

iSAQB Foundation Level

Entwurf und Entwicklung von
Softwarearchitekturen

Agenda

1. LZ 2-1: Vorgehen und Heuristiken zur Architekturentwicklung auswählen und anwenden können (R1, R3)
2. LZ 2-2: Softwarearchitekturen entwerfen (R1)
3. LZ 2-3: Einflussfaktoren auf Softwarearchitektur erheben und berücksichtigen können (R1-R3)
4. LZ 2-4: Querschnittskonzepte entwerfen und umsetzen (R1)
5. LZ 2-6: Entwurfsprinzipien erläutern und anwenden (R1-R3)
6. LZ 2-5: Wichtige Lösungsmuster beschreiben, erklären und angemessen anwenden (R1, R3)
7. LZ 2-7: Abhängigkeiten von Bausteinen managen (R1)
8. LZ 2-8: Qualitätsanforderungen mit passenden Ansätzen und Techniken erreichen (R1)
9. LZ 2-9: Schnittstellen entwerfen und festlegen (R1-R3)

Entwurf und Entwicklung von Softwarearchitekturen

- Dauer: 330 Min. Übungszeit: 90 Min.
- Wesentliche Begriffe:
 - Entwurf; Vorgehen beim Entwurf; Entwurfsentscheidung; Sichten; Schnittstellen; technische Konzepte und Querschnittskonzepte; Stereotypen; Architekturmuster; Entwurfsmuster; Mustersprachen; Entwurfsprinzipien; Abhängigkeit; Kopplung; Kohäsion; fachliche und technische Architekturen; Top-down und Bottom-Up-Vorgehen; modellbasierter Entwurf; iterativer/inkrementeller Entwurf; Domain-Driven Design

LZ 2-1: Vorgehen und Heuristiken zur Architekturentwicklung auswählen und anwenden können (R1,R3)

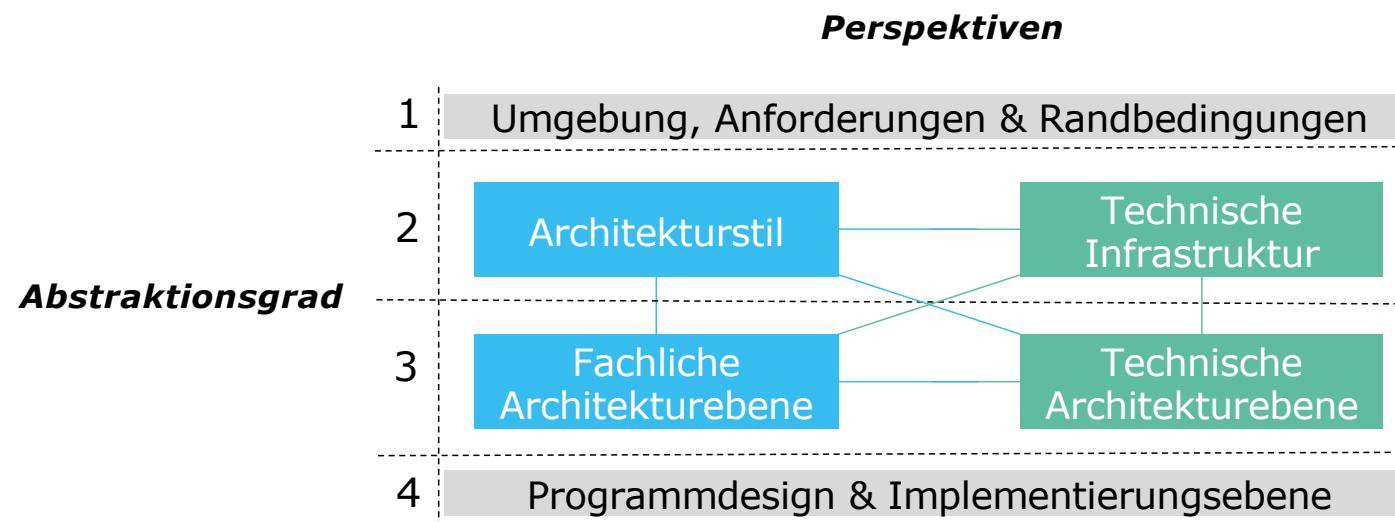
Softwarearchitekt:innen können grundlegende Vorgehensweisen der Architekturentwicklung benennen, erklären und anwenden, beispielsweise:

- Top-down- und Bottom-Up-Vorgehen beim Entwurf (R1)
- Sichtenbasierte Architekturentwicklung (R1)
- Iterativer und inkrementeller Entwurf (R1)
 - Notwendigkeit von Iterationen, insbesondere bei unter Unsicherheit getroffenen Entscheidungen (R1)
 - Notwendigkeit von Rückmeldungen zu Entwurfsentscheidungen (R1)
- Domain-Driven Design (R3)
- Evolutionäre Architektur (R3)
- Globale Analyse (R3)
- Modellgetriebene Architektur (R3)

Top-down und Bottom-up Vorgehen

Der **Architekt** kombiniert **Top-down** und **Bottom-up** Vorgehen und wechselt so die Abstraktionstufen und Architekturebenen

- Einflussfaktoren, Randbedingungen und Anforderungen wirken auf die Ebenen und Stufen
- Abstraktionsebene und Perspektiven stehen in Wechselwirkung zueinander



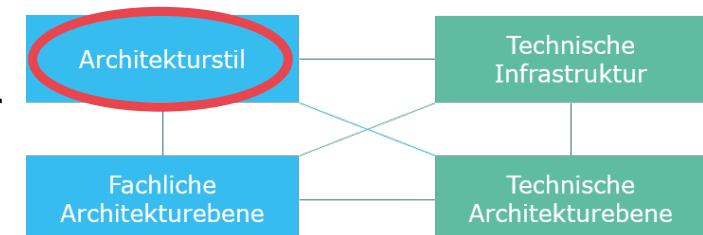
Zentrale Architekturmétapher durch Architekturstile



Zum Nachlesen

Architekturstil

- Umfasst Regeln, Muster und Prinzipien für die Basisstrukturierung eines Systems, die gleichartig anzuwenden sind
- Leitplanken für den Designraum der gesamten Architektur
- Prinzipien und Muster beziehen sich i.d.R. auf Modularität, Schnittstellen und Betrieb des Systems
- Klassisch: Client-Server, Pipes und Filter, Repository, Schichten
- Modern: REST, Event Driven, Reactive, Service-orientierung



„Unser Softwaresystem ist nach einer Drei-Schichten-Architektur unter Verwendung des Model-View-Controller [...]“
[Gharbi S.33]

6 „Wir haben eine Microservice Architektur, in dem jeder Microservice einen Bounded Context hat und ausschließlich über REST mit anderen Microservices kommuniziert.“

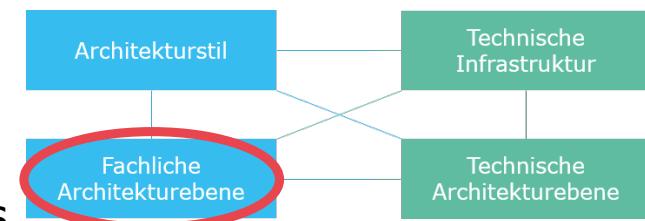
Fachlichkeit vor Technik



Zum Nachlesen

Fachliche Architekturebene

- Definition fachlicher Bausteine
- Definition fachlicher Abhängigkeiten zwischen Bausteinen (Schnittstelle)
- Analyse fachlicher Daten und Geschäftsobjekte
- Überführung der Fachlichkeit auf die Architektur (Paket- und Klassenstrukturen)
- Entwurf einer fachlich motivierten Architektur auf Basis eines Architektursichtenmodells (siehe auch LZ 2)
- Beispiele
 - Person, Kunde, Vertrag
 - PersonService, KundenService



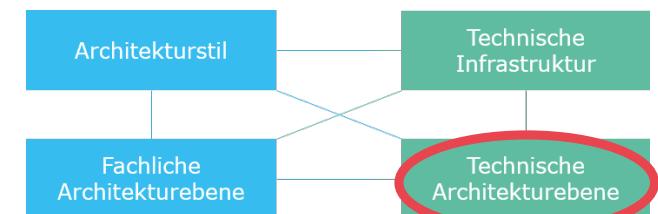
Fachliche und Technische Aspekte müssen zwingend getrennt werden



Zum Nachlesen

Technische Architekturebene

- Definition technischer Bausteine
(Querschnittskonzepte siehe auch LZ2 und LZ3)
- Fehlende Trennung zwischen technischen und fachlichen Bausteinen führt zu geminderter Wiederverwendbarkeit, Wartbarkeit und Verständlichkeit
- Ebenfalls Bestandteil des Entwurfs auf Basis eines Architektursichtenmodell
- Beispiele
 - Security Event Logging
 - Web Security Konfiguration
 - JPA bzw. JDBC Adapter



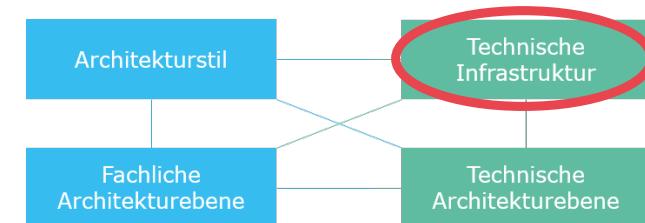
Technische Infrastruktur hat großen Einfluss auf die Softwarearchitektur



Zum Nachlesen

Technische Infrastruktur

- Möglichkeiten der Technischen Infrastruktur beeinflussen den Lösungsraum der Anwendungsarchitektur durch Technische und organisatorische Vorgaben
- Zentrale geregelter Einsatz von Komponenten wie z.B. Middleware, Build Tools, Server, Container, ESB und Caches
- Infrastruktur in Form von CPU, Speicher und Skalierung



Architekturebenen und Perspektive

Analyse, Identifikation von Risiken, Planungsinformationen und Anforderungslücken

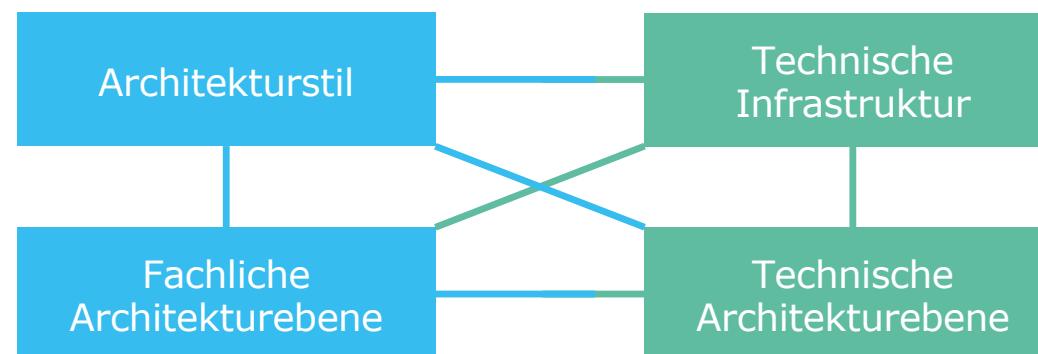
Architekturstil, Kommunikationsmodelle Geschäftsprozesse Fachliche Domäne Abbildung der Domäne auf Architektur Qualitätsanforderungen explizit machen

Architekturmuster Architektsichten Fachliche Dekomposition und Bausteine Packagestruktur Mustersprache Entitäten Modell

Umgebung, Anforderungen & Randbedingungen

Funktionale und nichtfunktionale Anforderungen, Referenzarchitekturen, Budget, Timeline, Toolchain, Systemkontext

Randbedingungen und Einschränkungen oder auch Möglichkeiten durch Infrastruktur



Logging & Logging Framework, Application Monitoring, Security, Persistenz & Caching Persistenzmodell Broker Clients

Programmdesign & Implementierungsebene

Beispiel Geschäftsobjekt Kunde

Umgebung, Anforderungen & Randbedingungen

Anforderungslücke:

Wie sensible sind die Kundendaten?

Anforderungen:

Kunde anlegen
Kundendaten bearbeiten
Kunde kontaktieren

Architekturstil:

REST

Kommunikationsmodell:

KundeResource
Kunde REST Schnittstelle
(CRUD Semantik)

Packagestruktur und

Mustersprache:

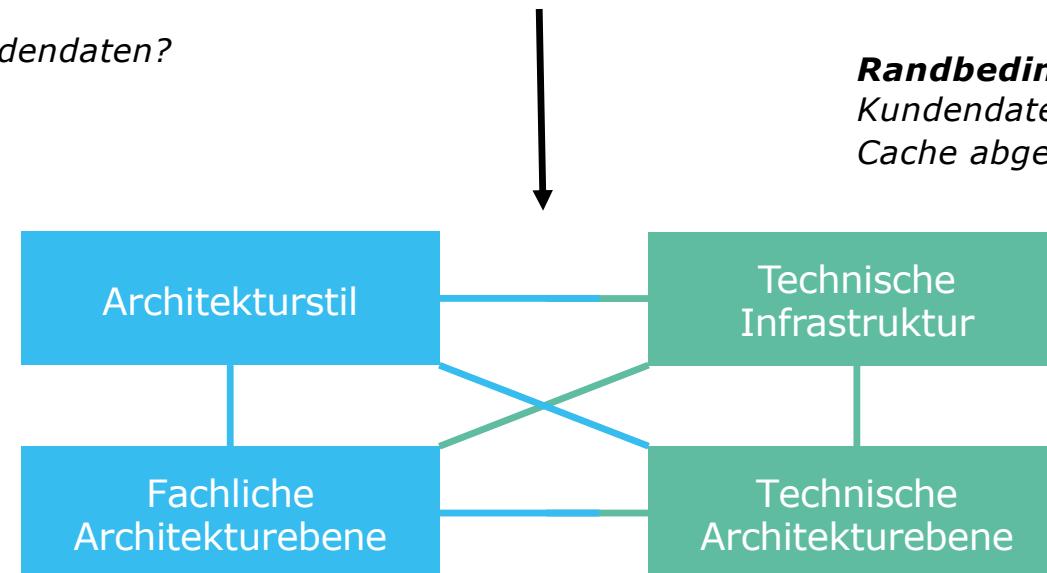
de.app.kunden.verwaltung
KundeService, KundeEntity

Randbedingung:

Kundendaten dürfen nicht in einem Cache abgelegt werden

Zentrale Komponenten:

CustomerAppointment
Management mit E-Mailversand



Technische Bausteine:

SendMailServiceClient
KundeJpaEntity, KundeRepository

Architektur-Mittel eines Softwarearchitekten

Architektur-Mittel sind **Instrumente**, die im **Architekturentwurf** zum Einsatz kommen und dabei helfen die **Komplexität der Problemstellungen** zu **reduzieren**.

- **Heuristiken**
„Die Kunst mit unvollständigen Informationen zu angemessenen Lösungen zu kommen“ [Starke 2015 S. 79]
Recherche, Hierarchische (De-)Komposition, Top-Down & Bottom-Up, Abstraktion, Black- & Whitebox, ...
- **Architekturprinzipien & Entwurfsprinzipien**
Konkrete und bewährte Regeln der Software-Entwicklung
Geheimnisprinzip, Trennung von Verantwortlichkeiten, Lose Kopplung & hohe Kohäsion, Schnittstellen-Zerlegungsprinzip, Gesetz von Demeter, ...
- **Architekturmuster & Entwurfsmuster**
Erprobte Lösung zu einem wiederkehrenden Problem
Schichtenarchitektur, Model View, Controller, Fassade, Adapter, Observer, Strategy, Event Sourcing, Ports & Adapter, Clean Architecture, Command Query Responsibility Segregation, ...

Hierarchische (De-)Komposition mit Hilfe von Black- und Whitebox Denken

- Die **Blackbox** Perspektive ermöglicht das Vernachlässigen von Details und eine Konzentration auf die **Schnittstelle** und **Bausteinverantwortung**
- Die **Whitebox** Perspektive ermöglicht die Konzentration auf die **Interna** eines Bausteins
- In Summe ermöglicht es Menschen, Probleme systematisch zu Analysieren und beherrschbare Lösungen zu erstellen

Ziele:

- Erstellung eines Modells zur Abstraktion (vereinfachten Darstellung) der Wirklichkeit
- In der Folge Komplexitätsreduktion, klar definierte Strukturen und Verantwortungen
- Positive Auswirkung auf Qualitätsmerkmale
Erweiterbarkeit, Änderbarkeit und Verständlichkeit

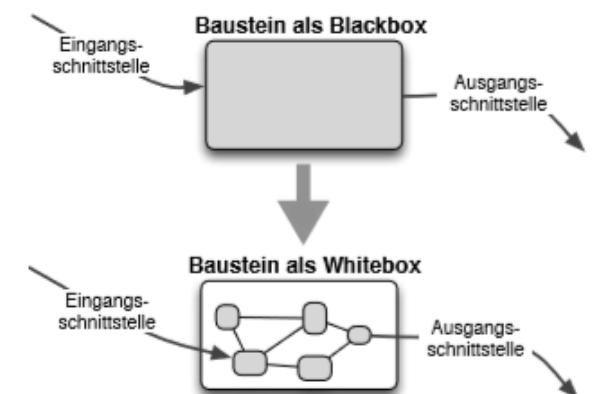


Abbildung aus
[Starke 2015]

Hierarchische Dekomposition

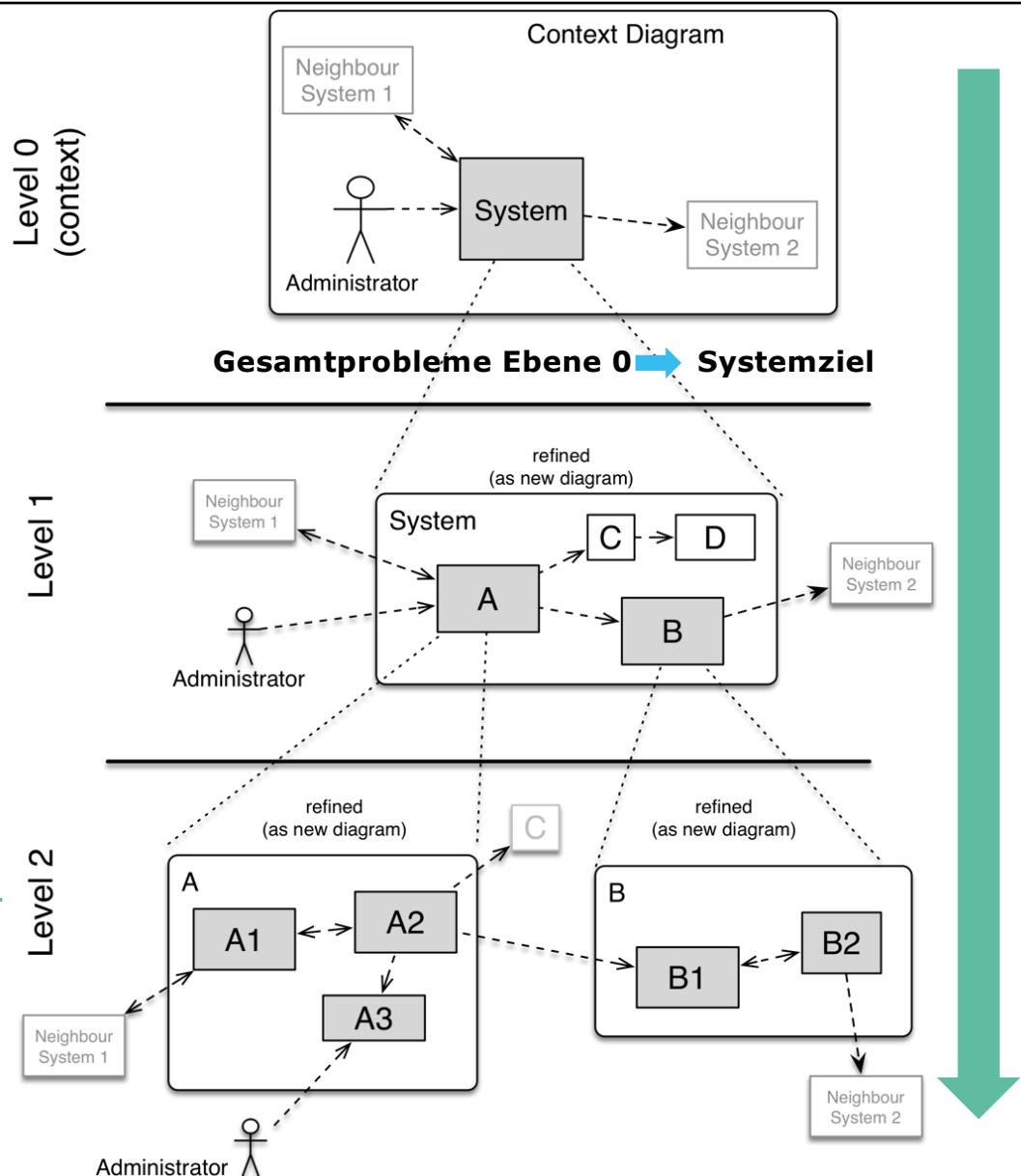
Top-Down Dekomposition

Ausgehend vom Ziel des Systems erfolgt die Zerlegung in Teilprobleme.

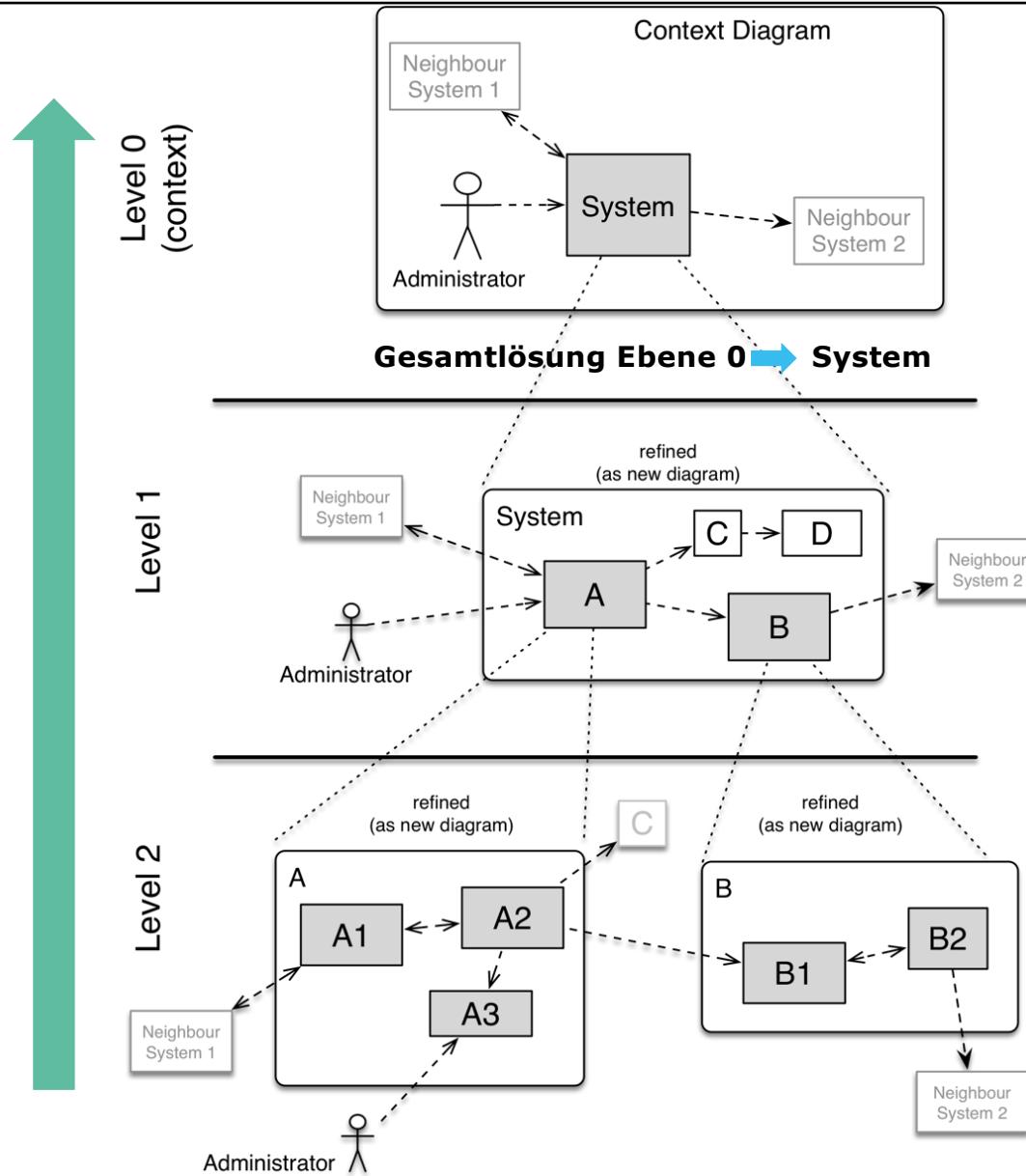
Teilproblem
Ebene 1

Teilproblem
Ebene 2

Abbildung von
<https://faq.arc42.org/questions/B-10/>



Hierarchische Komposition



Bottom-Up Komposition

Ausgehend von der Teillösung erfolgt der Zusammenbau des Systems

Teillösung
Ebene 1

Teillösung
Ebene 2

Abbildung von
<https://faq.arc42.org/questions/B-10/>

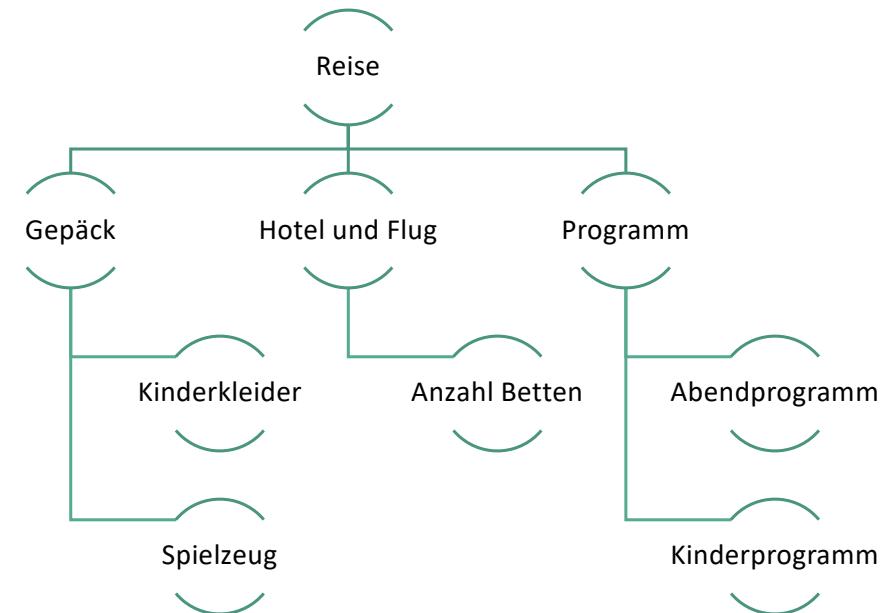
Hierarchische (De-)Komposition

Top-Down	Bottom-Up
Beginn mit der Problemstellung	Beginn mit der konkreten Maschine / Lösung
Zerlegung in immer weitere Teilprobleme	Aufbau abstrakter Bausteine auf der vorhandenen Basis
Ergebnisse liegen sehr spät vor	Sehr schnelle Ergebnisse
Geringes Risiko ungeeignete Ergebnisse zu erzeugen	Teilergebnisse können ungeeignet zur weiteren Verwendung sein

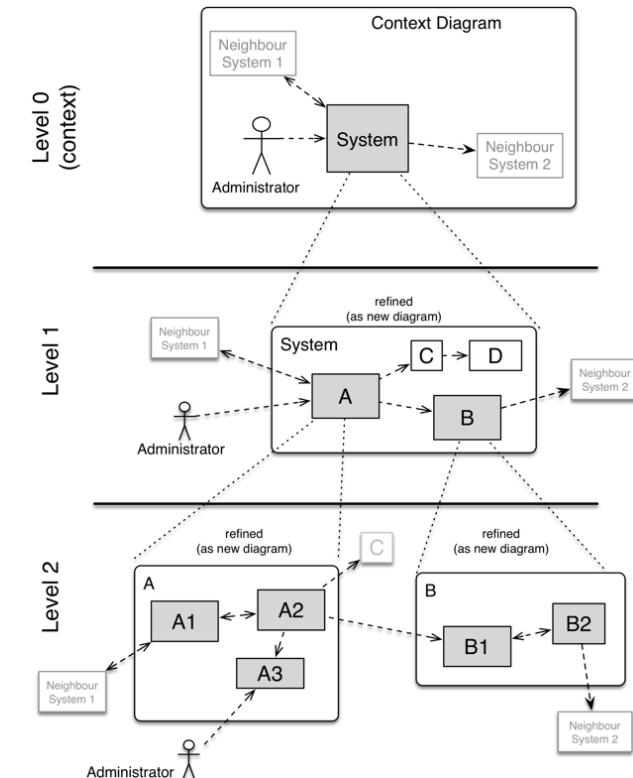
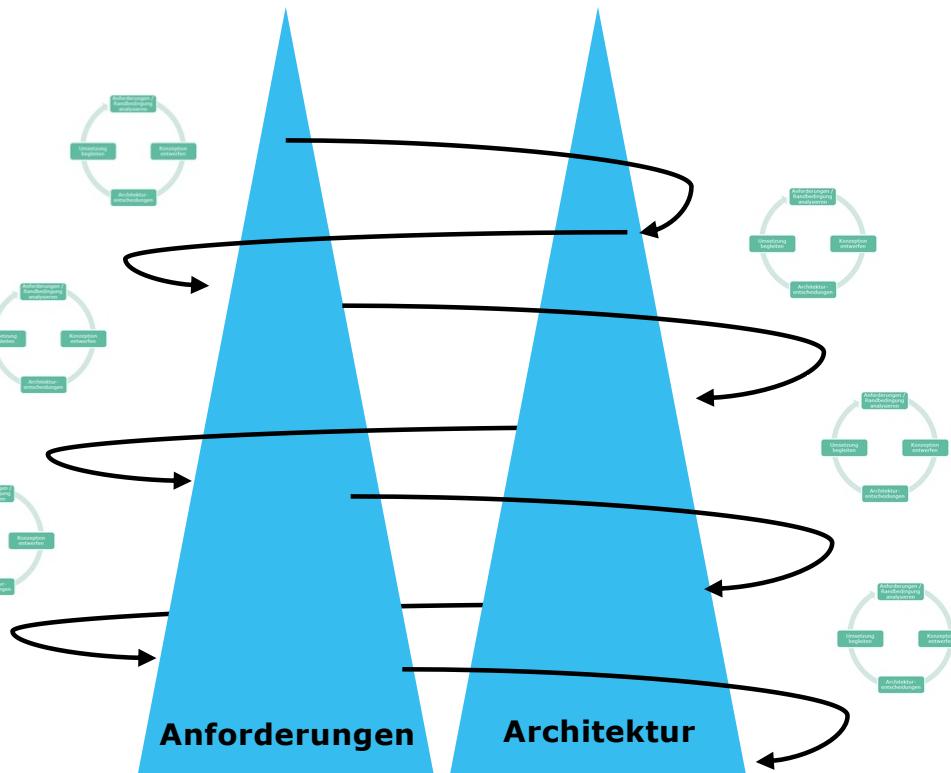
- Beide Verfahren müssen sich ergänzen, so dass ein System sowohl von oben her als auch von unten her entworfen wird.

Denken in Hierarchien ist menschlich...

- Hierarchisch geordnete Inhalte können Menschen leichter erlernen, verarbeiten, sich merken und abrufen
- Macht jeder im Alltag (Urlaubsplanung, Planung des Arbeitstag Projektplanung)
- Einheiten (z.B. Bausteine eines Softwaresystems) werden im Gedächtnis mit Hilfe von hierarchischen Strukturen abgelegt
- Ein hierarchisch gut strukturiertes System kann schneller verstanden werden und die Durchführung von Wartungsarbeiten ist effizienter
- Im Vordergrund stehen in diesem Kontext „Nutzungs-Hierarchien“ (nicht Vererbungs-Hierarchien)



Softwarearchitekturen müssen iterativ und inkrementell auf Basis von Abstraktionen erstellt werden!

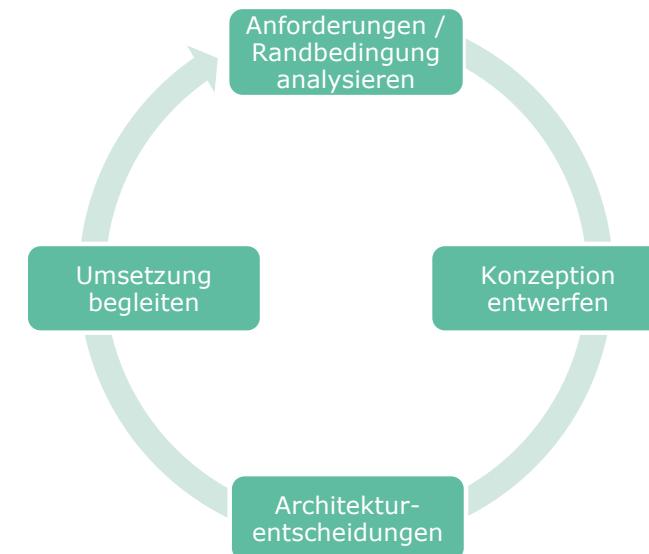


Architekturkonzeption im iterativen und inkrementellen Wechselspiel



Zum Nachlesen

- Viele Tätigkeiten des Architekturentwurfs sind nicht linear, sondern werden **gleichzeitig in sinnvoller Reihenfolge durchgeführt**
- Ein Architekt wechselt abhängig von der Situation und Kontext zwischen diesen Tätigkeiten
- Dies ist **kombiniert** mit einem **Top-Down und Bottom-up** Wechsel auf Abstraktionsebenen
- Iterative und inkrementelle Architekturarbeit ermöglicht das frühzeitige Erkennen von nicht tragenden Architekturentscheidungen und das zeitnahe Einholen von Feedback zu Entscheidung / Lösung sowie die Berücksichtigung des Feedbacks im weiteren Verlauf.



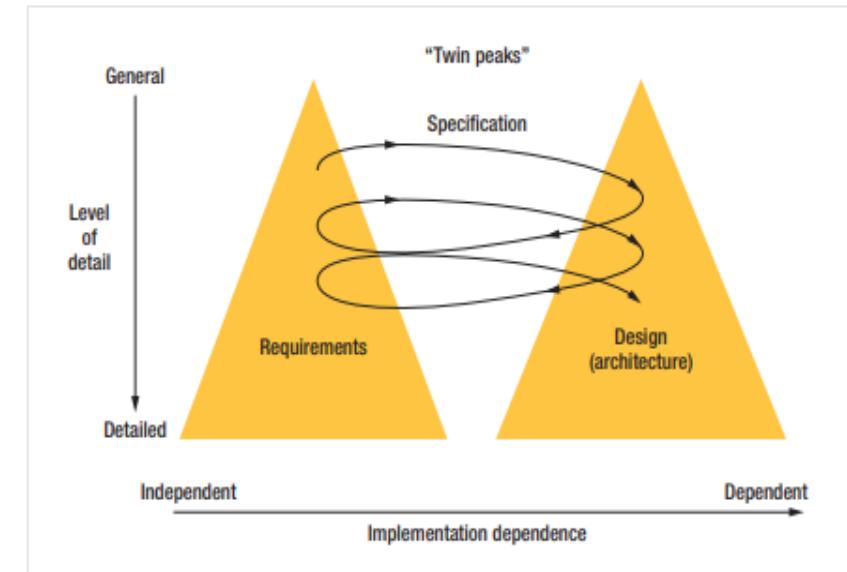
Architekturkonzeption im iterativen und inkrementellen Wechselspiel



Zum Nachlesen

- In jeder Iteration nur so viel Architektur wie notwendig festlegen und früh Feedback zu angedachten Lösungen einholen
- Dadurch langfristigen, zukunftsfähigen Bauplan finden, ohne dass der Architekturentwurf zu einem Zeitpunkt revidiert werden muss
- Bei innovativen, risikoreichen Projekten ist die Gefahr hoch, dass der Architekturentwurf komplett revidiert werden muss, wenn z.B. hoch priorisierte Qualitätsanforderung nicht erreicht werden können. Iteratives und inkrementelles Vorgehen unterstützt ein frühzeitiges Erkennen und Vermeiden dieses Problems.

Twin-Peaks Modell



<https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6470589>

Domain-Driven Design

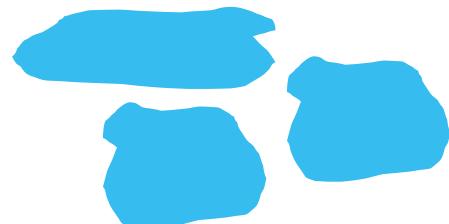
Domain first in architecture and design

Strategisches Design

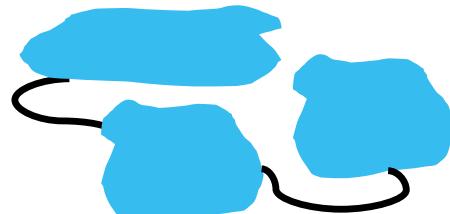


Taktisches Design

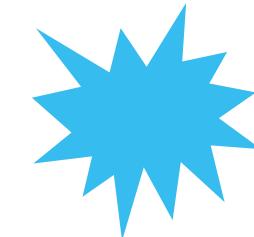
Bounded Context und
Ubiquitous Language



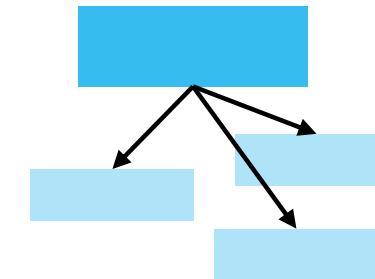
Context Mapping



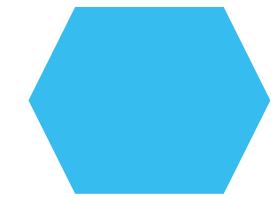
Domain Events



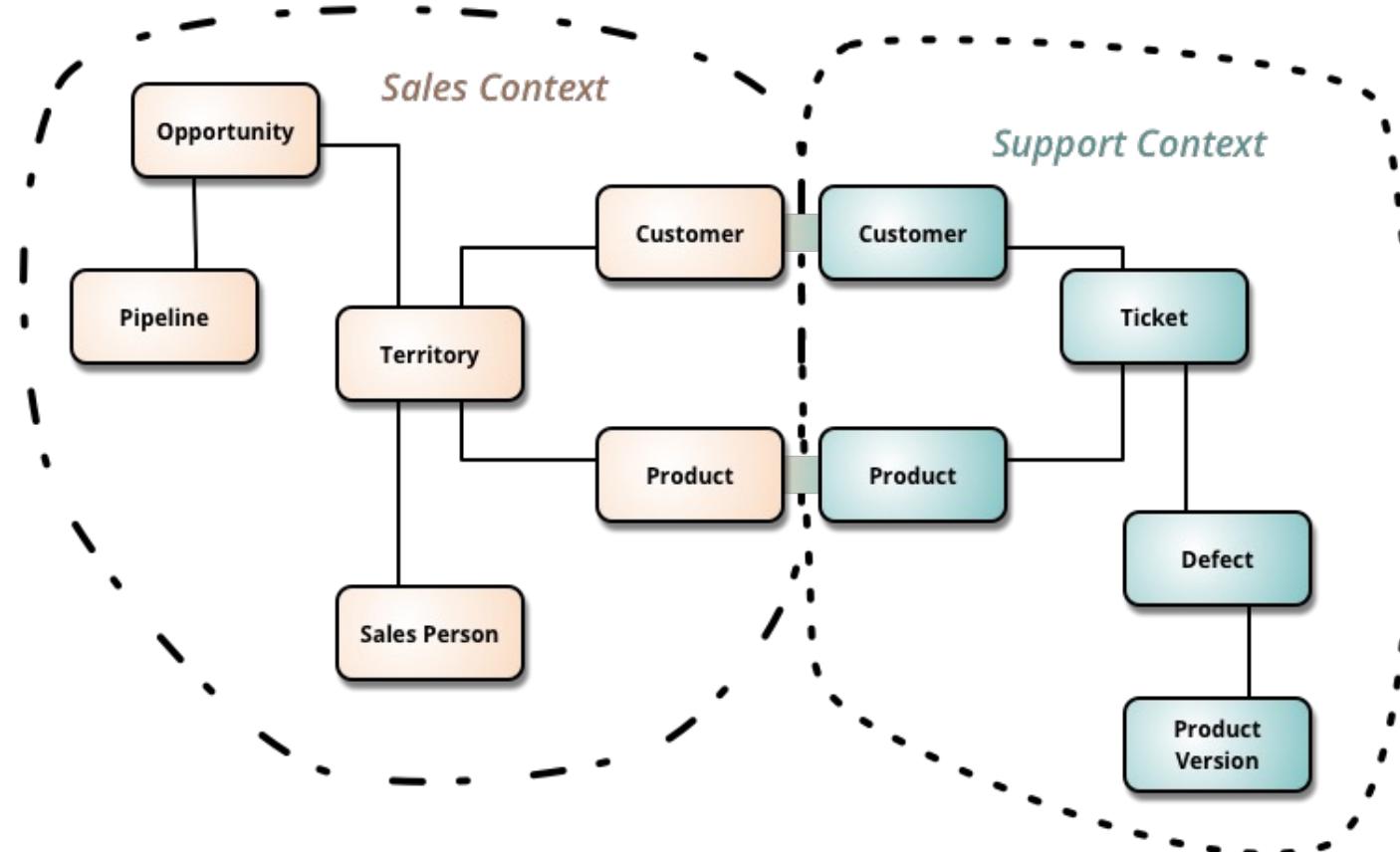
Building Blocks
und Aggregate



Architekturmuster
Hexagonale Architektur



Ubiquitous Language und Bounded Context



Domain-Driven Design



Zum Nachlesen

- Ganzheitlicher Architektur- und Entwicklungsansatz für Softwaresysteme in komplexen Domänen
- Domain-Driven Design unterteilt sich in **strategisches** und **taktisches Design**
- **Domain-Driven Design** ist eine Sammlung von **Prinzipien** und **Mustern**, die beim **Entwerfen hochwertiger Software** unterstützen
- Die Fachlichkeit, sprich die Domäne, ist ausschlaggebend für Architekturentscheidungen
- Die Domäne enthält eine Fachsprache, welche allgegenwärtig in der Kommunikation zwischen Entwicklung und Fachbereich sowie in der Architektur in Artefakt, Paket und Klassen ist
- **Domain-Driven Design** definiert für das **Klassendesign** sogenannte **Building Blocks**. Hierbei steht das Aggregate im Mittelpunkt.

Domänen und Subdomänen



Zum Nachlesen

Domäne

- Eigenständiger **Problem- und Lösungsbereiche**
- Hierarchiebildung durch **Subdomänen**
- Jede Domäne bzw. Subdomäne hat einen Bounded Context

Arten von Domänen

- **Core Domain**
Kernfunktionalität des Systems
- **Generic Domain**
Nicht Teil der Core Domain aber wichtig für das Business
- **Supporting Domain**
Unterstützende, untergeordnete Funktionen, die von der Core Domain getrennt werden müssen

Domain-Driven Design

Strategic Design



Zum Nachlesen

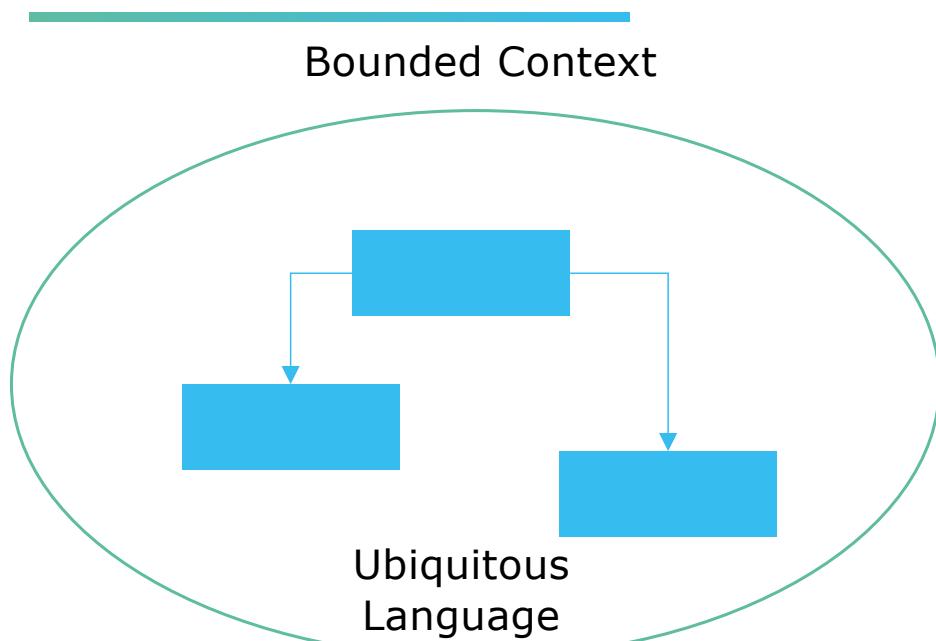


Abbildung angelehnt an [Vernon 2017]

Strategic Design beschäftigt sich mit dem Entwurf einer fachlichen Domäne und ihrer allgegenwärtigen Fachsprache.

Bounded Context

- Grenze eines fachlichen Bereichs
- Fachlichkeit des Kontextes wird ausgedrückt durch eine Fachsprache

Ubiquitous Language

- Fachsprache, die von allen Beteiligten (Entwickler, Architekten, Domänen Experten, Stakeholdern) verstanden wird
- Manifestiert sich in Artefakten-, Paket-, Klassen-, Methoden- und Variablennamen

Bounded Context, Teams und Ownership



Zum Nachlesen

Domain Driven Design hat das Ziel fachliche Domänen voneinander Abzugrenzen (Bounded Context) und fachlich ausdrucksstarke Architekturen zu erstellen. Dies verhindert monolithische Systeme und Systemstrukturen.

Ein **Bounded Context**...

- Hat ein **eigenes Quellcode-Repository**
- Hat ein **eigenes Datenbankschema** (oder auch Datenbank)
- Ist ein **eigenständiges Deployment-Artefakt**
- Kann nur von **einem Team** bearbeitet werden. Ein Team kann mehrere Bounded Contexte verantworten

Diese Aussage adressiert primär eine Enterprise Architecture. Beim Entwurf eines Softwaresystems steht oft ein (großes) Team zu Verfügung. Ein Bounded Context wird hier als Modularisierungsgrenze verwendet. Eine Aufteilung von Subteams auf Subdomänen ist möglich.

Taktisches Design mit Building Blocks

Das **taktische Design** von DDD widmet sich den internen Strukturen eines Bounded Context. Hierfür werden Building Blocks (Stereotypen für Klassen) eingesetzt!

Durch die Stereotypen werden Verantwortlichkeiten gemäß *Single Responsibility Principle* verteilt.

Dies unterstützt die Erstellung lose gekoppelter Klassenstrukturen.

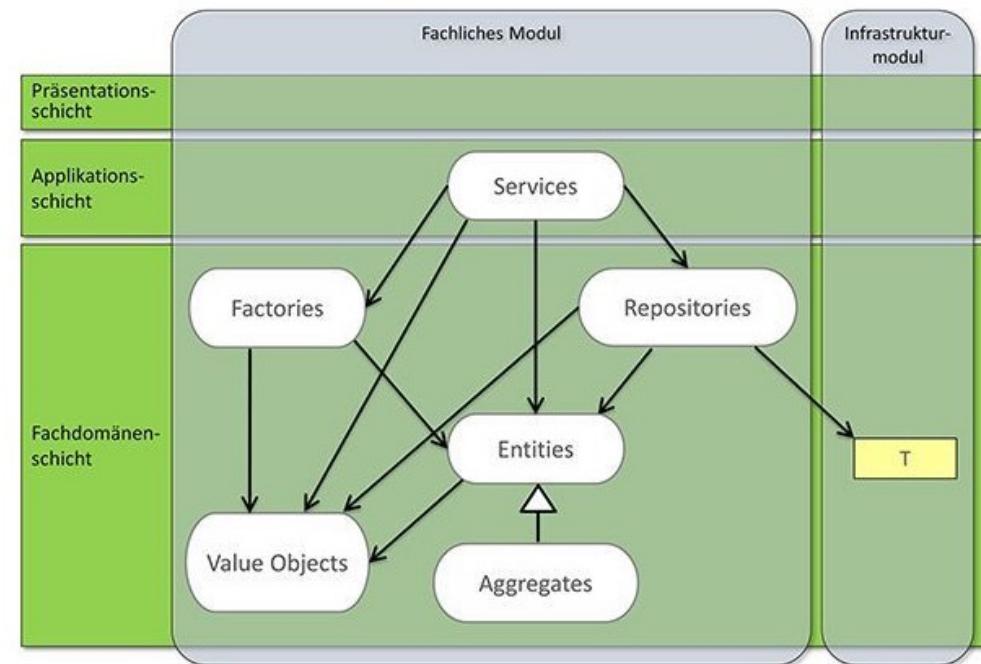
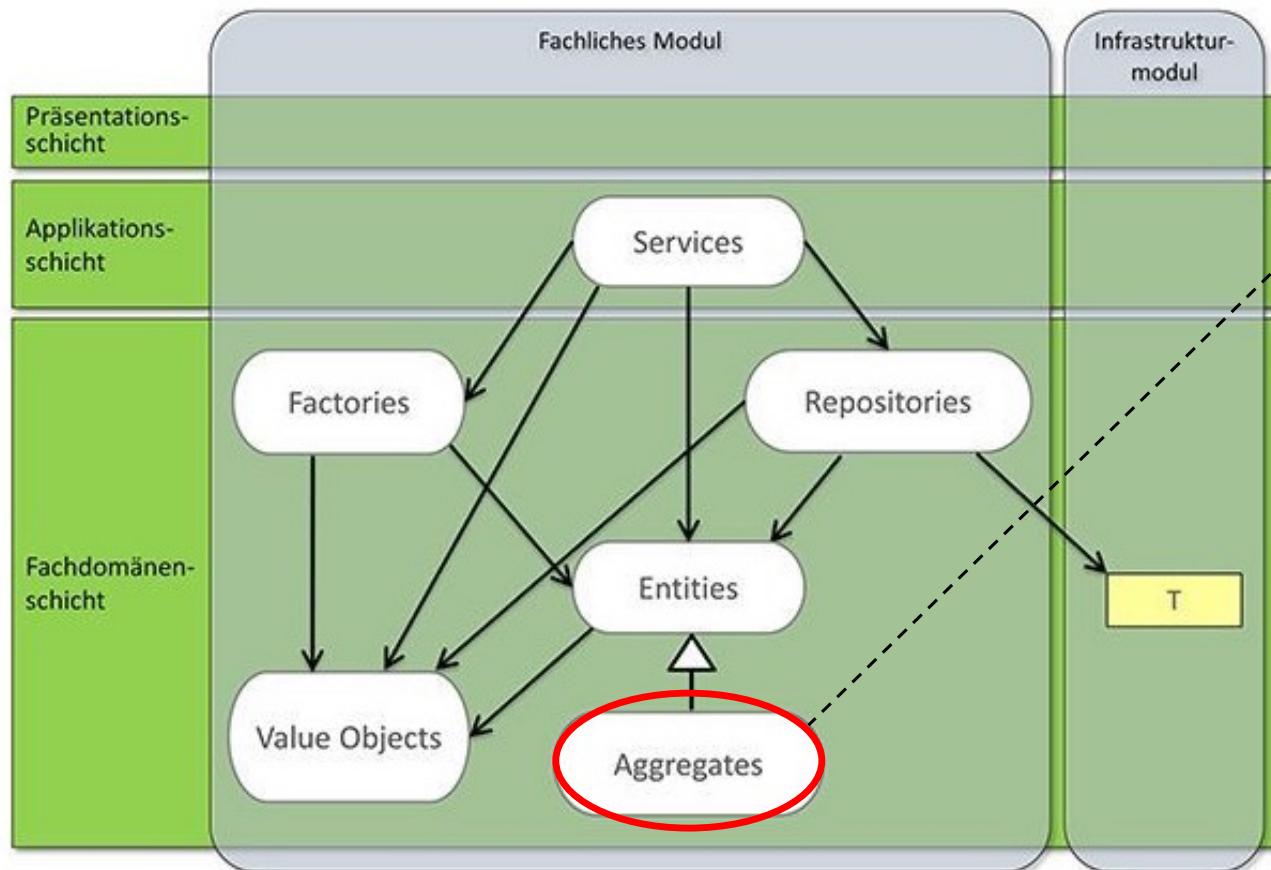


Abbildung aus
[Lilienthal 2016]

Taktisches Design mit Building Blocks



Zum Nachlesen



Aggregate

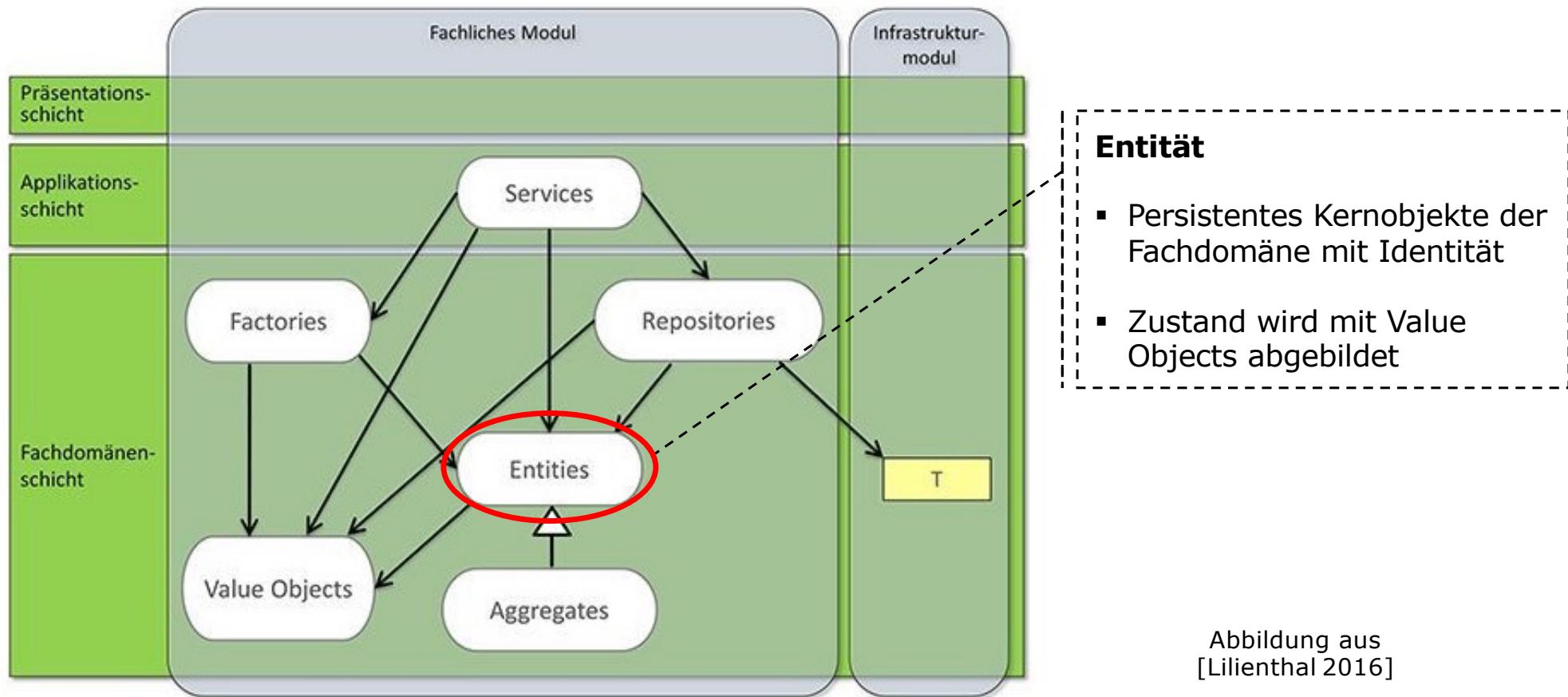
- Wurzel-Entität, die Entitäten und Value Objects kapselt
- Stellt die Klammer für Transaktionen und Serviceschnitt dar

Abbildung aus
[Lilienthal 2016]

Taktisches Design mit Building Blocks



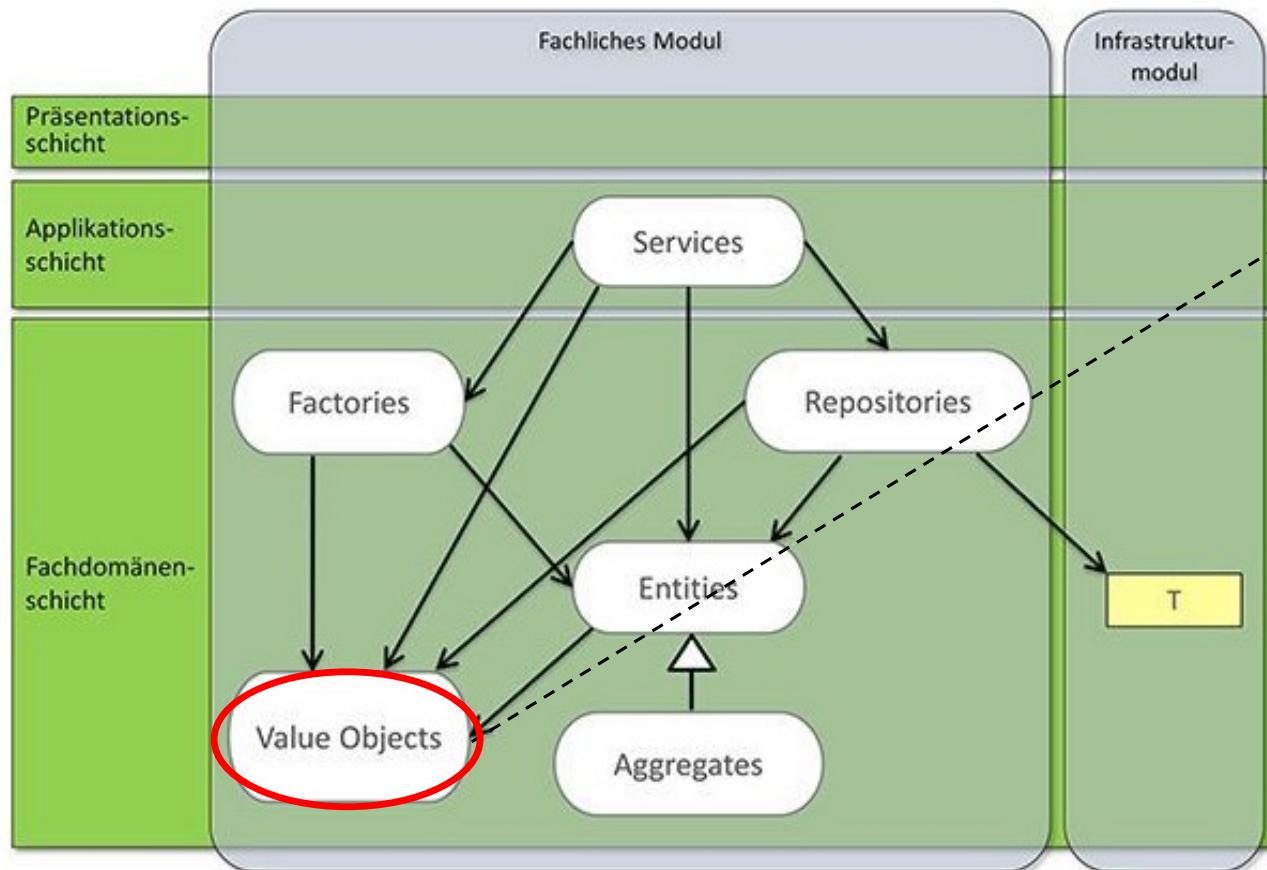
Zum Nachlesen



Taktisches Design mit Building Blocks



Zum Nachlesen



Value Object

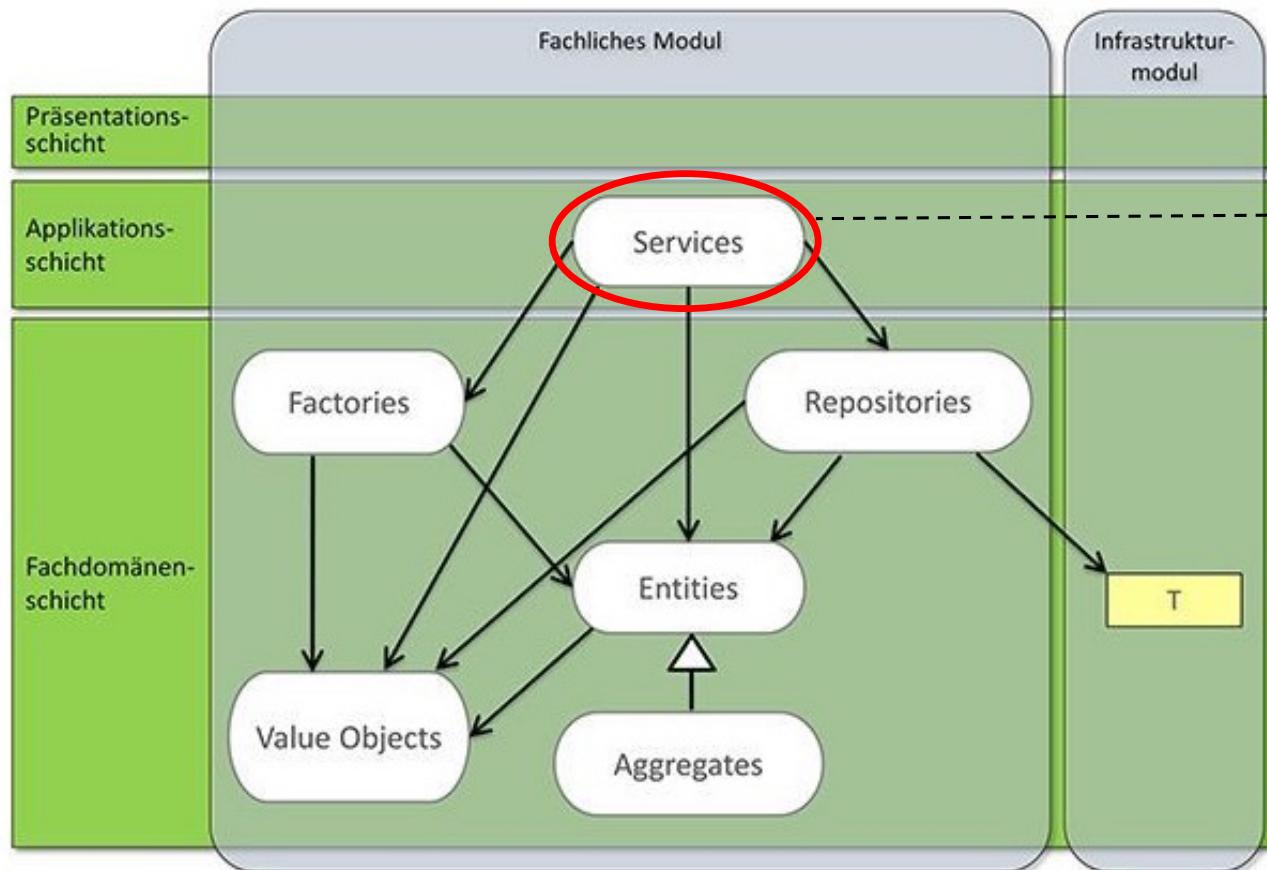
- Keine eigene Identität
- Können aus anderen Value Objects bestehen, aber nicht aus Entitäten

Abbildung aus
[Lilienthal 2016]

Taktisches Design mit Building Blocks



Zum Nachlesen



Service

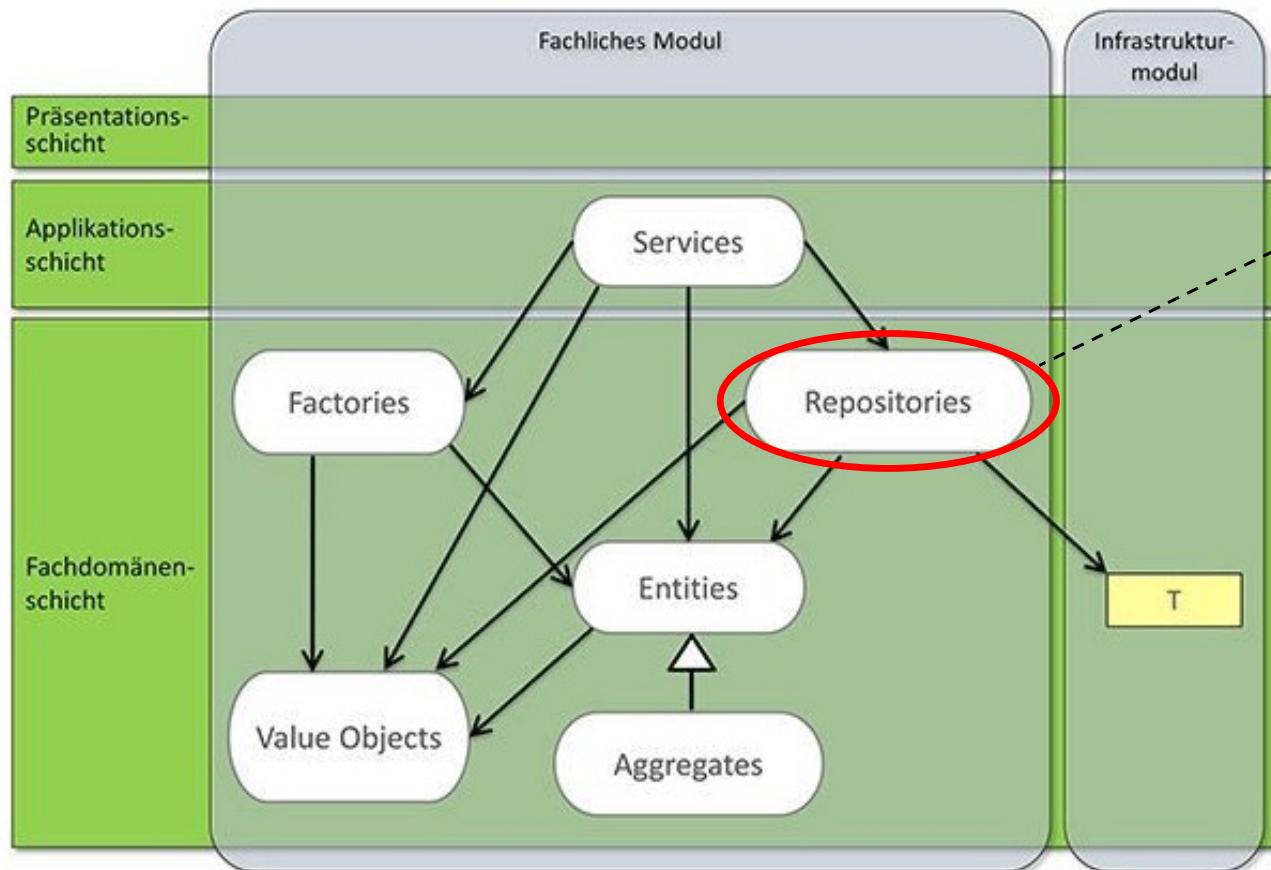
- Logik, Abläufe und Prozesse
- Ergebnisse werden durch Entitäten und Value Objects repräsentiert

Abbildung aus
[Lilienthal 2016]

Taktisches Design mit Building Blocks



Zum Nachlesen



Repository

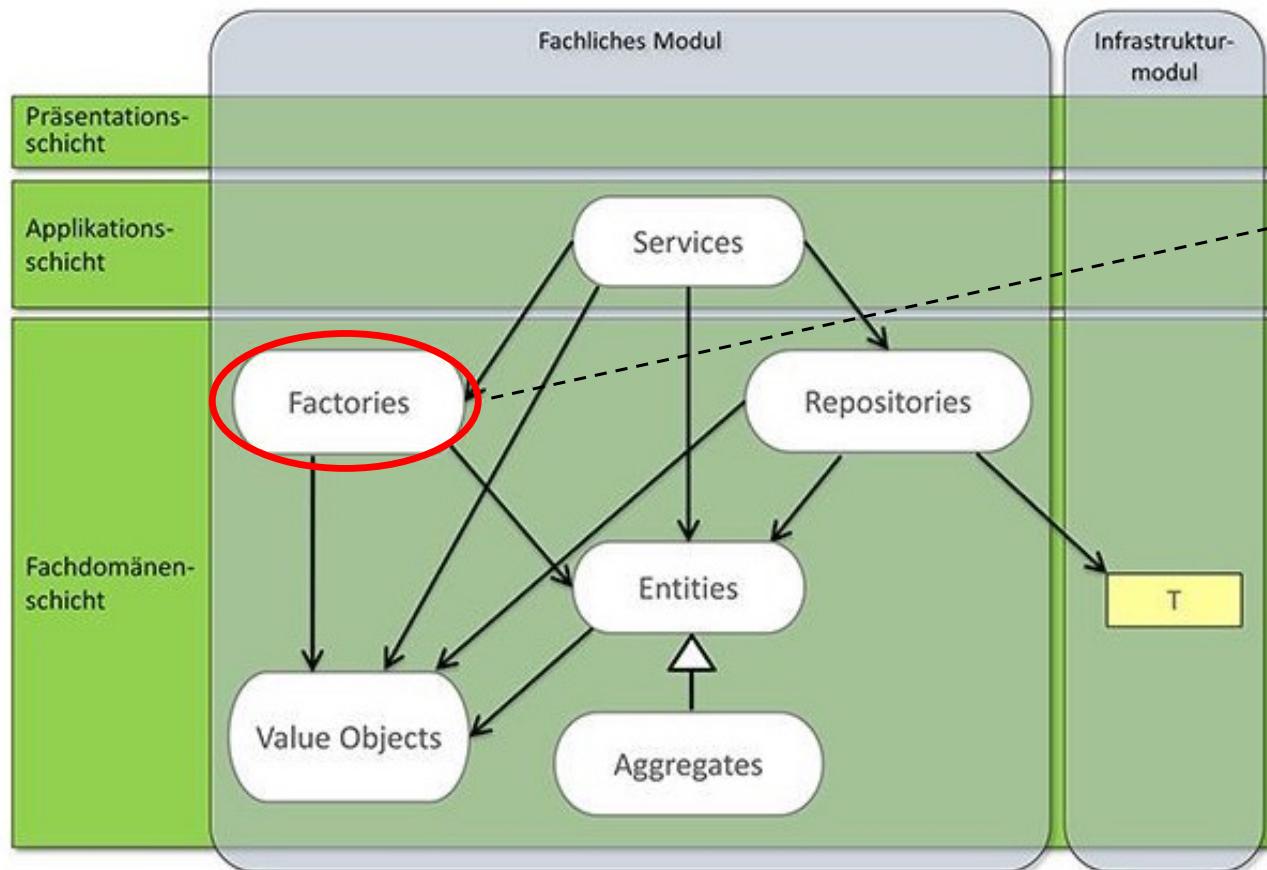
- Kapselung technischer Details der Infrastruktur
- Beschaffung von Objektreferenzen für Entitäten

Abbildung aus
[Lilienthal 2016]

Taktisches Design mit Building Blocks



Zum Nachlesen



Factory

- Erzeugung von Aggregates, Entitäten und Value Objects
- Nicht zwangsläufig Factory der GoF

Abbildung aus
[Lilienthal 2016]

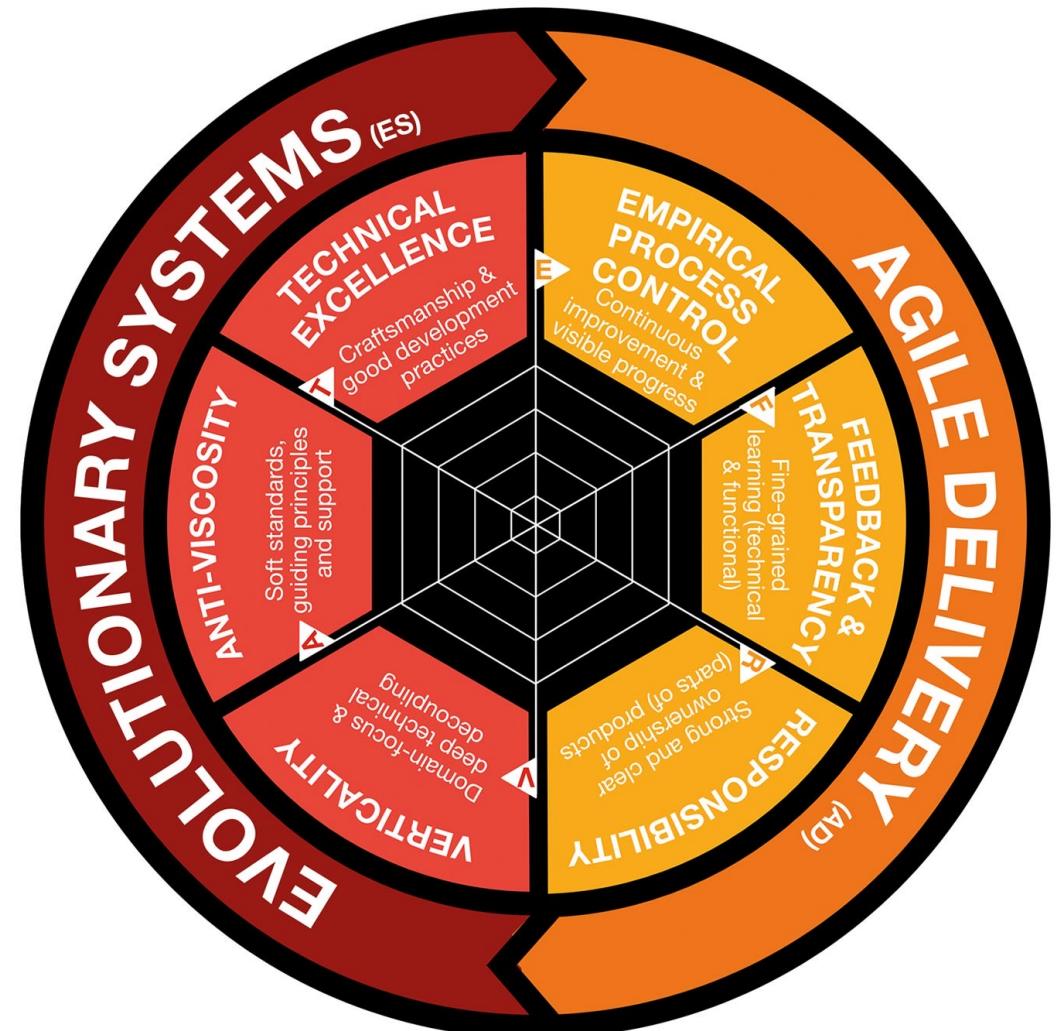
Evolutionäre Architektur

An evolutionary architecture supports
guided, incremental change
across multiple dimensions.

[Pearson, Ford, Kua 2017]

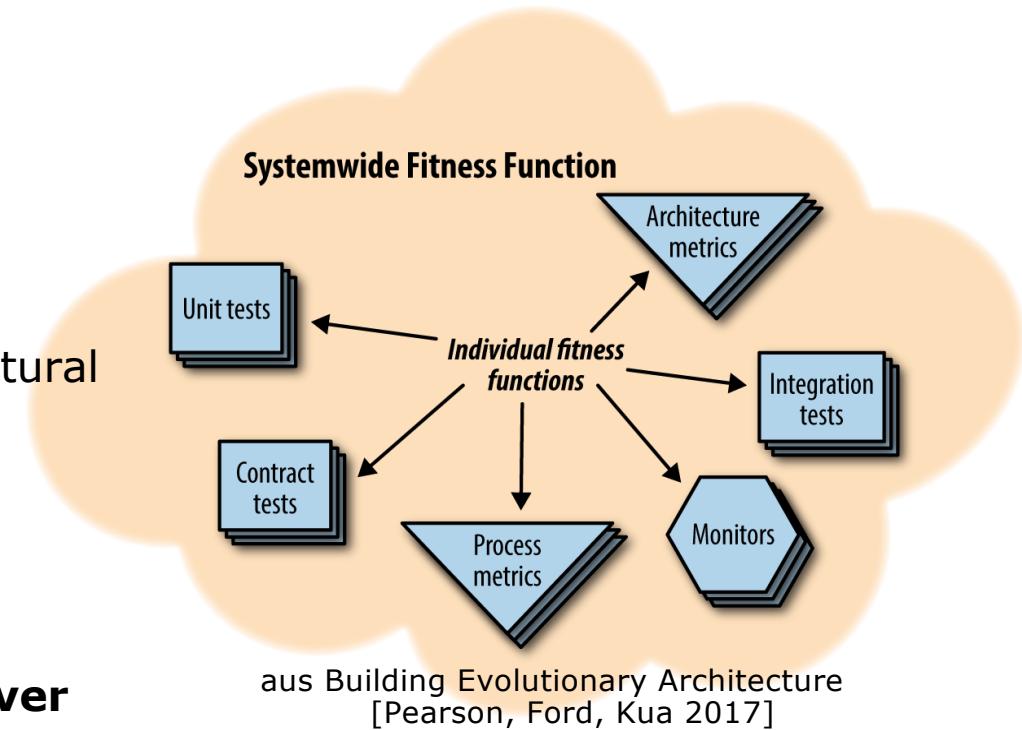
<https://kiosk.entwickler.de/java-magazin/java-magazin-11-2019/bewegung-im-architektur-genpool/>

<https://evolutionaryarchitecture.com/precis.html>

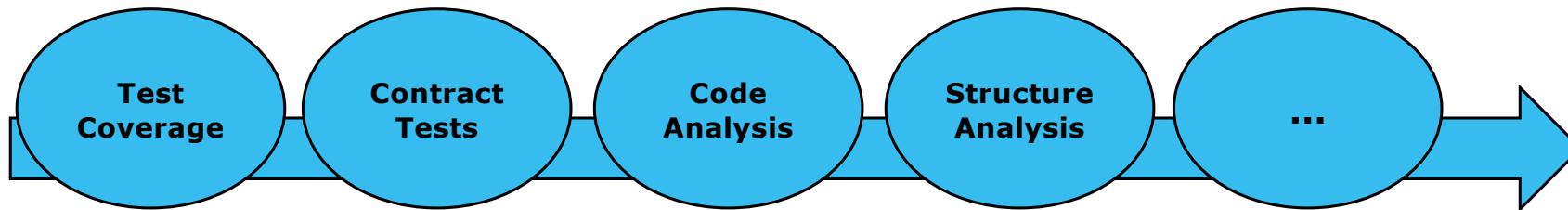


Evolutionäre Architektur und Fitness Functions

An architectural fitness function provides an objective integrity assessment of some architectural characteristic(s). [Pearson, Ford, Kua 2017]



Für jedes Inkrement wird sofort Feedback erhoben, hinsichtlich positiver oder negativer Auswirkung auf die Qualität!



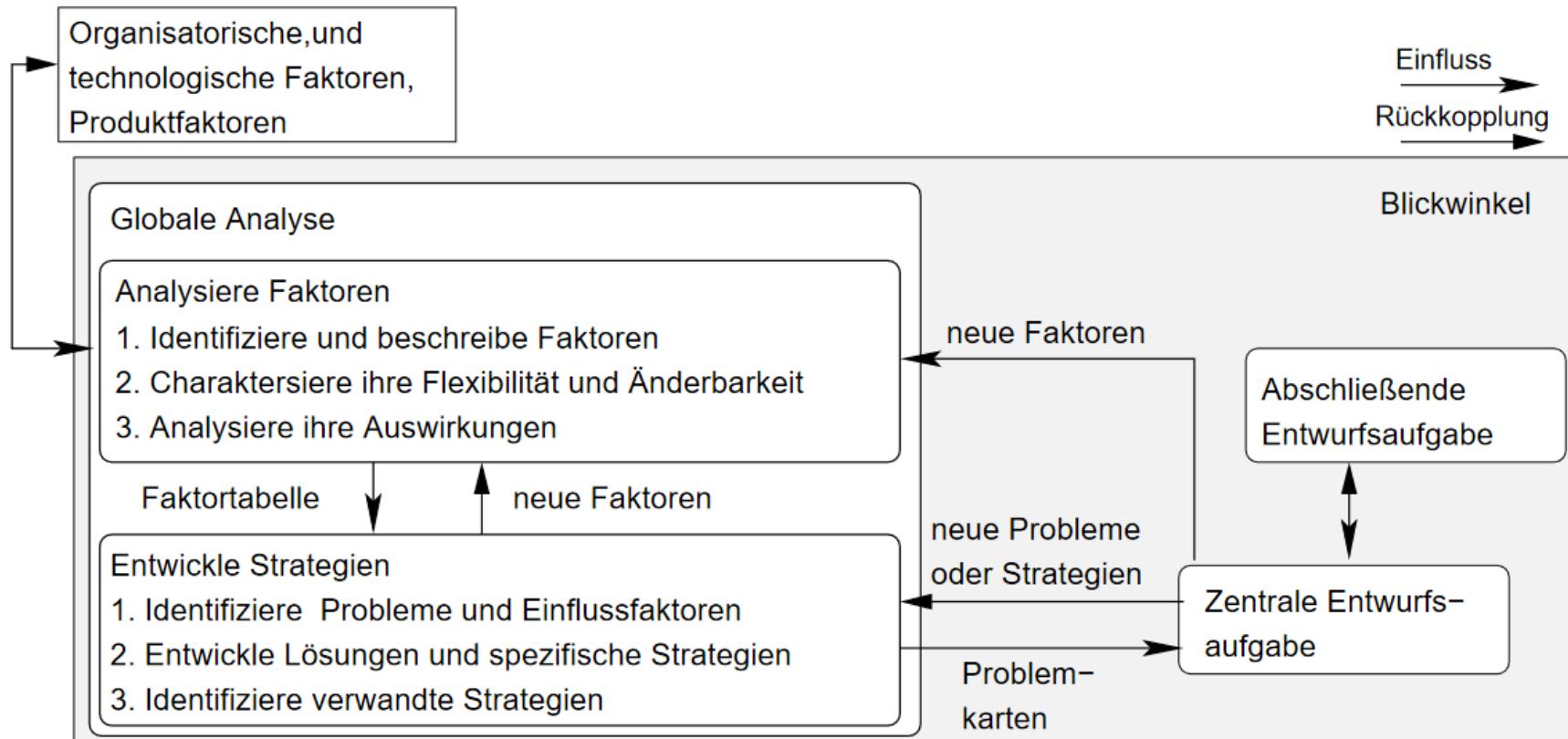
Evolutionäre Architektur und Fitness Functions



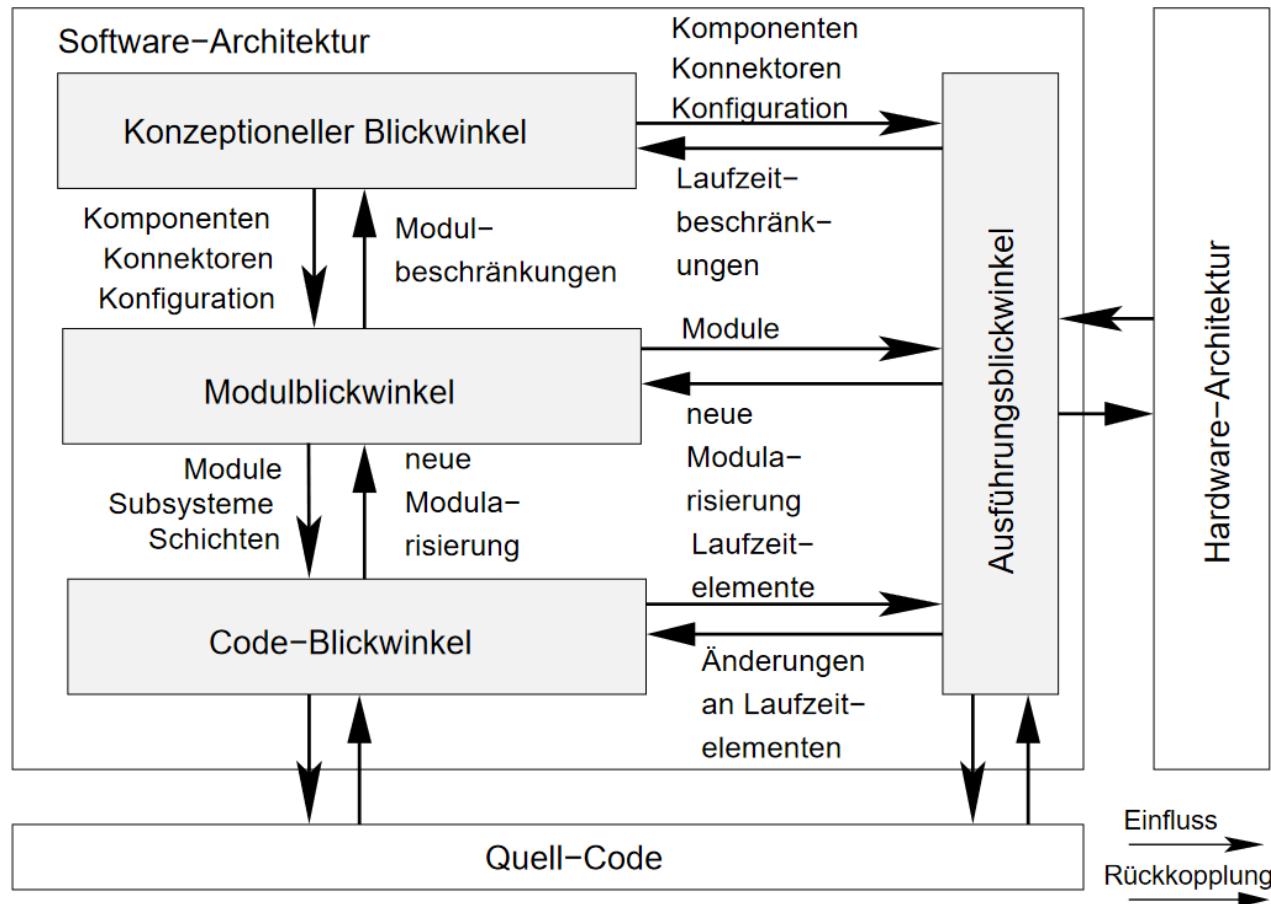
Zum Nachlesen

- (Neuer) Ansatz Architektur über die Zeit zu denken
(Architektur-, Qualität- und Zeitdimension)
- Fitness Functions sind Prüfungen und Messungen für Qualitätseigenschaften von Softwaresystemen
- Fitness Functions sind wiederkehrend, automatisiert und kleinteilig zu prüfen
- Fitness Functions müssen direktes Feedback hinsichtlich der Verbesserung oder Verschlechterung der Qualität aufgrund von Erweiterungen und Änderungen liefern
(Direktes Feedback bei jedem Inkrement)

Globale Analyse nach Hofmeister, Nord & Soni



Globale Analyse nach Hofmeister, Nord & Soni



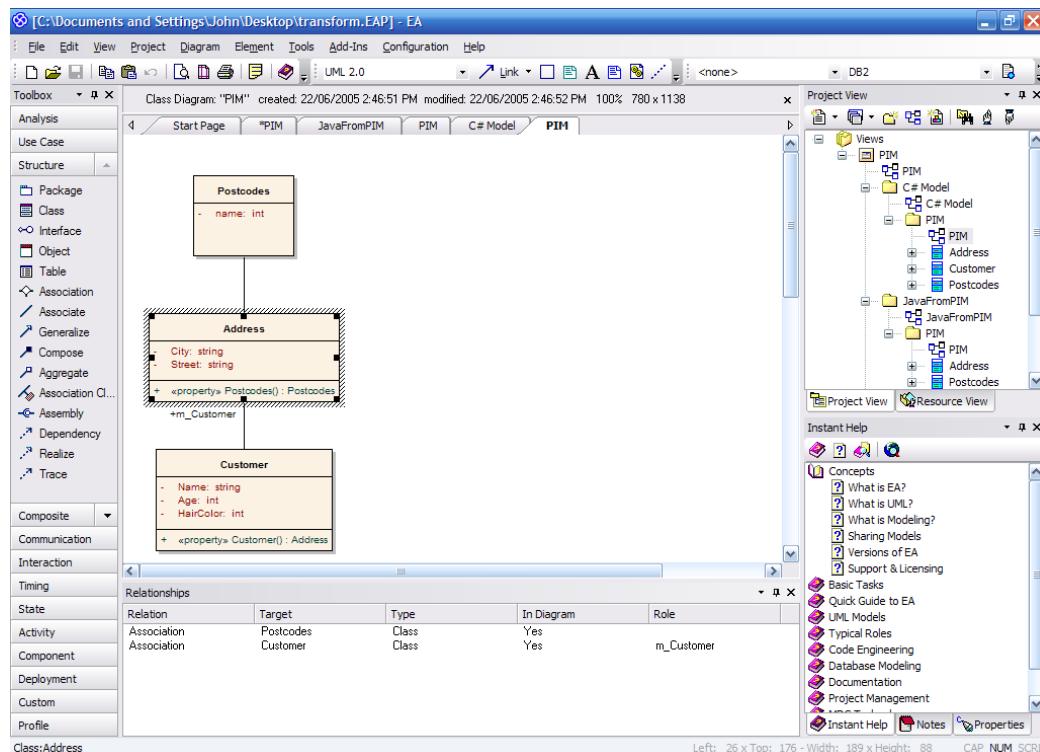
Model Driven Software Development

//Klassen werden generiert
public class Customer {

private String Name;
private int Age;
private int HairColor;
//Abhängigkeiten werden generiert
private Address address

//Methodenstubs werden generiert
public void addAddress() {
 // implement method stub
}

//Getter und Setter werden generiert
public String getName() {
 return this.Name;
}



Model Driven Software Development



Zum Nachlesen

- Geprägt von der Object Management Group (OMG) unter dem Namen Model Driven Architecture (MDA)
- Im Mittelpunkt steht die Entwicklung von Modellen mit UML oder domänenspezifischen Sprachen (DSL)
- Ausschließlich erfolgt die toolgestützte Transformation der Modelle in Quellcode (z.B. Klassen und Methoden bzw. Methodenhüllen)
- Auf dieser Basis erfolgt die Ausimplementierung der Methoden
- Es gibt aber auch vollständig ausführbare Modellsprachen (z.B. Executable UML der OMG)
- Z.B. Enterprise Architect unterstützt Codegenerierung von Java Klassen auf Basis eines UML Klassendiagramm

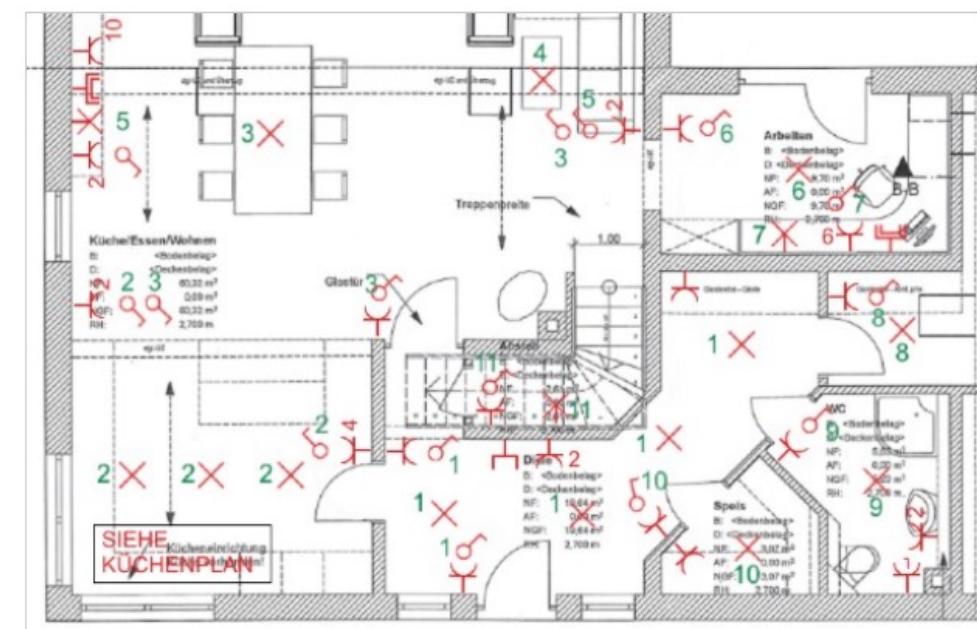
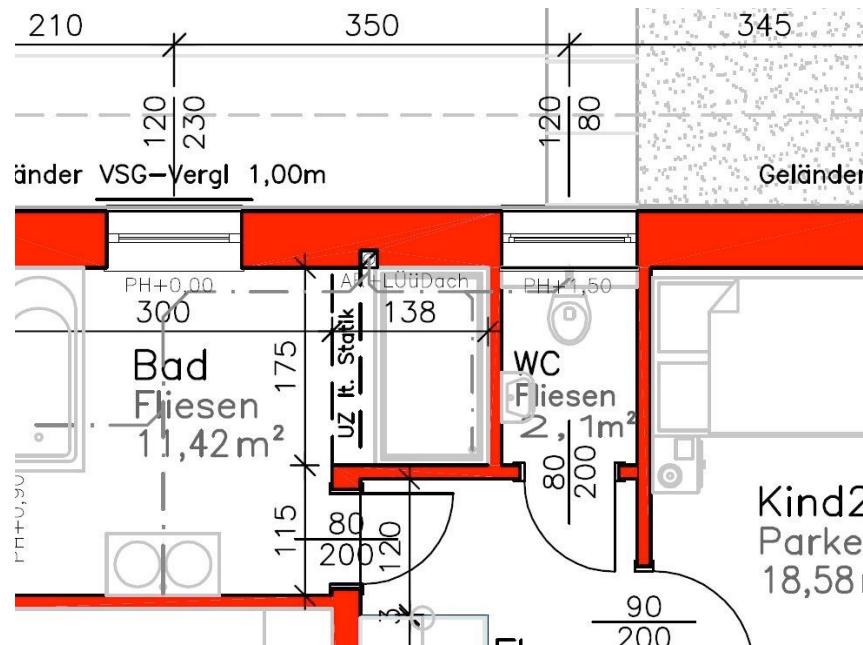
Diskussion

- Haben Sie bereits modellgetriebene Softwareentwicklung in Ihren Projekten eingesetzt?
- Wie waren Ihre Erfahrungen? Was hat gut und was hat weniger gut funktioniert?

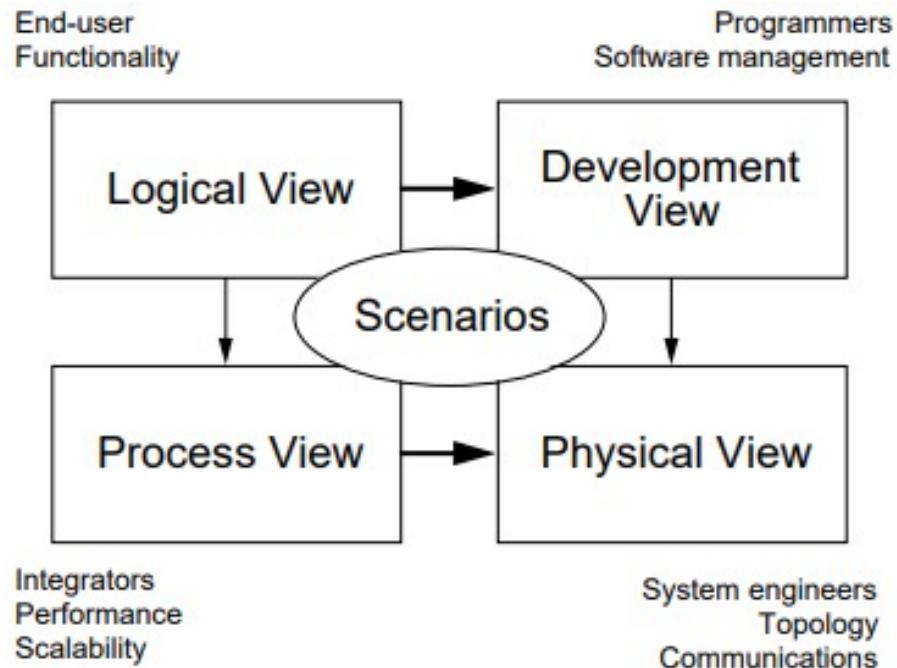
LZ 2-2: Softwarearchitekturen entwerfen (R1)

- Softwarearchitekt:innen können:
 - Softwarearchitekturen auf Basis bekannter funktionaler und Qualitätsanforderungen für nicht sicherheits- oder unternehmenskritische Softwaresysteme entwerfen und angemessen kommunizieren und dokumentieren
 - Strukturentscheidungen hinsichtlich Systemzerlegung und Bausteinstruktur treffen, dabei Abhängigkeiten zwischen Bausteinen festlegen
 - gegenseitige Abhängigkeiten und Abwägungen bezüglich Entwurfsentscheidungen erkennen und begründen
 - Begriffe *Blackbox* und *Whitebox* erklären und zielgerichtet anwenden
 - schrittweise Verfeinerung und Spezifikation von Bausteinen durchführen
 - Architektsichten entwerfen, insbesondere Baustein-, Laufzeit- und Verteilungssicht
 - die aus diesen Entscheidungen resultierenden Konsequenzen auf den Quellcode erklären
 - fachliche und technische Bestandteile in Architekturen trennen und diese Trennung begründen
 - Risiken von Entwurfsentscheidungen identifizieren.

Motivation Architektsichten



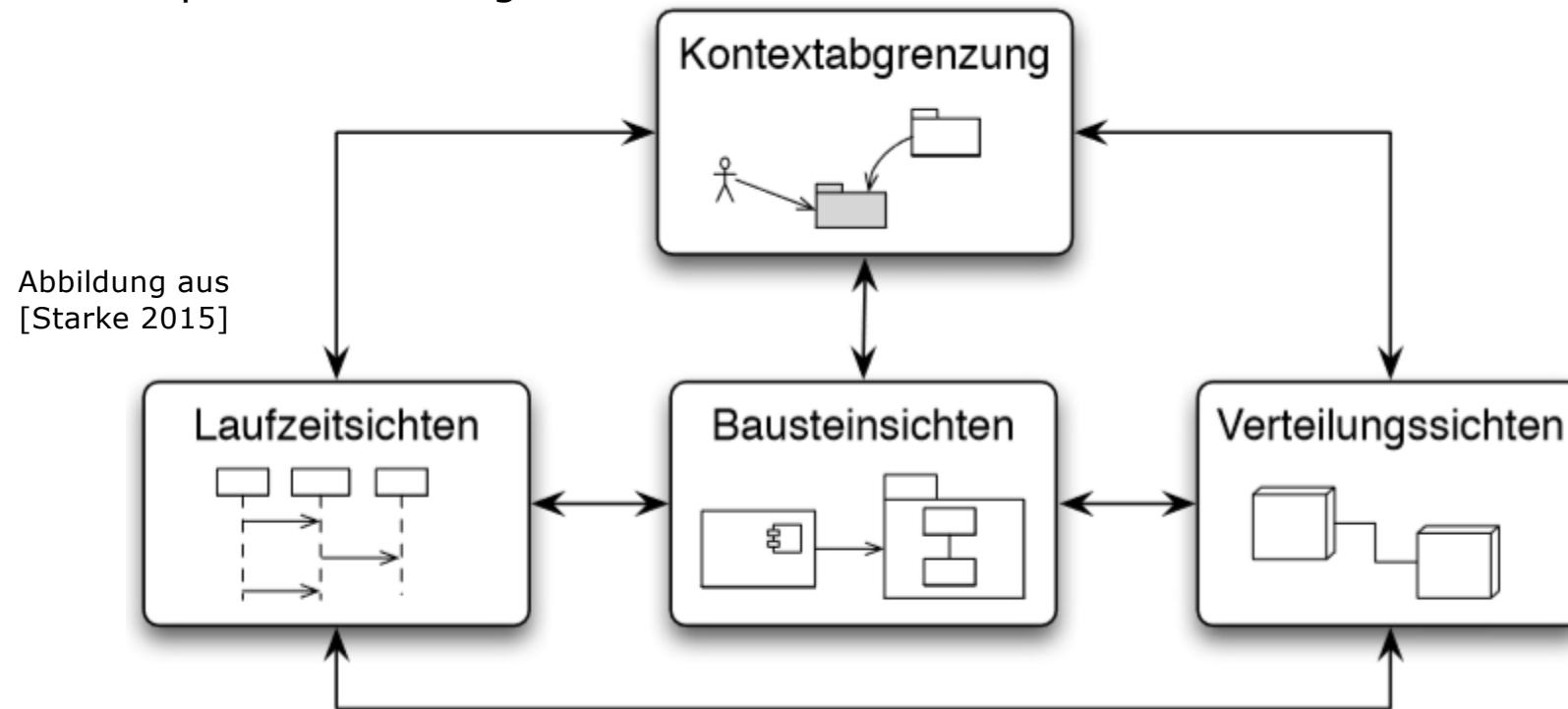
Abstraktion durch Architektsichten Bewährte Sichten nach iSAQB: Kruchten



<https://www.cs.ubc.ca/~gregor/teaching/papers/4+1view-architecture.pdf>

Abstraktion durch Architektsichten Bewährte Sichten nach iSAQB: arc42

Im Rahmen des iSAQB-Lehrplans werden vor allem vier Sichten behandelt, die als bewährt und praxisrelevant gelten.



Abstraktion durch Architektsichten

Sicht und Sichtenmodell

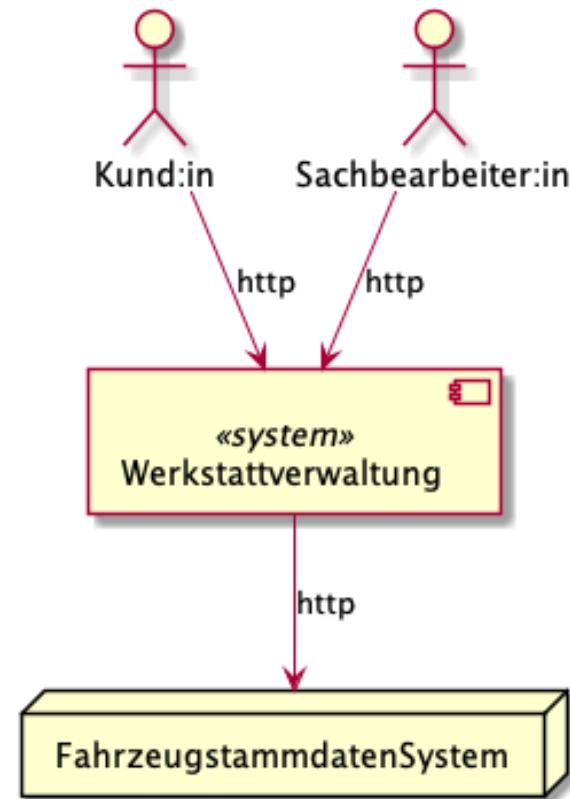


Zum Nachlesen

- Eine **Sicht** beschreibt eine Architektur von einem **bestimmten Standpunkt** aus
- Jede Sicht ist **unvollständig**, da nur Teilespekte des Systems beschrieben werden
- Eine Sicht **konzentriert** sich auf die **Architekturelemente ihrer Perspektive**
- Dadurch wird Komplexität reduziert und die Architektur eines Softwaresystems beherrschbar
- Die (sinnvolle) **Gruppierung** von zwei und mehreren Sichten wird **Sichtenmodell** genannt
- Eingesetzte Sichtenmodelle bestimmen wesentlich den Inhalt der Architekturdokumentation (siehe auch LZ 3)
- Zwischen Sichten eines Sichtenmodells existieren Wechselwirkungen. Dies bedeutet, dass der Architekt neben dem Wechselspiel der Architekturebenen auch ein Wechselspiel der Architektsichten hat.

Kontextabgrenzung

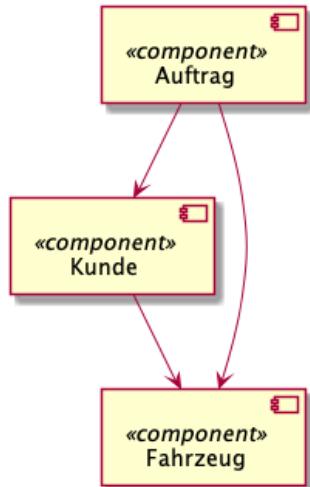
- Einbettung des Systems in eine Landschaft von IT-Systemen als Kommunikationspartner
- Interaktion mit Stakeholder
- Umgebende Infrastruktur
- UML Komponenten- und Verteilungsdiagramm



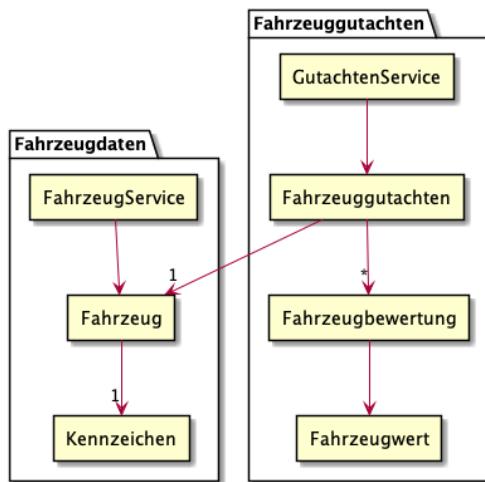
Bausteinsicht

- Statische Systemstruktur
- Subsysteme, Bausteine und deren Beziehungen
- Interner Aufbau des Systems und Organisation des Quellcode
- UML Komponenten- und Klassendiagramm

Architekturlevel 1

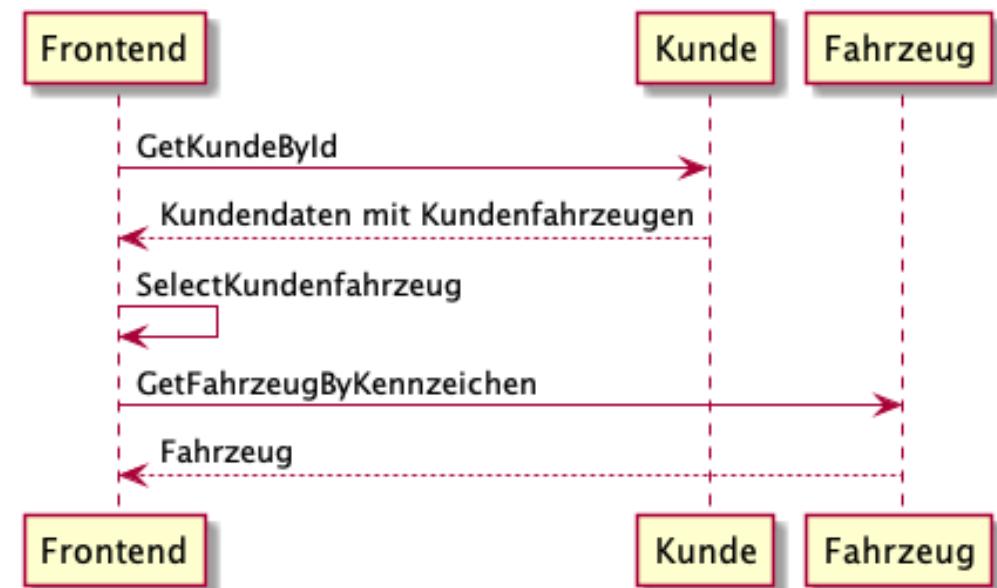


Architekturlevel 2 oder tiefer



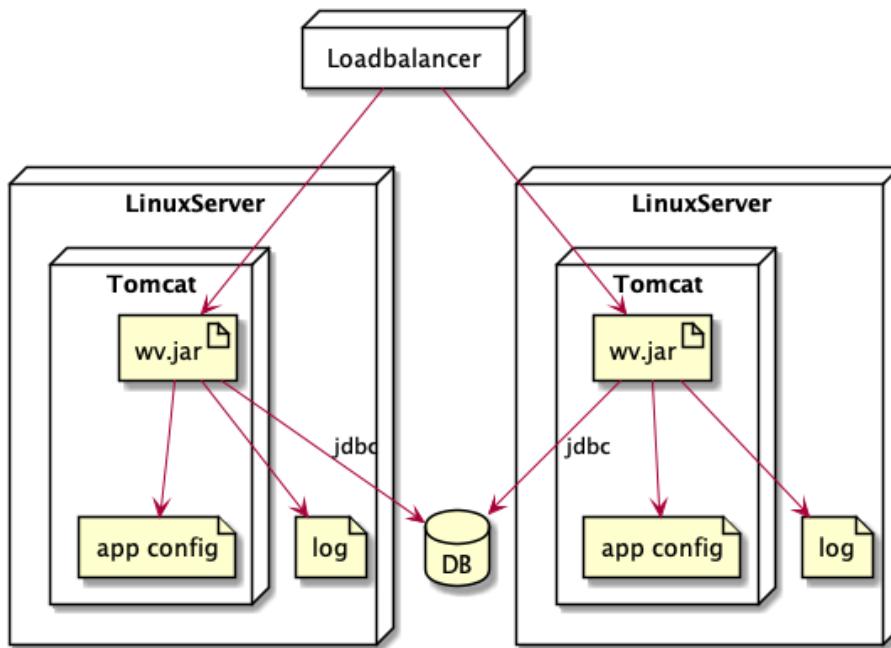
Laufzeitsicht

- Dynamische Strukturen
- Definition der zeitlichen Abfolge und des Datenflusses beim Zusammenwirken von zwei oder mehreren Subsystemen / Bausteinen
- UML Aktivitäts-, Sequenz-, Kommunikations- oder Kollaborationsdiagramm



Verteilungssicht

- Bereitstellung von Build-Artefakten auf Hardware
- Netzwerktopologie, Protokolle, Prozessoren, etc.
- UML Komponenten- und Verteilungsdiagramm



Weitere Sichten (nach iSAQB)

Datensicht

- Logische und technische Datenmodelle
- Kommunikationsdatenmodell
- UML Klassendiagramm, ER-Modell

Big Picture

- High-Level Architekturdiagramme zur Kommunikation mit der Managementebene
- UML Komponentendiagramme, Pfeile und Kästchen, PowerPoint

Geschäftsprozesssicht

- Geschäftsprozesse unterschiedlicher Abstraktionsebenen
- BPMN

Masken- oder Ablaufsicht

- Sequenz der Bildschirmmasken
- Ablaufdiagramm einer Website
- UML Aktivitätsdiagramm

Übung

- Identifizieren Sie die Komponenten / Domänen des Gesamtsystems (hierbei ist zwischen Bausteinen von Versicher24x7 und externen Bausteinen zu unterscheiden).
- Entwerfen sie die Blackbox Sicht für die Komponente Tarifierung
- Entwerfen Sie eine Whitebox Sicht für die Komponente Bestandssystem.
- Erläutern Sie das Zusammenspiel der Komponenten am Beispiel der Tarifierung eines Angebots.

LZ 2-3: Einflussfaktoren auf Softwarearchitektur erheben und berücksichtigen können (1) (R1-R3)

Softwarearchitekt:innen können Einflussfaktoren (Randbedingungen) als Einschränkungen der Entwurfsfreiheit erarbeiten und berücksichtigen. Sie verstehen, dass ihre Entscheidungen weitere Anforderungen und Einschränkungen für das zu entwerfende System, seine Architektur oder den Entwicklungsprozess mit sich bringen können.

Sie erkennen und berücksichtigen den Einfluss von:

- produktbezogenen Faktoren wie (R1)
 - funktionale Anforderungen
 - Qualitätsanforderungen und Qualitätsziele
 - zusätzliche Faktoren wie Produktkosten, beabsichtigtes Lizenzmodell oder Geschäftsmodell des Systems

Siehe LZ 1

LZ 2-3: Einflussfaktoren auf Softwarearchitektur erheben und berücksichtigen können (2) (R1-R3)

- technischen Faktoren wie (R1-R3)
 - extern beauftragte technische Entscheidungen und Konzepte (R1)
 - bestehende oder geplante Hardware- und Software-Infrastruktur (R1)
 - technologische Beschränkungen für Datenstrukturen und Schnittstellen (R2)
 - Referenzarchitekturen, Bibliotheken, Komponenten und Frameworks (R1)
 - Programmiersprachen (R3)
 - organisatorischen Faktoren wie
 - Organisationsstruktur des Entwicklungsteams und des Kunden (R1)
 - Unternehmens- und Teamkultur (R3)
 - Partnerschaften und Kooperationen (R2)
 - Normen, Richtlinien und Prozessmodelle (z.B. Genehmigungs- und Freigabeprozesse) (R2)
 - Verfügbarkeit von Ressourcen wie Budget, Zeit und Personal (R1)
 - Verfügbarkeit, Qualifikation und Engagement von Mitarbeitenden (R1)

LZ 2-3: Einflussfaktoren auf Softwarearchitektur erheben und berücksichtigen können (3) (R1-R3)

- regulatorischen Faktoren wie (R2)
 - lokale und internationale rechtliche Einschränkungen
 - Vertrags- und Haftungsfragen
 - Datenschutzgesetze und Gesetze zum Schutz der Privatsphäre
 - Fragen der Einhaltung oder Verpflichtungen zur Beweislast
- Trends wie (R3)
 - Markttrends
 - Technologietrends (z.B. Blockchain, Microservices)
 - Methodik-Trends (z.B. agil)
 - (potenzielle) Auswirkungen weiterer Stakeholderinteressen und vorgegebener oder extern festgelegter Designentscheidungen (R3)

LZ 2-4: Querschnittskonzepte entwerfen und umsetzen (R1)

- Softwarearchitekt:innen können:
 - die Bedeutung von Querschnittskonzepten erklären
 - Querschnittskonzepte entscheiden und entwerfen, beispielsweise Persistenz, Kommunikation, GUI, Fehlerbehandlung, Nebenläufigkeit
 - mögliche wechselseitige Abhängigkeiten dieser Entscheidungen erkennen und beurteilen.
- Softwarearchitekt:innen wissen, dass solche Querschnittskonzepte systemübergreifend wiederverwendbar sein können

Querschnittskonzepte

= systemübergreifende Konzepte

- Können wiederverwendet werden und sind i.d.R. einheitlich
- Nicht alle Konzepte immer benötigt
- Wirken über mehrere Teile oder die gesamte Architektur hinweg
- Konzepte sind oft orthogonal, aber es gibt auch Abhängigkeiten
- Architekt betrachtet das Konzept, nicht zwangsläufig die konkrete Umsetzung



57

GUI



Monitoring



Fehlerbehandlung



Persistenz

Technische und querschnittliche Architekturkonzepte

- Gelten Baustein-übergreifend, für das ganze System oder System-übergreifend
- Die Implementierungstechnologie beeinflusst die Beschreibung der technischen Konzepte (JEE / 2PC)
- Konzepte prägen die Bausteinstruktur und deren Implementierung (z.B. Persistenz, Internationalisierung, Geschäftsprozesssteuerung mit BPM Tool)
- Für ein System sind nicht alle dargestellten Themen relevant und abhängig vom System sind die Themen unterschiedlich wichtig

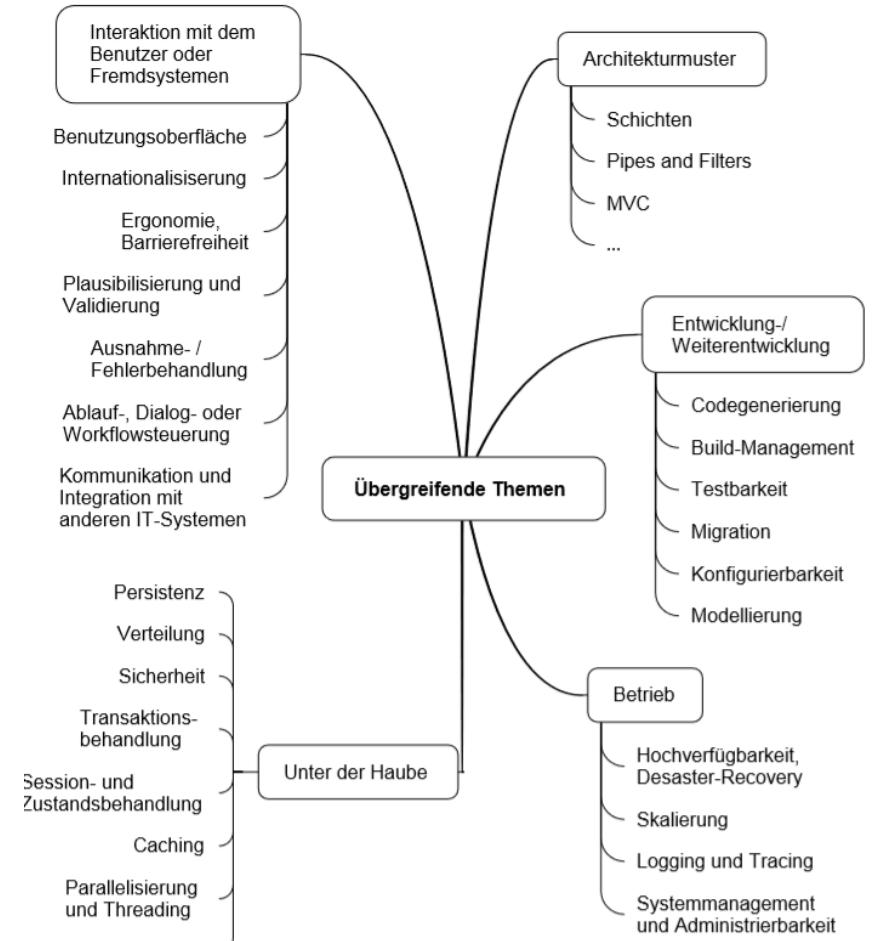
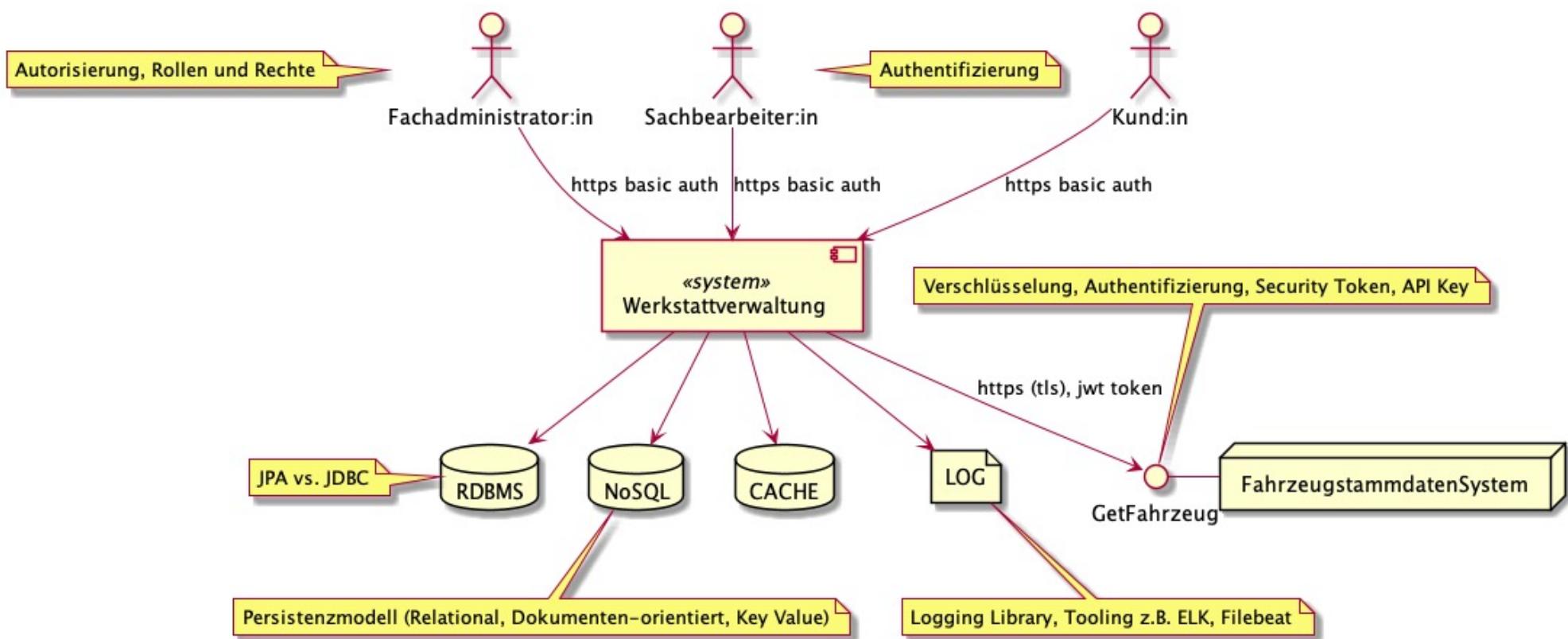
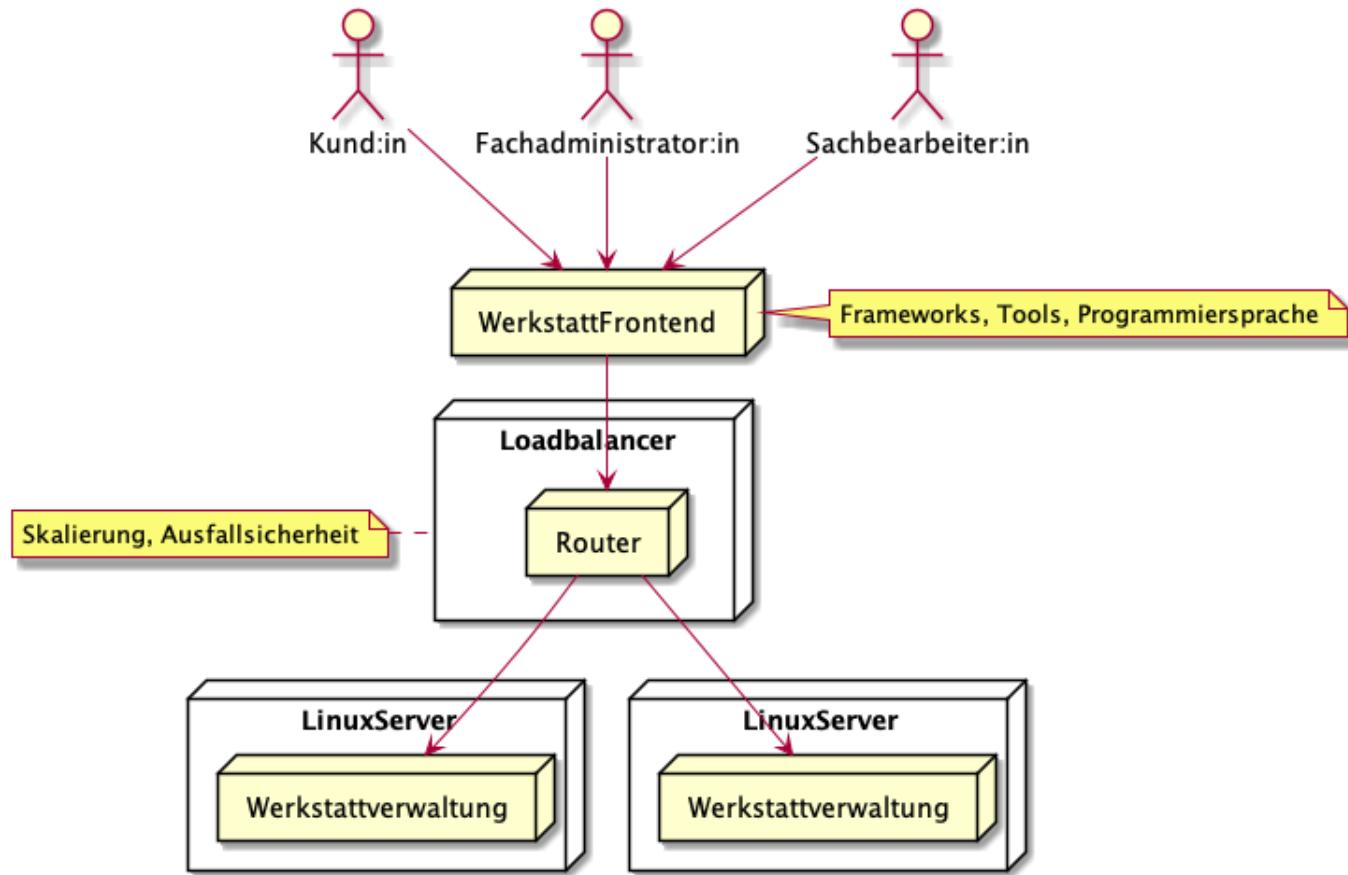


Abbildung aus [Zörner 2015]

Querschnittskonzepte in der Architektur

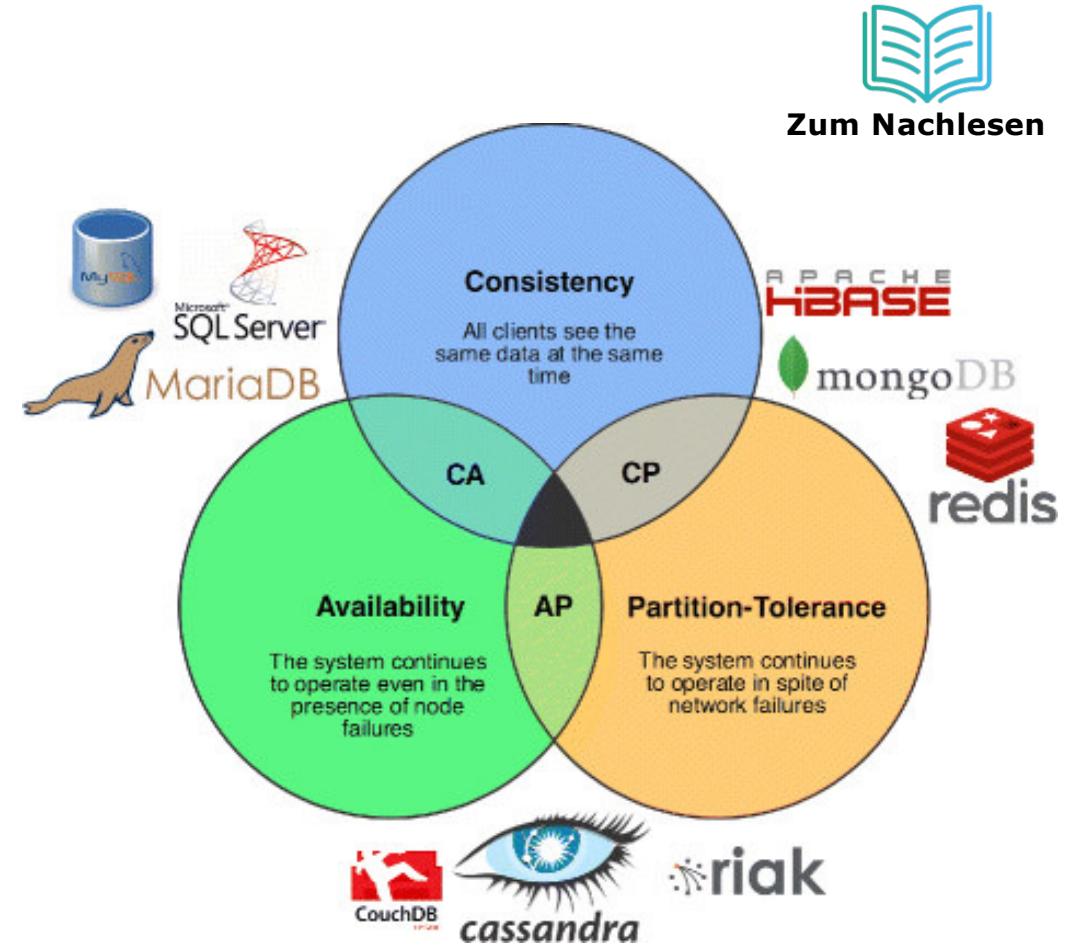


Querschnittskonzepte in der Architektur



Persistenz

- **Speichermodell**
SQL vs. NoSQL
- **Datenkonsistenz und Verfügbarkeit**
CAP-Theorem, ACID, BASE
- **Art des Datenbankzugriffs**
JPA vs. JDBC



<https://www.slideshare.net/ajung/why-we-love-arangodb-the-hunt-for-the-right-nosql-database>

Integration und Kommunikation



Zum Nachlesen

Backend Integration

- **Architekturstil**
Event-driven vs. Resource-oriented
vs. Message-oriented / Synchron vs.
Asynchron
- **Technologie**
SOAP vs. REST / REST vs. GraphQL /
Kafka vs. Rabbit MQ
- **Enterprise Application
Integration Patterns**
<https://www.enterpriseintegrationpatterns.com/>

Frontend Integration

- Clientseitige Frontend Integration
(Redirects, Links, JavaScript Modul
Systeme)
- Serverseitige Frontend Integration

GUI



Zum Nachlesen

- **Architekturmuster**
Single Page Application vs. Multi Page Anwendung /
Rich Client vs. Thin Client
- **Technologie und Framework**
Komponentenorientiert (JSF) vs. aktionsbasierte Frameworks
(Spring Thymeleaf) / JavaScript Framework (Angular, React.js,
Vue.js)
- **Usability**
Navigierbarkeit, intuitive Nutzbarkeit und Bedienelemente
- **Barrierefreiheit**
Schrift, Farben, Kontraste, Screen Reading, etc.

Sicherheit



Zum Nachlesen

- **Authentifizierung**
Username und Passwort, OAuth2, OpenID, SAML
- **Autorisierung**
Administratoren, Rollen und Rechte, Role based access control pattern, Attribute based access control, Frameworkunterstützung (Angular Guards, Annotations-basiert in Spring und Java EE), OAuth2
- **Datenklassifikation** und Umgang mit **sicherheitsrelevanten Informationen**
Öffentlich, Vertraulich, Streng Vertraulich, Kryptografie & Verschlüsselung, Technische User für Datenbanken und Tools
- **Web Application Security**
OWASP, statische und dynamische Sicherheitsprüfungen (z.B. Fortify Static Code Analyzer), Sicherheitsaudits

Basis Sicherheitsziele



Zum Nachlesen

Authentifizierung (authentication)

Feststellung der Absenderidentität

Autorisierung (authorization)

Einräumung von Nutzungsrechten anhand der Identität

Integrität (integrity)

Erkennung der Manipulationen von gesicherten Daten

Vertraulichkeit (confidentiality)

Daten unzugänglich für Unbefugte übertragen und speichern

Unleugbarkeit (non-repudiation)

Eingegangene Nachrichten können nicht vom Sender abgestritten werden.

Verfügbarkeit (availability)

Maßnahmen gegen unvorhergesehene oder mutwillige Systemstörung

aus [Gharbi 2018]

Siehe auch Schutzziele der Informationssicherheit in
[IT-Sicherheit: Konzepte - Verfahren – Protokolle; Eckert, Claudia,
9. Auflage, Oldenburg 2014]

Logging



Zum Nachlesen

- **Nachvollziehbarkeit** des Laufzeitverhaltens ermöglichen
- **Effiziente Auswertbarkeit** der protokollierten Informationen
Zugang über Frontend ermöglichen / Aussagekraft und analysierbares Mengengerüst der Informationen sicherstellen / Durchsuch- und Sortierbarkeit ermöglichen
- **Archivierung** von Informationen z.B. Online Vertragsabschlüsse
- **Logging Frameworks und Tooling**
log4J / slf4J / ngx-logger / ElasticSearch, Logstash & Kibana

Was ging schief?
Wo ist es passiert?
Warum ging es schief?

Logging Best Practices



Zum Nachlesen

- Fachliche Informationen von technischen Informationen (z.B. Exceptions) trennen
- Implementierungs-invasive Realisierung mit Aspektorientierung
- Performance-optimierte Verarbeitung von fachlichen und technischen Logs mit Log Appendern
 - Fire and Forget für Informationen dessen Verlust verkraftet werden kann, garantie Protokollierung bei Fehlerlogs

LZ 2-6: Entwurfsprinzipien erläutern und anwenden (R1-R3) (1)

Softwarearchitekt:innen sind in der Lage zu erklären, was Entwurfsprinzipien sind. Sie können deren grundlegende Ziele und deren Anwendung im Hinblick auf Softwarearchitektur skizzieren. (R2)

Mit einer Prüfungsrelevanz, die jeweils von dem unten aufgeführten konkreten Prinzip abhängt, sind Softwarearchitekt:innen in der Lage:

- die unten aufgeführten Gestaltungsprinzipien zu erläutern und mit Beispielen zu illustrieren
- zu erklären, wie diese Prinzipien angewendet werden sollen
- darzulegen, wie Qualitätsanforderungen die Anwendung dieser Prinzipien beeinflussen
- die Auswirkungen der Entwurfsprinzipien auf die Implementierung zu erläutern
- Quellcode- und Architekturentwürfe zu analysieren, um zu beurteilen, ob diese Entwurfsprinzipien angewendet wurden oder angewendet werden sollten

LZ 2-6: Entwurfsprinzipien erläutern und anwenden (R1) (2)

Abstraktion (R1)

- im Sinne eines Vorgehens zur Erarbeitung zweckmäßiger Generalisierungen
- als ein Entwurfskonstrukt, bei dem die Bausteine von Abstraktionen und nicht von Implementierungen abhängen
- Schnittstellen als Abstraktionen

LZ 2-6: Entwurfsprinzipien erläutern und anwenden (R1) (3)

Modularisierung (R1)

- Geheimnisprinzip (Information Hiding) und Kapselung (R1)
- Trennung von Verantwortlichkeiten (Separation of Concerns - SoC) (R1)
- Lose, aber funktionell ausreichende Kopplung (R1) von Bausteinen, siehe LG 2-7
- Hohe Kohäsion (R1)
- SOLID-Prinzipien (R1-R3), soweit sie auf architektonischer Ebene von Relevanz sind:
 - S: Single-Responsibility-Prinzip (R1) und seine Beziehung zu SoC
 - O: Offen/geschlossen-Prinzip (R1)
 - L: Liskovsches Substitutionsprinzip (R3) als eine Möglichkeit, Konsistenz und konzeptionelle Integrität beim objektorientierten Design zu erreichen
 - I: Interface-Segregation-Prinzip (R2) und seine Beziehung zu Lernziel 2-9 "Schnittstellen entwerfen und festlegen"
 - D: Dependency-Inversion-Prinzip (R1) - Umkehrung von Abhängigkeiten (R1) durch Schnittstellen oder ähnlichen Abstraktionen

LZ 2-6: Entwurfsprinzipien erläutern und anwenden (R1) (4)

Konzeptionelle Integrität (R2)

- bedeutet Einheitlichkeit (Homogenität, Konsistenz) von Lösungen für ähnliche Probleme zu erreichen (R2)
- als ein Mittel um Prinzips der geringsten Überraschung zu erreichen (principle of least surprise) (R3)

Einfachheit (R1)

- als Mittel zur Verringerung von Komplexität (R1)
- als Motiv der Prinzipien KISS (R1) und YAGNI (R2)

Erwarte Fehler (R1-R2)

- als Mittel für den Entwurf robuster und widerstandsfähiger Systeme (R1)
- als eine Verallgemeinerung des Robustheitsgrundsatzes (Postel's law) (R2)

Prinzipien des Software Designs

Prinzipien sind

- Bewährte und erprobte Grundsätze (Best-Practices) für die Anwendung auf Entwurfsfragen
- Handlungsmaxime bei der hierarchischen (De-)Komposition
- Effektives Mittel für die Reduktion der Komplexität und die Erhöhung der Flexibilität einer Architektur
- Werden angewendet auf Bausteine (Komponente, Package, Subsystem) und Klassen
- Es gibt allgemeine und abgeleitete Prinzipien

Übersicht Entwurfsprinzipien

Abstraktion

Schnittstelle als Abstraktion
Blackbox & Whitebox

Konzeptionelle Integrität

Homogenität & Konsistenz
in der Lösung
Principle of least surprise

Erwarte Fehler

Postel's Law
Robustheit

Modularisierung

Information Hiding
Lose Kopplung & hohe Kohäsion
Kapselung
SOLID

Einfachheit

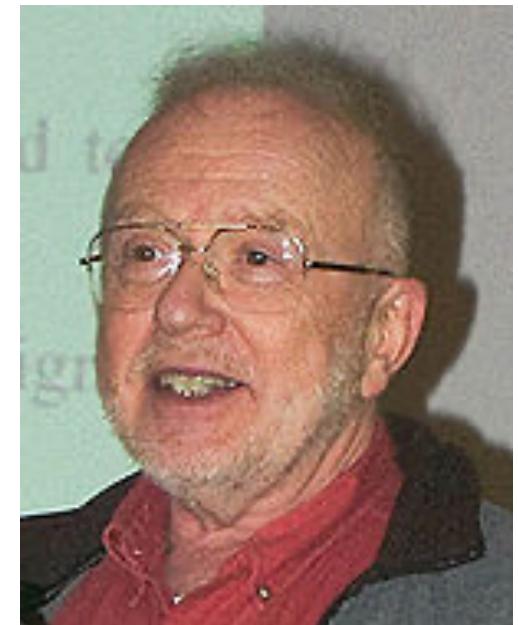
KISS
YAGNI

Modularisierung

Bausteine definieren und gestalten

Kapselung und Information Hiding

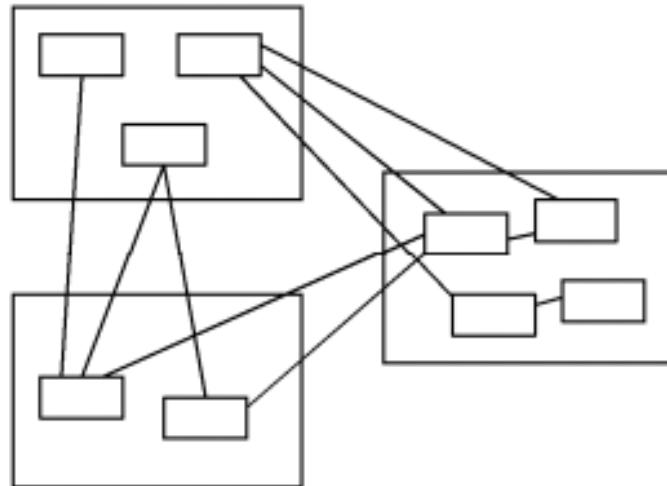
- Modularisierung wurde in den frühen 70er-Jahren von u.a. David Parnas beschrieben. Kernaussagen:
 - Anstreben von **loser Kopplung** und **hoher Kohäsion**
 - **Verberge Implementierungsdetails** gegenüber den Verwendern (**Geheimnisprinzip, Kapselung** und **Information Hiding**)
- **Minimiert die Komplexität** indem **klare Grenzen** und ein **Verantwortungsbereich** zwischen den Teilen eines Systems gezogen werden
 - Zwischen diesen Teilen werden klare, **schmale Schnittstellen** definiert
- Sauber modularisierte Systeme sind **einfacher zu warten** und zu **erweitern** da Änderungen lokal begrenzt sind und Fehler besser zu lokalisieren sind sowie das Gesamtsystem nur in geringerem Maße treffen



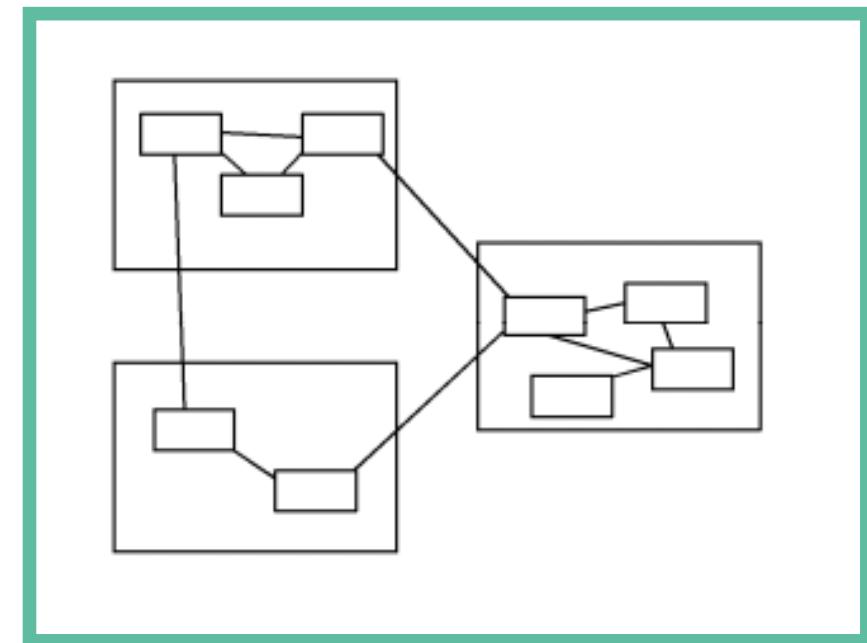
David Parnas. Source: Wikipedia

Lose Kopplung und hohe Kohäsion

Enge Kopplung und
niedrige Kohäsion



Lose Kopplung und
hohe Kohäsion



Abbildungen aus
[Vogel 2009]

Lose Kopplung und hohe Kohäsion



Zum Nachlesen

Kopplung

- Maß für den Stärkegrad der Abhängigkeit(en) zwischen Bausteinen
- Änderung der Schnittstelle eines Bausteins führt zu Änderungen in den nutzenden Bausteinen
- Lose Kopplung fördert Einfachheit, Wartbarkeit und Flexibilität
- Enge Kopplung bedeutet hohe Komplexität und Starrheit
 - Bausteine können nur in Verbindung mit anderen Bausteinen verstanden werden
 - Änderungen verursachen Seiteneffekte
 - Erweiterbarkeit und Wartbarkeit nimmt ab

Lose Kopplung und hohe Kohäsion



Zum Nachlesen

Kohäsion

- Zusammenhangskraft / Zusammenhängen
- Ein kohärenter Baustein / eine kohärente Klasse löst ein Problem
- Was dafür benötigt wird (Klassenverbund bei einem Baustein bzw. Methodenverbund bei einer Klasse) sollte eng zusammenhängen (inhaltliche und logische Zusammengehörigkeit)
- Eine lose Kopplung führt i.d.R. zu einer hohen Kohäsion

SOLID sollte jeder Architekt kennen!

S

SRP: Single Responsibility Principle: Das erkennen Sie wieder – wir haben das weiter vorne bereits als „Trenne Verantwortlichkeiten“ (Separation of Concerns) kennengelernt. Eine Klasse soll nur genau einen Grund haben, geändert zu werden.

O

OCP: Open-Close Principle: Sie sollten Klassen erweitern können, ohne bestehenden Code zu modifizieren (siehe Abschnitt 4.1.3.1).

L

LSP: Liskov Substitution Principle: Unterklassen müssen grundsätzlich für ihre Oberklassen eingesetzt werden können: wenn schon Vererbung, dann richtig (siehe Abschnitt 4.1.3.2).

I

ISP: Interface Segregation Principle: Verwenden Sie feingranulare, client-spezifische Schnittstellen statt ein großes Universalinterface (siehe Abschnitt 4.3.1.3).

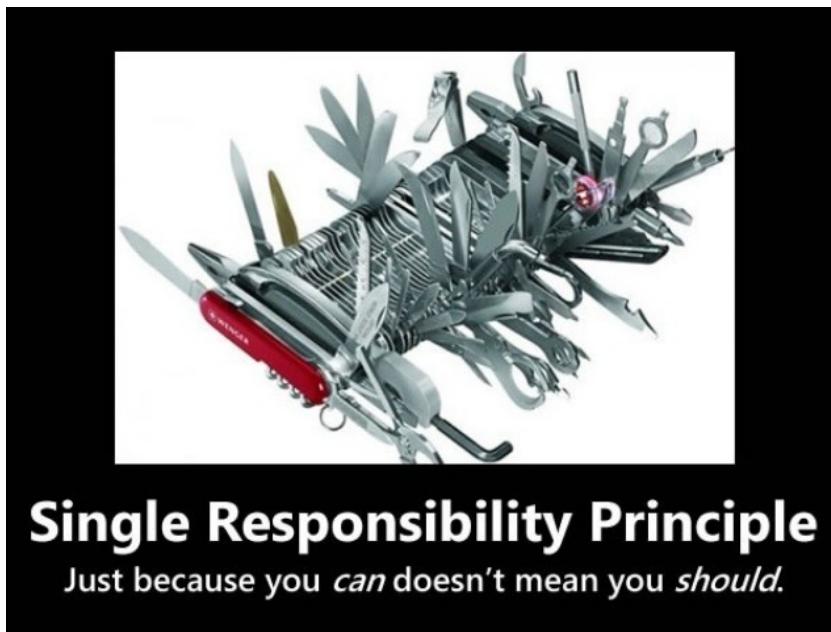
D

DIP: Dependency Inversion Principle: Erlaube Abhängigkeiten von Abstraktionen, nicht von konkreten Implementierungen (siehe Abschnitt 4.3.1.4).

Aus [Starke 2015]

Single Responsibility

- Ein Baustein / Klasse sollte nur eine Verantwortung haben
- Eine Technische Verantwortung z.B. **ein Controller** für alles entspricht nicht diesem Prinzip



**There should never be
more than one reason
for a class to change!**

Tom DeMarco &
Meilir Page-Jones

<https://blogs.msdn.microsoft.com/cdndev/s/2009/07/15/the-solid-principles-explained-with-motivational-posters/>

Separation of Concerns

- Trenne verschiedene Aspekte eines Problems und behandle jedes Teilproblem für sich
- Bausteine benötigen spezifische Verantwortungen, die explizit genannt werden müssen
Kontoverwaltung, Personverwaltung
- Wird angewendet bei der hierarchischen (De-)Komposition

Separation of Concerns Vertiefung



Zum Nachlesen

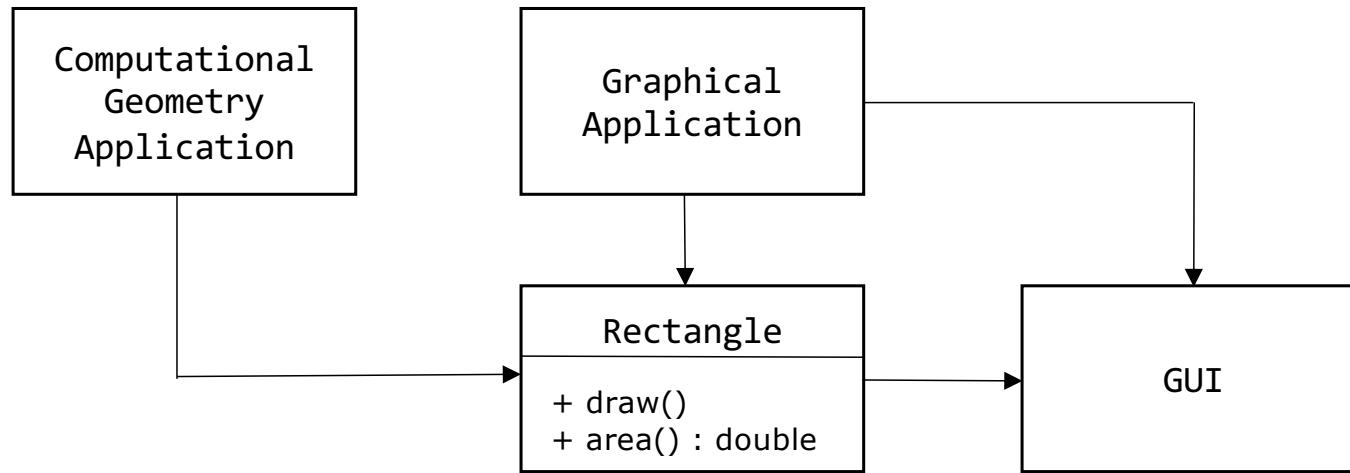
Separation of Concerns führt zu Grundprinzipien, auf denen eine Architektur aufgebaut sein sollte

- Objekt-orientierte Programmierung mit den Prinzipien der Kapselung (Information Hiding), bewusstem verstecken von Daten und definierten (public) Interfaces
- Aspekt-orientierte Programmierung in Form von Auslagerung von Aspekten, so dass diese einfach Objekten hinzugefügt bzw. weggenommen werden können
- Komponenten-orientierte sowie service-orientierte Architekturen, so dass größere Funktionsblöcke mit ggf. plattformunabhängigen Schnittstellen gekapselt werden
- Trennung einer Anwendung in verschiedene Schichten mit spezifischen Funktionalitäten (Schichtenarchitektur)
- Lose Kopplung durch architektonische Maßnahmen, den Einsatz von Middleware (Messaging-Systeme) sowie unabhängige Dienste.

Bilde Komponenten, die funktional voneinander getrennt sind und über klare, schmale Schnittstellen kommunizieren

Separation of Concerns & Single Responsibility

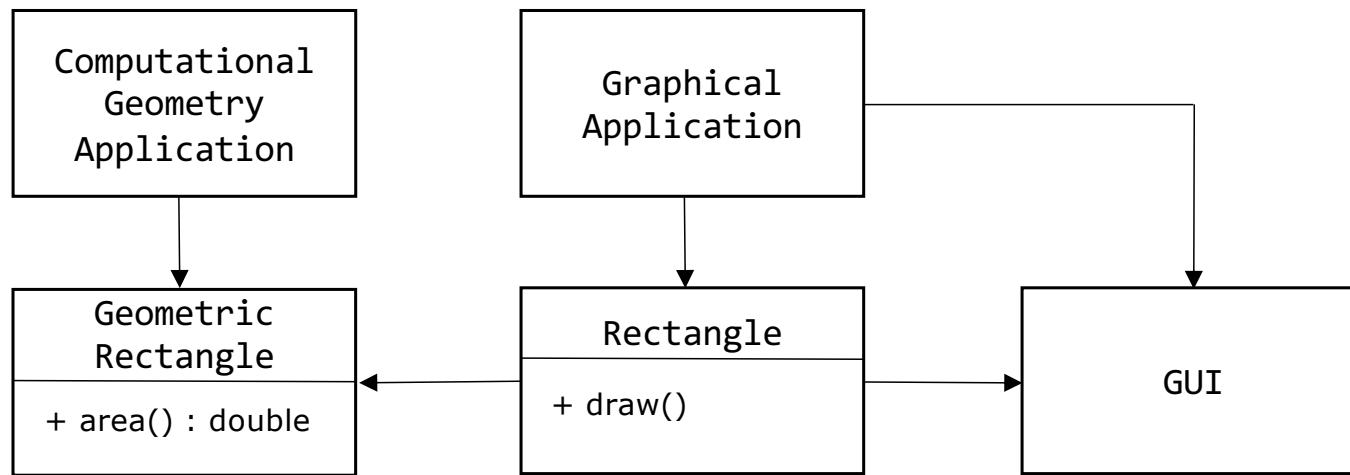
Beispiel – Was ist das Problem?



- `Rectangle::draw()` → Zeichnet das Rechteck auf dem Bildschirm
- `Rectangle::area()` → Berechnet die Fläche des Rechtecks

Separation of Concerns & Single Responsibility

Beispiel – Problemlösung



Open-/Closed-Principle

1. Offen für Erweiterung

- das Verhalten eines Bausteins kann erweitert werden ohne Gefahr für die Stabilität des Bausteins / Systems
- Dem Baustein kann bei Anforderungsänderungen neues Verhalten hinzugefügt werden, um neue Funktionen in die Anwendung zu integrieren

2. Geschlossen für Modifikation

- Der existierende Quellcode eines Bausteins bleibt unverletzt und muss bei Erweiterungen nicht modifiziert werden



<https://blogs.msdn.microsoft.com/cdndevs/2009/07/15/the-solid-principles-explained-with-motivational-posters/>

Open-/Closed-Principle



Zum Nachlesen

Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification

- Wenn eine einzelne Änderung zu einer Kaskade von Änderungen an abhängigen Bausteinen führt, wird das System fragil, starr, unvorhersehbar und nicht wiederverwendbar.
- Das Open-/Closed-Principle adressiert dieses Problem indem es vorsieht, dass **Erweiterungen nur durch neuen Code** hinzugefügt werden können und **bestehender Code nicht verändert** werden muss
- Einige Entwurfsmuster unterstützen dieses Prinzip, insbesondere das Decorator Pattern

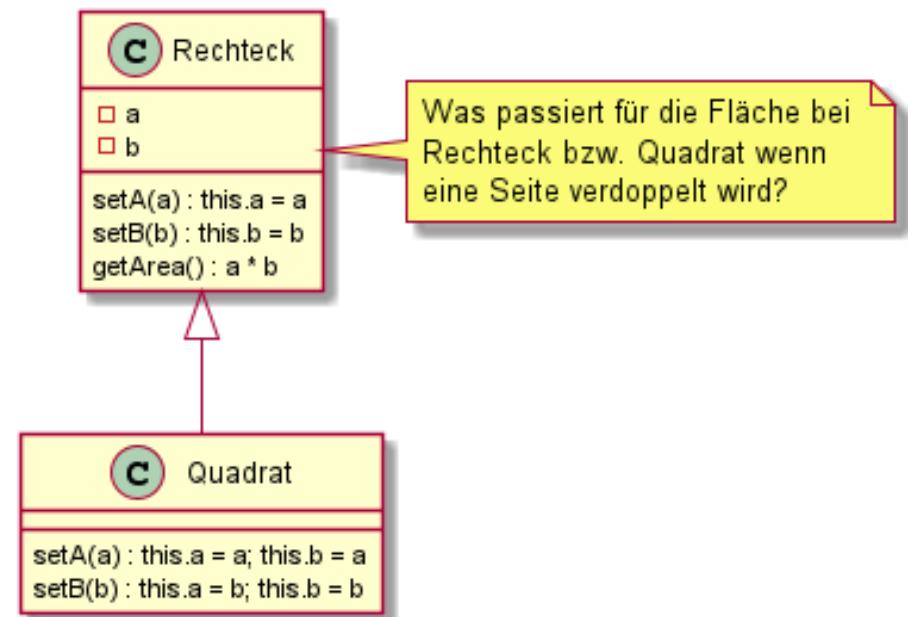
Liskov-Substitutionsprinzip



Liskov Substitution Principle

If it looks like a duck and quacks like a duck but needs batteries, you probably have the wrong abstraction.

<https://blogs.msdn.microsoft.com/cdndevs/2009/07/15/the-solid-principles-explained-with-motivational-posters/>



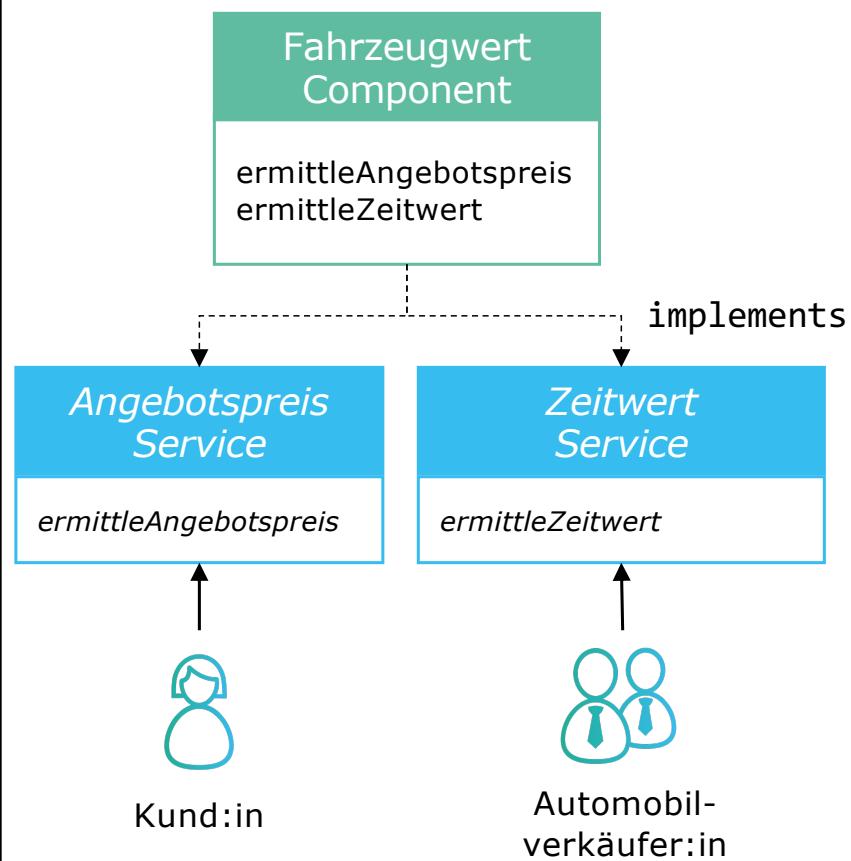
Liskov-Substitutionsprinzip



Zum Nachlesen

- Basisprinzip der objektorientierten Programmierung
- Eine Basisklasse muss immer durch ihre Unterklasse ersetzbar sein
- Unterklassen sollen nicht mehr erwarten und nicht weniger liefern. Die Verträge der Basisklassen müssen eingehalten werden!
- Dieses Prinzip wird durch falsche Anwendung von Vererbung verletzt, indem Methode nicht im korrekten Sinne der Generalisierung / Spezialisierung überschrieben werden
- OO Regelset
 - Wird eine Methode überschrieben, muss sie immer die Methode der Oberklasse nutzen (*super()* in Java)
 - Oberklassen dürfen nicht instanzierbar sein

Interface Segregation Principle



Prinzip nicht eingehalten

```
interface FahrzeugwertComponent{
    Fahrzeugwert ermittleFahrzeugwert
    (Bewertungsart bewertungsart) { ... }
}
```

Prinzip eingehalten mit getrennten Interfaces

```
interface AngebotspreisService {
    Angebotspreis ermittleAngebotspreis(FahrzeugId fahrzeugId);
}

interface ZeitwertService{
    Zeitwert ermittleZeitwert(FahrzeugId fahrzeugId);
}
```

Interface Segregation Principle

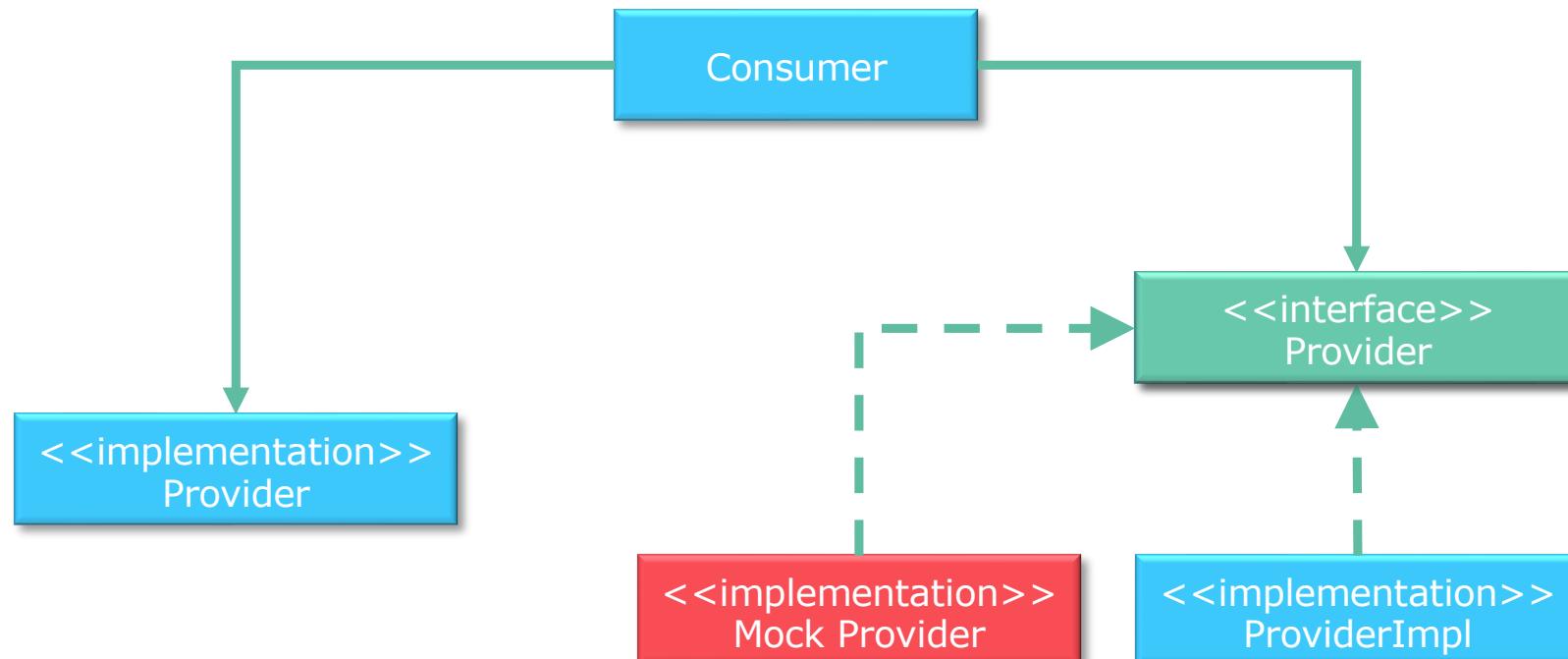


Zum Nachlesen

- Die Schnittstelle einer Komponente sollte nicht generell, sondern nach Zuständigkeiten aufgeteilt sein
- Hat eine Komponente mehrere Verantwortungsbereiche, sind diese in einer eigenen Schnittstelle zu exponieren
- Mehrere spezifische Schnittstellen sind wartbarer als eine Universalschnittstelle
- Anmerkung
 - „Komponente mehrere Verantwortungsbereiche“ vs. Single Responsibility Principles? ➡ Stellt sich die Frage überhaupt wenn SRP eingehalten wird?
 - Command Query Segregation Principle als Spezialform des Interface Segregation Principle

Dependency Inversion Principle

**Abstraktionen sollten nicht von Details abhängen,
Details sollten von Abstraktionen abhängen!**



Dependency Inversion Principle



Zum Nachlesen

- Abhängigkeits-Umkehr-Prinzip
- Im Allgemeinen wird Dependency Inversion beschrieben durch:
 - Bausteine höherer Ebenen sollten nicht von Bausteinen niedrigerer Ebenen abhängen.
 - Abstraktionen sollten nicht von Details abhängen. Details sollten von Abstraktionen abhängen.

Dependency Inversion Principle



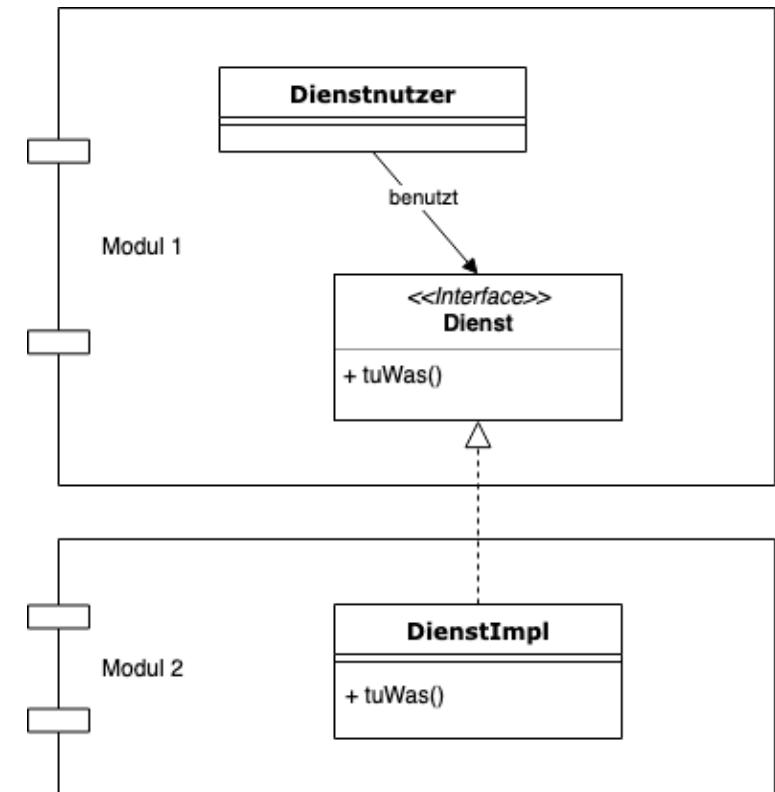
Zum Nachlesen

Invertierung der Abhängigkeit

- Das Modul der höheren Ebene definiert die Schnittstelle, mit der es arbeitet.
- Module niedrigerer Ebene realisieren die Schnittstelle.

OO-Regelset auf Klassenebene

- Variablen sollten keine Referenz auf eine konkrete Klasse halten, sondern auf eine Abstraktion (interface / abstract class)
- Keine Ableitung von konkreten Klassen – Oberklassen sollten immer abstrakt sein



Dependency Inversion Principle



Zum Nachlesen

Objektorientierte Entwürfe werden **in Bausteine (oder auch Module) strukturiert**, die unterschiedliche Verantwortlichkeiten umsetzen.

Eine gängige Praxis ist das **Anordnen der Bausteine in Ebenen**. Je niedriger die Ebene eines Baustein, desto spezieller sind die Vorgänge, die er definiert.

In Modulen niedrigerer Ebenen werden Abläufe definiert, welche von allgemeineren Abläufen in höheren Ebenen benutzt werden.

Falls diese **Anordnung falsch umgesetzt wird**, also Module höherer Ebenen von Modulen niedrigerer Ebenen abhängen, entsteht ein **Problem**:

Änderungen in Bausteine niedrigerer Ebenen führen unweigerlich zu Änderungen in Bausteinen höherer Ebenen.

Dies widerspricht einerseits dem eigentlichen Ansatz der Hierarchie, andererseits führt es zu **zyklischen Abhängigkeiten**. Dadurch kommt es zu einer **erhöhten Kopplung** der Bausteine, welche Änderungen in Architektur und Design verkomplizieren.

Einfachheit

Komplexität so gering wie möglich halten

Prinzipien für Einfachheit

KISS

- ***Keep it Simple and Stupid***
- Die Architektur / Lösungen sollten immer nur so komplex wie notwendig sein
- Ein zu starkes Vereinfachen ist jedoch auch Contra-Produktiv
- Die Architektur sollte bekannte Anforderungen adäquate begegnen und entsprechende Weitsicht in das Design einfließen lassen

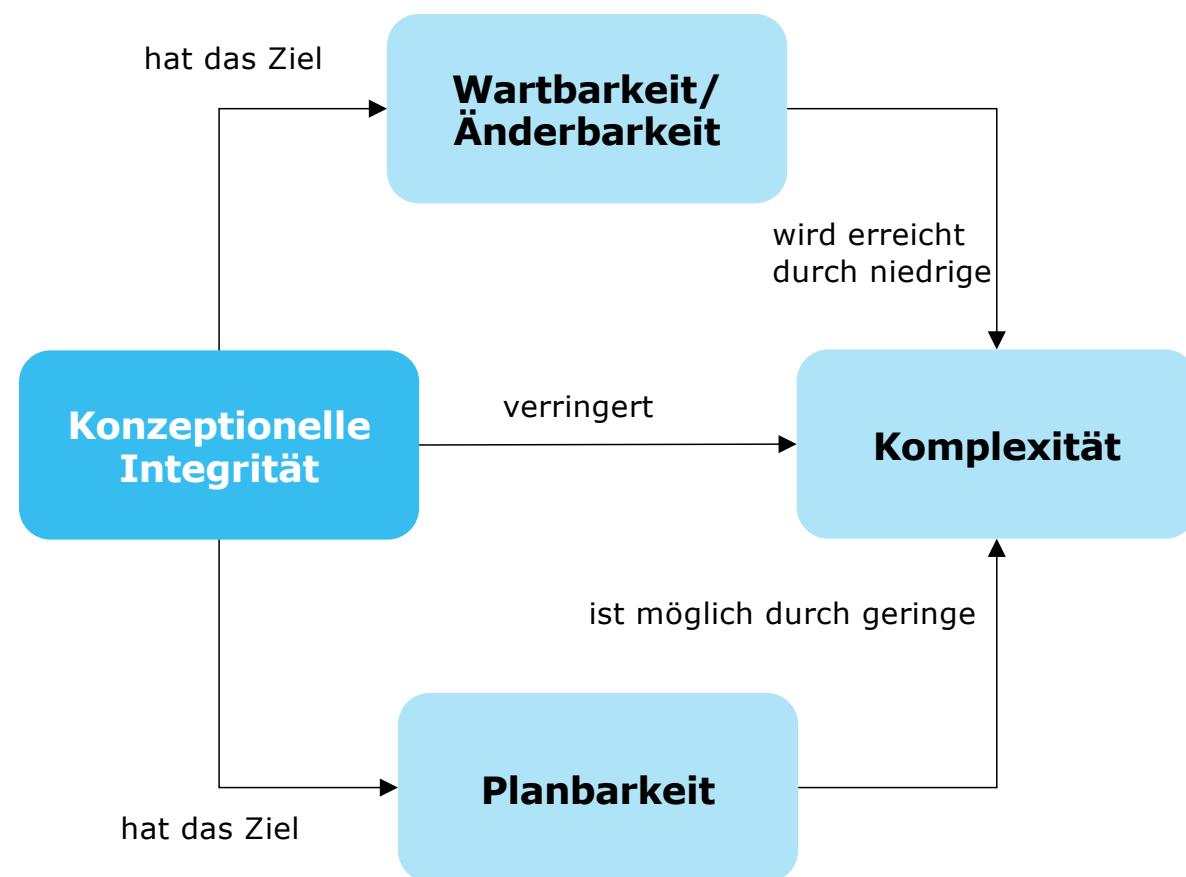
YAGNI

- ***You Ain't Gonna Need It***
- Nur tatsächliche Anforderungen werden im Entwurf berücksichtigt
- Auch bekannte aber unklare Anforderungen sollten erst dann berücksichtigt werden, wenn diese definiert wurden. Dies erhält den Lösungsraum.
- Berücksichtigt nur das, was wirklich gebraucht wird

Konzeptionelle Integrität

Gleichartigkeit im Design

Konzeptionelle Integrität



„Ein wichtiger Punkt ist vor allen Dingen die Bewahrung der konzeptionellen Integrität: Sie sollten möglichst **alle Teile nach ähnlichen Aspekten zerlegen und dieses Konzept konsistent anwenden** (und am besten dokumentieren).“ [Gharbi 2018]

„Softwarearchitekten verfolgen konzeptionelle Integrität (auch genannt Konsistenz): Die gesamte **Konstruktion von Software sollte einem einheitlichen Stil folgen**. Insbesondere sollten **ähnliche Aufgabenstellungen in Systemen ähnlich gelöst werden**. Dies erleichtert das **Verständnis und die langfristige Weiterentwicklung**.“ [Starke 2015]

Konzeptionellen Integrität



Zum Nachlesen

- **Gleichartige Anwendung** von **Lösungsmuster und -prinzipien** für Baustein- und Klassenstrukturen, sodass sich in allen Strukturen dieselben Konzeptideen wiederfinden
- **Wartbarkeit/Änderbarkeit**
Es ist einfach Änderungen einzubringen, und die entsprechenden Anpassungen an der Software vorzunehmen.
- **Komplexität**
Durch eine hohe Kohäsion, entsteht keine unnötige Komplexität
- **Planbarkeit**
Da die Features und ihre Abbildung auf die Software vom Team gut verstanden werden, sind auch die Aufwandschätzungen gut. Die Planbarkeit nimmt zu.

Konzeptionelle Integrität

*„Mangelnde konzeptionelle Integrität:
Identische Probleme werden innerhalb
eines Systems unterschiedlich gelöst, es
gibt mehrere unterschiedliche, teils
widersprüchliche Lösungsansätze.“
[Starke 2015]*

Erreichung von konzeptioneller Integrität durch:

- Gleichartige Anwendung von Architektur- und Entwurfsprinzipen
- Gleichartige Anwendung von Architekturstilen sowie Architektur- und Entwurfsmuster
- Verwendung von Stereotypen (Mustersprache, siehe z.B. taktisches DDD)
- Z.B. Event, Ressource, REST, Factory Pattern

Principle of least surprise

- Eine API bzw. Schnittstellen sollten intuitiv benutzbar sein und keine Überraschungen für den Konsumenten bereithalten

Schnittstelle mit vorhersagbarem Verhalten

```
Customer getCustomerOrNull(int id)
```

```
Customer getCustomer(int id)
```

```
void saveCustomer(int id, Customer out)
```

Schnittstelle mit Überraschungen

```
boolean runCustomerOperation  
(String operation, int id, Customer out)
```



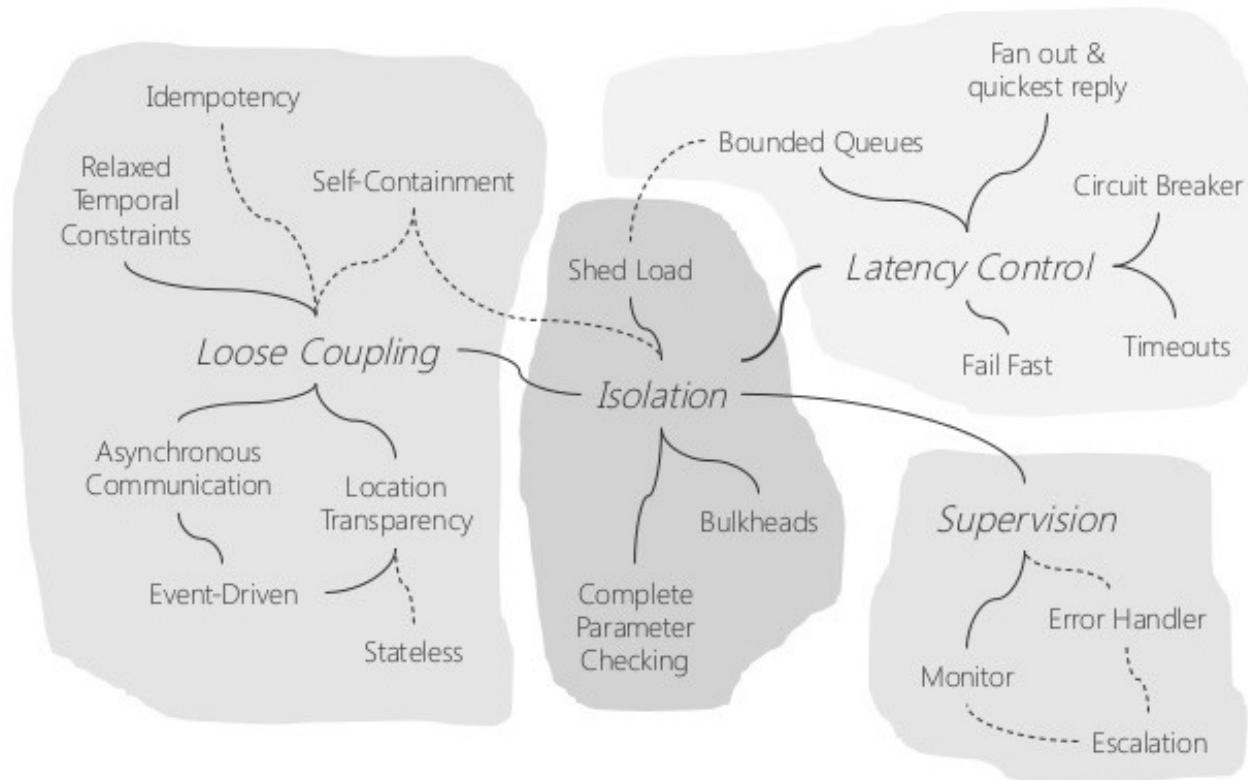
Erwarte Fehler

Postel's Law aka Robustness principle

- Findet Anwendung in der **Kommunikation zwischen Systemen** (z.B. im API Design)
- Das Robustheitsprinzip besagt, dass man sich **tolerant** bei der **Annahme von Nachrichten** und **konservativ** beim **Versenden von Nachrichten** verhalten sollte
- Die Verfolgung des Prinzips
 - Reduziert die Abhängigkeit zu Bestandteilen einer Nachricht und erhöht dadurch die Widerstandsfähigkeit gegenüber Änderungen
 - Verringert Inkompatibilitätsprobleme zwischen Versionen (Aufwärts- und **Abwärtskompatibilität**)
- **Konsumenten sollen nur die für sich relevanten Teile aus einer Nachricht extrahieren**

*Be conservative in what you send,
be liberal in what you accept
[Postel 1981]*

“Failures are a given, and everything will eventually fail over time.” Vogels, CTO Amazon



resilience(IT)
The ability of a system to handle unexpected situations [...]
[Friedrichsen 2015]

Übung

- Erläutern Sie die Begriffe Geheimnisprinzip, Trennung von Verantwortlichkeiten, Single Responsibility und lose Kopplung anhand Ihres Designs für die Komponente Bestandssystem.

LZ 2-5: Wichtige Architekturmuster und Architekturstile beschreiben, erklären und angemessen anwenden (R1,R3) (1)

Softwarearchitekt:innen kennen verschiedene Architekturmuster (siehe unten) und können sie gegebenenfalls anwenden.

Sie wissen (R3):

- dass Muster ein Weg sind, bestimmte Qualitäten für gegebene Probleme und Anforderungen innerhalb gegebener Kontexte zu erreichen
- dass es verschiedene Kategorien von Mustern gibt
- zusätzliche Quellen für Muster, die sich auf ihre spezifische technische oder Anwendungsdomäne beziehen

LZ 2-5: Wichtige Architekturmuster und Architekturstile beschreiben, erklären und angemessen anwenden (R1,R3) (2)

Softwarearchitekt:innen können die folgenden Muster erklären und Beispiele dafür liefern (R1):

- Schicht (Layers):
 - Abstraktionsschichten (Abstraction layers) verbergen Details, Beispiel: ISO/OSI-Netzwerkschichten
 - Eine andere Interpretation sind Schichten zur (physischen) Trennung von Funktionalität oder Verantwortung
- Pipes-and-Filter: repräsentativ für Datenflussmuster, die die schrittweise Verarbeitung in eine Reihe von Verarbeitungsaktivitäten ("Filter") und zugehörige Transport/Puffer ("Pipes") separieren.
- Microservices teilen Anwendungen in separate ausführbare Dienste auf, die über Netzwerk (remote) kommunizieren.
- Dependency Injection als eine mögliche Lösung für das Dependency-Inversion-Prinzip

LZ 2-5: Wichtige Architekturmuster und Architekturstile beschreiben, erklären und angemessen anwenden (R1,R3) (3)

Softwarearchitekt:innen können einige der folgenden Muster erklären, ihre Relevanz für konkrete Systeme erläutern und Beispiele dafür liefern (R3):

- Blackboard
- Kombinator
- CQRS (Command-Query-Responsibility-Segregation)
- Event-Sourcing
- Interpreter
- Integrations- oder Messaging-Patterns
- Die MVC-, MVVM-, MV-Update-, PAC-Musterfamilie
- Schnittstellenmuster wie Adapter, Fassade, Proxy.
- Observer
- Plug-In

LZ 2-5: Wichtige Architekturmuster und Architekturstile beschreiben, erklären und angemessen anwenden (R1,R3) (4)

- Plug-In
- Ports & Adapters
- Remote Procedure Call
- SOA
- Template and Strategy
- Visitor

Softwarearchitekt:innen kennen wesentliche Quellen für Architekturmuster, beispielsweise die POSA-Literatur (z.B. [Buschmann+1996]) und PoEAA ([Fowler 2003]) (für Informationssysteme) (R3)

Muster für den Architekturentwurf

Muster sind eine dreiteilige Regel, die die Beziehung zwischen einem bestimmten Kontext, einem Problem und einer Lösung ausdrückt.

- **Architekturmuster** (Ebene 0 und 1) und **Entwurfsmuster** (Ebene 1 und tiefer)
- Kombinierte Anwendung von Mustern in einer Mustersprache
- Es gibt weitere Muster in der Softwareentwicklung, welche aber nicht im CPSA-F relevant sind:
 - Enterprise Integration Patterns
 - Vorgehensmuster
 - Microservice Patterns
 - Datenarchitekturmuster
 - Stabilitätspattern (siehe LZ 2-6)

Muster für den Architekturentwurf

Architekturstil vs. Architekturmuster

Architekturstil

- Ein Architekturstil ist ein Muster der strukturellen Organisation einer Familie von Systemen
- Fundamentale Struktur und Eigenschaften eines Softwaresystems

Architekturmuster

- Bilden Vorlagen für die Systemstrukturen und Verteilung der Verantwortlichkeiten
- Haben den Charakter einer Basisstrukturierung / Basisarchitektur
- Verwendung als Klassifikationsschemas (Z.B. Model View Controller)

Grenzen sind fließend

Anhand der Ähnlichkeit der Beschreibungen ist erkennbar, dass die Grenze „Was ist ein Stil und „Was ist ein Muster“, fließend sind. Insbesondere die „klassischen“ Architekturstile (Schichtenarchitektur, Pipes and Filters, Blackboard) aber auch z.B. Microservices werden in der Literatur bei Stilen und Muster aufgeführt, im Vergleich zu REST und Service-orientierung, die in der Regel als Stil aufgeführt werden. In der Praxis sind die Idee dahinter entscheidend und nicht die Frage, ob es sich um ein Stil oder Muster handelt!

Muster für den Architekturentwurf

Architekturstil vs. Architekturmuster

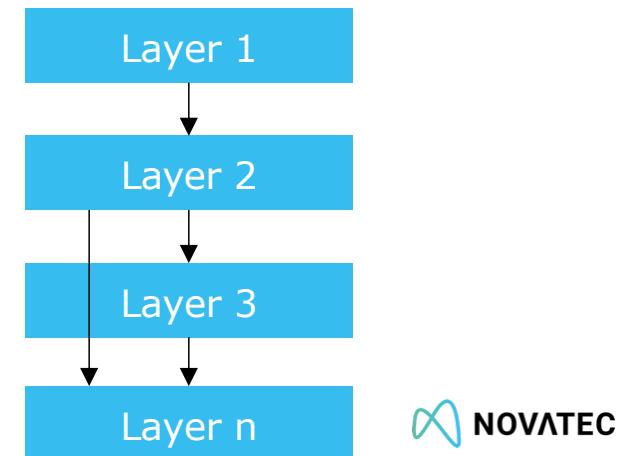
Architekturstil

- **Datenfluss-zentrische Systeme**
Batch-Sequentiell, Pipes and Filters
- **Hierarchische Systeme**
Schichtenarchitektur, Ports & Adapter, Master-Slave, Komponentenorientierung, Objektorientierung
- **Verteilte Systeme**
Command Query Segregation Principle, service-orientierte Architektur, Microservices
- **Ereignisbasierte Systeme**
Publish-Subscribe, Event-Driven Architecture
- **Interaktionsorientierte Systeme**
Model View Controller, Model View Presenter, Presentation Abstract Control
- **Sonstige**
REST, Remote Procedure Call (RPC)

Architekturmuster

Schichtenarchitektur

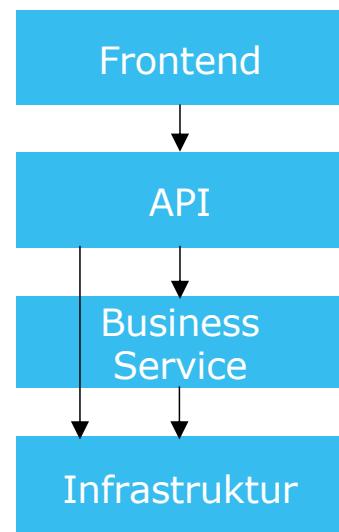
- **Basisstrukturierung** großer Systeme
- In Abhängigkeit stehende High- und Low-Level-Operationen lassen sich in Ebenen gleichen Abstraktionsgrads anordnen
- Zugriffe dürfen nur von High- nach Low erfolgen
- Eine **übergeordnete Schicht** (High Level) darf nur auf eine **darunter liegende Schicht** zugreifen (Low Level)
- Fördert Verständlichkeit, Wiederverwendbarkeit und Portabilität
- Änderung beschränken sich auf eine Schicht und wirken sich nicht auf das gesamte System aus
- Datenänderungen können sich aber vertikal durch alle Schichten ziehen
- Performance muss im Auge behalten werden



Schichtenarchitektur

Technische Strukturierung großer Systeme

- Präsentation
Backend
Persistenz
- Frontend
API
Service
Infrastruktur
- Frontend
Gateway
Service
Repository
Data

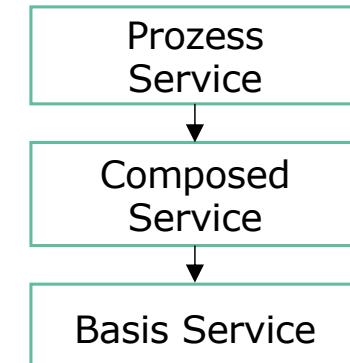
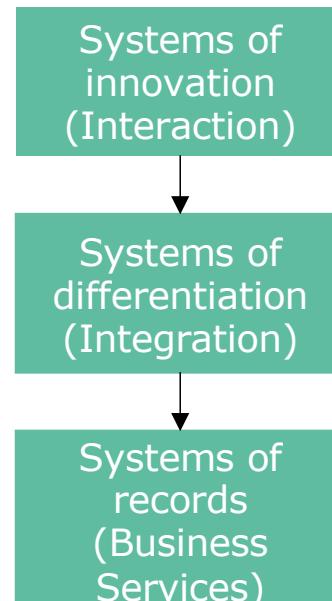


Strukturierung nach Verantwortlichkeiten

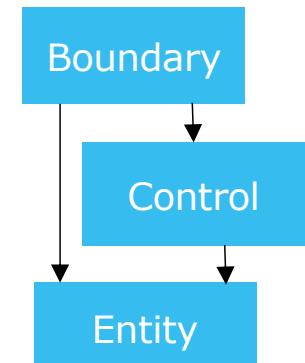
BI Modale IT /
2-Speed-Architecture

SOA

Microservice



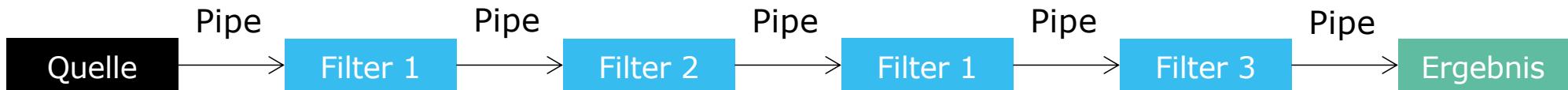
Nachzulesen in „SOA in der Praxis“ von Nicolai Josuttis



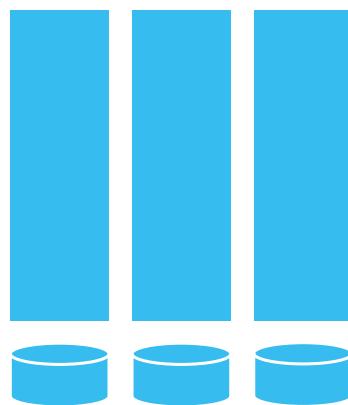
Nachzulesen in „Architecting modern Java EE Applications“ von Sebastian Daschner

Pipes and Filters

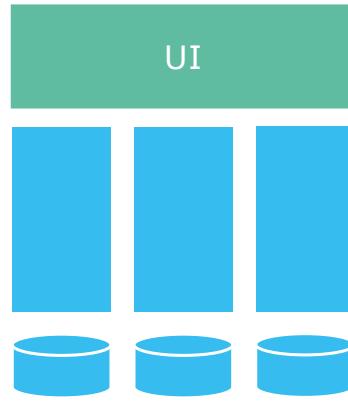
- Sequenz von Verarbeitungsschritten (Filter) sind mit Datenkanälen (Pipes) miteinander verbunden
- Pipes transportieren Zwischenergebnisse von einem Filter zum nächsten
- Filter kennen sich nicht
- Führt zu Entkopplung auf Ebene der Zeit, des Transportmechanismus und in der Bestimmung des Folgefitters
- Compiler / Batch Verarbeitung



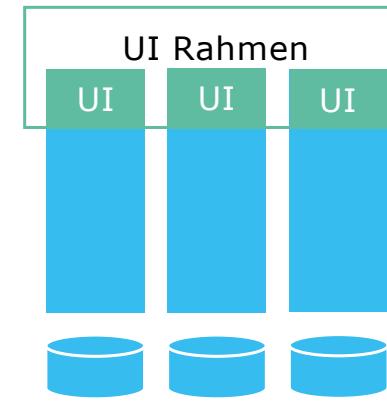
Microservices Modularisierungskonzepte für große Systeme



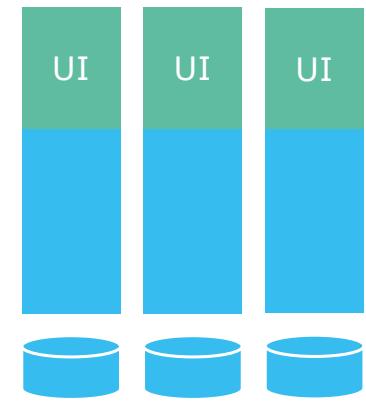
Microservice ohne UI



Full Client



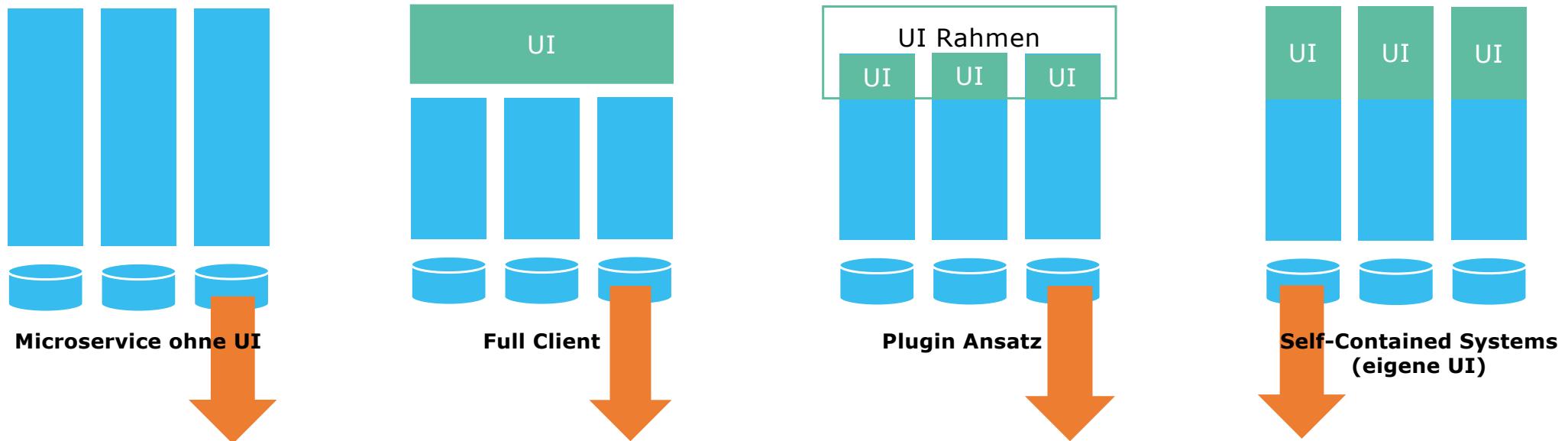
Plugin Ansatz



**Self Contained Systems
(eigene UI)**

<https://scs-architecture.org/>

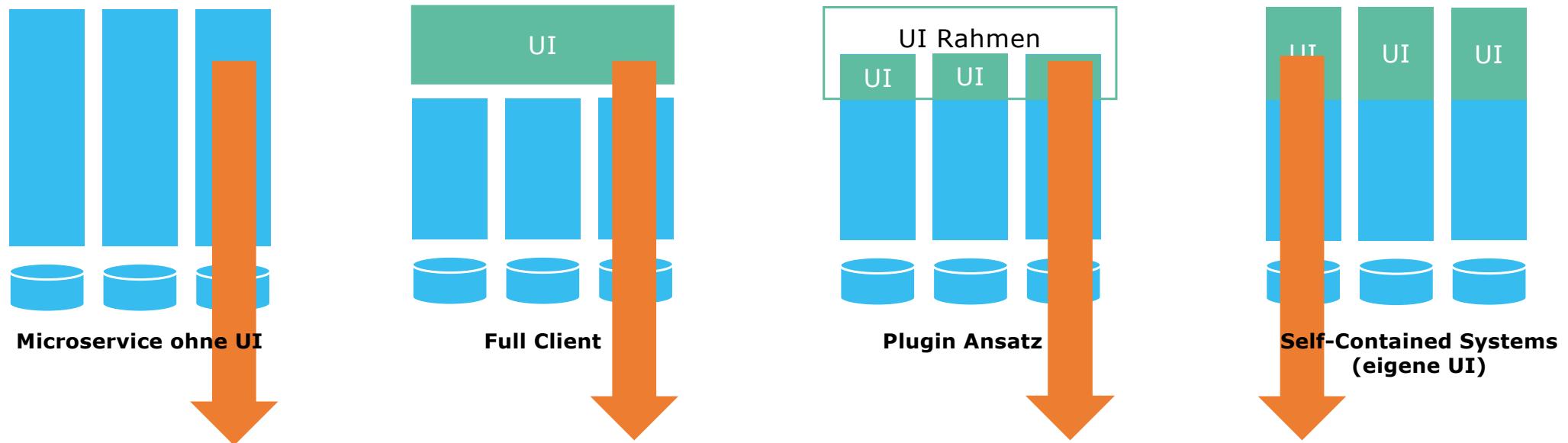
Microservices sind maximal unabhängig! Eine eigene Datenhaltung ist eine grundlegende Eigenschaft von Microservices!



Owning their Own State [Newman 2021]

(Datenowner, eigene Datenbank, mindestens eigenes Schema)

Microservices teilen ein Softwaresystem in fachliche Module auf!



Modelled Around a Business Domain [Newman 2021]

(Vertikale E2E Schnitte, Bounded Context
bzw. Aggregate als Microservice)

Weitere Eigenschaften von Microservices

Ein Microservice ist unabhängig in

- der **Versionierung (SCM)**,
- im **Deployment**,
- bei der Veröffentlichung eines **Releases**
- sowie in der **technologischen Umsetzung**
- und **kommuniziert** ausschließlich über das **Netzwerk** mit anderen Microservices

Orientierung bieten die 12 Factor App Principles!

<https://12factor.net/de/>

I. Codebase

Eine im Versionsmanagementsystem verwaltete Codebase, viele Deployments

II. Abhängigkeiten

Abhängigkeiten explizit deklarieren und isolieren

III. Konfiguration

Die Konfiguration in Umgebungsvariablen ablegen

IV. Unterstützende Dienste

Unterstützende Dienste als angehängte Ressourcen behandeln

V. Build, release, run

Build- und Run-Phase strikt trennen

VI. Prozesse

Die App als einen oder mehrere Prozesse ausführen

VII. Bindung an Ports

Dienste durch das Binden von Ports exportieren

VIII. Nebenläufigkeit

Mit dem Prozess-Modell skalieren

IX. Einweggebrauch

Robuster mit schnellem Start und problemlosen Stopp

X. Dev-Prod-Vergleichbarkeit

Entwicklung, Staging und Produktion so ähnlich wie möglich halten

XI. Logs

Logs als Strom von Ereignissen behandeln

XII. Admin-Prozesse

Admin/Management-Aufgaben als einmalige Vorgänge behandeln

Stile der Microservice Kommunikation

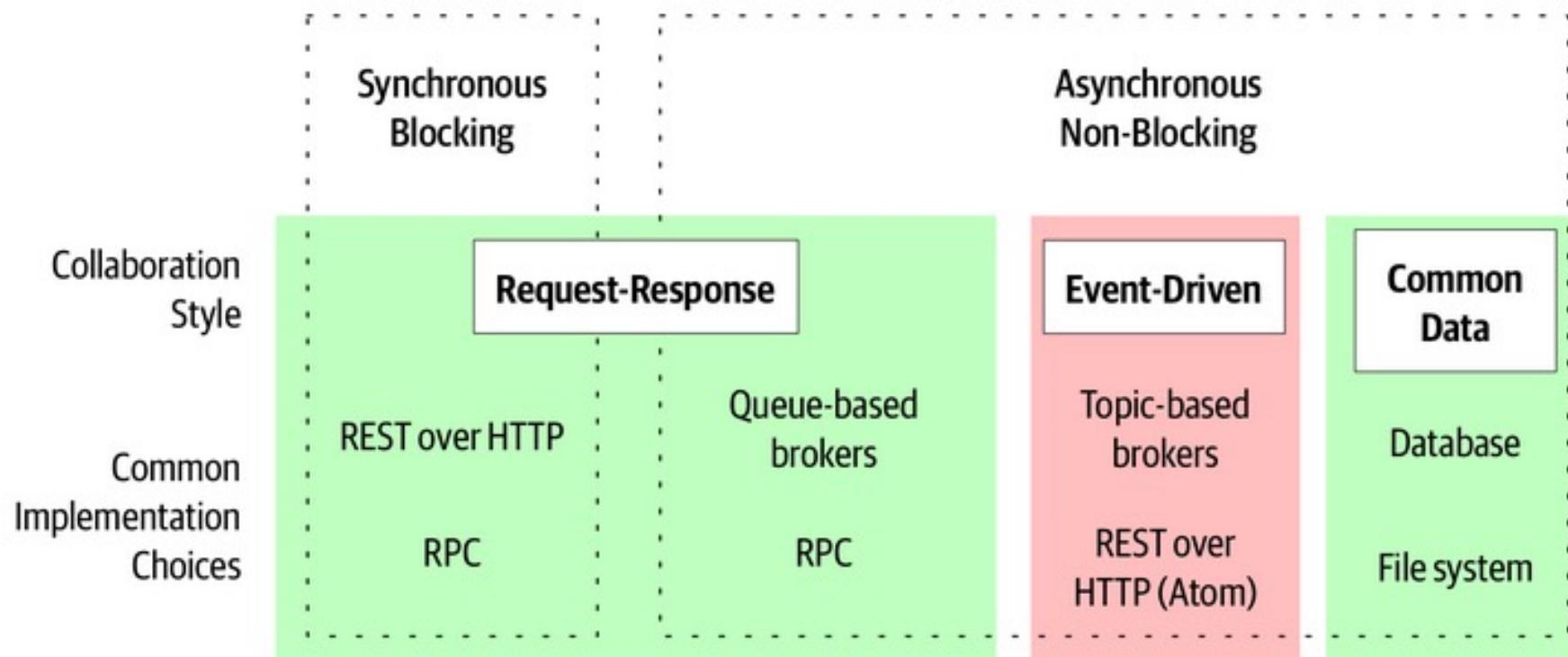


Abbildung aus [Newman 2021]

Größe eines Microservice

- Ein Microservice muss von **einem Entwickler verstanden** und **weiterentwickelt** werden können
- Ein Microservice darf nur von **einem Team** mit einer Größe gemäß Agiler Vorgehensmodelle (7+/-2 Entwickler) verantwortet werden
(Collective Code Ownership)
- Ein Team kann mehrere Microservices verantworten
- Ein Microservice sollte eine **abgrenzbare fachliche Funktion** bereitstellen (Bounded Context, Aggregate)
- Darf nicht so groß werden, dass er nicht mehr **ersetzbar** ist, aufgrund zu hoher Kosten für Neuimplementierung oder Refactoring
(sonst das gleiche Problem wie bei Monolithen)
- Darf nicht so klein werden, dass eine ACID Transaktion verteilt ausgeführt werden muss.

122

Einflussfaktoren

Teamgröße

Fachliche Komplexität

Fachliche Verantwortung

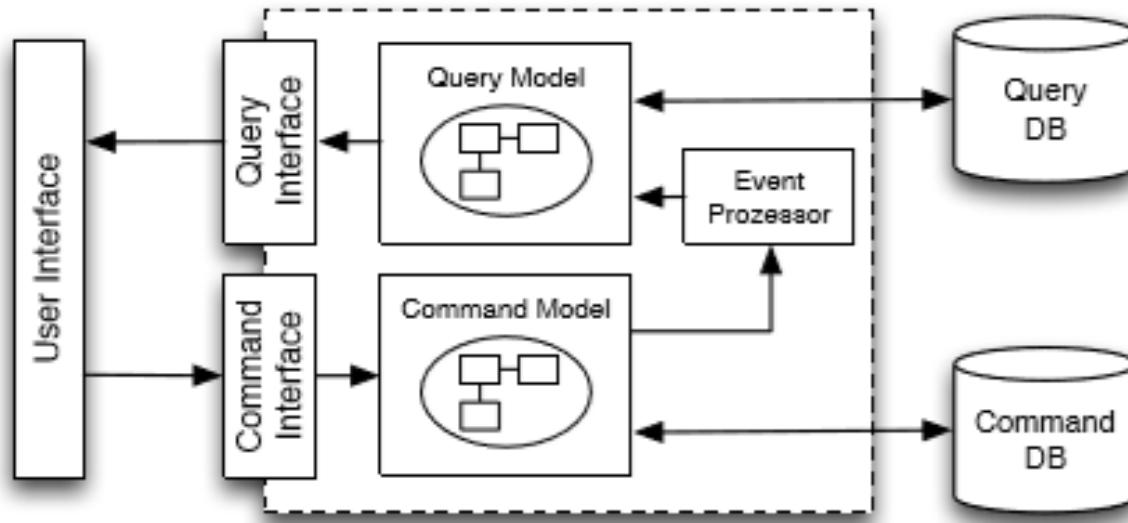
Ersetzbarkeit

Transaktionsklammer

Abbildung angelehnt an [Wolff 2016]



Architekturmuster Command Query Responsibility Segregation



Abbildungen aus [Starke 2015]

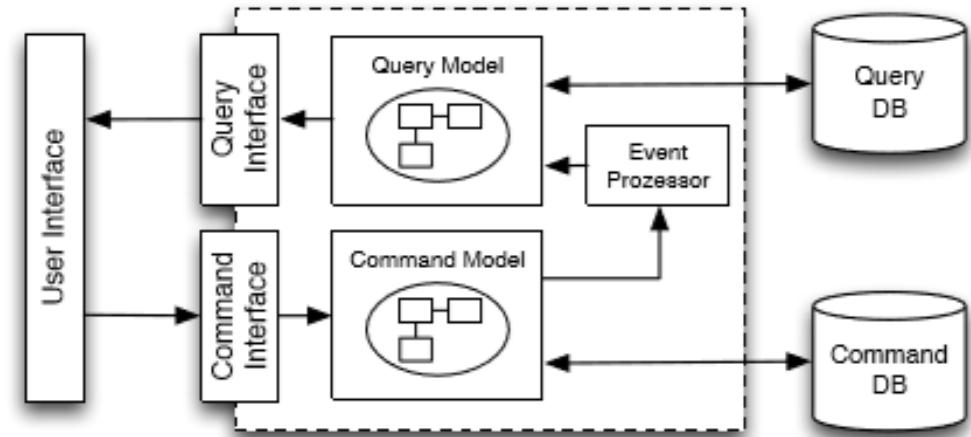


Zum Nachlesen

Architekturmuster Command Query Responsibility Segregation

- Klassifiziert die Funktionalität in Commands und Queries
- **Commands** verändern Daten auf Basis eines Command Model
- **Queries** lesen Daten auf Basis eines Query Model
- **Skalierbarkeit**
I.d.R. hat eine Anwendung mehr lesende Zugriffe auf Daten. In einer Microservice-Architektur kann eine Query-Service einzeln skaliert und so die Ressourcenausnutzung optimiert werden
- **Performance**
Query-Model kann auf Performance und Konsumenten optimiert werden. Zusätzlich Einsatz von Caching möglich. Command-Model muss Erweiterbarkeit beachten. Nutzung optimaler Technologien (schnelle Abfragen vs. transaktionssichere Commands / NoSQL vs. SQL) möglich.

- Herausforderung der **Datenkonsistenz**
Synchronisation bei Caches, physische Datenbanken und Schema (Einschränkung der Entkopplung), Eventual Consistency, etc.



Abbildungen aus [Starke 2015]

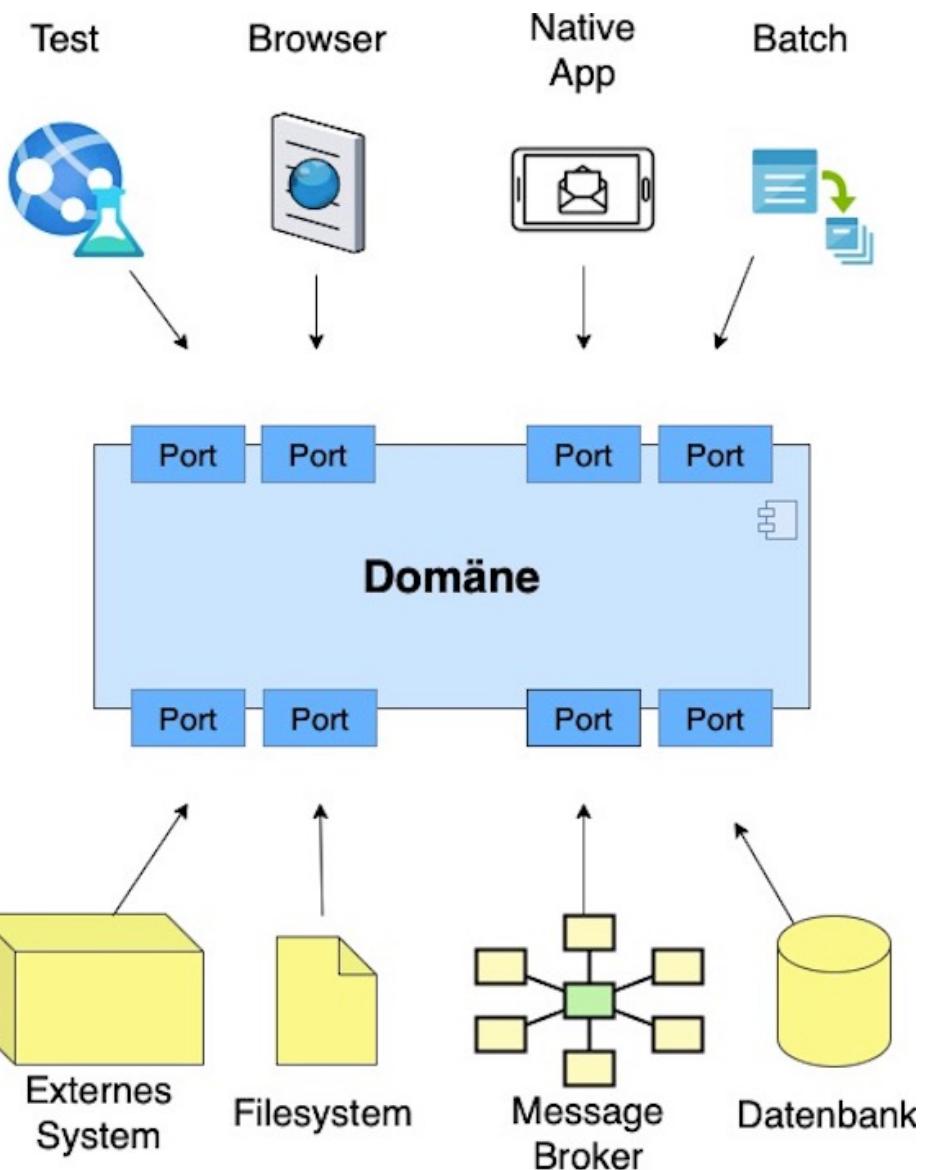
Ports & Adapter Muster

Entkopplung der Domäne von aller Art Infrastruktur, wie z.B.:

- DB
- Third Party Library
- Externen Systemen
- Web / UI

Sowie Entkopplung von übergeordneten Anwendungsfällen!

Kein Anpassungsbedarf in der Domäne, bei Änderung der Infrastruktur!



Robert C. Martin's Clean Architecture

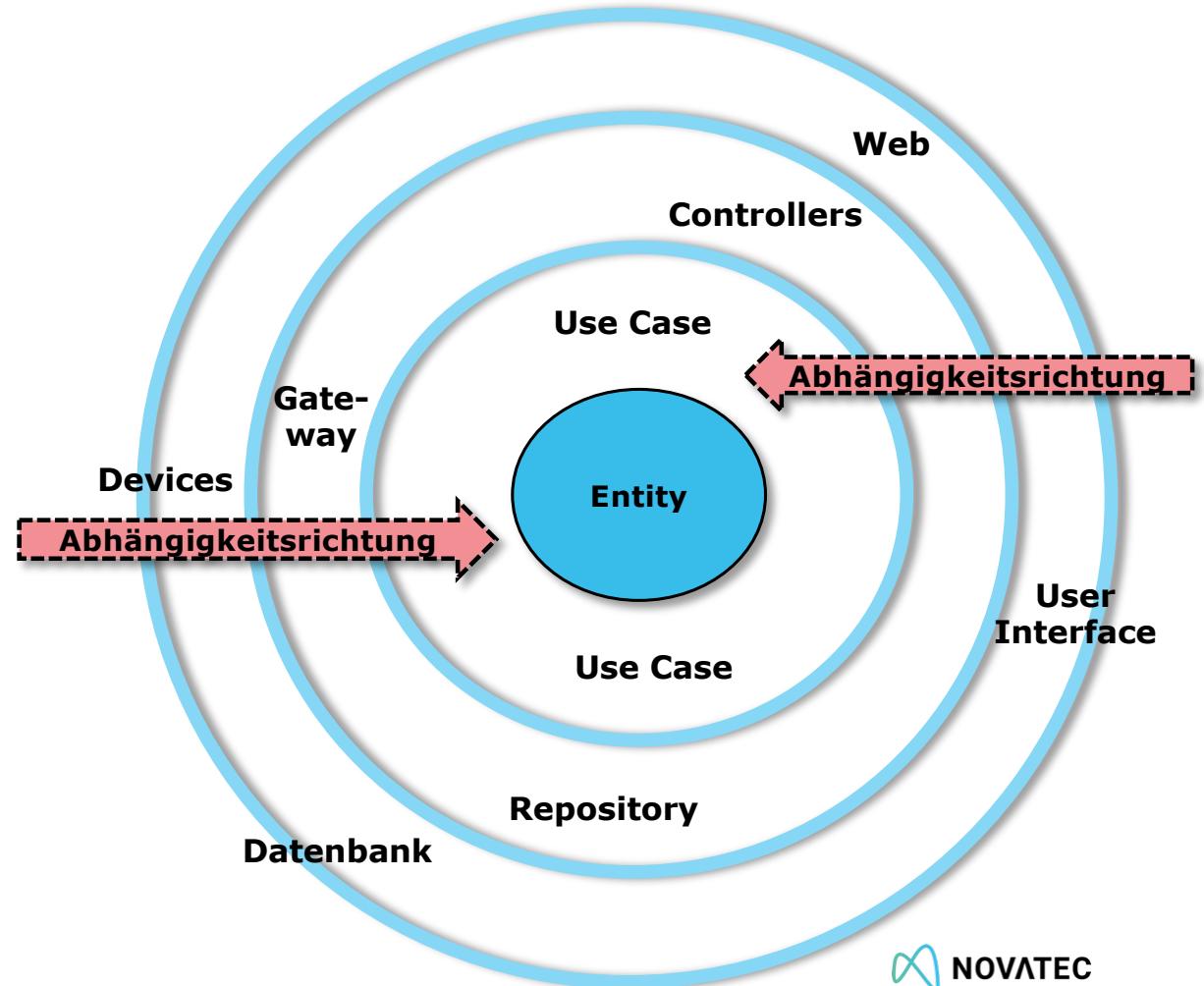
Kernmotivation

Fachlichkeit ist unabhängig von Database, Framework und UI Technologie

Strategie

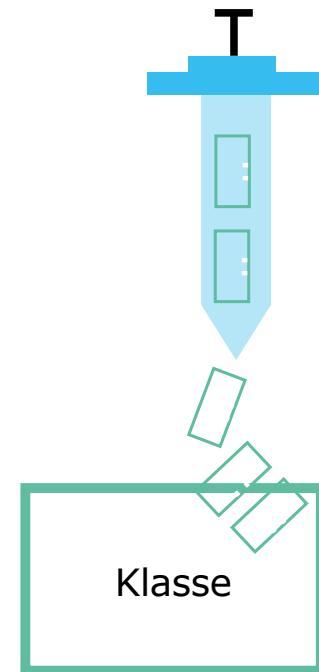
Domäne hat keine ausgehenden Abhängigkeiten!

Alle Abhängigkeiten gehen auf die Domäne!



Dependency Injection

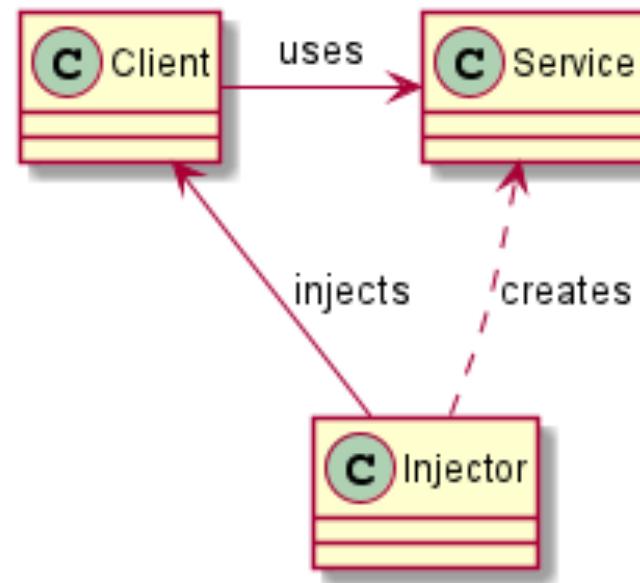
- Ziel: Lose Kopplung
- Anderer Name: Inversion of control
- Die Bezeichnung **Dependency Injection** wurde 2004 von Martin Fowler eingeführt, um den Begriff Inversion of Control zu präzisieren:
- *„Inversion of Control is too generic a term, and thus people find it confusing. As a result with a lot of discussion with various [Inversion of Control] advocates we settled on the name Dependency Injection..“*
- Die Konfiguration und die Verbindung mit anderen Komponenten wird aus der Komponente selbst herausgenommen
- Die Erzeugung von Klassen werden außerhalb einer nutzenden Klasse durchgeführt und als Parameter übergeben (Constructor-, Setter-, Field Injection)



Dependency Injection

Spring als Injector

```
@Configuration  
public class Config {  
    @Bean  
    public Service service() {  
        return new Service(...);  
    }  
}  
  
@Component  
public class Client {  
    @Autowired  
    public Client(Service service) {  
        this.service = service;  
    }  
}
```

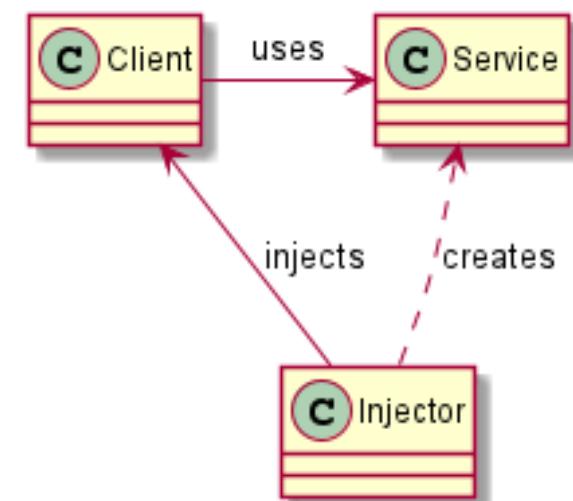


Dependency Injection



Zum Nachlesen

- Martin Fowler beschreibt drei verschiedene Arten zum Setzen benötigter Referenzen, die er mit dem Begriff Dependency Injection verbindet:
Constructor Injection, Field Injection und Setter Injection
- Es existieren verschiedene **Frameworks** für diverse Programmiersprachen und Plattformen als fertige Lösungen
- Diese implementieren das Muster mit zum Teil umfassender, weiterführender Funktionalität, wie beispielsweise das Einlesen der Konfiguration aus Dateien und deren Prüfung auf formale Korrektheit.



Dependency Injection



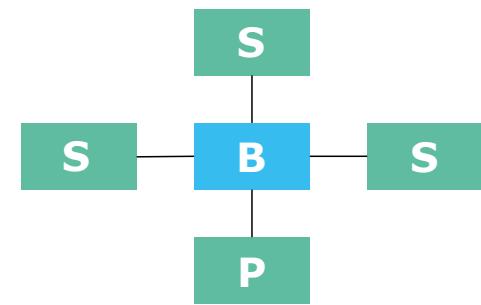
Zum Nachlesen

Mit Dependency Injection ist es möglich – entsprechend dem Single Responsibility Prinzip – die **Verantwortlichkeit** für den **Aufbau des Abhängigkeitsnetzes zwischen den Objekten** eines Programmes aus den einzelnen Klassen in eine **zentrale Komponente** zu überführen.

- In einem herkömmlichen System objektorientierter Programmierung ist dagegen jedes Objekt selbst dafür zuständig, seine Abhängigkeiten, also benötigte Objekte und Ressourcen, zu verwalten. Dafür muss jedes Objekt einige Kenntnisse seiner Umgebung mitbringen, **die es zur Erfüllung seiner eigentlichen Aufgabe normalerweise nicht benötigen würde.**
- Dependency Injection überträgt die Verantwortung für das Erzeugen und die Verknüpfung von Objekten an eine eigenständige Komponente, wie beispielsweise ein extern konfigurierbares Framework. Dadurch wird der Code des Objektes unabhängiger von seiner Umgebung.

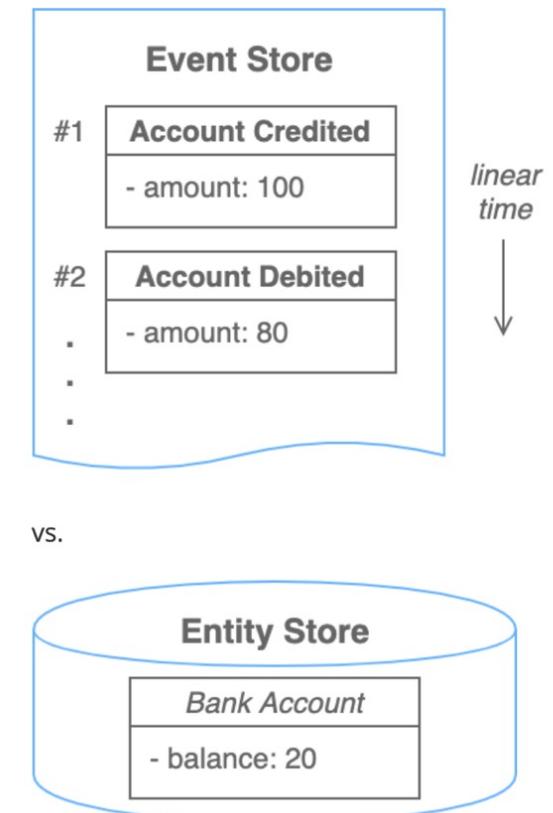
Broker & Messaging

- Broker haben die Aufgabe der Vermittlung von Nachrichten zwischen Interessenten und Anbietern
- Nachrichten-basierte Service Architekturen nutzen i.d.R. eine Middleware, die die Broker-Funktionalität umsetzt
- Interne und/oder externe Kommunikation eines verteilten Systems erfolgt i.d.R. asynchron durch Nachrichten
- Oft mit Anwendung des Publish-Subscribe-Pattern
 - Interessenten (Subscriber) registrieren sich auf eine Nachricht
 - Erzeuger (Publisher) einer Nachricht übergeben diese dem Broker
 - Broker veröffentlichen dies an die Interessenten



Event Sourcing

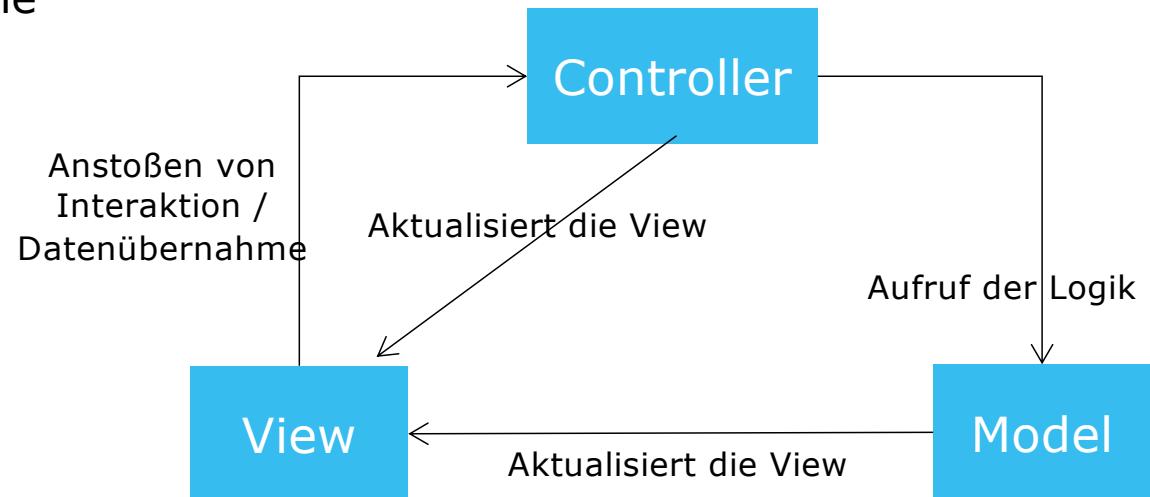
- Event Sourcing beschreibt einen Persistenzmechanismus, der nicht zu einer Veränderung des Zustands führt, sondern die einzelnen Zustandsänderungen (Events) in einem Event Store speichert.
- Der eigentliche Zustand eines Objektes kann somit durch sequentielles Anwenden der gespeicherten Zustandsänderungen immer wieder hergestellt werden.
- Einsatzszenarien:
 - Nachvollziehbarkeit von Änderungen
 - Rückgängigmachen von einzelnen Änderungen
 - Snapshots von Datenbeständen entlang der Zeit erstellen



<https://www.sitepen.com/blog/architecture-spotlight-event-sourcing>

Model View Controller (MVC)

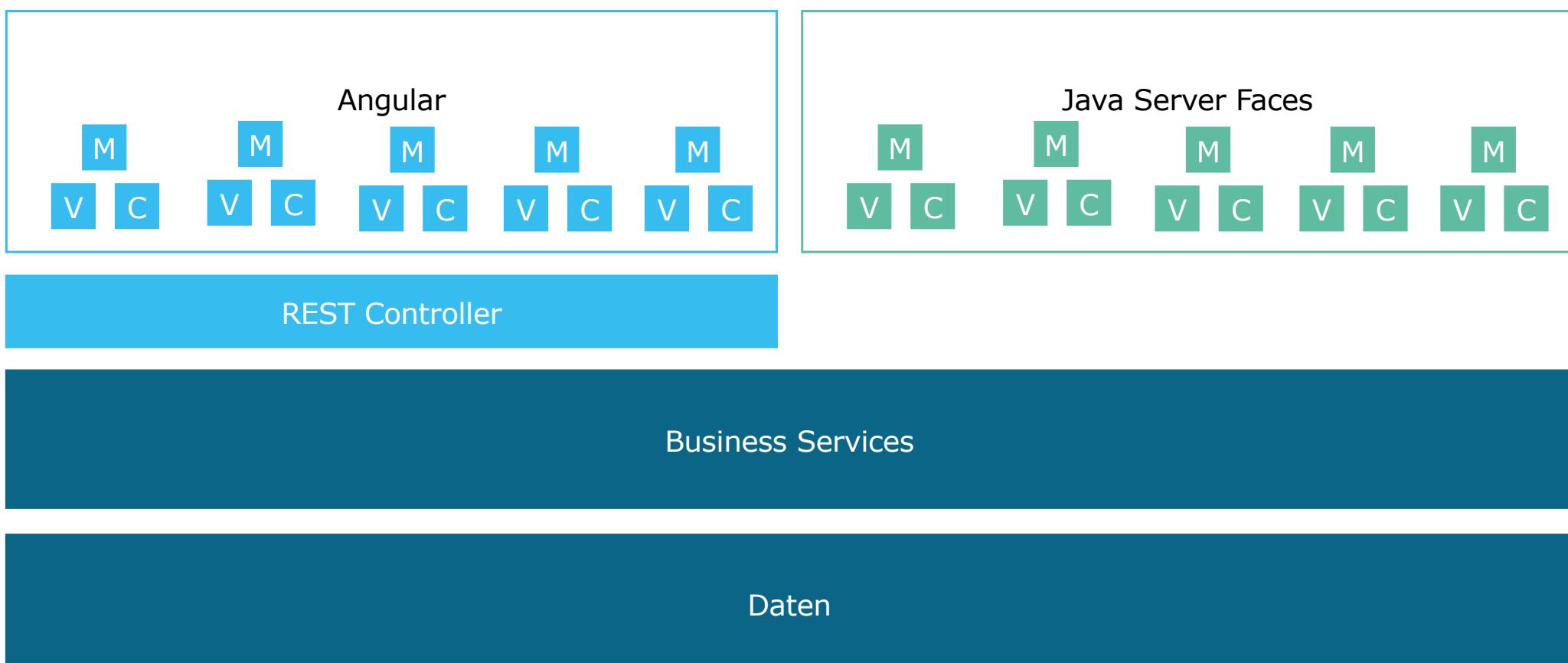
- **Muster für interaktive Anwendungen**
- Aufteilung der Verantwortung im Frontend in drei Zuständigkeitsbereiche
- **View**
 - Ansicht des Models
 - Änderungshäufigkeit hoch
- **Model**
 - Geschäftslogik und Daten
 - eher stabil
 - Aktualisierung der View
- **Controller**
 - Reaktion auf Benutzerereignisse
 - Aufruf der Geschäftslogik



Unterschiedliche MVC Ansätze

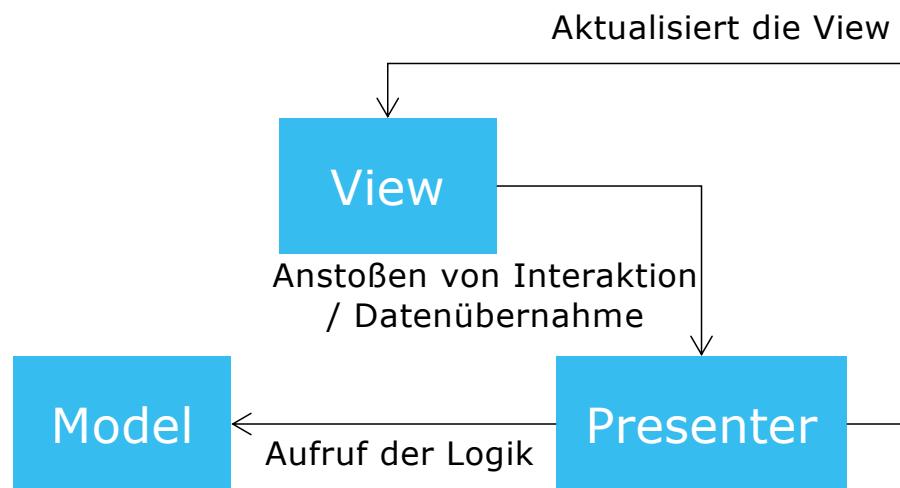


Zum Nachlesen



Model View Presenter (MVP)

- Basiert auf MVC und ist auch ein **Interaktionsmuster**
- Fokussiert (stärker) die Trennung von Benutzeroberfläche und Businesslogik
- **Model**
 - Businesslogik und Daten
 - kennt weder View noch Presenter
- **View**
 - Darstellung
 - Erfassung von Benutzereingaben
- **Presenter**
 - Verbindet View und Model
 - Steuert den Ablauf



Übung

- Erläutern Sie die Schichtenarchitektur Ihres Systems und ihrer Bausteine.

Entwurfsmuster

Entwurfsmuster

- Ein fachlich kontextfreier und bewährter Lösungsansatz für Klassendesign und Implementierung
- Basiert auf Objektorientierung als Design-/Programmierparadigma
- Gang of Four 1994 & später weitere (Gamma, Helm, Johnson, Vlissides)



Musterkategorien

- **Erzeugungsmuster**
 - Erzeugung von Objekten
 - Unterstützt Flexibilität, Open/Closed Principle, Separation of Concerns, Kapselung und Information Hiding
- **Strukturmuster**
 - Muster für strukturelle Zusammenhänge
 - Unterstützt Abstraktion, Kapselung, Information Hiding, Open/Closed Principle und Separation of Concerns
- **Verhaltensmuster**
 - Muster für Interaktion und Verhalten von Objekten untereinander
 - Unterstützt Single Responsibility, Separation of Concerns

Entwurfsmuster

Strukturmuster

Adapter



Zum Nachlesen

Problem

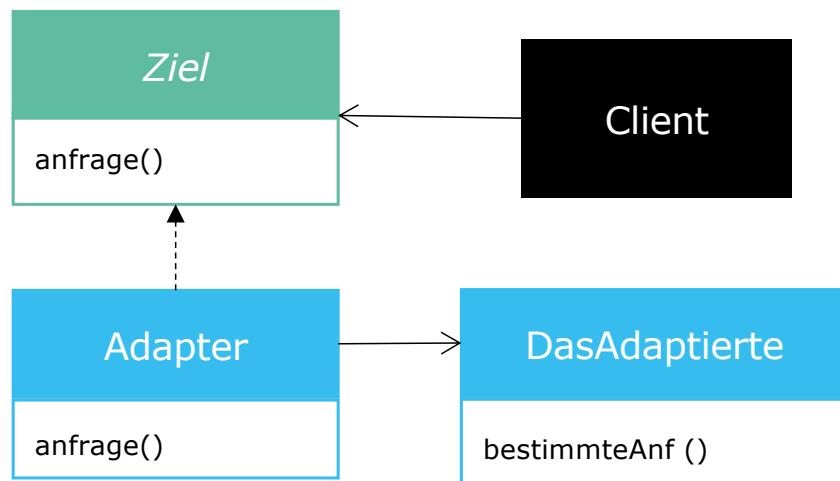
- Interaktion zwischen Modulen mit inkompatiblen Schnittstellen
- Schnittstellen von Modulen ändern sich häufig und verursachen Anpassungsbedarf in mehreren Clients

Lösung

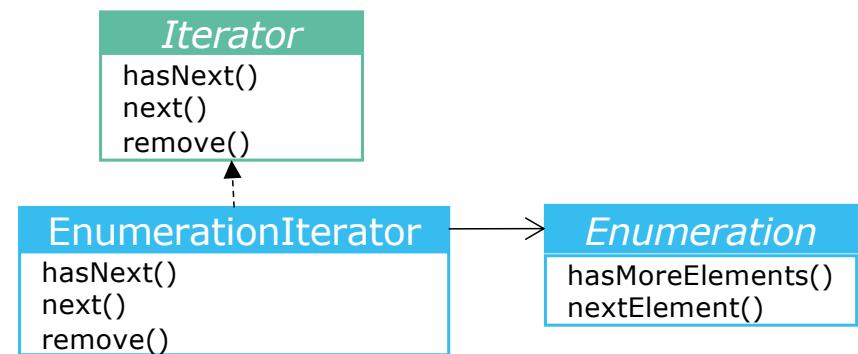
- Konvertierung der Schnittstelle in einer vom Client erwartete Schnittstelle durch einen Adapter
- **Kapselung** der Integration in einem Adapter, um Änderungen auf diesen zu beschränken

Adapter

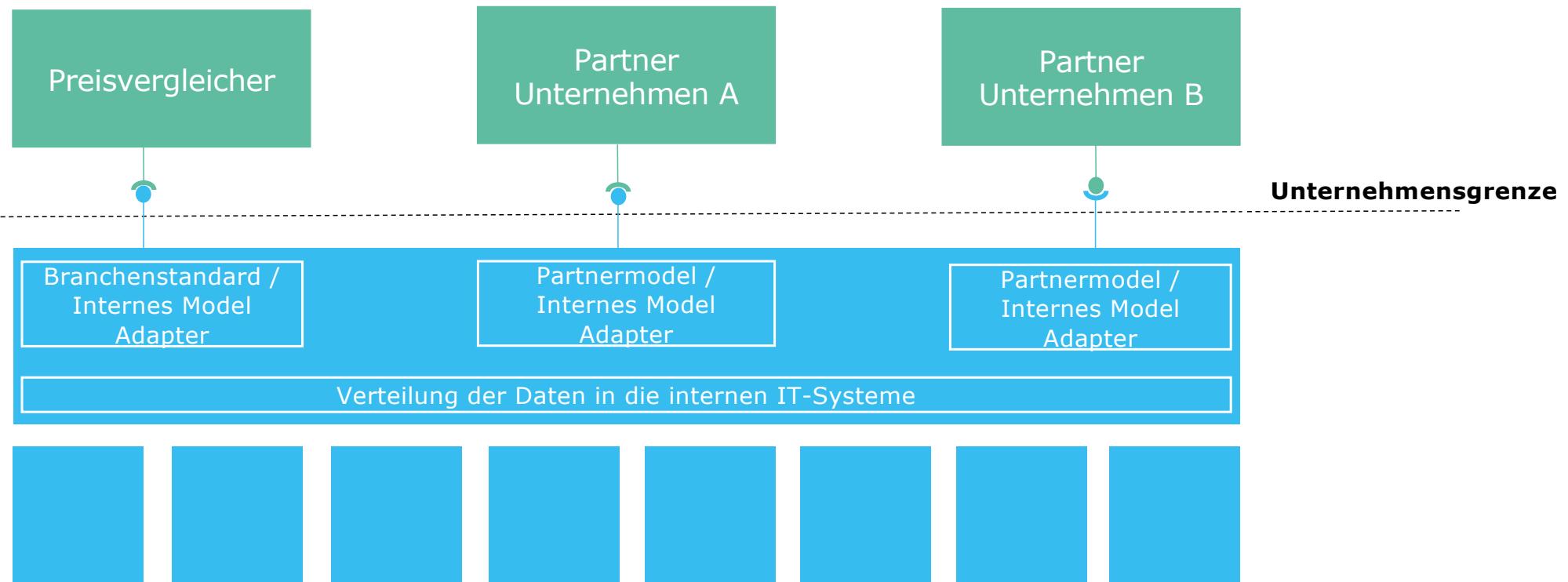
Mustervorlage



Beispiel



Adapter für externe & interne Datenmodell



Fassade



Zum Nachlesen

Problem

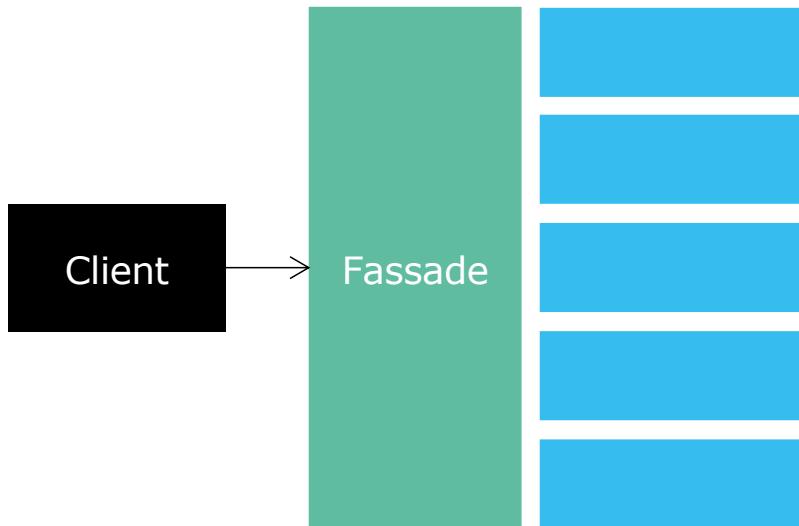
- Für Clienten ist die fachliche Verknüpfung unterschiedlicher Schnittstellen zu einem Basissystem zu aufwendig und Änderungen betreffen immer eine Vielzahl an Clients
- Information bzgl. der Nutzung der Schnittstellen sowie benötigte Metadaten und Daten sollen nicht öffentlich gemacht werden
- Klienten fordern eine einfache und performante Schnittstelle

Lösung

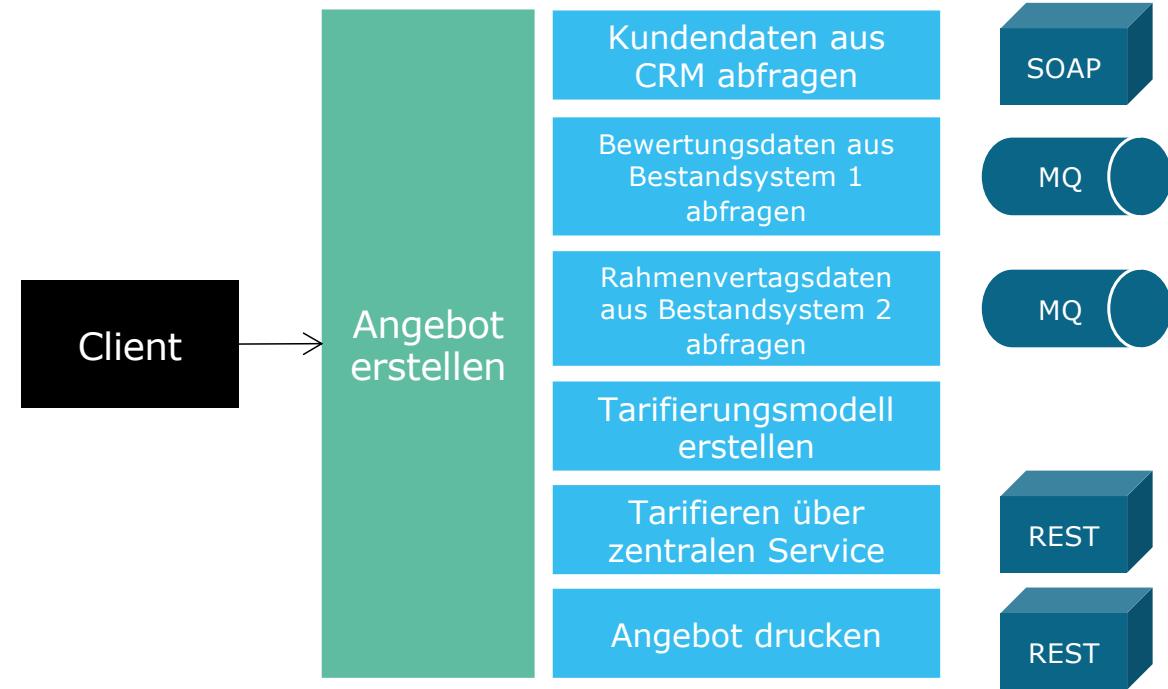
- Vereinfachte Schnittstelle (Fassade) zu einem Satz von Schnittstellen des Basissystems anbieten
- Schnittstelle der Fassade ist höherstufig und soll die Nutzung des Basissystems vereinfachen
- Kapselung und Information Hiding auf hohem Abstraktionsniveau
- Unterstützt lose Kopplung und hohe Kohäsion

Fassade

Mustervorlage



Beispiel



Proxy



Zum Nachlesen

Problem

- Zugriff auf eine Komponente / Klasse / Ressource ist teuer und ineffizient
- Die Komponente, die benutzt wird, ist nur über ein Netzwerk erreichbar
- Zugriff auf Operationen muss besonders abgesichert werden

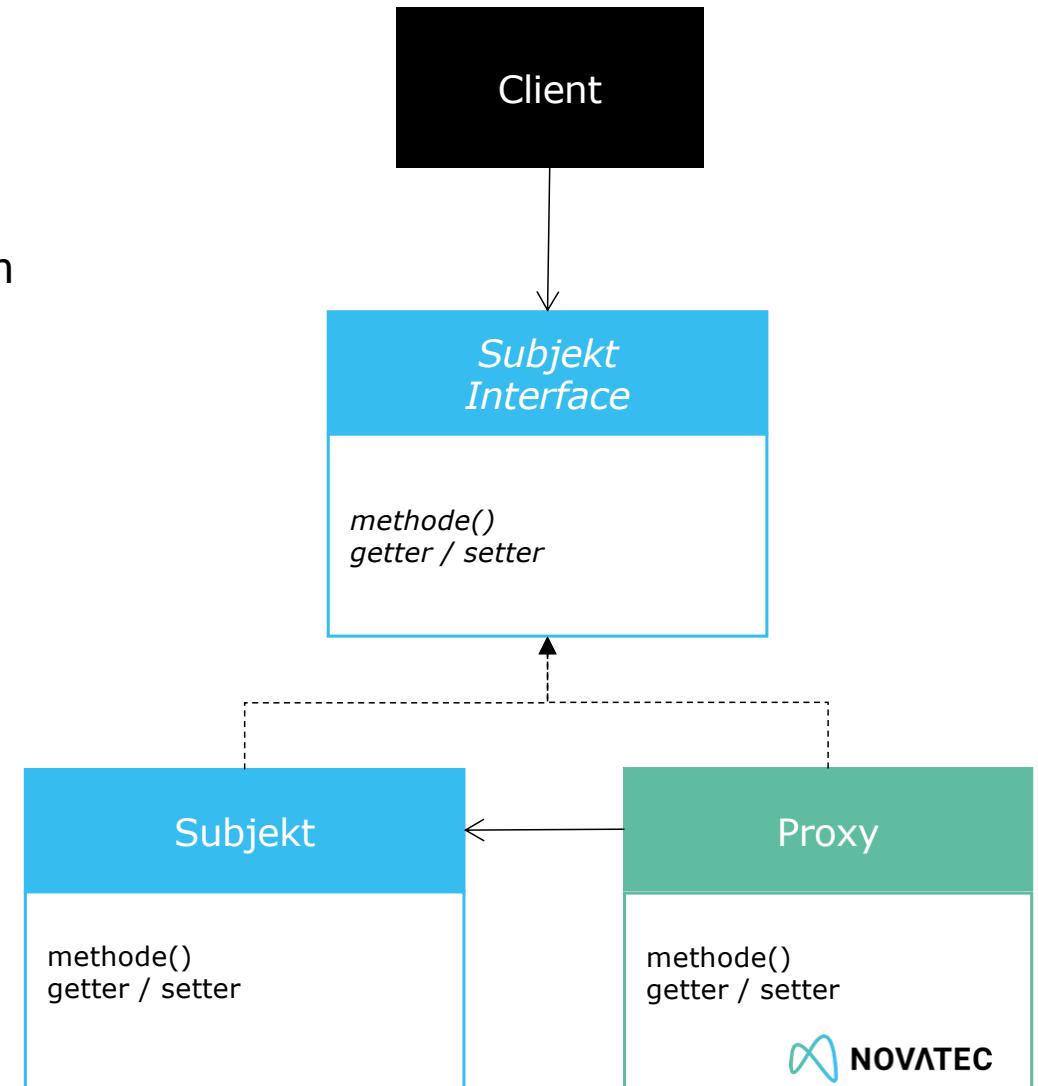
Lösung

- Einführung eines Stellvertreters (Proxy) auf den ein Zugriff erfolgen darf und der auf die gewünschte Komponente zugreifen darf
- Der Proxy definiert die gleichen Schnittstellen, wie die dahinter liegende Komponente und ist nur Proxy für diese
- Unterstützt Separation of Concerns Z.B. Trennung Security und Business
- Schutz-Proxy, Remote-Proxy, Virtueller Proxy, Dynamic Proxy

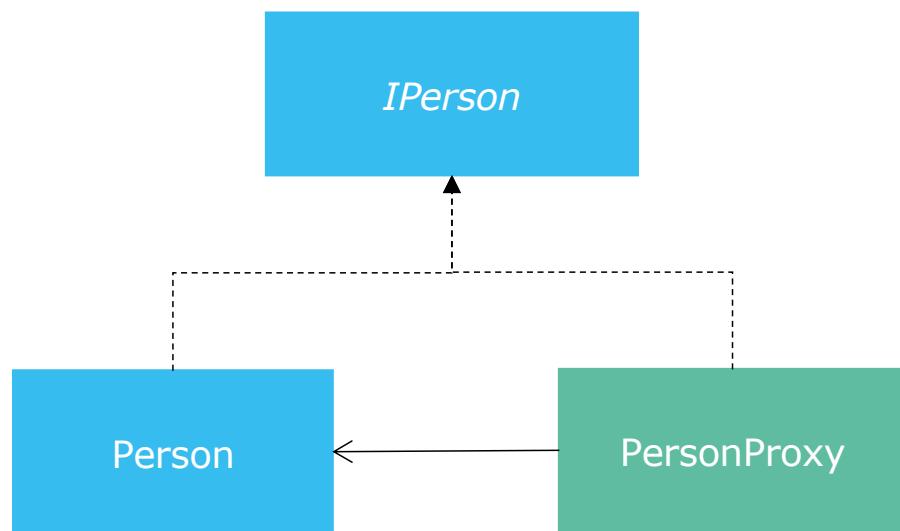
Proxy

- Problem
 - Objekterzeugung ist teuer bis extrem teuer
 - Hardware-Ressourcen sind meist geteilt
- Lösung

Lazy Loading via Proxy Pattern



Beispiel: Hibernate



1. Client lädt eine Entität
2. Hibernate erzeugt einen Proxy-Objekt und gibt dieses zurück
3. Das Proxy-Objekt ist nicht initialisiert
4. Bei Zugriff auf Datenfelder erfolgen im Hintergrund Abfragen auf die Datenbank (sprich auf die eigentliche Person)

<https://www.baeldung.com/hibernate-lazy-eager-loading>

Entwurfsmuster

Verhaltensmuster

Observer



Zum Nachlesen

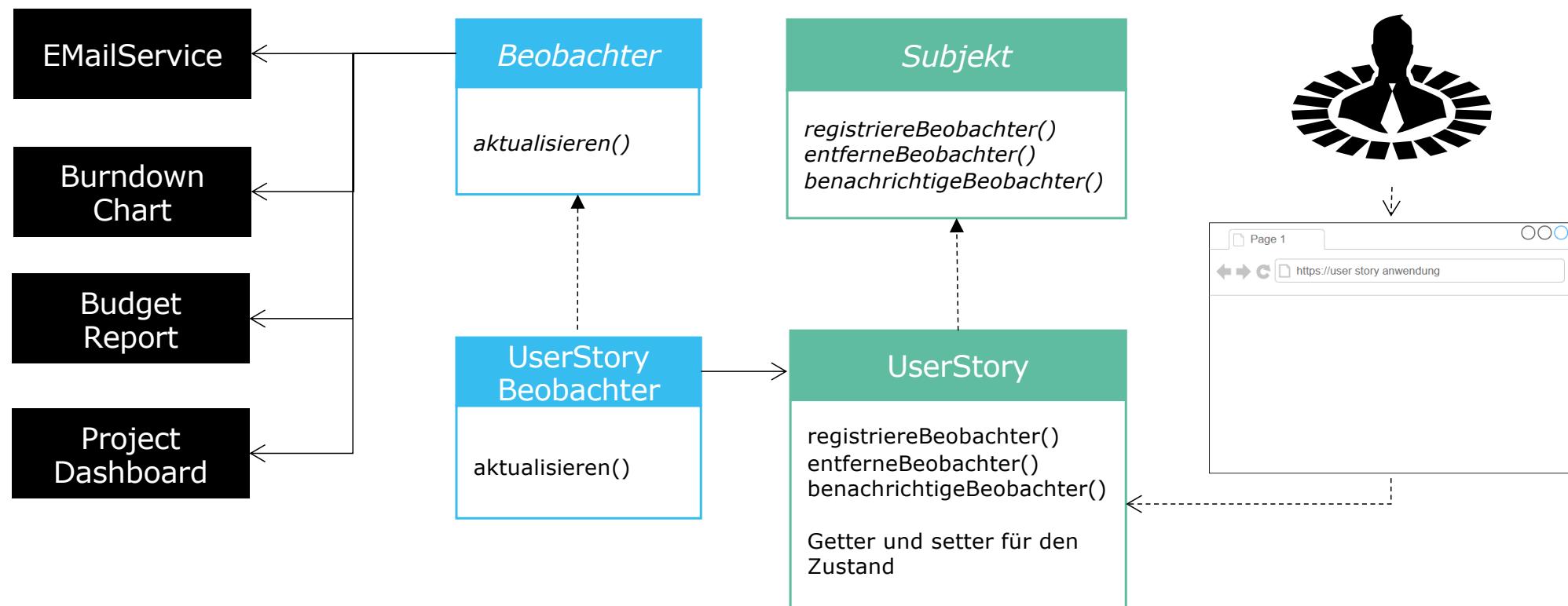
Problem

- Komponenten müssen miteinander kommunizieren, die Kopplung soll aber so gering wie möglich gehalten werden
- Eine Komponente interessiert sich für ein zu einem beliebigen Zeitpunkt eintretende Zustandsänderung einer anderen Komponente
- Darüber hinaus kann die Zustandsänderung diverse Auslöser haben, sodass die interessierten Komponenten ebenfalls von mehreren Auslösern gerufen werden muss

Lösung

- „Don't call us, we call you“!
(Hollywood-Regel)
- Realisierung einer Eins-zu-Viele Abhängigkeit zwischen Komponenten, die miteinander kommunizieren müssen, sodass alle abhängigen Komponenten benachrichtigt werden, wenn sich der Zustand ändert
- Unterstützt lose Kopplung zwischen Komponenten
- Inzwischen oft als Framework-Feature oder im JDK vorhanden

Observer



Strategy



Zum Nachlesen

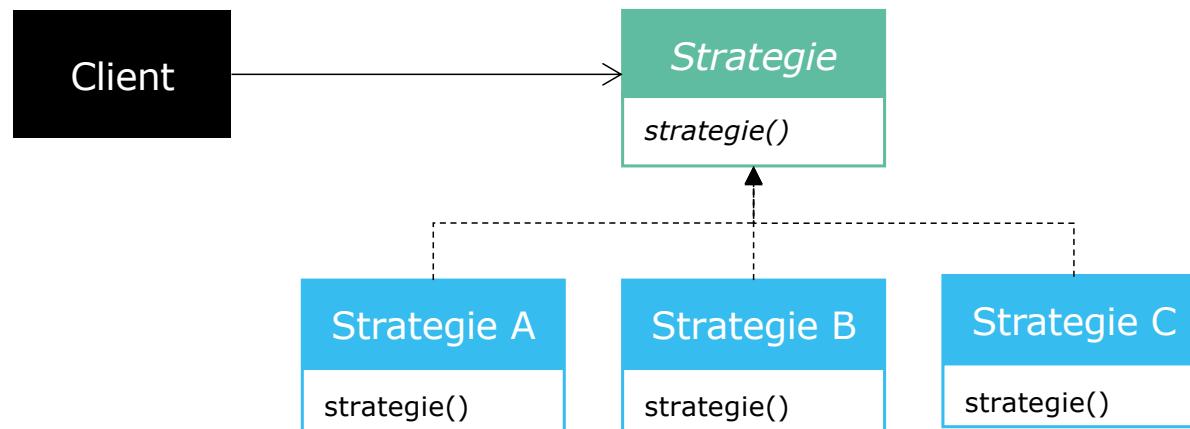
Problem

- Clients sollen nicht Abhängig von einer konkreten Implementierung (Algorithmus) sein. Vielmehr soll dieser Austauschbar sein.
- Gemeinsamkeiten von Klassen / Objekten sind zu klein um Vererbung richtig und ohne schwächen anzuwenden, aber zu groß um die Dinge völlig entkoppelt voneinander zu betrachten

Lösung

- Eine Familie von Algorithmen wird in einer 'Strategy' einzeln gekapselt und ermöglicht die Austauschbarkeit
- Der Entwurf gegen Schnittstellen ermöglicht Austauschbarkeit (zur Laufzeit)
Rechtlich bindende Kalkulation einer Versicherungs-police vs. Schnellberechnung bei Anfragen
- Composition over Inheritance bevorzugen
 - Vererbung hat Nachteile, vor allem wenn es sehr viel eingesetzt wird
 - Intransparente Wiederverwendung
 - Änderungen an Oberklassen wirken sich auf Unterklassen aus

Strategy



Das Problem mit Vererbung

Das Design ist bei 3, spätestens bei 4 nicht zu gebrauchen und muss grundlegend überarbeitet werden. Sonst werden mögliche Ziele wie Flexibilität, Erweiterbarkeit und Wartbarkeit nicht erreicht!

1



2

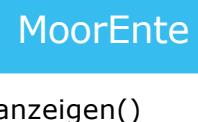
- Neue Ente „Gummi Ente“ quietscht statt quakt.
- Die anderen Enten sollen ab jetzt fliegen können.

3

- Neue Ente „Lock Ente“ die nicht fliegen, nicht quaken (ist stumm) und nicht schwimmen kann.

?

Enten mit weiteren Unterschieden und Ausnahmen



- fliegen() kommt in die Ente und wird bei GummiEnte überschrieben
- Das quaken() Problem wird dann analog behandelt

Lösung durch bekannte Prinzipien



Zum Nachlesen

Dependency-Inversion-Principle

- Implementierung gegen eine Schnittstelle und nicht gegen eine konkrete Klasse

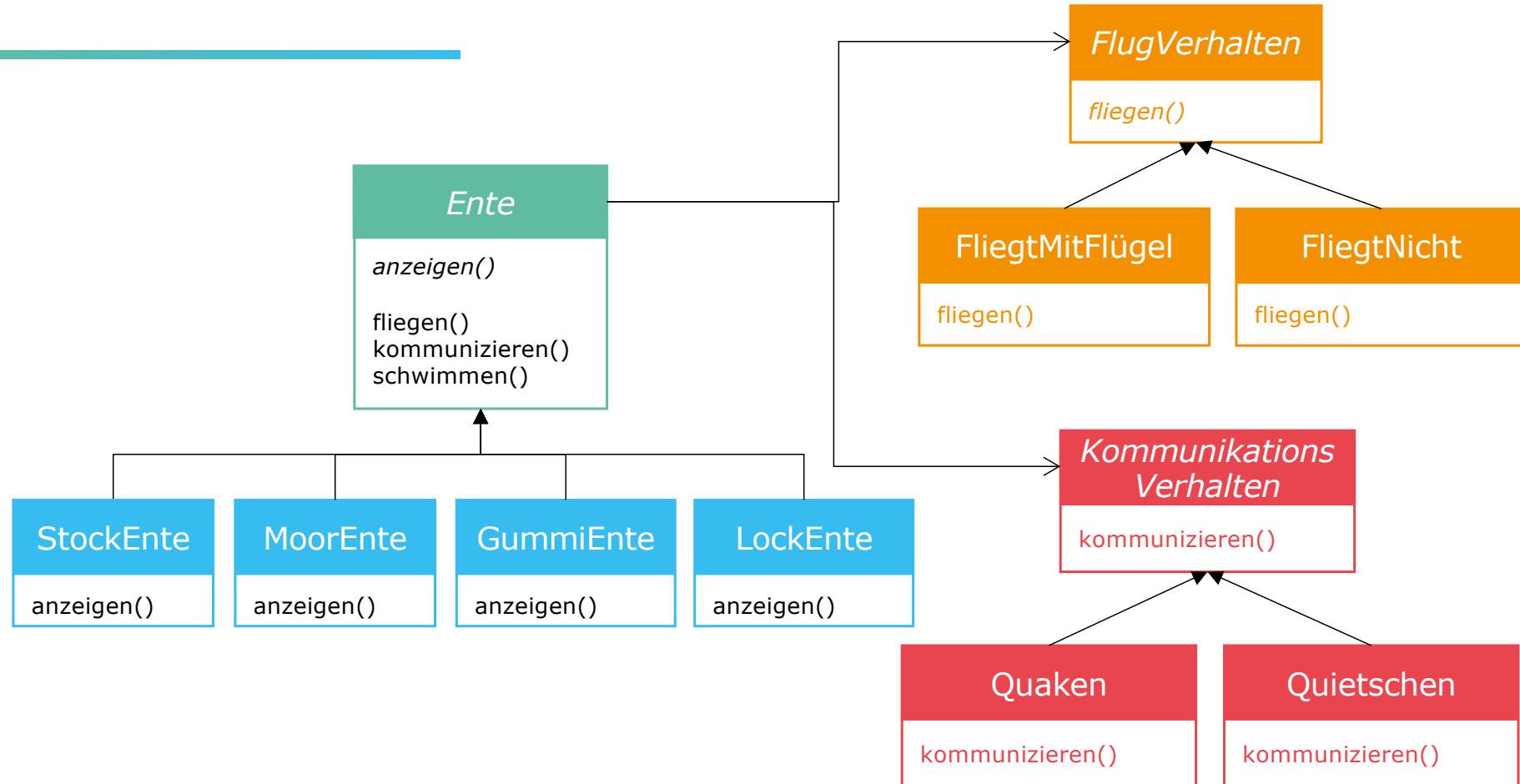
Komposition vor Vererbung

- Veränderlichen Teile identifizieren, auslagern und via Komposition nutzen

Beispiel für den Gewinn an Flexibilität:

- Eine Ente könnte aufgrund einer Verletzung ihr Flugfähigkeit verlieren
- Die quietschende Gummiente könnte mit einem Sprachmodul ausgestattet werden, dass sie quaken lässt

Strategy to the rescue



Empfehlungen für die Anwendung von Entwurfsmustern

- Entwürfe sollten immer so einfach wie möglich sein (KISS)!
- Ziehe Einfachheit dem Einsatz von Muster vor
- Verwende das Muster, wenn es die einfachste Lösung ist
- Refactoring – Bewertung des aktuellen Entwurfs auf Verbesserung durch geeigneten Einsatz von Mustern
- Zu viele Muster können auch zu unnötiger Komplexität führen

Rückblick: Muster für den Architekturentwurf

Architekturmuster

- Inversion of control, Dependency Injection
- Model View Controller, Model View Presenter, Presentation Abstraction Control
- Publish-Subscribe-Pattern, Messaging, Event-Driven
- Schichtenarchitektur, Microservices
- Pipes and Filters, Blackboard und Repository
- Client-Server
- Remote Procedure Call (RPC), REST

Entwurfsmuster

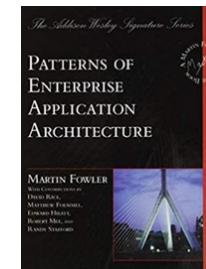
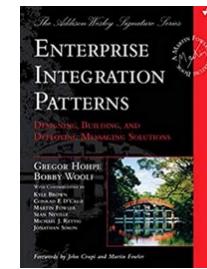
- Adapter
- Proxy
- Fassade
- Strategy



Zum Nachlesen

Literatur zu Entwurfs- und Architekturmuster

- Design Patterns von Gamma, Melm, Johnson und Vissides
(Deutsch und Englisch)
- Head First Design Patterns von Freeman und Robson
(Deutsch und Englisch)
- Entwurfsmuster - Das umfassende Handbuch von Geirhors
- Pattern Oriented Software Architecture von Buschmann, Henney, Schmidt
- Patterns of Enterprise Application Architecture von Fowler
- Enterprise Integration Patterns von Hohpe und Woolf



LZ 2-7: Abhangigkeiten von Bausteinen managen (R1)

Softwarearchitekt:innen verstehen Abhangigkeiten und Kopplung zwischen Bausteinen und konnen diese gezielt einsetzen. Sie:

- kennen unterschiedliche Arten der Abhangigkeiten von Bausteinen (beispielsweise strukturelle Kopplung uber Benutzung/Delegation, Schachtelung, Besitz, Erzeugung, Vererbung, zeitliche Kopplung, Kopplung uber Datentypen oder uber Hardware)
- konnen solche Arten der Kopplung gezielt einsetzen und die Konsequenzen solcher Abhangigkeiten einschatzen
- kennen Moglichkeiten zur Auflösung bzw. Reduktion von Kopplung und konnen diese anwenden, beispielsweise:
 - Muster (siehe Lernziel 2-5)
 - Grundlegende Entwurfsprinzipien (siehe Lernziel 2-6)
 - Externalisierung von Abhangigkeiten, d.h. konkrete Abhangigkeiten erst zur Installations- oder Laufzeit festlegen, etwa durch Anwendung von Dependency Injection.

Welche Abhängigkeiten gibt es?

```
public class ConsoleGreeter extends BaseGreeter implements Greeter {  
    private final Console console;  
  
    public GreetResult greet(GreetMessage message) {  
        var consoleMessage = new ConsoleMessage(message.getText());  
  
        console.write(consoleMessage);  
        return GreeterUtils.createSuccess();  
    }  
}
```

Grad der Kopplung

Kommunikation erfolgt über Nachrichten auf Basis von Ereignissen

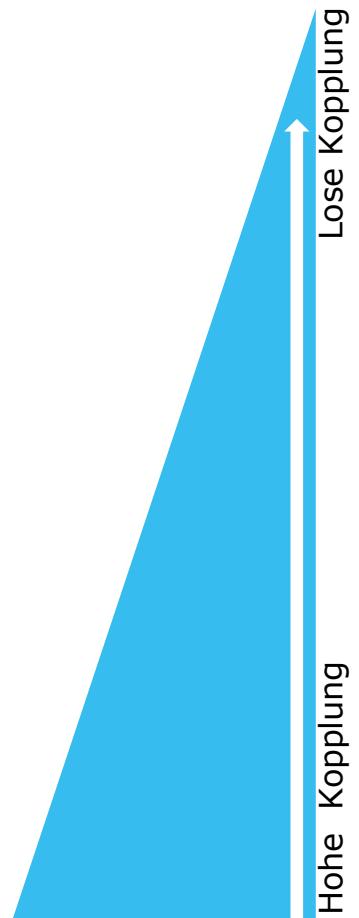
Asynchrone Kommunikation ist Entkopplung auf Ebene der Zeit, im vgl. zu synchroner Kommunikation.

Abhängigkeiten z.B. zu Services oder Hilfsklassen werden als Parameter übergeben (Dependency Injection)

Kommunikation zwischen Bausteinen erfolgt über Methodenparameter

Kommunikation zwischen Bausteinen erfolgt über globale Datenstruktur / Vererbung

Kommunikation erfolgt durch Zugriff auf die Internas (z.B. private Daten) des anderen Bausteins



Umgang mit Kopplung

Bewusster Einsatz

- Wiederverwendung dank Abhängigkeiten
- Funktionalität durch Komposition kleinerer Funktionalitäten
- Abhängigkeiten im Blick behalten

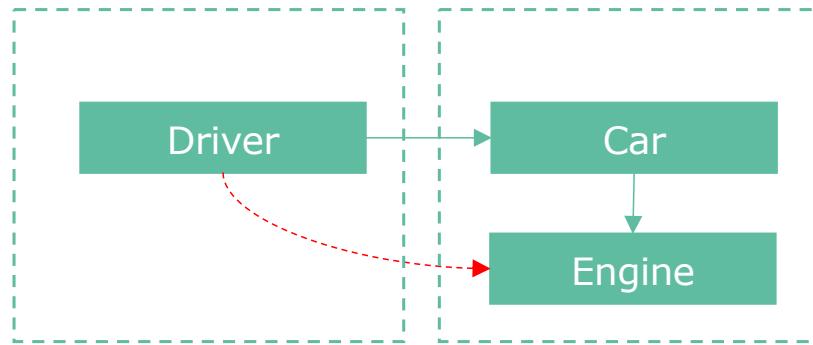
Vermeidung/Reduktion

- Kopplung lockern durch:
 - Muster
 - Prinzipien
 - Externalisierung

Kopplung reduzieren durch das Law of Demeter

- Spezialfall des Information Hiding
- Schnittstellen kapseln Dienste (=Funktionalität)
- Dabei ist es wichtig, dass der **Dienst vollständig übernommen** wird
- Dienste, die dem Aufrufer Internas zurückliefern, die der Aufrufer für weitere Aufrufe nutzt, verletzten das Prinzip

Indirekte Abhängigkeiten
vermeiden
Verletzung des Law of
Demeter



Law of Demeter

Verletzung des Law of Demeter

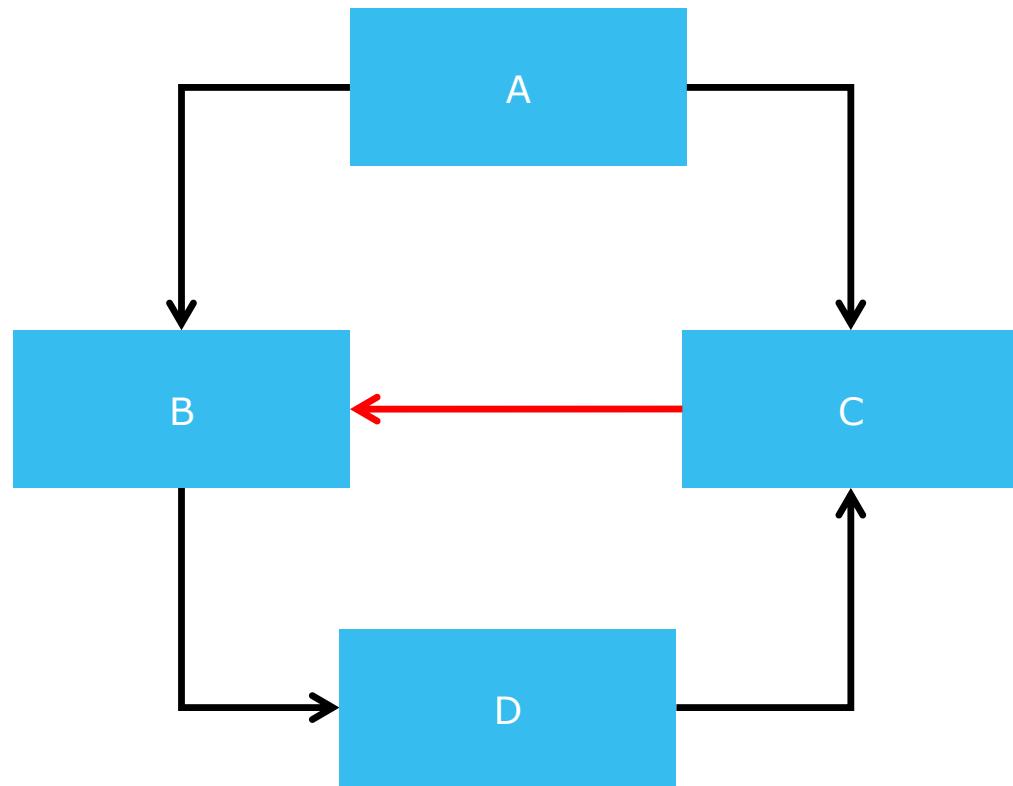
```
public class Car {  
    private Engine engine = new Engine();  
  
    public Car() {}  
  
    public Engine getEngine() {  
        return engine;  
    }  
  
    }  
  
Car car = new Car();  
car.getEngine().start();
```

Erfüllung des Law of Demeter

```
public class Car {  
    private Engine engine = new Engine();  
  
    public Car() {}  
  
    public void start() {  
        engine.start();  
    }  
  
    }  
  
Car car = new Car();  
car.start();
```

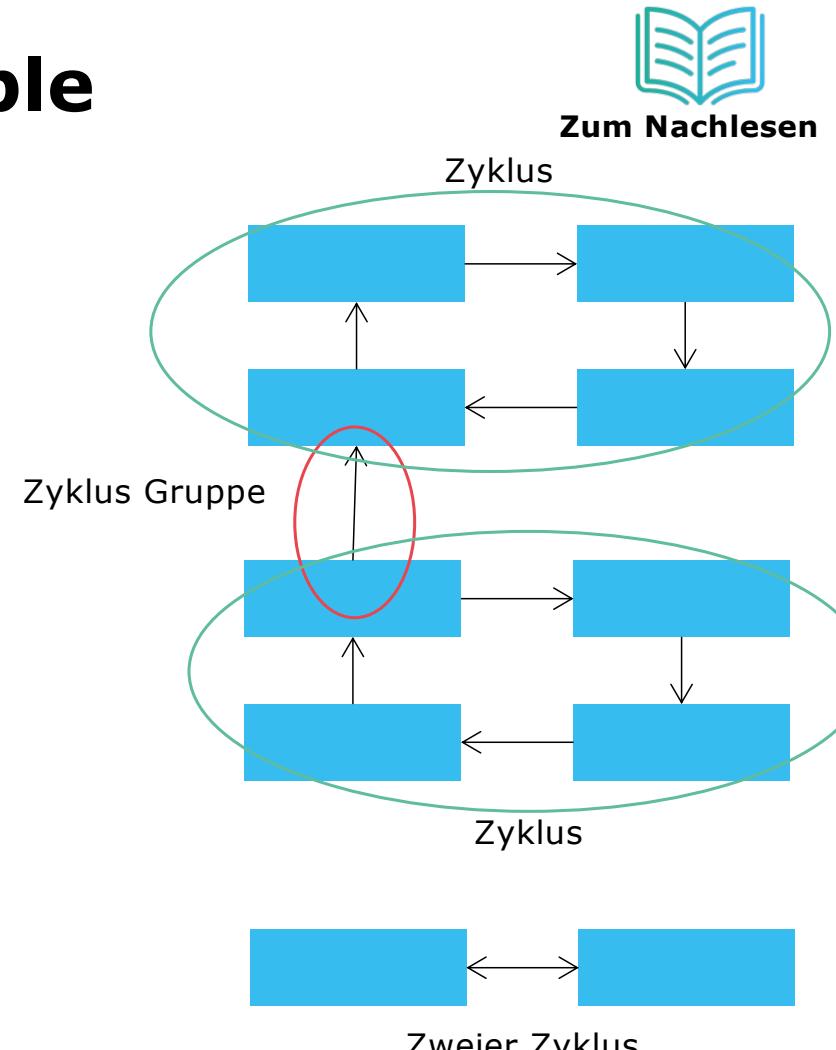
Acyclic dependencies principle (Robert C. Martin)

The **dependency** graph of packages or components should have no cycles



Acyclic dependencies principle (Robert C., Martin)

- **Zyklus**
Gelangt Element A durch seine Beziehungen zu anderen Elementen und dessen Beziehungen indirekt zu sich selbst zurück, befindet sich das Element in einem Zyklus.
- Zyklen können auf Klassenebene und Bausteinenebene vorkommen
- Es wird zwischen Zyklus und Zyklus Gruppe unterschieden
- Zyklus ist ein Beweis für eine gebrochene Modularisierung
Klassen / Bausteine in Zyklen sind größer und haben mehrere Aufgaben. Sie folgen nicht dem Single Responsible Principle, sie sind schwer verständlich und testbar.



Zyklen



Zum Nachlesen

Negative Effekte von Zyklen

- **Erweiterbarkeit** leidet, die Gefahr von Seiteneffekten nimmt zu
- **Testbarkeit** nimmt ab, da das isolierte Testen in einem Zyklus nicht möglich ist. Mocking ist sehr aufwendig und kompliziert.
- **Verständlichkeit** nimmt ab, da Zyklus / Zyklusgruppe als Ganzes verstanden werden muss anstatt jeder Baustein für sich
- **Evolution** - Zyklen ändern sich häufiger und Erweiterungen lassen den Zyklus wachsen.

Zyklen verhindern

- Single Responsibility „leben“ und Bausteine in einer hierarchischen Struktur ohne Rückbeziehung anordnen
- Wichtigste Regel einer Schichtenarchitektur einhalten – Kommunikation nur in niedrigere Schichten!
- Außerdem Anwendung von Composition over Inheritance, Dependency Inversion Principle sowie Entwurfsmustern

LZ 2-8: Qualitätsanforderungen mit passenden Ansätzen und Techniken erreichen (R1)

Siehe LZ 4

- Softwarearchitekt:innen kennen und berücksichtigen den starken Einfluss von Qualitätsanforderungen in Architektur- und Entwurfsentscheidungen, beispielsweise für:
 - Effizienz / Performance
 - Verfügbarkeit
 - Wartbarkeit, Modifizierbarkeit, Erweiterbarkeit, Adaptierbarkeit
- Sie können:
 - Lösungsmöglichkeiten, *Architectural Tactics*, angemessene Praktiken sowie technische Möglichkeiten zur Erreichung wichtiger Qualitätsanforderungen von Softwaresystemen (unterschiedlich für eingebettete Systeme bzw. Informationssysteme) erklären und anwenden
 - mögliche Wechselwirkungen zwischen solchen Lösungsmöglichkeiten sowie die entsprechenden Risiken identifizieren und kommunizieren.

LZ 2-9: Schnittstellen entwerfen und festlegen (R1-R3)

Softwarearchitekt:innen kennen die hohe Bedeutung von Schnittstellen. Sie können Schnittstellen zwischen Architekturnbausteinen sowie externe Schnittstellen zwischen dem System und Elementen außerhalb des Systems entwerfen bzw. festlegen.

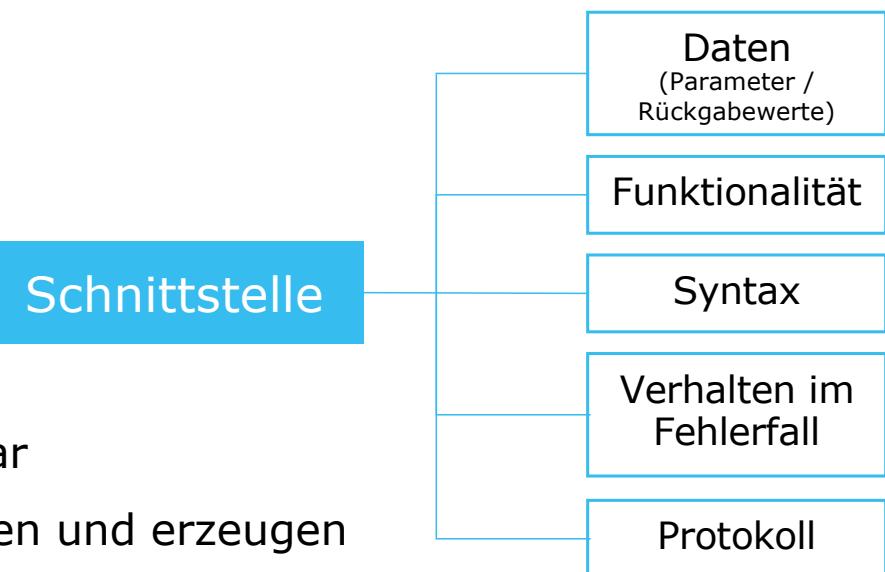
Sie kennen:

- wünschenswerte Eigenschaften von Schnittstellen und können diese beim Entwurf einsetzen:
 - einfach zu erlernen, einfach zu benutzen, einfach zu erweitern
 - schwer zu missbrauchen
 - funktional vollständig aus Sicht der Nutzer oder nutzender Bausteine.
 - die Notwendigkeit unterschiedlicher Behandlung interner und externer Schnittstellen
- unterschiedliche Implementierungsansätze von Schnittstellen (R3):
 - ressourcenorientierter Ansatz (REST, REpresentational State Transfer)
 - serviceorientierter Ansatz (wie bei WS-*/SOAP-basierten Webservices).

Schnittstellen

Schnittstellen

- Ist ein wohldefinierter Zugangspunkt zum System, Service oder Baustein
- Beschreibt die Eigenschaften des Zugangspunkts (Vertrag)
- Sind Grundlage für ein modulares System und lose Kopplung
- Spiegeln den Verantwortungsbereich von Baustein / Service wider
- Stellt eine Consumer und Provider Beziehung dar
- Über Schnittstellen arbeiten Bausteine zusammen und erzeugen letztendlich den Mehrwert des Gesamtsystems
- Über Schnittstellen werden Daten ausgetauscht und / oder Funktionen anderer Komponenten genutzt



Schnittstellen sind erfolgsentscheidendes Designmittel für eine gute Architektur

- Schnittstellen entstehen unter Anwendung von Prinzipien
Geheimnisprinzip, Interface Segregation Principle, Law of Demeter, Command Query Responsibility Segregation
- Das Finden der Schnittstellendefinition basiert auf Kommunikation zwischen Consumer und Provider
- Eine Schnittstelle stellt eine direkt verwendbare atomare Informationseinheiten bereit
Kein String mit mehreren Informationsobjekten, welcher erst geparst werden muss
- Eine Schnittstelle sollte sprechende Namen verwenden
(z.B. aus dem Domänenmodell)
- Schnittstellen können innerhalb eines Systems gelten (intern) oder auch öffentlich sein (extern/public API)

Entwerfen und festlegen

Varianten

- Synchrone Kommunikation
Funktion, Methode, Service, Request / Response
- Asynchrone Kommunikation
Callback, Return Address
- Fire and Forget
- Publish Subscribe

Mindestanforderungen

- Intuitiv verständlich
- Darf nicht falsch oder missbräuchlich nutzbar sein
- Einfachen Client Code ermöglichen
- Leicht erweiterbar
- Vollständig aus Nutzersicht (Law of Demeter)
- Minimal
- Stabil
- Dokumentiert
- Konsistent
Namensgebung / Datenformat

Was bedeutet einfacher Client Code? Beispiel: Abfrage einer Liste von Fahrzeugen

```
HTTP 200 : Fahrzeug  
{  
  "fahrzeuge": []  
}
```



```
HTTP 200 : Fahrzeug  
{  
  "errorCode": 400  
  "errorMessage": "NOT FOUND"  
}  
  
//404 : INTERNAL_SERVER_ERROR
```

173

**Standardisierte
Nachrichten
für alle Services**

HTTP 200: Fahrzeug
HTTP 200: Fahrzeug[]
HTTP 404: Not found

Service

HTTP 200: Fahrzeug

Service

Individualablauf für einen Service

1. Objekt lesen und Status prüfen
2. Aussage der Message verstehen
3. Eigentlich erwartetes Verhalten herstellen (Leere Liste)
4. Logik zum Aussteuern eines möglichen Fehlers implementieren

Beispiel: Einfachen Client Code ermöglichen



Zum Nachlesen

- Der Service Provider liefert bei einer leeren Liste den HTTP Status Code 404 zurück, anstatt gemäß REST Architekturstil den HTTP Status Code 200 und eine leere Liste
- Den Client stellt dies vor erhöhte Komplexität als Notwendig. Die Response muss unterschieden werden zu 404 mit leerer Liste und wirklichen 404 Fehlern
- Client Code bleibt hierfür zwangsläufig nicht einfach!

Qualitätsanforderungen an Schnittstellen

- Fehlerbehandlung und Kommunikation (Robustheit)
- Authentifizierung und Autorisierung (Sicherheit)
- Datenverschlüsselung (Sicherheit)
- Zertifikats- und / oder Tokenbehandlung (Sicherheit)
- Service Level Agreement, wie Durchsatz oder Antwortzeit (Effizienz)
- Protokollierung für Kostenabrechnung und Nachvollziehbarkeit (Funktionalität, Sicherheit)

Technologische Umsetzungsmöglichkeiten

- Öffentliche Java Methode einer Klasse
- Enterprise Java Beans Remote Interface
- JSON over Http / XML over Http
- SOAP over Http
- Advanced Message Queuing Protocol (AMQP)
- Websockets

Quellen

[Dowalil 2018] Dowalil, Herbert: Grundlagen des modularen Softwareentwurfs. Der Bau langlebiger Mikro- und Makro-Architekturen wie Microservices und SOA 2.0. 1. Aufl. München

[Ford, Parsons und Kua 2017] Neal Ford, Parsons Rebecca, Kua Patrick: Building Evolutionary Architectures. 1. Aufl. Sebastopol.

[Freeman 2006] Freeman, Eric [u.a.]: Entwurfsmuster von Kopf bis Fuß. 1. Aufl. Köln

[Geirhos 2015] Geirhos, Matthias: Entwurfsmuster. Das umfassende Handbuch. 1. Aufl. Bonn

[Gharbi 2018] Gharbi, Mahbouba [u.a.]: Basiswissen für Softwarearchitekten. Aus- und Weiterbildung nach iSAQB-Standard zum Certified Professional for Software Architecture – Foundation Level. 3. überarb. Aufl. Heidelberg

[Lilienthal 2016] Lilienthal, Carola: Langlebige Software-Architekturen. Technische Schulden analysieren, begrenzen und abbauen. 1. Aufl. Heidelberg

[Newman 2021] Newman, Sam: Building Microservices. 2. Aufl. Sebastopol.

[Starke 2015] Starke, Gernot: Effektive Software-Architekturen: Ein praktischer Leitfaden. 7. überarb. Aufl. München

[Vernon 2017] Vernon, Vaughn: Domain-Driven Design kompakt. Leitfaden. 1. Aufl. Heidelberg

[Vogel 2009] Vogel, Oliver [u.a.]: Software-Architektur. Grundlagen – Konzepte – Praxis. 2. Aufl. Heidelberg

[Wolff 2016] Wolff, Eberhard: Microservices: Grundlagen flexibler Softwarearchitekturen. Leitfaden. 1. Aufl. Heidelberg

Icons made by <https://www.freepik.com/> , <https://www.flaticon.com/> & <https://icons8.de>

iSAQB Foundation Level

Softwarearchitekturen und Qualität
Beispielfragen



Beispielfragen

Frage 1

A-Frage: Wählen Sie die beste Option aus 1 Punkt

Welches Prinzip beschreibt, dass nur Abhängigkeiten zu Schnittstellen (Interfaces) bestehen sollen.

- (a) Das Dependency-Inversion Prinzip
- (b) Das Dependency-Injection Prinzip
- (c) Das Liskovsche Substitutionsprinzip

Beispielfragen

Frage 1

A-Frage: Wählen Sie die beste Option aus 1 Punkt

Welches Prinzip beschreibt, dass in Softwaresystemen nur Abhängigkeiten zu Schnittstellen (Interfaces) bestehen sollen.

- (a) Das Dependency-Inversion Prinzip
- (b) Das Dependency-Injection Prinzip
- (c) Das Liskovsche Substitutionsprinzip

Beispielfragen

Frage 2

P-Frage: Wählen Sie die **drei korrekten** Begriffe aus 1 Punkt

Welche Begriffe sind Bestandteile einer Whitebox-Sicht eines Bausteins

- (a) Beschreibung
- (b) Nicht-funktionale-Anforderungen
- (c) Interne Struktur
- (d) Implementierungstechnologie
- (e) Schnittstelle
- (f) Kommunikationsprotokolle

Beispielfragen

Frage 2

P-Frage: Wählen Sie die **drei korrekten** Begriffe aus 1 Punkt

Welche Begriffe sind Bestandteile einer Whitebox-Sicht eines Bausteins

- (a) Beschreibung
- (b) Nicht-funktionale-Anforderungen
- (c) Interne Struktur
- (d) Implementierungstechnologie
- (e) Schnittstelle
- (f) Kommunikationsprotokolle

Beispielfragen

Frage 3

K-Frage: Wählen Sie für jede Zeile „Richtig“ oder „Falsch“ aus 1 Punkt

Welchen Aussagen über das Proxy Pattern sind korrekt?

Richtig

Falsch

- | | | |
|--------------------------|--------------------------|--|
| <input type="checkbox"/> | <input type="checkbox"/> | (a) Das Pattern kann für den Remote-Aufruf verwendet werden, so dass der Proxy diese vor dem Aufrufer verbirgt. |
| <input type="checkbox"/> | <input type="checkbox"/> | (b) Das Pattern kann für Caching verwendet werden, indem Aufrufe einmal durchgeführt werden und anschließend aus dem Cache bedient werden. |
| <input type="checkbox"/> | <input type="checkbox"/> | (c) Das Pattern kann eingesetzt werden, um komplexe Schnittstellen für den Aufrufer zu vereinfachen. |
| <input type="checkbox"/> | <input type="checkbox"/> | (d) Das Pattern kann zur Zugriffskontrolle eingesetzt werden, um vor einem Aufruf die Rechteprüfung durchzuführen. |

Beispielfragen

Frage 3

K-Frage: Wählen Sie für jede Zeile „Richtig“ oder „Falsch“ aus 1 Punkt

Welchen Aussagen über das Proxy Pattern sind korrekt?

Richtig



Falsch



- (a) Das Pattern kann für den Remote-Aufruf verwendet werden, so dass der Proxy diese vor dem Aufrufer verbirgt.
- (b) Das Pattern kann für Caching verwendet werden, indem Aufrufe einmal durchgeführt werden und anschließend aus dem Cache bedient werden.
- (c) Das Pattern kann eingesetzt werden, um komplexe Schnittstellen für den Aufrufer zu vereinfachen.
- (d) Das Pattern kann zur Zugriffskontrolle eingesetzt werden, um vor einem Aufruf die Rechteprüfung durchzuführen.



iSAQB Foundation Level

Entwurf und Entwicklung von
Softwarearchitekturen

Selbstlernkontrolle



Beispielfragen

Wann ist eher ein Top-Down Vorgehen beim Architekturentwurf sinnvoll und in welchen Fällen ein Bottom-Up Vorgehen?

Beispielfragen

Wann ist eher ein Top-Down Vorgehen beim Architekturentwurf sinnvoll und in welchen Fällen ein Bottom-Up Vorgehen?

Top-Down	Bottom-Up
Beginn mit der Problemstellung	Beginn mit der konkreten Maschine / Lösung
Zerlegung in immer weitere Teilprobleme	Aufbau abstrakter Bausteine auf der vorhandenen Basis
Ergebnisse liegen sehr spät vor	Sehr schnelle Ergebnisse
Geringes Risiko ungeeignete Ergebnisse zu erzeugen	Teilergebnisse können ungeeignet zur weiteren Verwendung sein

Beispielfragen

Wie kann lose Kopplung in einer Programmiersprache wie Java erzielt werden?

Beispielfragen

Wie kann lose Kopplung in einer Programmiersprache wie Java erzielt werden?

Factory, Strategy, Dependency Injection, Inversion of Control

Beispielfragen

Welche Services sind am engsten gekoppelt:

- Synchrone Aufrufe
- Asynchrone Aufrufe
- Die Kopplung ist unabhängig von der Art des Aufrufs

Beispielfragen

Welche Services sind am engsten gekoppelt:

- **Synchrone Aufrufe**
- Asynchrone Aufrufe
- Die Kopplung ist unabhängig von der Art des Aufrufs

Beispielfragen

Code-Duplizierung ist ein Zeichen schwacher Kohäsion.

- Wahr
- Falsch

Beispielfragen

Code-Duplizierung ist ein Zeichen schwacher Kohäsion.

- **Wahr**
- Falsch

Beispielfragen

Was beinhaltet die Abkürzung SOLID?

- Single Responsibility Principle (SRP), Open Closed Principle (OCP), Liskov Substitution Principle (LSP), Interface Segregation Principle (ISP), Dependency Inversion Principle (DIP)
- Single Responsibility Principle (SRP), Open Closed Principle (OCP), Lose coupling (LC), Interface Segregation Principle (ISP), Dependency Inversion Principle (DIP)
- Single Responsibility Principle (SRP), Open Closed Principle (OCP), Liskov Substitution Principle (LSP), Interface Segregation Principle (ISP), Don't Repeat Yourself (DRY)
- Single Responsibility Principle (SRP), Open Closed Principle (OCP), Lose coupling (LC), Interface Segregation Principle (ISP), Don't Repeat Yourself (DRY)

Beispielfragen

Was beinhaltet die Abkürzung SOLID?

- **Single Responsibility Principle (SRP), Open Closed Principle (OCP), Liskov Substitution Principle (LSP), Interface Segregation Principle (ISP), Dependency Inversion Principle (DIP)**
- Single Responsibility Principle (SRP), Open Closed Principle (OCP), Lose coupling (LC), Interface Segregation Principle (ISP), Dependency Inversion Principle (DIP)
- Single Responsibility Principle (SRP), Open Closed Principle (OCP), Liskov Substitution Principle (LSP), Interface Segregation Principle (ISP), Don't Repeat Yourself (DRY)
- Single Responsibility Principle (SRP), Open Closed Principle (OCP), Lose coupling (LC), Interface Segregation Principle (ISP), Don't Repeat Yourself (DRY)

Beispielfragen

Das "Law of Demeter" hilft die Kopplung zwischen Klassen zu reduzieren.

- Wahr
- Falsch

Beispielfragen

Das "Law of Demeter" hilft die Kopplung zwischen Klassen zu reduzieren.

- **Wahr**
- Falsch

Beispielfragen

Nennen Sie drei Anwendungsbeispiele für Pipes und Filters.

Beispielfragen

Bei folgende drei Mustern handelt es sich um Strukturmuster:

- **Adapter, Fassade, Proxy**
- Fassade, Observer, Adapter
- Adapter, Mediator, Fassade

Beispielfragen

Mediator ist ein Strukturmuster.

- Wahr
- Falsch

Beispielfragen

Mediator ist ein Strukturmuster.

- Wahr
- **Falsch**

Beispielfragen

Dependency Injection wird auch oft beim Testen verwendet.

Beschreiben Sie den Zusammenhang zwischen Testen und Dependency Injection.

Beispielfragen

Dependency Injection wird auch oft beim Testen verwendet.

Beschreiben Sie den Zusammenhang zwischen Testen und Dependency Injection.

Mit entsprechender Frameworkunterstützung lassen sich Tests i.d.R. einfacher schreiben, da Mock-Abhängigkeit als auch die Testklasse nur deklariert aber nicht „manuell“ mit new erzeugt werden müssen.



Novatec Consulting GmbH
Dieselstraße 18/1
D-70771 Leinfelden-Echterdingen

T. +49 711 22040-700
info@novatec-gmbh.de
www.novatec-gmbh.de