

# Entwurfsprinzipien für Moderne Verteilte Anwendungen

(📖 *Design Principles and Design Patterns for Distributed Applications*)

Dozent: Prof. Dr. Michael Eichberg  
Kontakt: [michael.eichberg@dhbw.de](mailto:michael.eichberg@dhbw.de)  
Version: 1.0.1

---


Folien: <https://delors.github.io/ds-se-entwurfsprinzipien/folien.de.rst.html>  
<https://delors.github.io/ds-se-entwurfsprinzipien/folien.de.rst.html.pdf>  
Fehler melden: <https://github.com/Delors/delors.github.io/issues>

# 1. Entwurfsprinzipien

---

# Entwurfsziele

Die Entitäten (Klassen, Module, Komponenten, Services...) unseres Entwurfs können:

- unabhängig von einander von einem "kleinen" Team iterativ entwickelt werden  
Dies setzt die - unabhängige Testbarkeit - voraus.
- unabhängig von einander bereitgestellt ( *to deploy*) werden
- unabhängig von einander gewartet und weiterentwickelt werden

## Hinweis

Diese Kriterien erlauben es uns einen „fertigen“ Entwurf zu beurteilen ohne zu sagen, wie das Ziel erreicht werden kann.

---

Die unabhängige Testbarkeit ersetzt aber nicht die Notwendigkeit von Integrationstests.

# Design needs Principles!

Oder welche Entität ist von wem, wann, warum und in welcher Weise abhängig?

Welche Entität soll mit welcher zusammen definiert werden?

[Martin2017]

---

Es muss Leitlinien geben, die uns helfen einen guten Entwurf zu erstellen, der die genannten Ziele erreicht. Weiterhin muss klar sein wie dieser beurteilt werden kann. d. h. Code darf nicht beliebig „platziert“ werden; Schnittstellen sollten nicht aus dem Bauch heraus entworfen werden.

Die folgenden Prinzipien wurden (zumindest teilweise) im Kontext der objekt-orientierten Programmierung identifiziert und beschrieben; passen jedoch auf verschiedensten Abstraktionsgeraden, deswegen ist im Folgenden auch von Entitäten die Rede.

## Kopplung (🚩 *coupling*)

Kopplung beschreibt die Stärke der Abhängigkeit zwischen verschiedenen Entitäten<sup>[1]</sup>.

Eine Entität E1 ist mit Entität E2 verbunden, wenn E1 direkt oder indirekt E2 benötigt.

Jedoch ist Kopplung nicht gleich Kopplung:

- statische und dynamische Kopplung
- Code-basierte und Daten-basierte
- ...

---

*Dynamische Kopplung* entsteht zur Laufzeit durch den Austausch von Nachrichten, statische Kopplung zur Compilezeit.

(*Temporale Kopplung* bezieht sich darauf, dass etwas gleichzeitig ausgeführt wird.)

---

[1] Eine Entität kann z. B. eine Methode, Klasse, Modul, Package, Komponente oder Service sein.

## Hohe statische Kopplung (🇺🇸 *high (static) coupling*)

Eine Entität mit hoher Kopplung ist nicht wünschenswert:

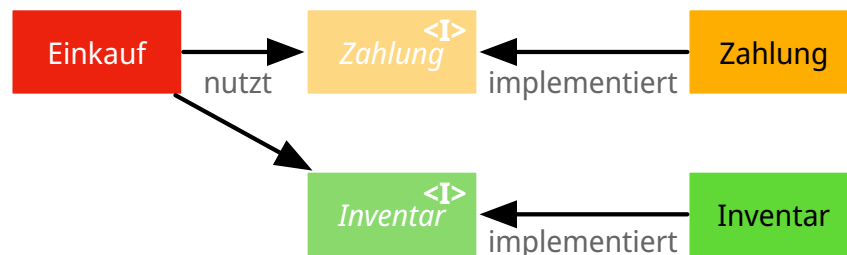
- Änderungen in verwendeten Entitäten erfordern (oft) lokale Anpassungen (mind. neues Testen)
- sie sind schwerer zu verstehen
- sie sind schwerer wiederzuverwenden, da die Verwendung auch aller weiteren Entitäten notwendig ist von denen die Entität abhängt

---

Hohe Kopplung ist aber nicht per-se schlecht! Eine hohe Kopplung an Dinge, die extrem stabil sind, ist im Allgemeinen unkritisch.

## Niedrige statische Kopplung (🇺🇸 *low (static) coupling*)

- Eine niedrige Kopplung unterstützt den Entwurf von vergleichsweise unabhängigen und deswegen besser wiederverwendbaren Entitäten.
- „generische“ Entitäten mit einer hohen Wiederverwendungswahrscheinlichkeit sollten eine geringe Kopplung aufweisen.

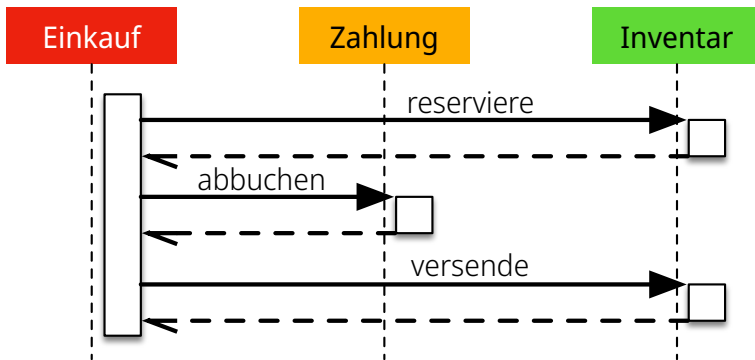


---

Keine Kopplung ist (auch) nicht wünschenswert, da dies zu Entitäten führt, die alle Arbeit durchführen; weiterhin führt dies auch dazu, dass sich ggf. die Arbeit sehr viel schlechter aufteilen lässt und dann eine agile Entwicklung mit einem kleinen Team nicht mehr möglich ist.

Relevante Frage: Wer ist/sollte der Eigentümer der Schnittstellen sein? D. h. aus welcher Perspektive sollte die Schnittstelle entworfen werden?

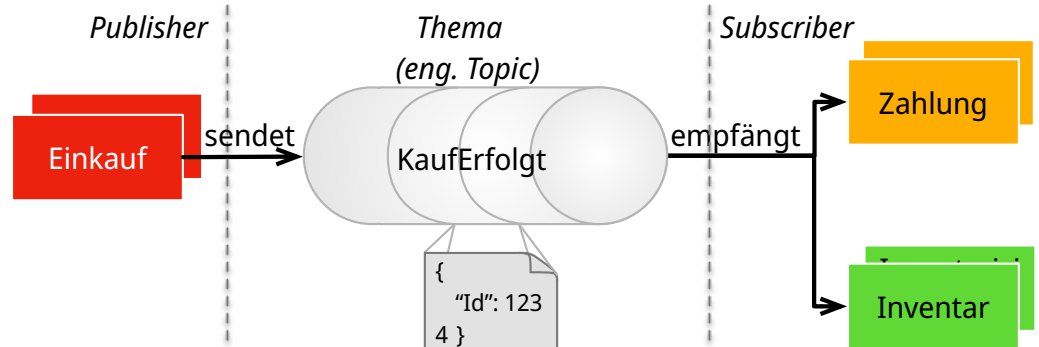
# Niedrige vs. hohe dynamische Kopplung



Anforderung-  
Antwort (Synchron)  
(📧 Request-response)

PubSub (Asynchron)

(📧 Publisher-subscriber)



## Beobachtungen

Die Skalierbarkeit der ersten Lösung hängt direkt von der Performance von Zahlung und Inventarisierung ab. Für die Verfügbarkeit der Gesamtlösung gilt das Gleiche.

PubSub ist hier deutlich mächtiger; Nachteile von PubSub liegen im Bereich:

- Indirektion
- garantierter Nachrichtenverbleib bzw. garantierte Nachrichtenverarbeitung
- verteilter Zustand bei Fehlern

In diesem Fall führt lose Kopplung zu sehr viel höherer Komplexität bei der Fehlerbehandlung.



# Zusammenhalt / Kohäsion (🇺🇸 Cohesion)

Der Zusammenhalt ist ein Maß der Stärke zwischen den Elementen einer Entität.

Ausgewählte Typen von Zusammenhalt:

- **Funktionale Kohäsion**

Die Elemente realisieren eine logische Funktion.

- ...

- **Logische bzw. technische Kohäsion**

Die Elemente stehen aus technischer Sicht in enger Beziehung.

- **Zufällig**

Es gibt keine relevante Beziehung zwischen den Elementen.

---

Eine wesentliche Frage ist: „Worin besteht der abgeschlossene Kontext, um etwas auf einer entsprechenden Abstraktionsebene kohäsiv erscheinen zu lassen?“

Technische Kohäsion entsteht zum Beispiel an der Schnittstelle für den Zugriff auf die Datenbank.

## Geringer Zusammenhalt (🇺🇸 *Low Cohesion*)

Entitäten mit geringem Zusammenhalt sind nicht wünschenswert!

Sie sind:

- schwer zu verstehen
- schwer wiederzuverwenden
- schwer zu warten und oft von Änderungen betroffen

---

Services mit einer geringen Kohäsion repräsentieren häufig Dinge auf sehr grober, abstrakter Ebene und haben Verantwortlichkeiten übernommen für Dinge, die sie bessere delegieren sollten.

## Hoher Zusammenhalt (🇺🇸 *High Cohesion*)

Alle Funktionalität und alle Daten sollten „natürlich“ zum Konzept gehören, das von der Entität realisiert wird.

---

Eine sehr niedrige Kopplung führt zwangsweise dazu, dass man zu viel Funktionalität in ein Modul/einen Service/eine Klasse/eine Funktion packt. Eine hohe Kohäsion führt zwangsweise dazu, dass man (sehr) viele Module/Services/Klassen/Funktionen benötigt, die häufig viele (starke) Kopplungen haben. Es gilt also die richtige Balance zu finden.

Kopplung und Kohäsion erlauben es uns einen Entwurf auf allen (Abstraktions-)ebenen zu beurteilen.

# Von Verantwortung und Zuständigkeit

- Der Verteilung von Zuständigkeiten ist die zentrale Tätigkeit während des Entwurfs.
- Entwurfsmuster, Idiome und Prinzipien helfen dabei die Zuständigkeiten zu verteilen.
- Bei der Verteilung von Zuständigkeiten gibt es eine große Bandbreite:
  - Deswegen gibt es gute und schlechte Entwürfe, schöne und hässliche, effiziente und ineffiziente.
  - Eine schlechte Wahl führt zu fragilen Systemen, welche schwer zu warten, zu verstehen, wiederzuverwenden oder zu erweitern sind.

## ? Frage

Wie verteilt man die Zuständigkeiten auf verschiedene Entitäten?

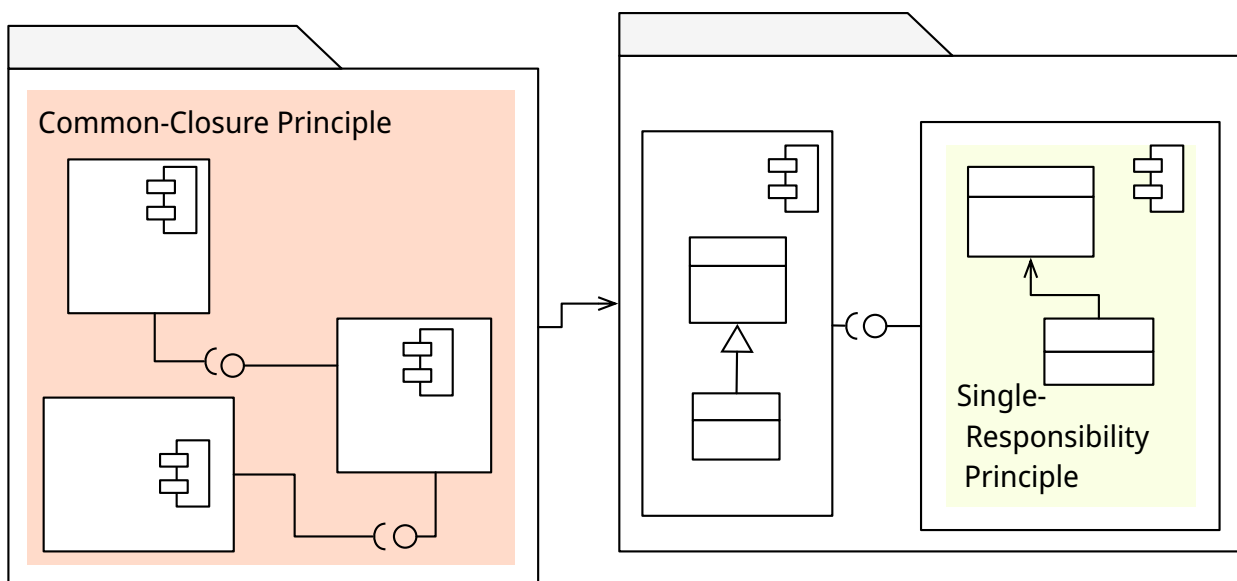
---

Bei der Verteilung der Zuständigkeiten gibt es eine große Bandbreite in Hinblick darauf, wie die nicht-funktionalen - und die funktionalen Eigenschaften einer Software realisiert werden.

Leitgedanke bzgl. funktionaler Kohäsion

**Code, der sich gemeinsam ändert,  
bleibt zusammen.**

Fasse die Dinge zusammen, die sich aus dem gleichen Grund und zur selben Zeit ändern.



---

#### Single Responsibility Principle (SRP):

Ein Modul sollte nur einem einzigen Akteur gegenüber verantwortlich sein. D. h. es sollte nur eine wohldefinierte Gruppe von Akteuren geben, die eine Veränderung veranlassen/verlangen können. Code, von dem verschiedene Akteure abhängen, sollte aufgeteilt werden.

#### Common Closure Principle (CCP):

Fasse in Komponenten solche Klassen zusammen, die sich aus dem gleichen Grund und zur gleichen Zeit ändern. Z. B. weil sie die gleichen Stakeholder haben oder die gleichen rechtlichen Grundlagen haben.

Die beiden Prinzipien sind eng miteinander verwandt. Das CCP ist ein Prinzip, das auf allen Abstraktionsgeraden angewendet werden kann. Das SRP ist - zumindest ursprünglich - ein Prinzip, das nur auf der Ebene von Klassen und Modulen angewendet wurde.



# Dependency Inversion Principle (DIP)

*...all well-structured [object-oriented] architectures have clearly defined layers, with each layer providing some coherent set of services through a well-defined and controlled interface...*

—Grady Booch

*High-Level-Module sollten nicht von Low-Level-Modulen abhängen. Beide sollten von Abstraktionen abhängen.*

*Abstraktionen sollten nicht von Details abhängen. Details sollten von Abstraktionen abhängen.*

—Agile Software Development; Robert C. Martin; Prentice Hall, 2003

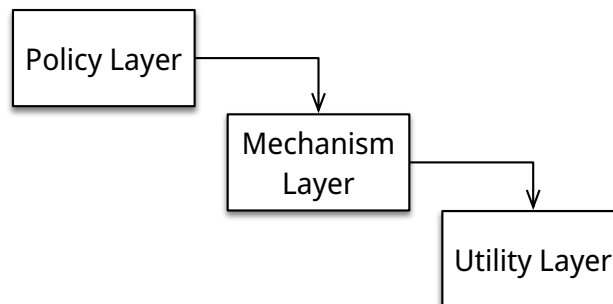
---

## Mögliche Interpretation

Je höher das Modul in einer Schichtenarchitektur positioniert ist, desto allgemeiner ist die Funktion, die es implementiert.

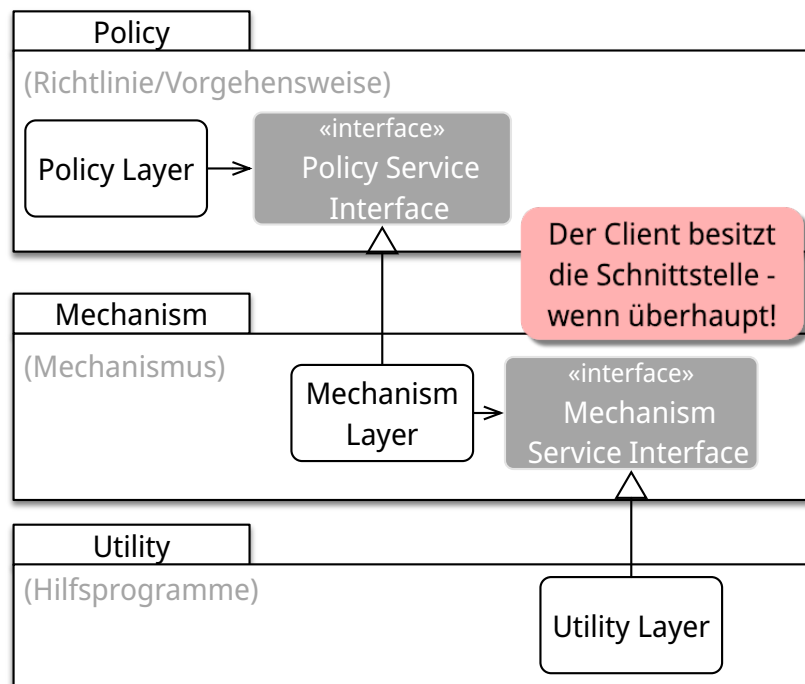
Je niedriger das Modul, desto detaillierter ist die Funktion, die es implementiert.

## Ein Klassendesign, dass das DIP verletzt:



**Die Einhaltung des DIP sollte auf allen Ebenen der Architektur sichergestellt werden.**

# Dependency Inversion Principle



---

## Begründung

Gute Softwarekonzepte sind in Module gegliedert.

High-Level-Module enthalten die wichtigen politischen Entscheidungen und Geschäftsmodelle einer Anwendung. Sie definieren die Identität der Anwendung.

Low-Level-Module enthalten detaillierte Implementierungen einzelner Mechanismen, die zur Umsetzung der Richtlinie benötigt werden.

# Open-closed Principle (OCP)

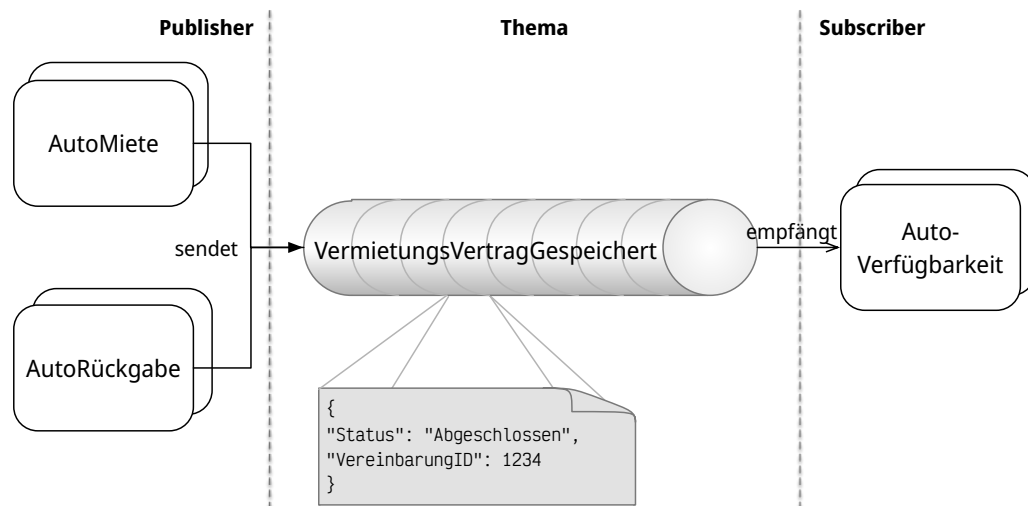
*Ein Softwareartefakt sollte offen für Erweiterungen, aber abgeschlossen gegenüber Veränderungen sein.*

*—Bertrand Meyer 1988, Robert C. Martin 1996*

---

D. h. es sollte möglich sein neue Erweiterungen zu realisieren ohne dass man die Software verändern, rekompilieren, neu bereitstellen (📦 *to deploy*) oder vergleichbares muss. Klassisches Beispiel ist ein Texteditor wie VS Code, welcher durch *Extensions/Plug-Ins* erweitert werden kann; d. h. es die Software is erweiterbar ohne das man diese neu kompilieren muss.

# Open-closed Principle - Case Study<sup>[2]</sup>



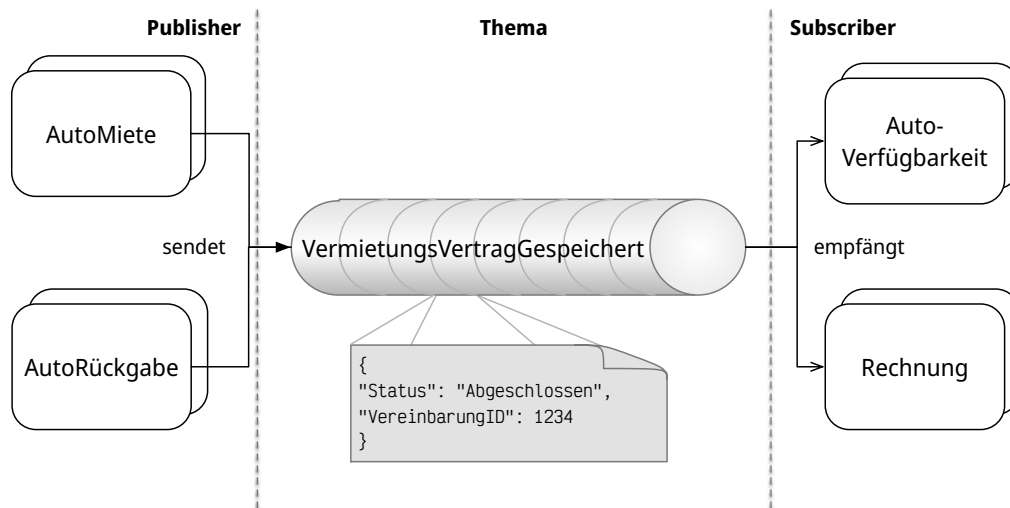
## ? Frage

Ist dieses Design offen für Erweiterungen?

In diesem Fall haben wir eine Architektur, die auf "Services" aufbaut welche lose gekoppelt sind und über Nachrichten kommunizieren.

[2] Beispiel nach David Llobrega, 2019

# Open-closed Principle - Case Study



## ? Frage

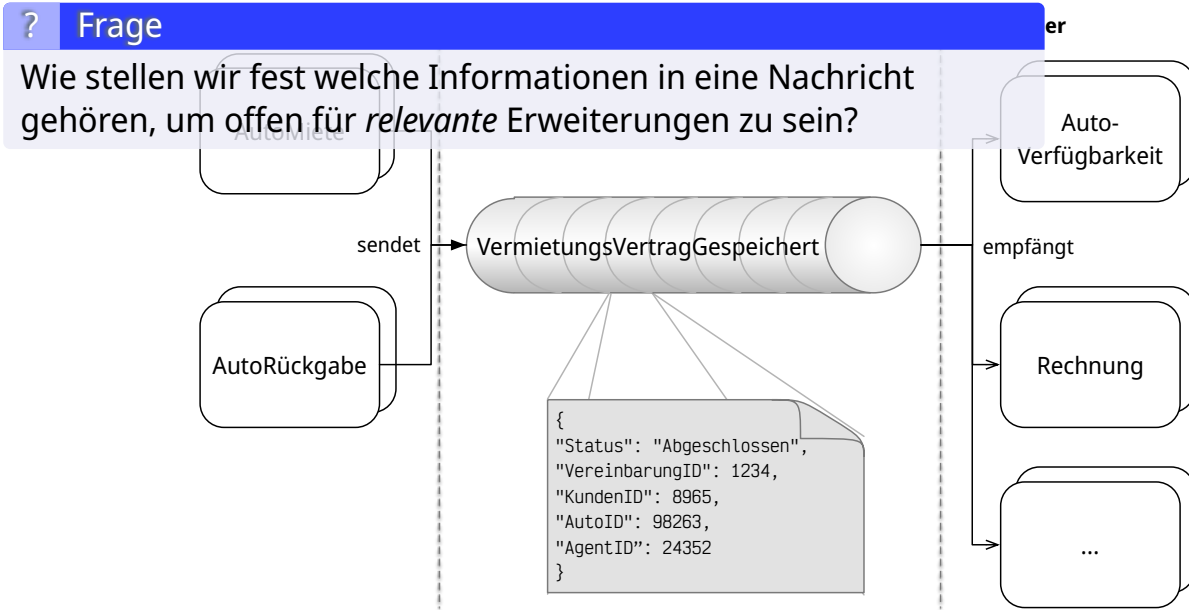
Ist dieses Design *wirklich* offen für Erweiterungen?

Das Problem ist, dass wir hier die Nachrichten - welche im Prinzip die Schnittstelle modellieren - relativ exakt an den Anforderungen des Services zur Bestimmung der Verfügbarkeit von Autos ausgerichtet haben.

Wie sähe in diesem Fall z. B. eine Erweiterung um einen Dienst für Kundenprämienberechnung aus?

Über die `VereinbarungID` bekommen wir Zugriff auf die Daten des Kunden aber dies fordert dann mehr als einen *Lookup* in einer Datenbank und ggf. auch das Einbinden mehrerer Dienste, was es zu vermeiden gilt, da die Kopplung unnötig ansteigen würde.

# Open-closed Principle - Case Study



Eine Antwort darauf liefern ggf. *Bounded-Context* aus dem *Domain-driven Design*

Ein *Bounded Context* ist ein Gültigkeitsbereich eines Domänenmodells, einer **Ubiquitous Language** und die Basis für die Organisation des Projekts.[...]

Eine Modellierung nach den Daten führt nicht zu sinnvollen Bounded Contexts, sondern eher zu komplexen Modellen. Wichtig ist, die Daten als Folge der Funktionalitäten zu modellieren.

Domain-driven Design behandelt Beziehungen zwischen *Bounded Contexts* im sogenannten *Strategic Design*.

<https://www.heise.de/hintergrund/Domain-driven-Design-und-Bounded-Context-Eigentlich-ganz-einfach-oder-4634258.html?seite=all>

# Liskov Substitution Principle (LSP)

*Subtypes must be substitutable for their base types.*

—*Barbara Liskov, 1988*

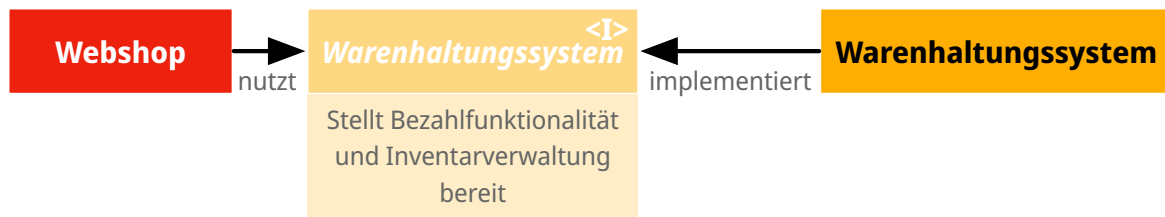
## **Moderne Interpretation**

Die Implementierungen von Schnittstellen müssen austauschbar sein.

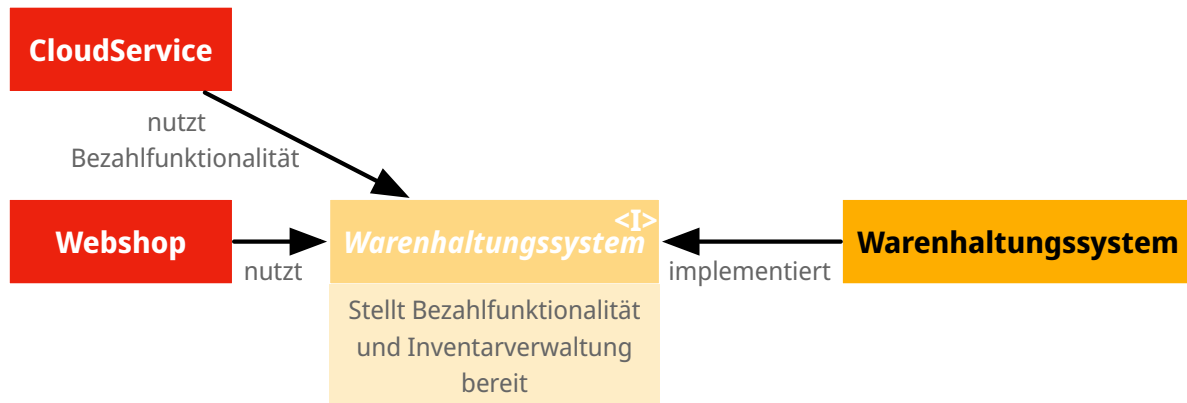
---

Im Original wird auf die Substituierbarkeit von Subtypen im Kontext der objekt-orientierten Programmierung eingegangen. Das Prinzip lässt sich aber auch auf andere Abstraktionsgeraden übertragen. Insbesondere auch auf die Ebene von Services deren Schnittstellen und Implementierungen.

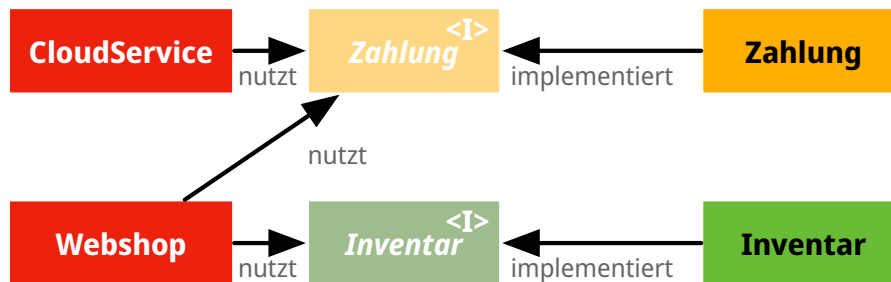
## Interface Segregation Principle & Common Reuse Principle



(Ausgangszustand)




(Geplante Erweiterung)



(Teilung der Schnittstelle)

## Achtung!

Hänge nicht von Dingen ab, die du nicht benötigst.

Segregation ( *Abtrennung*) bezeichnet hier die Aufspaltung eines bestehenden Interfaces bei dem die Teile abgespalten werden, die logisch zu einer anderen Funktionalität gehören. d. h. die von der Schnittstelle zur Verfügung gestellte Funktionalität ist nicht homogen und wird deswegen in verschiedene Teile aufgeteilt.



# Command-Query Separation (CQS)

*Methoden werden strikt aufgeteilt in:*

**Abfragen** (📄 Queries), die keine Veränderung des Objektzustandes erlauben

**Kommandos** (📄 Commands), die den Zustand verändern, aber keine Werte zurückliefern

—Bertrand Meyer, 1988

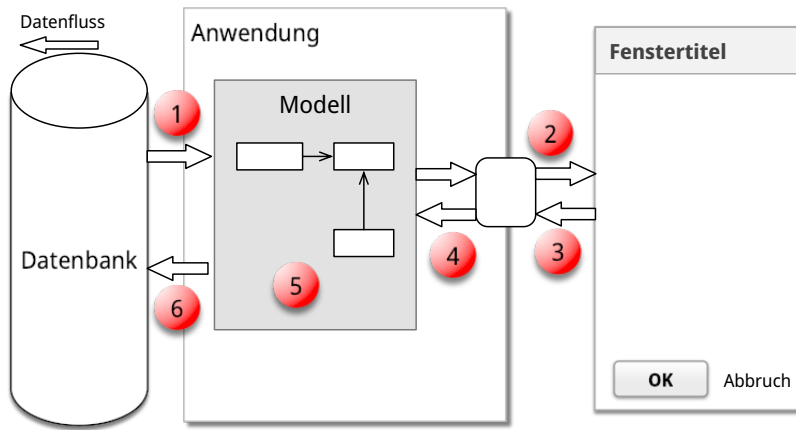
## Bemerkung

Auf der Ebene von nachrichten- bzw. ereignisgetriebenen Systemen wird CQS zum CQRS erweitert (Command-Query Responsibility Segregation).

---

Ein Java Iterator mit seiner „next“ Methode verletzt ganz klar dieses Prinzip!

# Traditionelle Interaktion mit Informationssystemen (CRUD)

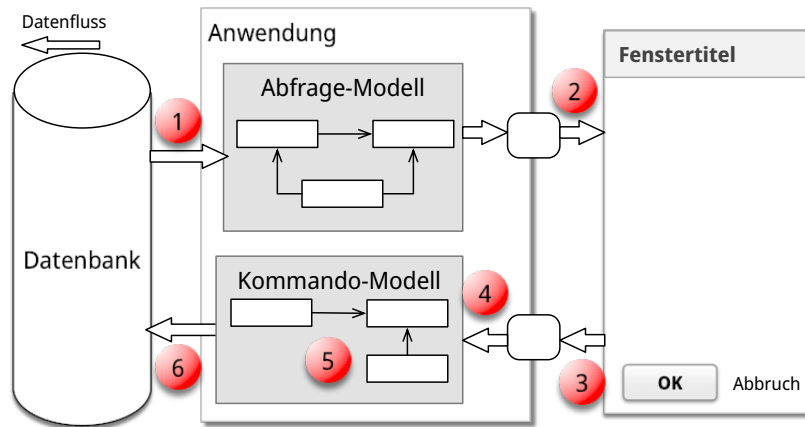


1. Modell liest von DB
2. Service stellt Information für Präsentationsschicht bereit
3. Nutzer hat Änderung vorgenommen
4. Weiterleitung der Änderung
5. Modell validiert
6. Modell aktualisiert DB

Darstellung nach **Martin Fowler**.

Darstellung einer Anwendung mit traditioneller Architektur.

# Command-Query Responsibility Segregation Principle



Darstellung nach **Martin Fowler**.

1. Abfrage-Modell liest von DB
2. Abfrage-Service stellt Information für Präsentationsschicht bereit
3. Nutzer hat Änderung vorgenommen
4. Weiterleitung der Änderung
5. Kommando-Modell validiert
6. Kommando-Modell aktualisiert DB

---

Command-Query-Responsibility-Segregation (CQRS) wendet das CQS-Prinzip an, indem es separate Abfrage- und Befehlsnachrichten zum Abrufen bzw. Ändern von Daten verwendet.

# Command-Query Responsibility Segregation Principle (CQRS)

## Einsatzszenarien

- Die Anzahl an Schreibe- und Leseoperationen ist extrem unterschiedlich.
- Die Datenmodelle bzgl. Abfragen und „Kommandos“ unterscheiden sich deutlich und es kommen ggf. mehrere Datenbanken zum Einsatz.
- Die Validierung der Daten ist komplex.

## Vorteile/Möglichkeiten

- Die Modelle können von unterschiedlichen Teams entwickelt werden (im Rahmen einzelner Services).
- Unterschiedliche Skalierung bzgl. Abfragen und Kommandos ist möglich.
- Passt sehr gut zu ereignisgetriebenen Programmiermodellen/Architekturen.  
Erlaubt sehr einfache Unterstützung von *Event Sourcing*.

## 2. Moderne Architekturprinzipien für verteilte Anwendungen

---

# Gute Anwendungsarchitekturen

Die (technischen) Ziele einer guten Anwendungsarchitektur sollten der Minimierung des Aufwands dienen, der notwendig ist, um das System zu entwickeln und zu warten bzw. weiterzuentwickeln.

## Ein einfacher RESTful Web Service mit Spring[3]

```
1 import java.util.concurrent.atomic.AtomicLong;
2 import org.springframework.web.bind.annotation.*;
3
4 @RestController
5 public class GreetingController {
6
7     private static final String template = "Hello, %s!";
8     private final AtomicLong counter = new AtomicLong();
9
10    @GetMapping("/greeting")
11    public Greeting greeting(
12        @RequestParam(value = "name", defaultValue = "World") String name
13    ) {
14        return new Greeting(counter.incrementAndGet(), String.format(template, name));
15    }
16 }
```

[3] Beispiel von <http://spring.io>.

Die (technischen) Ziele einer guten Anwendungsarchitektur dienen der Minimierung des Aufwands, der notwendig ist, um das System zu entwickeln und zu warten bzw. weiterzuentwickeln.

Eine gute Anwendungsarchitektur erlaubt es Entscheidungen, die sich *nicht* aus den Geschäftsanforderungen ergeben, zu verzögern bzw. „leicht“ anpassbar zu machen.

---

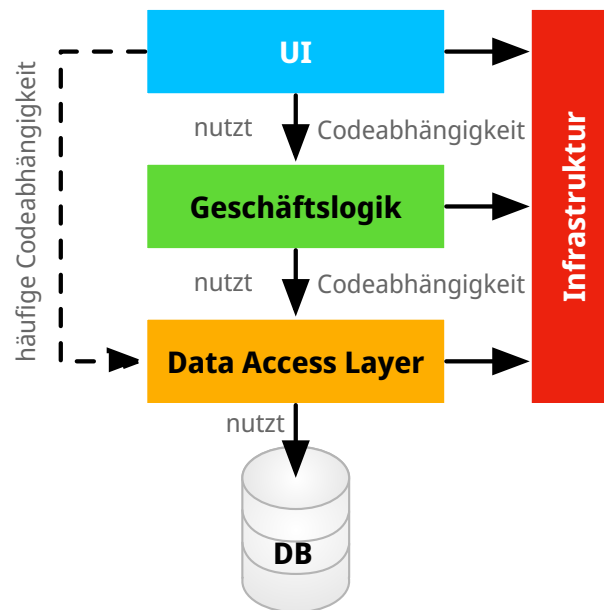
Entscheidungen, die nicht am Anfang final getroffen werden sollten, da sie ggf. die Architektur dominieren:

- Frameworks
- Datenbanken
- Webserver
- Kommunikationsprotokolle
- ...

Im RESTful-Beispiel hatten wir einen technischen Service for Augen - er implementiert keine wesentliche Geschäftslogik!



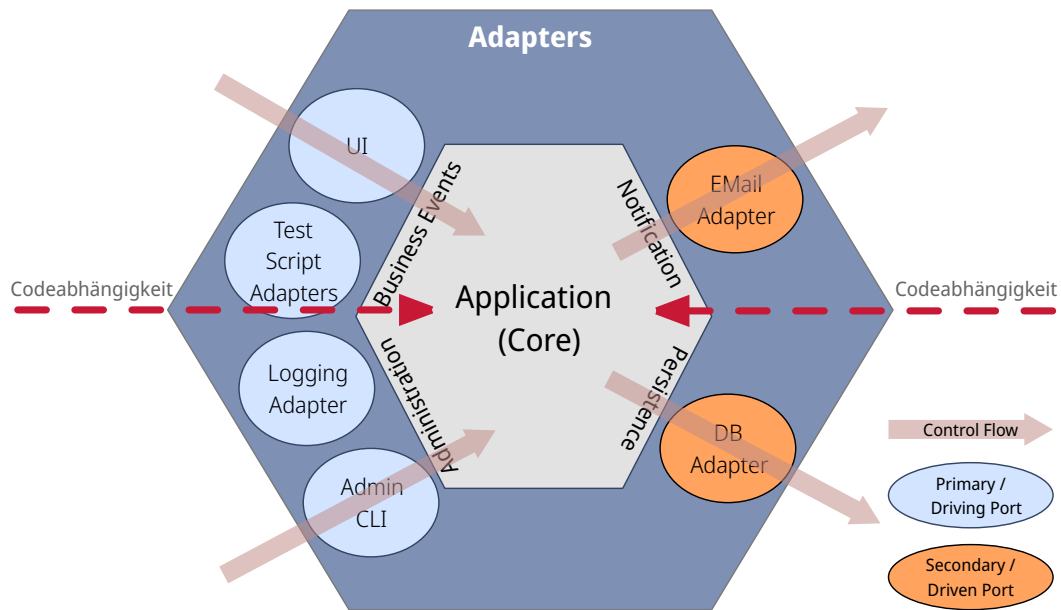
## Traditionelle n-Schichten Architektur (Wiederholung)



Codeabhängigkeiten ergeben sich zum Beispiel beim Verwenden eines Object-relational Mappers (ORM).

Solch eine Architektur war Ende der 90er/Anfang der 2000er Standard und ist für einfache Programme auch heute noch akzeptabel, da diese häufig sehr schnell zu entwickeln sind und viel Erfahrung mit dieser Architektur vorhanden ist. Besser ist es jedoch gleich eine der folgenden Architekturen anzuwenden, um ggf. vorbereitet zu sein, wenn das System wächst.

# Hexagonal Architecture (Ports & Adapters) [4]



Ziel der hexagonalen Architektur ist es die Anwendungslogik unabhängig von der UI und den Datenbanken etc. zu machen. Die Anwendungslogik/die Anwendungskomponenten sollen lose gekoppelt sein und einfach mit Ihrer Umgebung verbunden werden können durch die Nutzung von *Ports & Adapters*.

Für die Implementierung von *Primary Ports* werden oft *Inversion of Control Frameworks* verwendet. Die Implementierung von *Secondary Ports* erfordert üblicherweise den Einsatz von *Dependency Inversion*.

Im Allgemeinen ist es oft notwendig in den Adaptern Entity Klassen hin und zurück „zu Mappen“, um sicherzustellen, dass keine technischen Abhängigkeiten in den Kern einsickern.

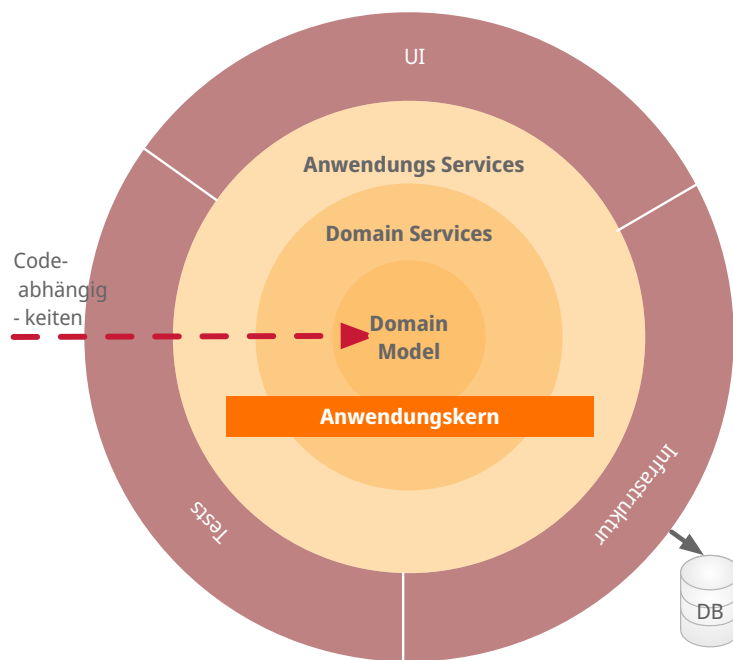
Die hexagonale Architektur wird von einigen als Ausgangsarchitektur für *Microservices* gesehen, da häufig einzelne Services nach diesem Architekturmuster implementiert werden.

*Meine Herangehensweise für die Planung einer komplexen Geschäftsanwendung ist in der Regel eine Kombination aus Domain Driven Design, Microservices und hexagonaler Architektur: Einsatz von Strategic Design zur Planung von Core Domain, Sub Domains und Bounded Contexts. Aufteilung eines Bounded Contexts in einen oder mehrere Microservices. Ein Microservice kann ein oder mehrere Aggregates enthalten, aber auch den kompletten Bounded Context, sofern dieser nicht zu groß ist (und statt des gewünschten Microservices wieder ein Monolith entsteht).*

—<https://www.happycoders.eu/de/software-craftsmanship/hexagonale-architektur/>



## Onion Architecture [5]



---

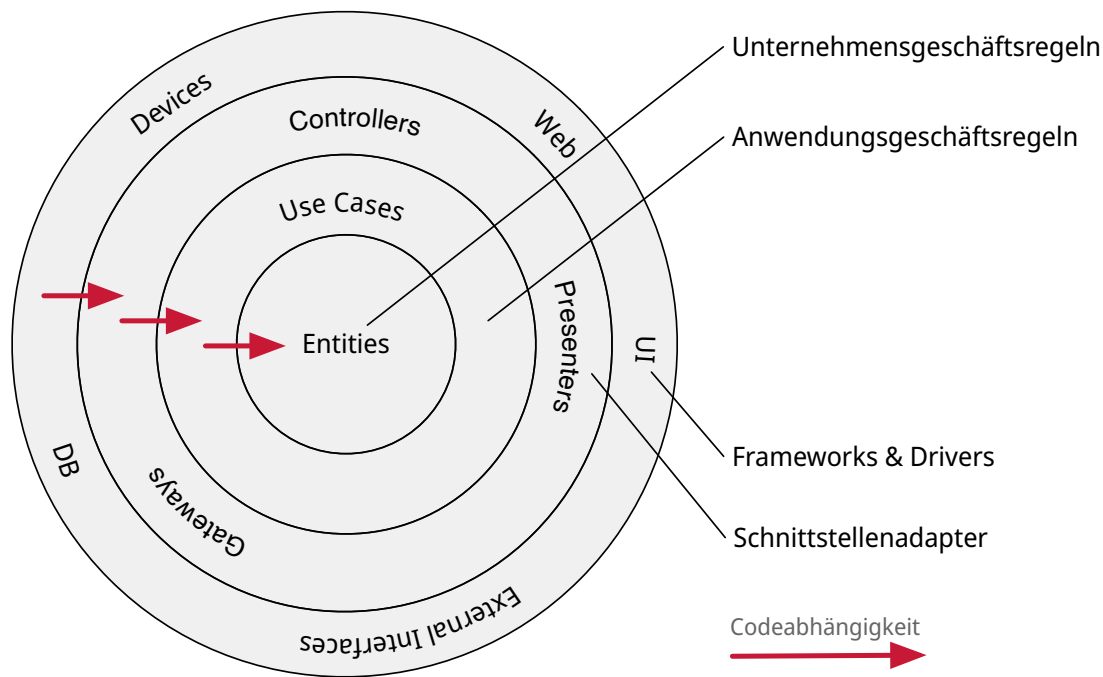
Schlüssellehren der *Onion Architecture* (Zwiebelarchitektur):

- Die Anwendung ist rund um ein unabhängiges Objektmodel gebaut.
- Innere Schichten definieren Schnittstellen.
- Äußere Schichten implementieren Schnittstellen.
- Die Richtung der Kopplung ist immer in Richtung zum Zentrum!
- Der Anwendungskern (*Application Core*) kann immer ohne die Infrastruktur kompiliert und davon unabhängig ausgeführt werden.

---

[5] Jeffrey Palermo, 2008

# Clean Architecture [Martin2017]



---

## Entities

Entitäten (📦 *Entities*) kapseln unternehmensweite kritische Geschäftsregeln.

- Objekte mit Methoden
- Datastrukturen
- Funktionen
- ...

Entitäten sind „Dinge“, die sich nicht aufgrund externer (technischer) Änderungen ändern sollten. Zum Beispiel aufgrund von geänderten Sicherheitsanforderungen oder der verwendeten Datenbank.

## Use Cases

Anwendungsspezifische Geschäftsregeln orchestrieren den Fluss der Daten von und zu den Entitäten; Änderungen an den Anwendungsfällen (*Use Cases*) sollten auf die Entitäten keinen Einfluss haben.

## Controllers, Gateways, Presenters

Die Aufgabe des Rings der Schnittstellen und Adapter ist die Konvertierung der Daten der Anwendungsfälle/Use Cases bzw. Entitäten und dem Format, dass für die externen Funktionalitäten sinnvoll ist.

In diesem Ring erfolgt zum Beispiel die Implementierung des MVC Patterns für eine GUI, oder das ORM Mapping.

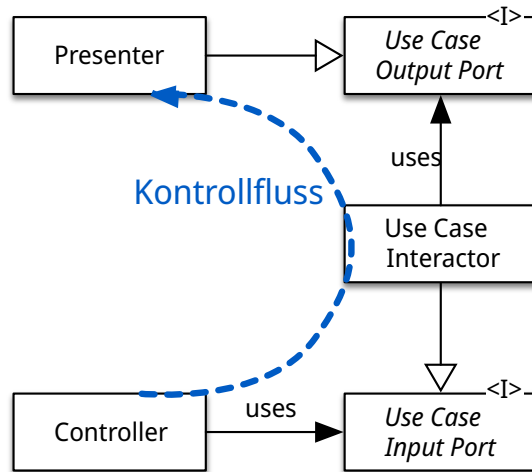
## DBs, Web, Devices

In diesem Ring befinden sich die externen Details, in der Regel gibt es hier keinen oder nur minimalen *Glue Code*.

## Code Abhängigkeiten

Wie bei den anderen Architekturen auch, gehen auch hier die Abhängigkeiten immer von außen nach innen. D. h. die Entitäten sind von nichts abhängig, die Anwendungsfälle von den Entitäten, die Schnittstellen von den Anwendungsfällen und die externen Details von den Schnittstellen.

## Clean Architecture - Prototypische Implementierung



---

Mit einer solchen Implementierung sind auch echte initiale Kosten verbunden - mehrere Interfaces müssen implementiert und gewartet werden. Partielle Lösungen sind denkbar, müssen aber wohl überlegt sein, um ungewünschte Abhängigkeiten zu vermeiden, die häufig zu einer schlechten Wartbarkeit und langfristigen bzw. verzögerten Kosten führen.

# Gemeinsamkeiten aktueller Architekturen

- Unabhängig von Frameworks
- Testbar
- Unabhängig von der Benutzerschnittstelle
- Unabhängig von Datenbanken
- Unabhängig von jeglichen externen Agenten/Systemen



# Literatur

[Martin2017] (1,2) Clean Architecture: A Craftsman's Guide to Software Structure and Design; Robert C. Martin, Addison-Wesley, 2017