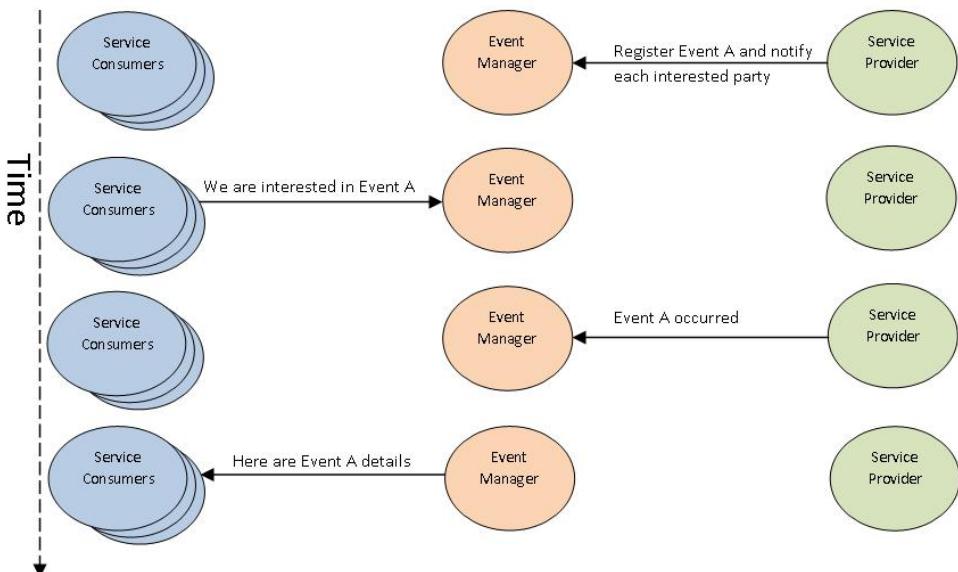
# Ereignis-basierte Architektur

## Wichtige Eigenschaften

- Ereignisquelle muss Ereignissenken nicht kennen (impliziter Aufruf)
- Unterstützt Wiederverwendung und Evolution

## Nachteil

- Komponenten haben keine Kontrolle von Berechnungsreihenfolge



Quelle: wikimedia.org

# Schichten Architektur

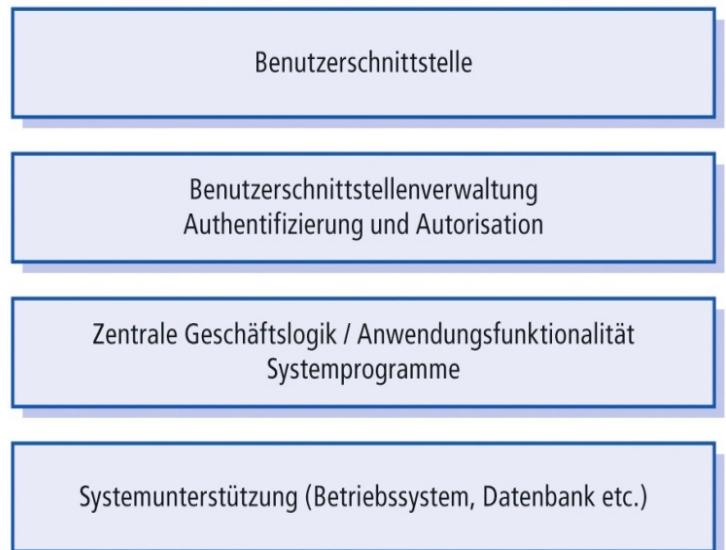
Schichtenarchitektur organisiert System in Menge von Schichten, wobei jede Schicht der darüber liegenden Schicht Funktionalität zur Verfügung stellt.

## Beispiele

- Betriebssysteme
- Kommunikationsprotokolle

## Wichtige Eigenschaften

- Normale Schichten beschränkt, da Element nur:
  - andere Elemente in gleichen Schicht oder
  - Elemente aus darunterliegenden Schicht sehen
- Unterstützen wachsende Abstraktion während Entwurf



Layer Diagramm

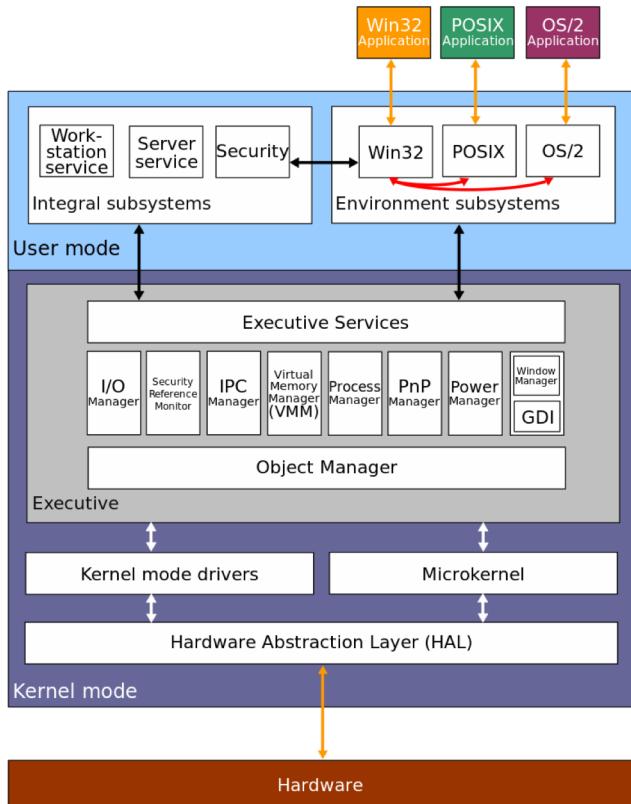
# Schichten Architektur

- Erlauben Erweiterungen (additive Funktionalität) und Wiederverwendbarkeit
- Möglichkeit Standardschicht Schnittstellen zu definieren
- Callbacks zur Kommunikation

## Nachteil

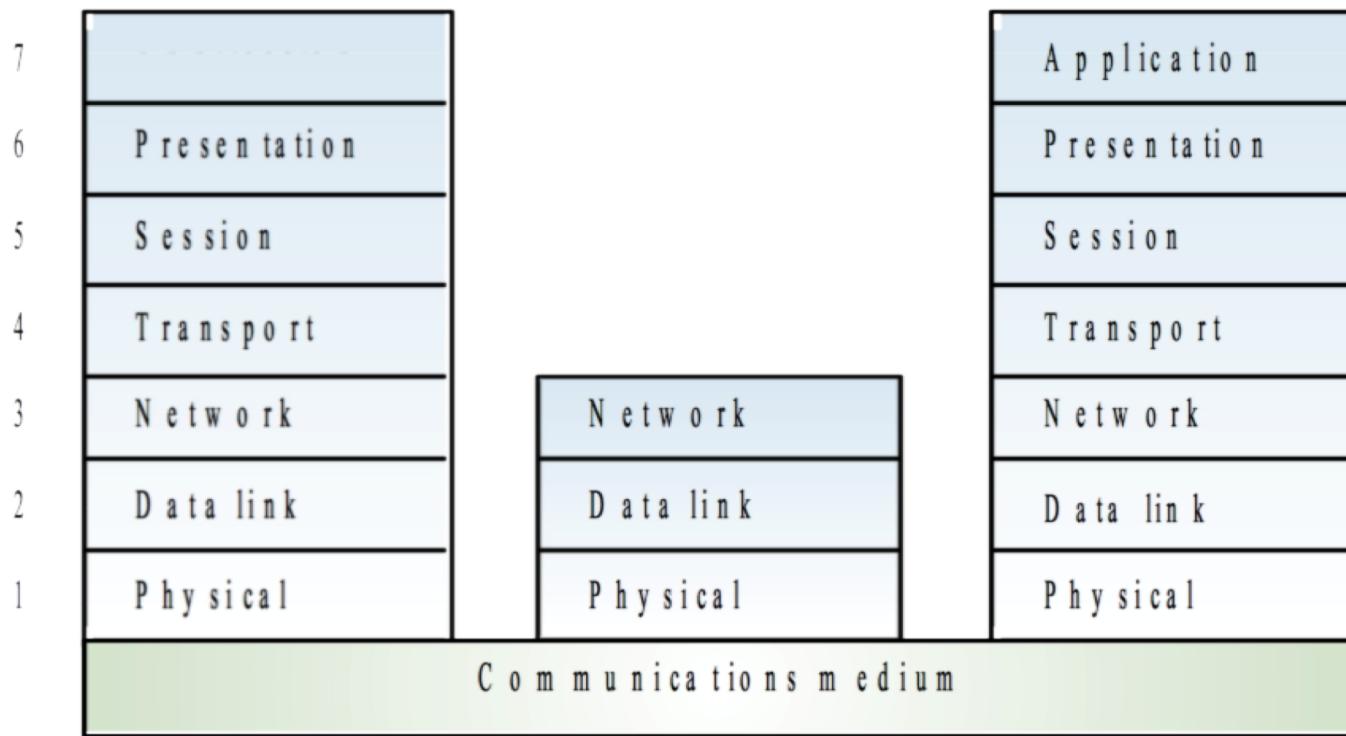
- Schwierig saubere Schichttrennungen zu definieren

Beispiel: MS Windows 2000 OS



Quelle: MS Windows 2000 OS, [wikimedia.org](https://commons.wikimedia.org)

# Beispiel: ISO/OSI



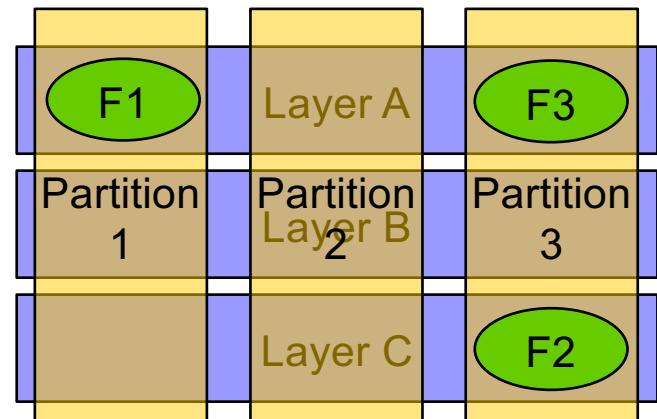
# Schichten vs. Partitionen

## Schichten

- Hierarchische Dekomposition → Baum von Subsystemen
- „Horizontale“ Unterteilung
- Offen vs. geschlossene Schichten

## Zerlegung (Partition)

- „Vertikale“ Unterteilung:  
Jede Teilmenge behandelt  
eine Funktion
- Extremform der Entkoppelung:  
semi-unabhängig



## Geschlossene Architektur

- Jede Schicht verwendet ausschließlich Dienste der direkt darunterliegenden Schicht
- Minimiert Abhängigkeiten zwischen den Schichten und reduziert Auswirkungen von Veränderungen

## Offene Architektur

- Eine Ebene nutzt Dienste von darunterliegenden Schichten
- Kompakterer Code, da darunterliegende Services direkt zugreifbar sind
- Weicht Kapselung der Schichten auf und erhöht Abhängigkeiten

# Repositories

## Beispiele

- Datenbanken
- Blackboard Expertensysteme
- Programmierumgebungen

## Wichtige Eigenschaften

- Ort der Kontrolle wählbar  
(Agenten, Blackboard, beides)
- Reduziert Notwendigkeit  
komplexe Daten zu duplizieren

## Nachteil

- Blackboard wird Engpass

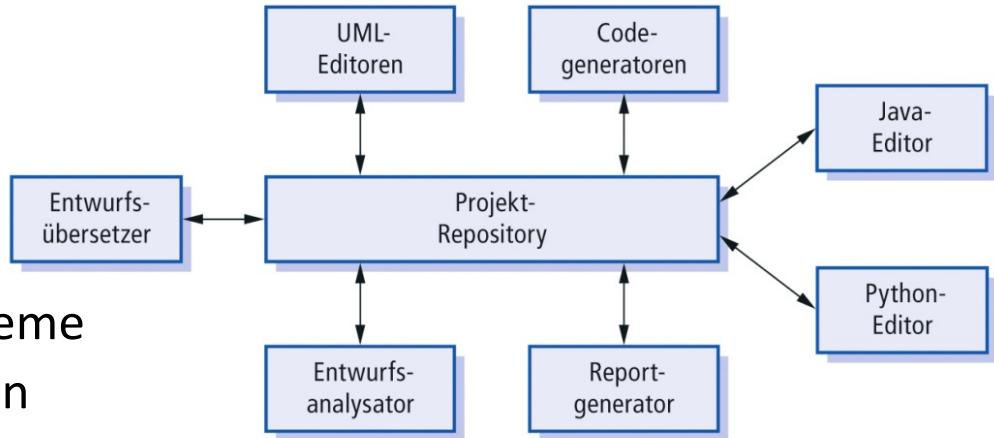
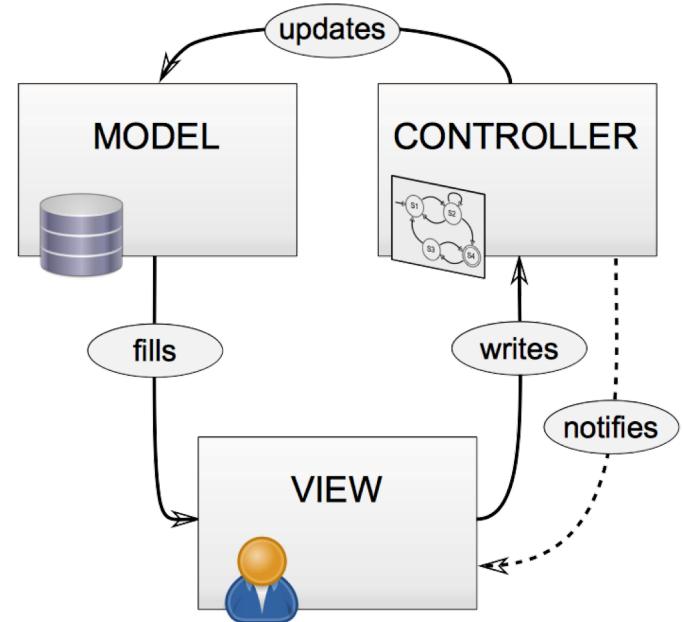


Abbildung 6.9: Eine Repository-Architektur für eine IDE.

# Blackboard/Repository

- **Blackboard Architektur (Repository)** verteilt Anwendungslogik auf unabhängige Subsysteme, verwaltet jedoch alle Daten zentral in **einzelнем, gemeinsamen Repository**
  - Subsysteme greifen und modifizieren einzelne Datenstrukturen
  - Effiziente Möglichkeit für komplexe, wandelbare Daten
  - Nebenläufigkeit und Datenkonsistenz
  - Kontrolle durch Subsystem oder Blackboard
  - Nachteil: Möglicher Engpass und reduzierte Modifizierbarkeit
  - Beispiele: DBs, IDE's, tuple spaces

# Model-View-Controller



## Wichtige Eigenschaften:

- Ein zentrales Modell und mehrere Sichten
- Jede Sicht hat eigenen Controller
- Controller verarbeitet Updates der Nutzer (Sicht)
- Controller ändert Modell und propagiert Änderungen an Sichten

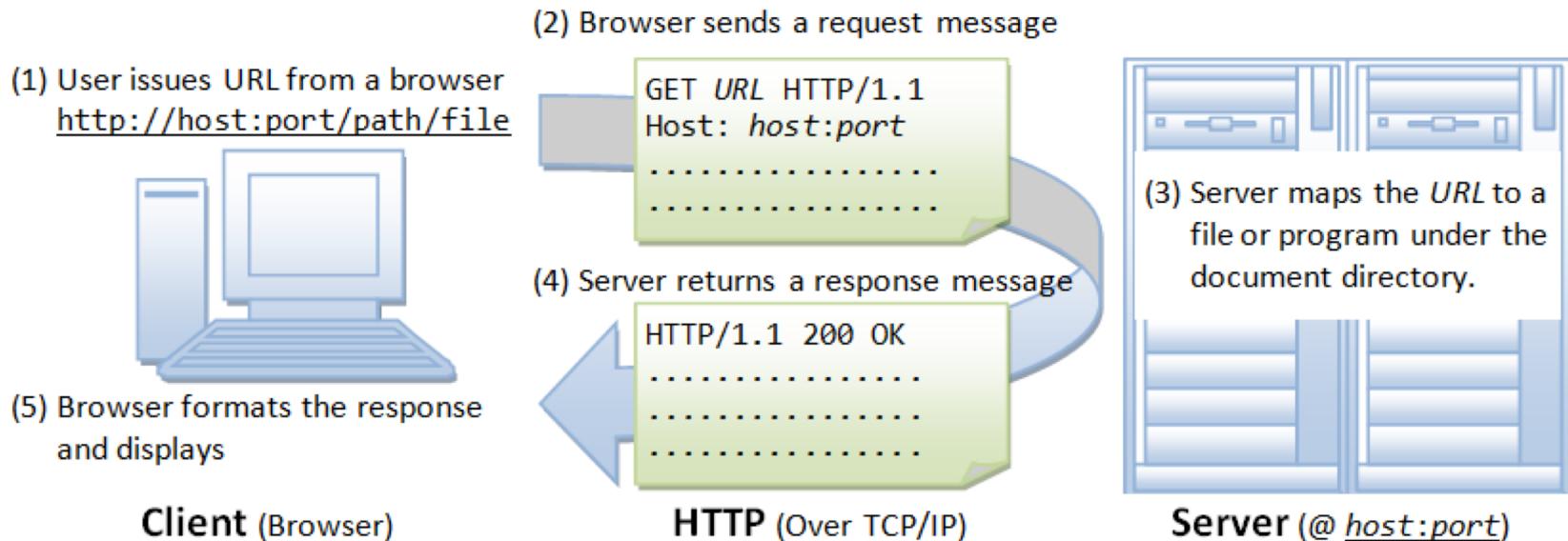
# REST

## Representational State Transfer

- REST-Paradigma von Roy Fielding 1994 als rationale Rekonstruktion des HTTP Object Models: GET, POST, ...
- Seit 2000 bekannt als REST-Architekturstil
- Prinzipien
  - Client-Server
  - Zustandslosigkeit
  - Caching
  - Einheitliche Schnittstelle
  - Mehrschichtige Systeme
  - Code on Demand (optional)



# HTTP



- Zustandsloses Protokoll
- Request Arten: GET, POST, HEAD, PUT, DELETE, ...
- Ursprünglich zur Auslieferung von Internetseiten
- Webserver kann Requests an beliebige Programme weitergeben

# RESTful Webservices

- REST-basierte Webservices über HTTP
- Beispiel:  
Google Drive

<code>get</code>	<code>GET /files/<i>fileId</i></code>	Gets a file's metadata by ID.
<code>insert</code>	<code>POST https://www.googleapis.com/upload/drive/v2/files</code> and <code>POST /files</code>	Insert a new file.
<code>patch</code>	<code>PATCH /files/<i>fileId</i></code>	Updates file metadata. This method supports <a href="#">patch semantics</a> .
<code>update</code>	<code>PUT https://www.googleapis.com/upload/drive/v2/files/<i>fileId</i></code> and <code>PUT /files/<i>fileId</i></code>	Updates file metadata and/or content.
<code>copy</code>	<code>POST /files/<i>fileId</i>/copy</code>	Creates a copy of the specified file.
<code>delete</code>	<code>DELETE /files/<i>fileId</i></code>	Permanently deletes a file by ID. Skips the trash. The currently authenticated user must own the file or be an organizer on the parent for Team Drive files.
<code>list</code>	<code>GET /files</code>	Lists the user's files.
<code>touch</code>	<code>POST /files/<i>fileId</i>/touch</code>	Set the file's updated time to the current server time.
<code>trash</code>	<code>POST /files/<i>fileId</i>/trash</code>	Moves a file to the trash. The currently authenticated user must own the file or be an organizer on the parent for Team Drive files.
<code>untrash</code>	<code>POST /files/<i>fileId</i>/untrash</code>	Restores a file from the trash.
<code>watch</code>	<code>POST /files/<i>fileId</i>/watch</code>	Start watching for changes to a file.
<code>emptyTrash</code>	<code>DELETE /files/trash</code>	Permanently deletes all of the user's trashed files.
<code>generateIds</code>	<code>GET /files/generateIds</code>	Generates a set of file IDs which can be provided in insert requests.
<code>export</code>	<code>GET /files/<i>fileId</i>/export</code>	Exports a Google Doc to the requested MIME type and returns the exported content. Please note that the exported content is limited to 10MB.

# Agenda

- Zum Begriff Architektur
- Architekturmuster
- Qualität von Architektur
- Weitere Muster

- Wie unterstützt man geringe Abhangigkeit, geringe Auswirkung von nderungen und hohe Wiederverwendbarkeit?

# Kopplung

**Kopplung (coupling)** : Menge von Beziehungen zwischen Subsystemen

**Maß**, wie stark ein Element verbunden ist, besitzt Wissen über andere Elemente oder hängt von anderen Elementen ab.

# Wann sind zwei Klassen gekoppelt?

Typische Formen der Kopplung zwischen TypX und TypY:

- TypX besitzt Attribut mit Verweis auf Instanz von TypY
- TypX Objekt ruft Dienst eines TypY Objekts auf
- TypX hat Methode welche TypY Instanz referenziert
  - z.B. Parameter, lokale Variable, Rückgabewert
- TypX ist direkte oder indirekte Subklasse von TypY
- TypY ist Schnittstelle und TypX implementiert diese Schnittstelle

# Starke Kopplung (schlecht)

Klasse mit hoher oder starker Kopplung ist auf viele andere Klassen angewiesen. Solche Klassen nicht wünschenswert und haben folgende Probleme:

- Lokale Änderungen notwendig bei Änderungen in verbundenen Klassen
- Isoliertes Verstehen der Klassen schwierig (big picture)
- Geringere Wiederverwendbarkeit, da Klassenverwendung Anwesenheit weiterer verbundener Klassen benötigt

# Lose Kopplung

Wie unterstützt man geringe Abhängigkeit, geringe Auswirkung von Änderungen und hohe Wiederverwendbarkeit?

**Lose Kopplung = wenig Beziehungen zwischen Subsystemen**

- allgemeines Entwurfsziel:
  - Unabhängigkeit
  - Instandhaltungsvermögen / Wartbarkeit

# Diskussion

- Lose Kopplung ermöglicht Entwürfe mit hoher Unabhängigkeit, was die Auswirkungen von Änderungen verringert.
- Implementierungsvererbung erhöht Kopplung – insbesondere zwischen Domänenobjekten welche von technischen Diensten erben (z.B. PersistentObject)
- Hohe Kopplung mit stabilen, globalen Objekten ist unproblematisch (z.B. Java Bibliothek java.util)

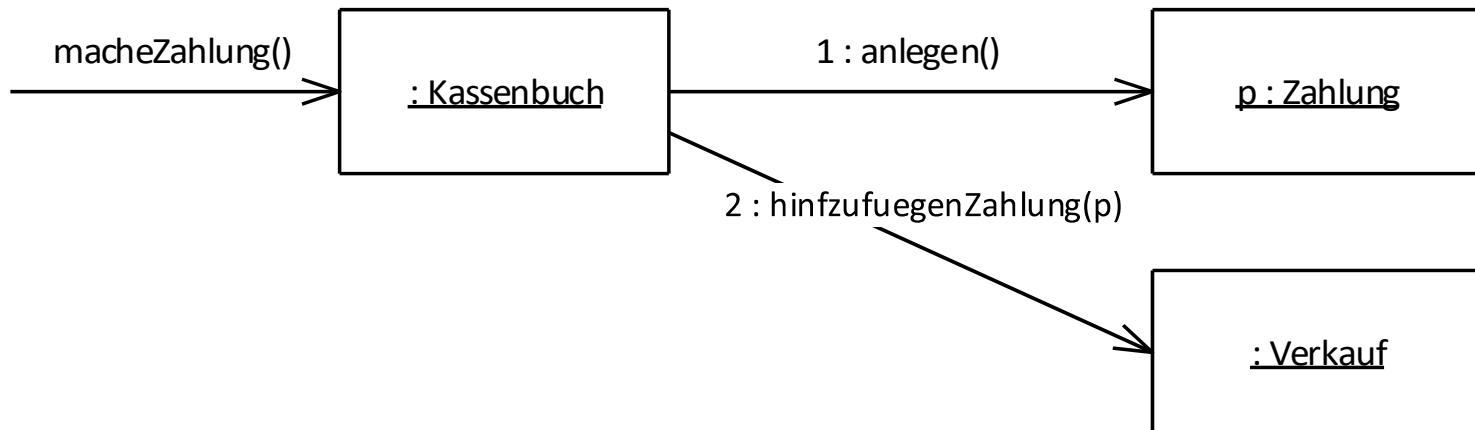
# Beispiel

Gegeben seien folgende Klassen



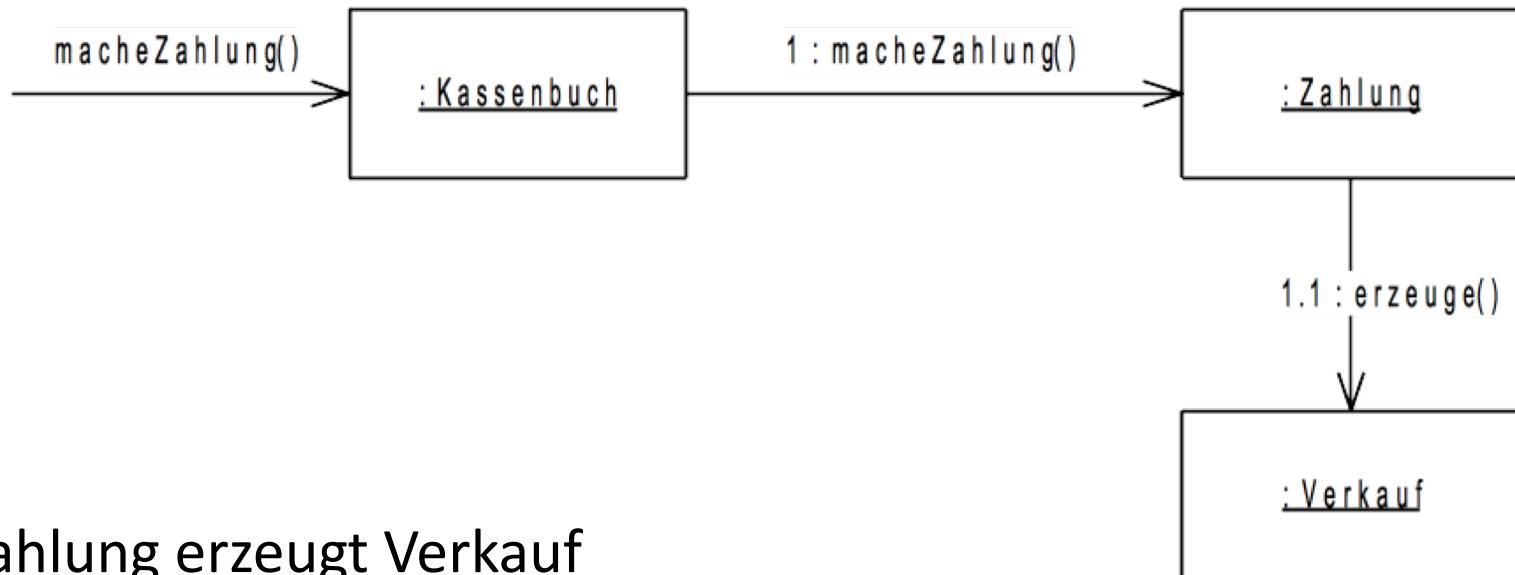
- Instanz „Zahlung“ muss erzeugt werden und mit Verkauf verbunden werden.
- Welche Klasse ist die Verantwortliche?

# Möglichkeit #1



Diese Lösung ist Erzeuger-Muster (creator pattern)

# Möglichkeit #2



Zahlung erzeugt Verkauf

# Welche Lösung ist besser?

Angenommen jeder *Verkauf* ist mit *Zahlung* verbunden.

Lösung #1 besitzt Kopplung zwischen *Kassenbuch* und *Zahlung*, welche nicht in Lösung #2 vorkommt.

Lösung #2 besitzt daher geringere Kopplung.

- Wie bündelt man ein Objekt, macht es verständlich, handhabbar und erreicht – ganz nebenbei – eine lose Kopplung?

# Kohäsion

**Kohäsion** (cohesion): Menge von Beziehungen innerhalb eines Subsystems

**Kohäsion** ist Maß, wie stark zusammenhängend und gebündelt die Verantwortlichkeiten im Element sind.

# Geringe Kohäsion

Element mit geringer Kohäsion bietet verschiedene, unzusammenhängende Funktionalitäten oder wenig Funktionalität. Solche Elemente sind unerwünscht und erzeugen folgende Probleme:

- schwer verständlich
- schwer wiederzuverwenden
- schwer zu warten
- anfällig, andauernd von Änderungen beeinflusst

# Hohe Kohäsion

Element mit hohen verbundenen Verantwortlichkeiten, das nicht viel Funktionalität bietet, hat eine hohe Kohäsion.

- allgemeines Entwurfsziel:
  - Reduzierte Komplexität

# Diskussion

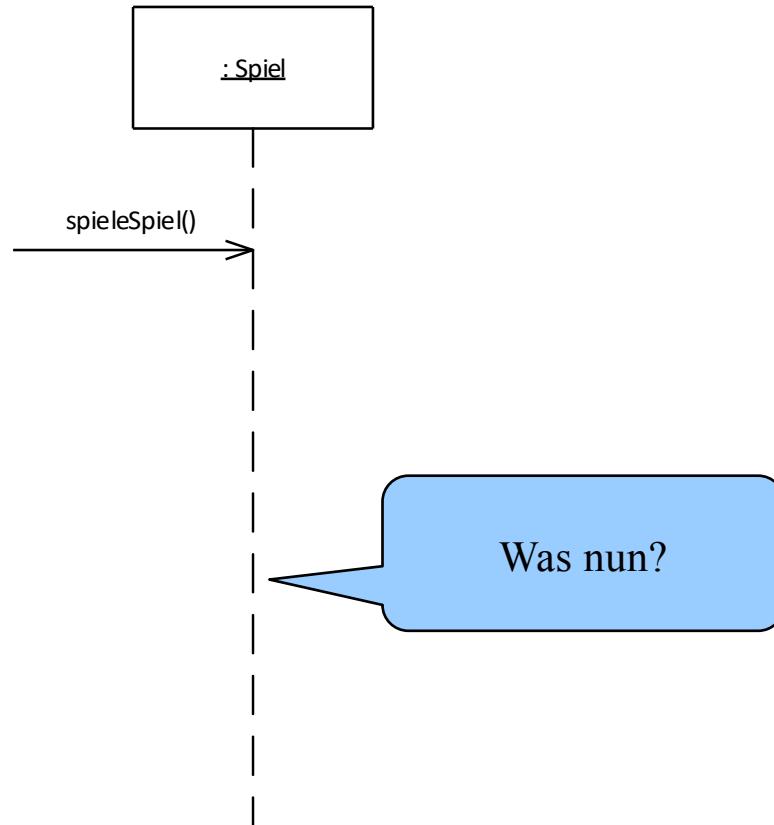
- **Sehr niedrige Kohäsion:** Klasse bietet vollständig unterschiedliche Funktionalität, z.B. Datenbankverbindung und RPC.
- **Niedrige Kohäsion:** Klasse hat eine Verantwortlichkeit für eine komplexe Aufgabe in einer komplexen Funktionalität, z.B. eine einzelne Schnittstelle für alle Datenbankzugriffe.
- **Mittlere Kohäsion:** Leichtgewichtige Klasse verantwortlich für einige wenige logische Bereiche, z.B. *Unternehmen* kennt Mitarbeiter und Finanzdetails.
- **Hohe Kohäsion:** Klasse mit moderater Verantwortlichkeit in einem funktionalen Bereich, welche mit anderen Klassen zusammenarbeitet.

# Faustregeln

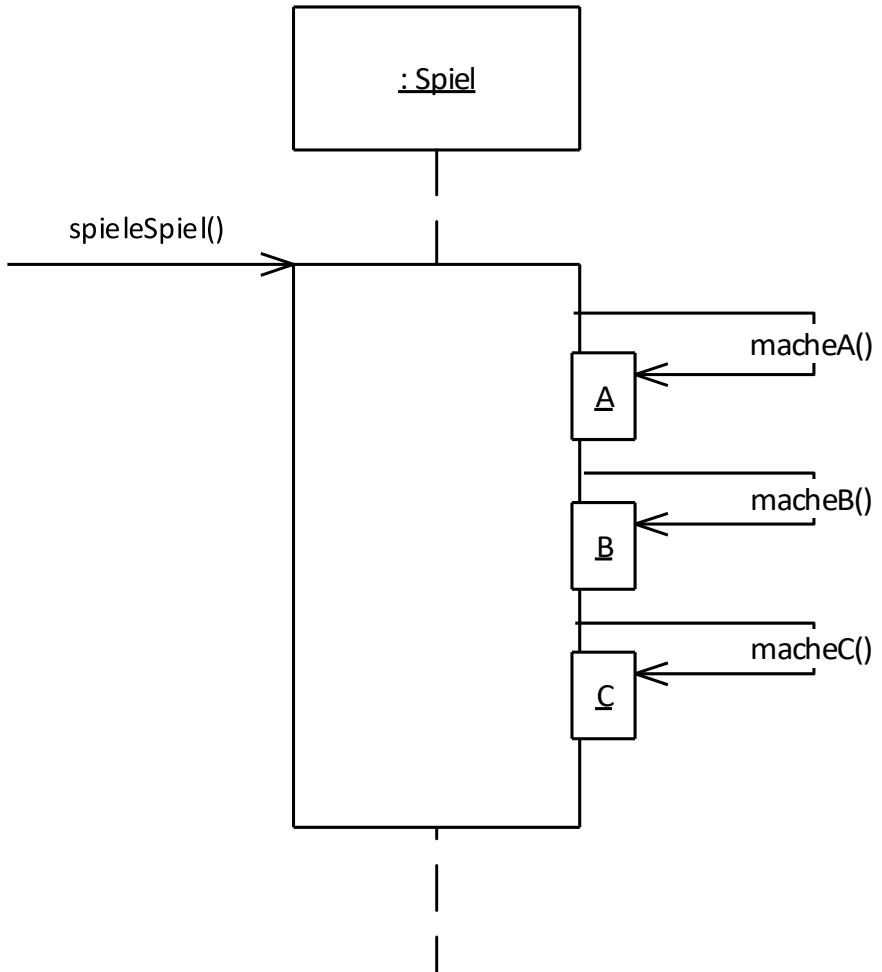
Um hohe Kohäsion aufzuweisen, muss eine Klasse

- wenige Methoden besitzen,
- wenige Codezeilen groß sein,
- nicht zu viel Funktionalität bieten,
- eine hohe Abhängigkeit des Programmcodes besitzen.

# Beispiel



# Beispiel

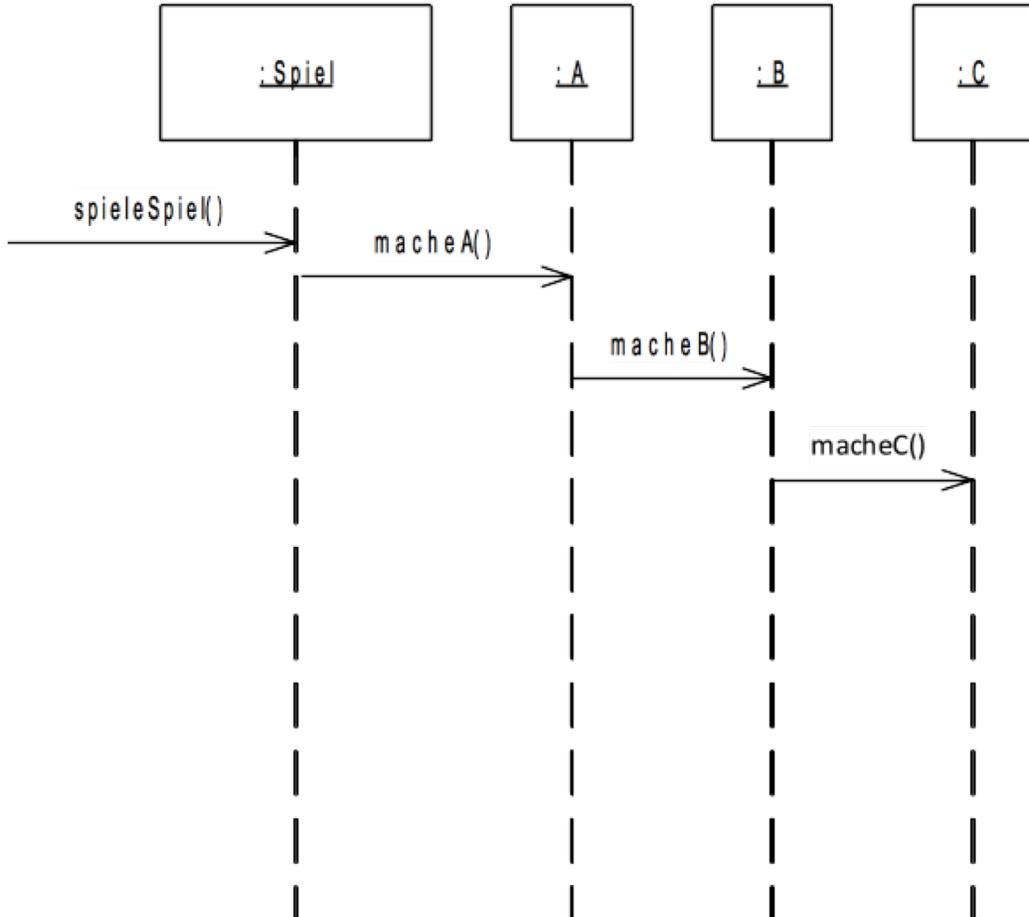


Object Spiel macht alles.

Wie hängen Aufgaben A, B und C zusammen?

Falls geringe Abhängigkeit,  
dann auch geringe Kohäsion!

# Erhöhung der Kohäsion



A,B und C kapseln  
einzelne Aspekte =>  
Hohe innere Kohäsion

# Eigenschaften einer guten Architektur



- **Minimieren** der Kopplung zwischen Modulen
  - Ziel: Module müssen wenig über Interaktion mit anderen Modulen wissen
  - Lose Kopplung erleichtert zukünftige Änderungen
- **Maximieren** der Kohäsion in Modulen
  - Ziel: Inhalte jeder Komponente sollten stark zusammenhängend sein
  - Hohe Kohäsion erleichtert das Verstehen einer Komponente

Oft: Zielkonflikt!

# Agenda

- Zum Begriff Architektur
- Architekturmuster
- Qualität von Architektur
- Weitere Muster

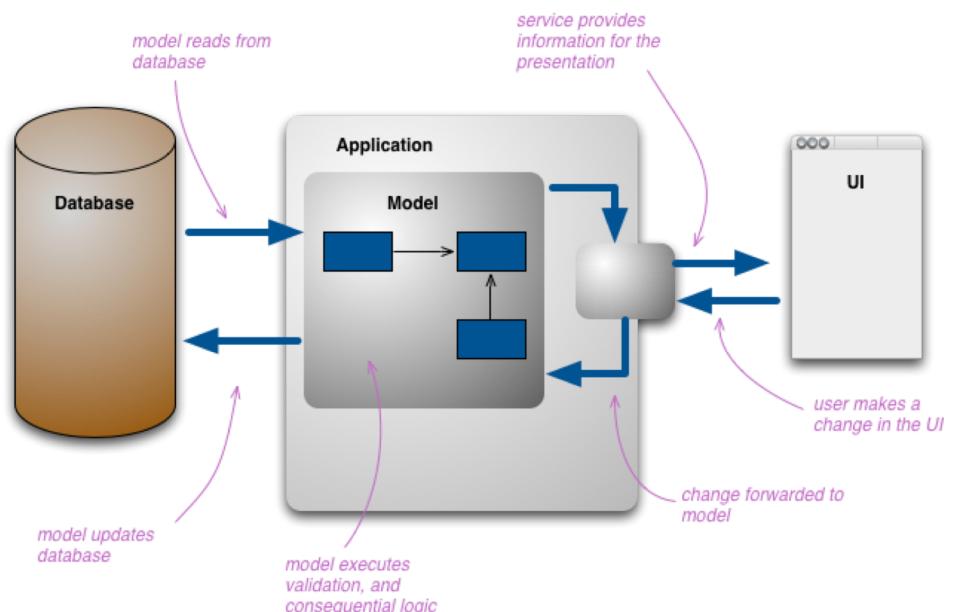
Skalierbarkeit

# CQRS

## Command Query Responsibility Segregation

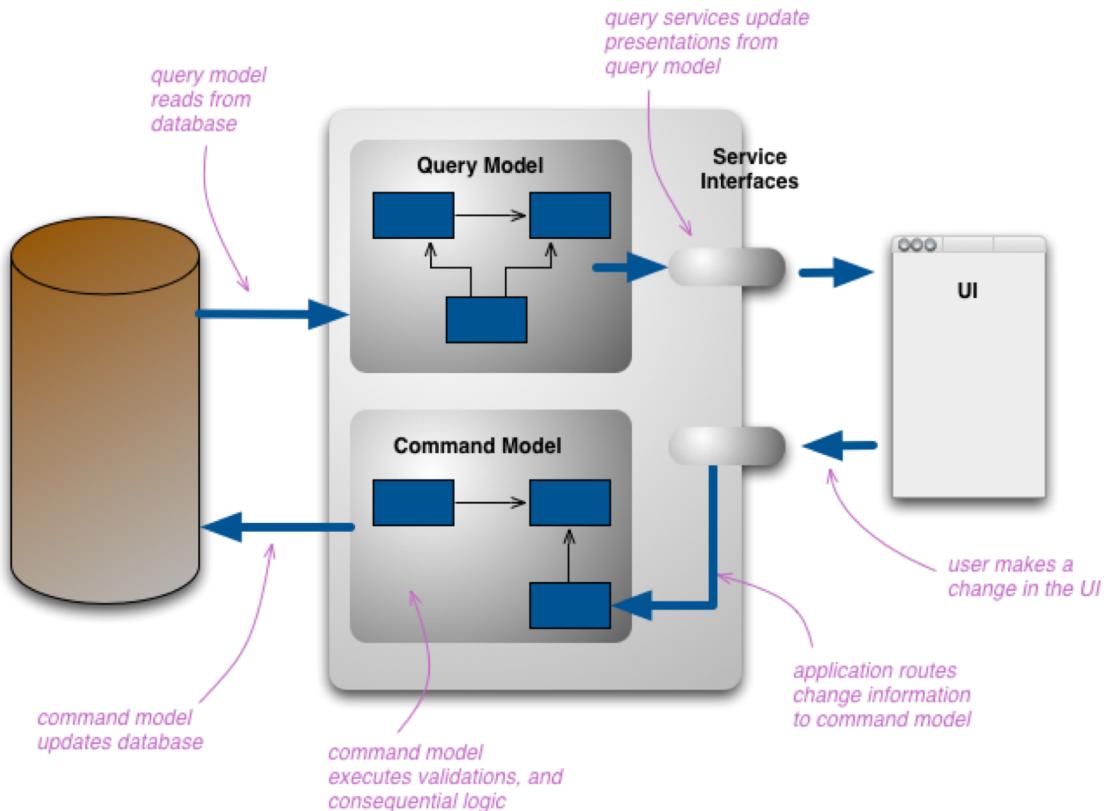
**Problem:**

Gemischtes Modell  
für Ansicht  
und  
Manipulation



Quelle: <http://martinfowler.com/bliki/CQRS.html>

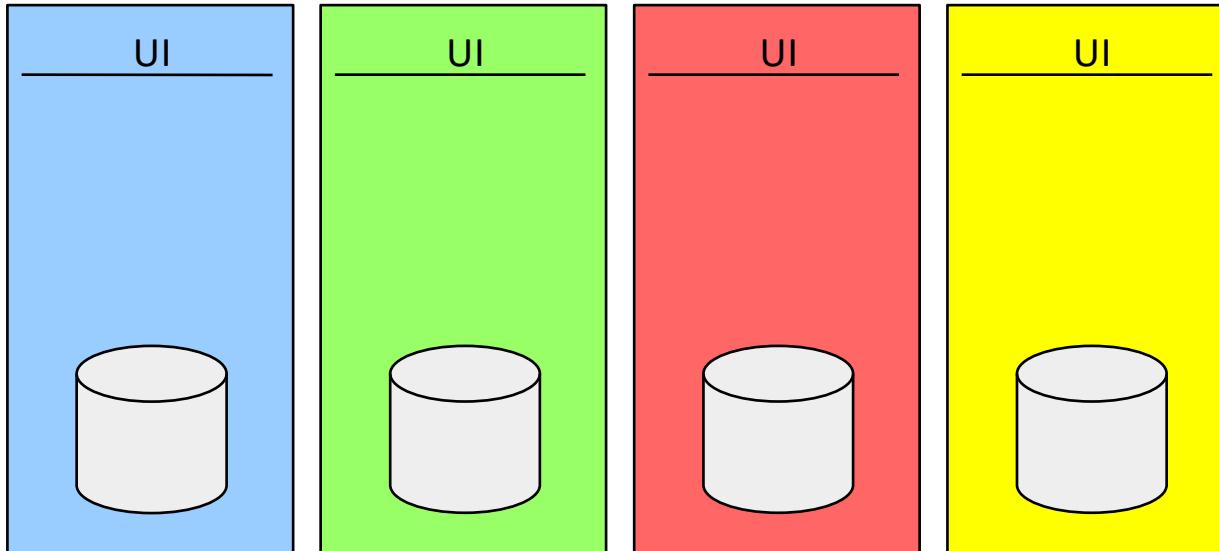
Lösung:  
Trennung von  
Sicht- und  
Manipulations  
befehlen



Quelle: <http://martinfowler.com/bliki/CQRS.html>

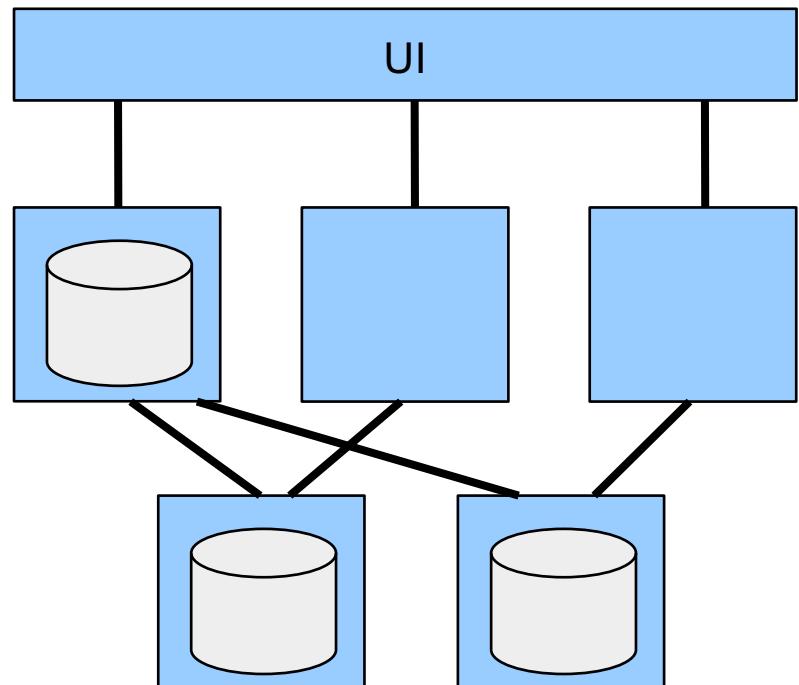
# Vertikale Dekomposition

Anstelle einer Anwendung, in der alle Features in **einen Prozess** gepackt werden, zerlege Anwendung von vornherein in **mehrere**, voneinander möglichst **unabhängige** Anwendungen. (Auch oft Partition genannt)



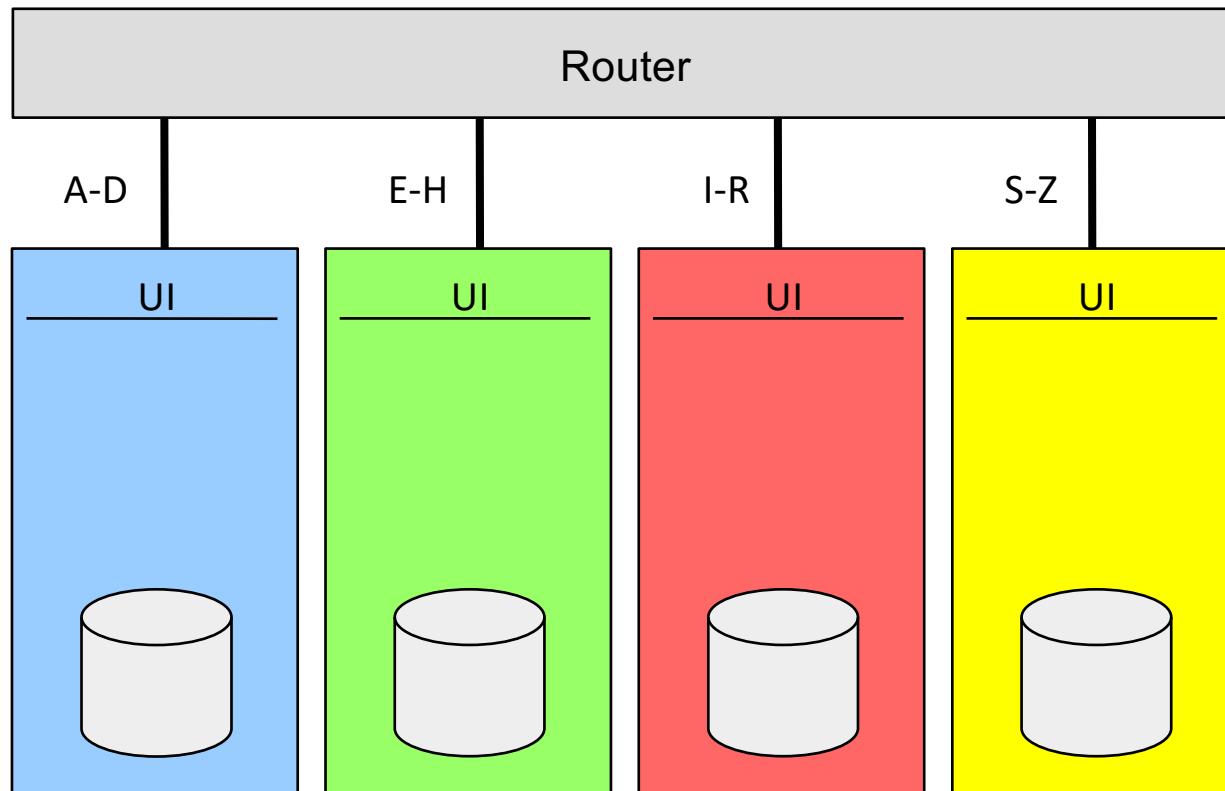
# Distributed Computing

- Vertikale Schnitte reichen u.U. nicht aus → daher auch horizontale Schnitte
- Kommunikation z.B. über REST (Representational state transfer)



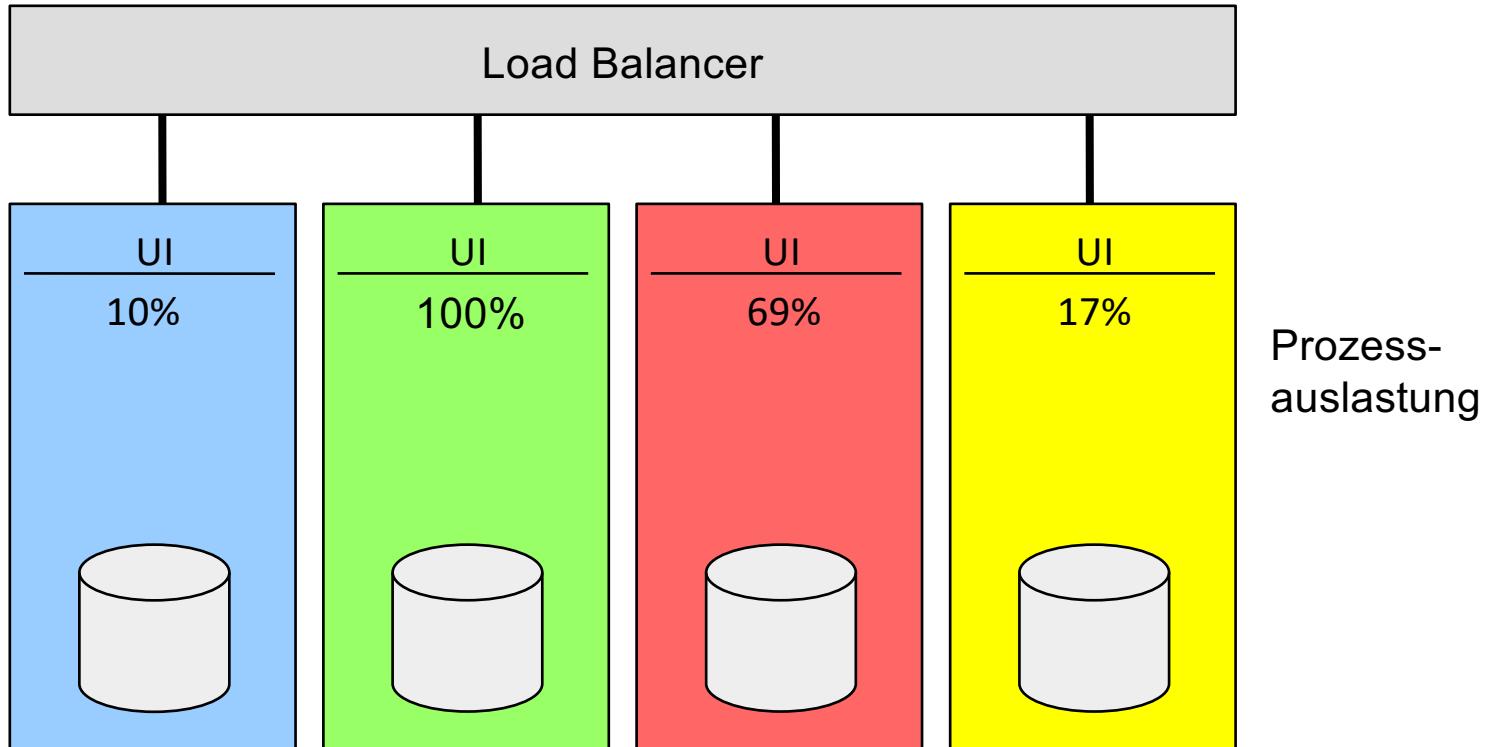
# Sharding

- Aufteilung der Eingaben auf dedizierte Prozesse
- Beispiel: Aufteilung nach Anfangsbuchstaben



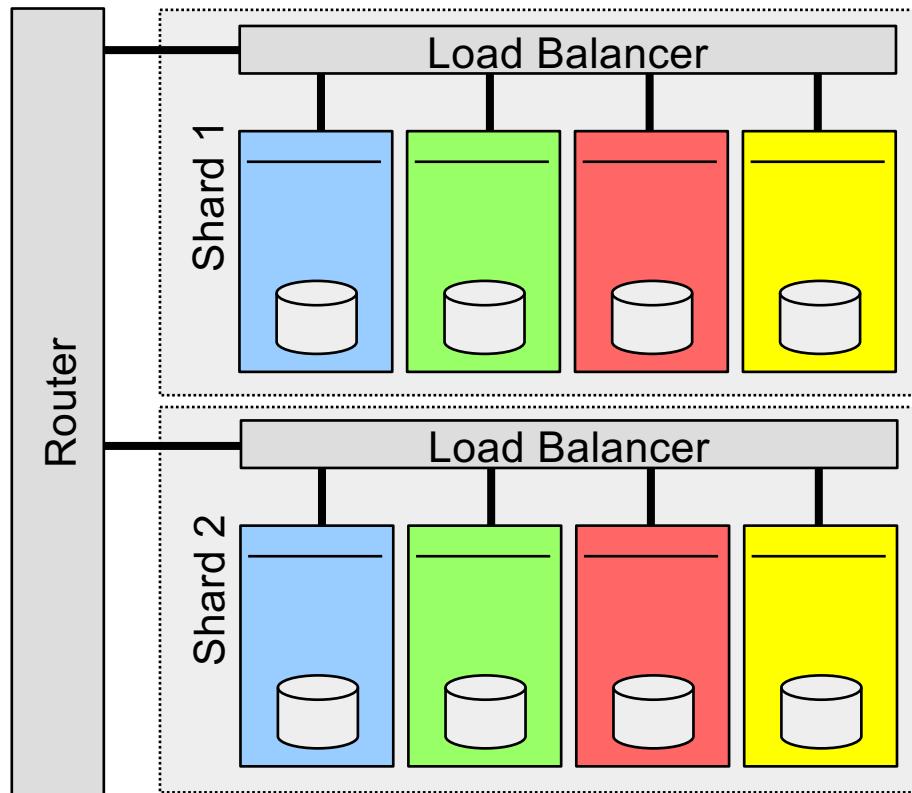
# Load Balancing

Anstatt Sharding mit festem Kriterium, z.B. Anfangsbuchstaben → dynamische Zuordnung der Bearbeitung, z.B. nach Auslastung



# Kombinationen

- Varianten können kombiniert werden um Skalierbarkeit zu erhöhen



# Microservice Architektur

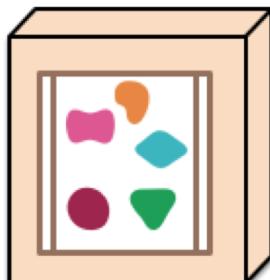
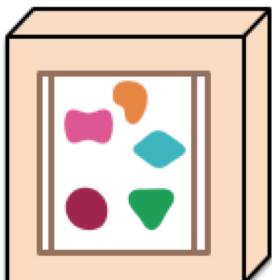
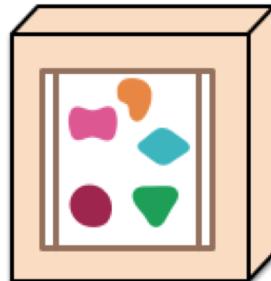
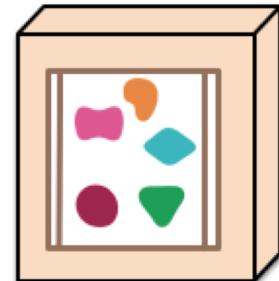
- In Microservice Architekturen sind Softwareapplikationen eine Folge von unabhängig einsetzbaren (deployable) Diensten
  - Wichtige Eigenschaften:
    - Komponentenzerlegung via Dienste leicht möglich
    - Organisiert um betriebswirtschaftliche Funktionen
    - Entwicklung eher Produkte als Projekte
    - Smarte Endpunkte und dumme Verbindungen
    - Ausfallsicher und Evolutionsfähig
    - Dezentrale Anwendungs- und Datenverwaltung

# Microservice Architektur

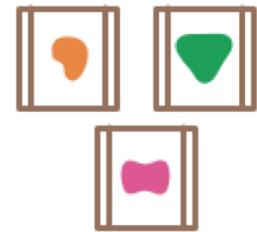
*A monolithic application puts all its functionality into a single process...*



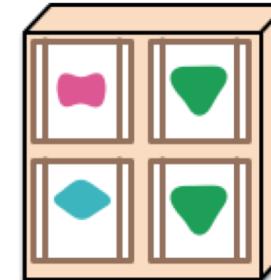
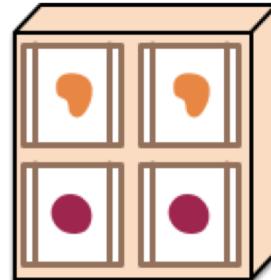
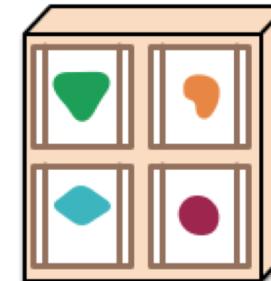
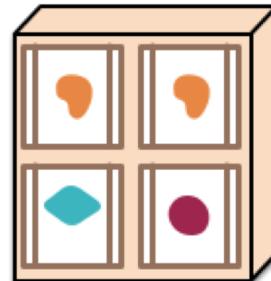
*... and scales by replicating the monolith on multiple servers*



*A microservices architecture puts each element of functionality into a separate service...*



*... and scales by distributing these services across servers, replicating as needed.*



Quelle: <http://martinfowler.com/articles/microservices.html>

# Zusammenfassung

- Architektur
  - Umsetzbarer Entwurf der eines Systems
- Architekturmetriken
  - Kopplung: Abhängigkeiten zwischen Systemteilen
  - Kohäsion: Zusammenhang innerhalb von Systemteilen
- Architekturstile
  - Verschiedene Stile addressieren verschiedene Herausforderungen im Systementwurf