

# Software-Architektur

Maurice Müller

2021-10-15

## Geschichte



"Those who fail to learn from history are condemned to repeat it."

— Sir Winston Churchill

"Wer aus der Geschichte nichts lernt, ist dazu verdammt, sie zu wiederholen."

## Evolution der Code-Basis

### Geschichtlich

- einzelne Dateien / Skripte
- Komposition von Dateien / Skripten
- Anbindung externer Komponenten (Datenbanken, ...)
- Big Ball of Mud

### In Projekten

- einzelne Klassen
- Komposition von Klassen
- Anbindung externer Komponenten
- Big Ball of Mud

- verbreitete Entschuldigung: **"Historisch gewachsen."**
- eigentlich → **"Niemand hat sich um die Architektur gekümmert."**

# Big Ball of Mud



"A Big Ball of Mud is a haphazardly structured, sprawling, sloppy, duct-tape-and-baling-wire, spaghetti-code jungle."

— Brian Foote & Josphe Yoder

"Eine Riesenmatschkugel ist ein planlos strukturierter, wuchernder, schludriger, mit Isolierband und Draht umwickelter Spaghetti-Code Dschungel."

- Software braucht eine richtige Architektur
- Software muss gepflegt werden
- jeder Entwickler muss grundlegendes Verständnis von Software-Architektur haben
- ohne Architektur, Pflege und Verständnis → **Big Ball of Mud**

## Was ist Software-Architektur?

Es gibt keine einheitliche Definition.

Mögliche Definitionen:

- was das große Ganze betrifft
- was am Anfang eines Projekts grundlegend entschieden werden muss
- worüber sich die Hauptentwickler einig sind
- IEEE-1471: "The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution."

### NOTE

IEEE-1471 Quelle: <http://www.iso-architecture.org/ieee-1471/docs/all-about-ieee-1471.pdf> S.13

## Was macht Software-Architektur?

entnommen aus dem Buch „Effektive Software-Architekturen“ von Gernot Starke

### Architektur gibt Strukturen vor

- Definition der Komponenten eines Systems

- Beschreibung der wesentlichen Merkmale
- Beschreibung der Schnittstellen
- Beziehung der Komponenten untereinander
- gibt somit einen Überblick über den *Bauplan* und *Ablaufplan*

## Architektur beschreibt eine Lösung

- enthält übergreifende Prinzipien
- ist keine konkrete Implementierung

## Architektur basiert auf Entwurfsentscheidungen

- Architektur ist das Resultat zahlreicher Entscheidungen
  - z.B. interner Aufbau einzelner Komponente oder eingesetzte Technologien
- Entscheidungen müssen begründet und dokumentiert werden
  - die Auswirkungen sind häufig erst später sichtbar
  - erleichtert / ermöglicht eine (Neu-)Bewertung

### NOTE

(Neu-)Bewertung: Hat die Architektur die gesteckten Ziele erreicht? Warum wurde X anstatt Y eingesetzt? Eignet sich eine neue Technologie Z um Probleme / Funktionalität besser umzusetzen?

## Architektur unterstützt den Übergang von der Analyse zur Realisierung

- Brücke zwischen Fachdomäne und Implementierung
- Vorgaben zur Erreichung von fachlichen und qualitativen Anforderungen

### NOTE

- fachliche Anforderung: gehört zur Domäne / zum Business
- Qualitätsanforderung: gehört nicht zur Domäne → z.B. Verarbeitungszeit, Sicherheit, Benutzbarkeit, ...

## Architektur kann Systeme aus unterschiedlichen Sichten zeigen

- Kontextabgrenzung: Vogelperspektive und Nachbarsysteme
- Laufzeitsicht: Wie arbeiten Komponenten zusammen?
- Bausteinsicht: Statische Struktur der Bausteine und ihrer Beziehungen
- Verteilungssicht: In welcher Umgebung läuft das System ab?

### NOTE

Unterschiedliche Sichten: \* erhöhen das Verständnis \* ermöglichen die Darstellung spezifischer Punkte \* erleichtern das Finden von Informationen

## Architektur basiert auf Abstraktionen

- nicht benötigte Informationen werden weggelassen

## Architektur kann Soll- und/oder Ist-Zustand beschreiben

- bei einem bestehendem System den Ist-Zustand dokumentieren
- mögliche Strukturen, Konzepte und Entscheidungen in einem Soll-Zustand beschreiben

# Vorgehen

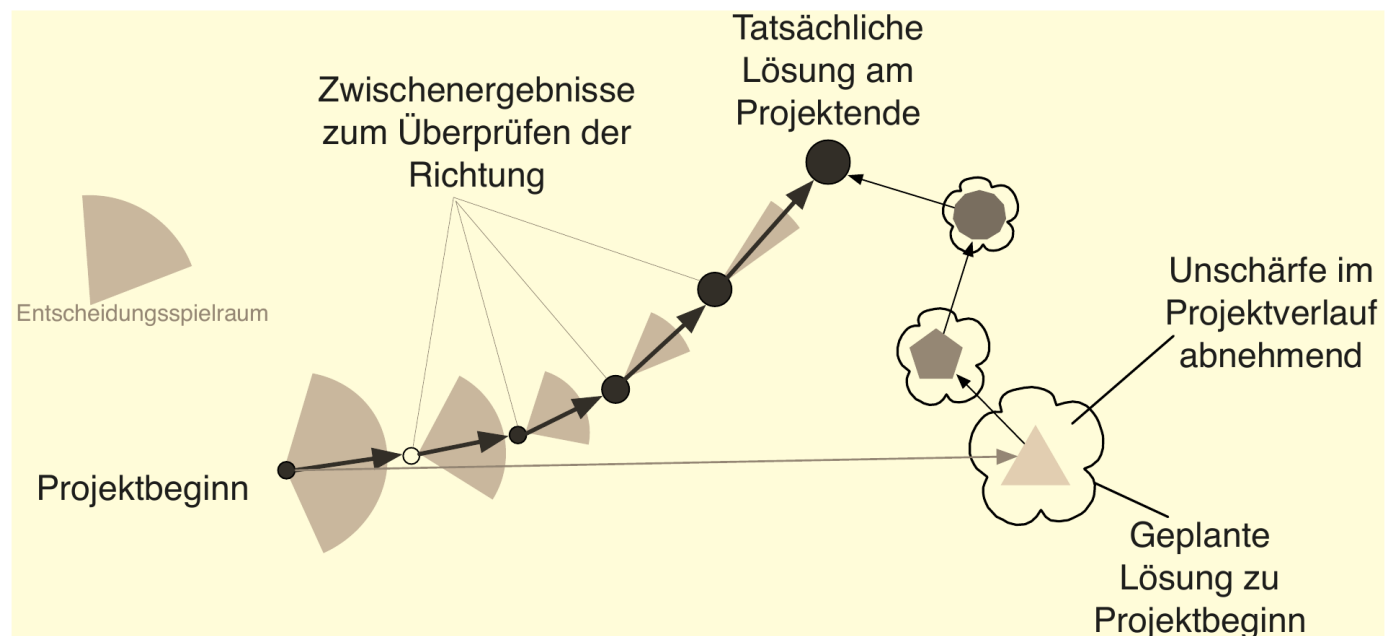
## Problem: Moving Target

**Problem:** auf ein sich bewegendes Ziel zielen

*"aiming at a moving target"*

- Anforderungen sind schwammig
- Auswirkungen von Entscheidungen sind nicht voll absehbar
- Kunde weiß nicht wirklich, was er will

## Lösung: Iterativ und inkrementell



Quelle: Effektive Software-Architekturen von Gernot Starke

## ABC: Architecture Business Cycle

- Architektur beeinflusst das System
- das System beeinflusst die Organisation und Randbedingungen

- Organisation & Randbedingungen beeinflussen den Architekten
- Architekt beeinflusst Architektur
- (fast) alles beeinflusst die Erfahrung des Architekten
- die Erfahrung beeinflusst die Architektur

#### NOTE

- Organisation im Sinne von Firma
- System im Sinne von Applikation(en)

## Konkretes Vorgehen

1. Anforderungen & Randbedingungen klären
2. Entwerfen und kommunizieren
3. Umsetzung begleiten
4. Lösungsstrategien für organisatorische Probleme
5. Architektur bewerten

#### NOTE

Konkretes Vorgehen nach *Effektive Software-Architekturen* von Gernot Starke

## Anforderungen & Randbedingungen klären

**Immer zu erst nach vorhandenen Lösungen suchen!**

- funktionale Anforderungen klären
  - = Kernaufgabe / Was wird verkauft?
  - in 2-3 Sätzen ausformulieren (Begriffe Fachdomäne verwenden)
  - Beispiel: Das System unterstützt auf möglichst einfache Art und Weise den Austausch von Dokumenten. Die Datenhoheit soll dabei jederzeit beim Sender eines Dokuments liegen. Neben der Datenhoheit werden auch gesetzliche Vorgaben (DSGVO, Vorhalteplichten, ...) bei Bedarf automatisch umgesetzt.
- Kategorie des Systems
  - z.B. online & interaktiv, mobil, Entscheidungsunterstützungssystem, Hintergrundsystem, embedded, Echtzeit-System

#### NOTE

Für viele Probleme (sowohl fachlich als auch technisch) existieren bereits Lösungen. Diese sollten zu erst analysiert und bewertet werden.

- Stakeholder identifizieren
  - Stakeholder = jemand der „etwas“ mit dem System zu tun hat
  - z.B. Anwender, Betreiber, Projektleiter, Auftraggeber, Entwickler, Management, ...
  - inklusive Bezug notieren

- z.B. **Auftraggeber** will das System **innerhalb 10 Monaten auf den Markt** bringen
- erwartet nur **Grundfunktionalität**
- spielt die **wichtigste Rolle bei der Abnahme**

#### NOTE

Egal ob man Entwickler, Architekt, Projektleiter, ... ist: es ist immer wichtig, dass man die Stakeholder kennt - v.a. die, die die Abnahme erlauben, fördern oder behindern können.

- Qualitätsanforderungen ermitteln
  - Qualität entsteht nicht von selbst, sondern muss Teil des Entwurfs sein
  - nur indirekt messbar
  - unterschiedliche Stakeholder haben unterschiedliche Anforderungen / Auffassungen

## Qualitätsmerkmale nach ISO 9126

in ISO 25000 aufgegangen

- Funktionalität
  - Angemessenheit, Richtigkeit, Interoperabilität, Ordnungsmäßigkeit, Sicherheit
- Zuverlässigkeit
  - Reife, Fehlertoleranz, Wiederherstellbarkeit
- Benutzbarkeit
  - Verständlichkeit, Erlernbarkeit, Bedienbarkeit
- Effizienz
  - Zeitverhalten, Verbrauchsverhalten
- Änderbarkeit
  - Analysierbarkeit, Modifizierbarkeit, Stabilität, Prüfbarkeit
- Übertragbarkeit
  - Anpassbarkeit, Installierbarkeit, Konformität, Austauschbarkeit

## Beispiele konkreter Qualitätsmerkmale

- (*schlecht*) das System muss bei durchschnittlicher Last schnell antworten
- (*gut*) bei kontinuierlich 10.000 Anfragen pro Sekunde muss das System jede Anfrage unter 200ms beantworten
- (*ok*) parallele Entwicklung muss möglich sein
- (*besser*) 3 unterschiedliche Entwickler-Teams sollen parallel an dem System arbeiten können
- der Hauptalgorithmus muss austauschbar sein
- das System muss sich innerhalb von 2 Minuten neustarten können

- ein Daten-Backup kann während dem laufenden Betrieb gezogen werden

## Fachlicher und technischer Kontext ermitteln

- Benutzergruppen
- Nachbarsysteme
- Schnittstellen zu externen Systemen
  - in beide Richtungen: die zu integrierenden und die angebotenen
  - stabil oder volatil
  - synchron oder asynchron
  - Performance
- organisatorische Faktoren
  - Umgebung, in der entwickelt wird
  - Aufbau, Struktur, Politik eines Unternehmens
  - wird häufig unterschätzt
  - Budget
  - wechselnde Verantwortliche
  - beste technische Lösung **nicht** zwangsläufig beste politische Lösung
- technische Faktoren
  - vorhandenes Wissen zu bestimmten Technologien
  - Deployment
  - Infrastruktur
  - firmeninterne Standards

## Zusammenfassung: Anforderungen & Randbedingungen klären

- vorhandene Lösungen analysieren
- funktionale Anforderungen klären
- Qualitätsanforderungen klären
- Stakeholder identifizieren
- fachlicher und technischer Kontext klären

## Konkretes Vorgehen

1. Anforderungen & Randbedingungen klären
2. **Entwerfen und kommunizieren**

3. Umsetzung begleiten
4. Lösungsstrategien für organisatorische Probleme
5. Architektur bewerten

## Entwerfen und kommunizieren

- Stakeholder beurteilen Erfolg von Lösungen / dem System
  - **nicht** der Architekt und **nicht** die Entwickler
  - kurzfristige und langfristige Ziele ausbalancieren
- KISS - Keep it small and simple
  - Einfachheit gewinnt → so einfach wie möglich
  - *gold plating* vermeiden
- spezifische Lösungen entwickeln
  - es gibt nicht die eine Lösung

### NOTE

Kurzfristige Ziele werden von den Stakeholdern häufig höher priorisiert als langfristige Ziele. Daher ist es sehr wichtig, die langfristigen Ziele zu kennen und in die Planung miteinzubeziehen.

**Gold plating:** auch over-engineering; das Hinzufügen unnötiger Bestandteile (s. Nespresso)

- explizit statt implizit
  - implizite Annahmen vermeiden und Entscheidungen / Entwurf explizit festhalten
  - z.B. fachliche Zusammenhänge zwischen Domänobjekten, klare Verantwortlichkeiten von Komponenten, Qualitätsanforderung
- Änderungen erwarten
  - iteratives Vorgehen mit Feedback-Schleifen zu den Stakeholdern
  - auch scheinbare feste Entscheidungen können (von außen) geändert werden
- Fehler erwarten
  - Was kann ausfallen?
  - Debugging / Fehlersuche erleichtern (→ Einfachheit!)
  - Fehler möglichst lokal halten
- Qualitätsanforderungen immer einhalten
  - *Continuous Compliance*
  - die wichtigsten Ziele müssen jederzeit eingehalten werden
  - z.B. durch interne Reviews sicherstellen
  - Systeme „verfaulen“ ohne Achtsamkeit / Pflege





- Prinzipien der Software-Entwicklung gelten im Großen auch für die Architektur
  - KISS, SOLID, lose Kopplung, hohe Kohäsion, SoC, Geheimnisprinzip, DRY, ...
- **Konsistenz**
  - gleiche / ähnliche Dinge heißen gleich / ähnlich
  - gleiche / ähnliche Probleme werden gleich / ähnlich gelöst
  - führt zu: leichterem Einlernen, schnellerer Entwicklung, weniger Fehlern, erhöhter Verständlichkeit
- Kommunikation mit allen Stakeholdern ist entscheidend
  - Entscheidungen müssen erklärt werden
  - Prinzipien müssen verinnerlicht werden (v.a. von den Entwicklern)
  - das Ziel muss vor Augen gehalten werden
    - kontinuierlicher Prozess, um sich nicht in bedeutungslosen Details zu verlieren
  - generell Top-Down kommunizieren
    - von der Vogelperspektive beginnen und schrittweise Details hinzufügen
  - Dokumentation ist Teil der Kommunikation

## Konkretes Vorgehen

1. Anforderungen & Randbedingungen klären

2. Entwerfen und kommunizieren
3. **Umsetzung begleiten**
4. Lösungsstrategien für organisatorische Probleme
5. Architektur bewerten

## Umsetzung begleiten

- „Kai-Zen“ / Goldgräber-Prinzip
  - bessere Ideen von Entwicklern (Stakeholdern allgemein) mit in die Architektur übernehmen
- Missverständnisse aufdecken
  - es gibt leicht Missverständnisse, was die Architektur, Anforderungen und Ziele angeht
  - wer Nahe am Team ist (z.B. bei Stand Ups) kann Missverständnisse leichter erkennen und aufklären
- Quell-Code lesen & Peer-Reviews
  - stichprobenartig sollte der Quellcode gelesen werden
  - Code-Reviews selbst oder mit erfahrenen Entwicklern durchführen

## Konkretes Vorgehen

1. Anforderungen & Randbedingungen klären
2. Entwerfen und kommunizieren
3. Umsetzung begleiten
4. **Lösungsstrategien für organisatorische Probleme**
5. Architektur bewerten

## Lösungsstrategien für organisatorische Probleme

Die drei größten organisatorischen Probleme:

- ZEIT
- BUDGET
- ERFAHRUNG

### NOTE

Die drei Faktoren beeinflussen sich gegenseitig. Verbessert man einer der Faktoren, tritt häufig eine Verbesserung in mind. einem der anderen Faktoren ebenfalls auf.

## Problemlösungen für „normale kritische Situationen“

- iteratives Vorgehen, um Probleme / Missverständnisse frühzeitig zu erkennen
- eng mit der Projektleitung und/oder Projektmanagement arbeiten

- Einfluss auf das Risikomanagement nehmen
- alternative Ziele vereinbaren (z.B. erstmal einen Teilschritt ausliefern)
- Auswirkungen kritischer Anforderungen diskutieren (um Verständnis zu schaffen)
- Prioritäten neu verhandeln (z.B. muss evtl. keine Hochverfügbarkeit gewährleistet sein)

## Hilfestellungen für „echte kritische Situationen“

- Druck abbauen
  - unter Druck entstehen mehr Fehler und schlechtere Lösungen
- keine neuen Mitarbeiter hinzunehmen
  - neue Mitarbeiter verzögern das Projekt weiter
- Politik / Management muss „mitspielen“, sonst ist alle Mühe umsonst

### NOTE

Politik wird von Technikern häufig unterschätzt bzw. vernachlässigt. Techniker müssen aber Einfluss auf die Politik nehmen, sonst entscheiden im schlimmsten Fall nicht-Techniker über technische Details.

## Konkretes Vorgehen

1. Anforderungen & Randbedingungen klären
2. Entwerfen und kommunizieren
3. Umsetzung begleiten
4. Lösungsstrategien für organisatorische Probleme
5. **Architektur bewerten**

## Architektur bewerten

Qualität — Quantität

Ziele — Metriken

Soll/Ist — Zahlen

### NOTE

Generell kann man Qualität und Quantität messen. Qualität entspricht der Erfüllung von Zielen für die Software-Architektur. Quantität sind Messungen wie z.B. Grad der Kopplung, Performance, ...

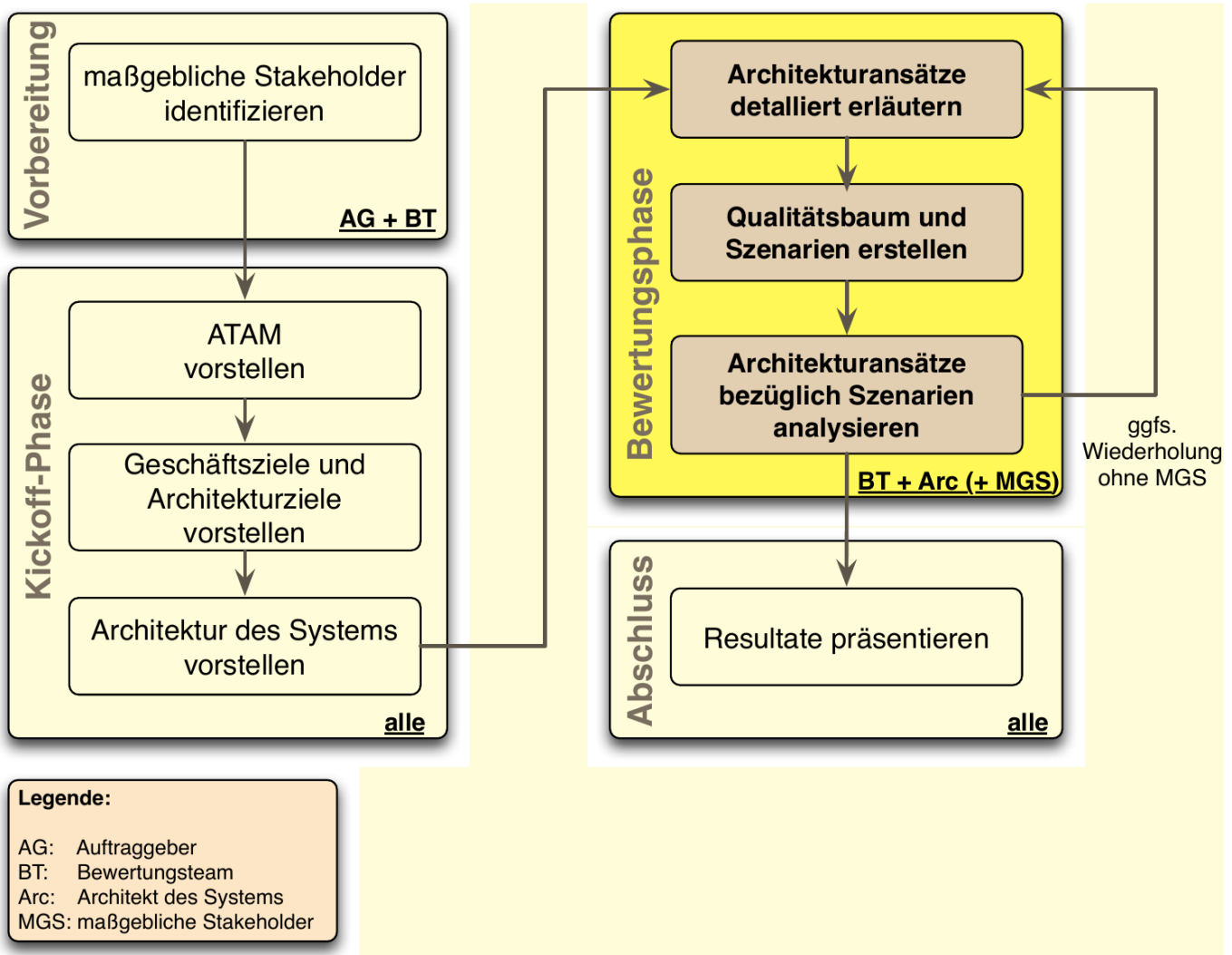
## Architektur bewerten: Qualität

### ATAM: Architecture tradeoff analysis method

- tradeoff (dt. Kompromiss): Architekturziele können sich oder anderen Anforderungen

widersprechen → Priorisierungen & Kompromisse nötig

- sollte eigentlich *vorher* statt finden
- kann aber auch zur Bewertung eingesetzt werden



Quelle: Effektive Software Architekturen von Gernot Starke

## Maßgebliche Stakeholder identifizieren

- i.d.R. nur wenige
- sollten zusammen mit dem Auftraggeber identifiziert werden
- ggf. Endanwender und Betreiber berücksichtigen

## ATAM vorstellen

- es geht um Risiken und Abwägungen und diesbzgl. Strategien
- Unterschied zw. qualitativer und quantitativer Bewertung hervorheben

## Geschäftsziele vorstellen

- Idealfall: Auftraggeber stellt geschäftliche (Qualitäts-)Ziele vor

- geschäftlicher Kontext & Einordnung in größere Systeme / Prozesse
- häufig sitzen erstmals alle maßgeblichen Stakeholder an einem Tisch

## Architektur vorstellen

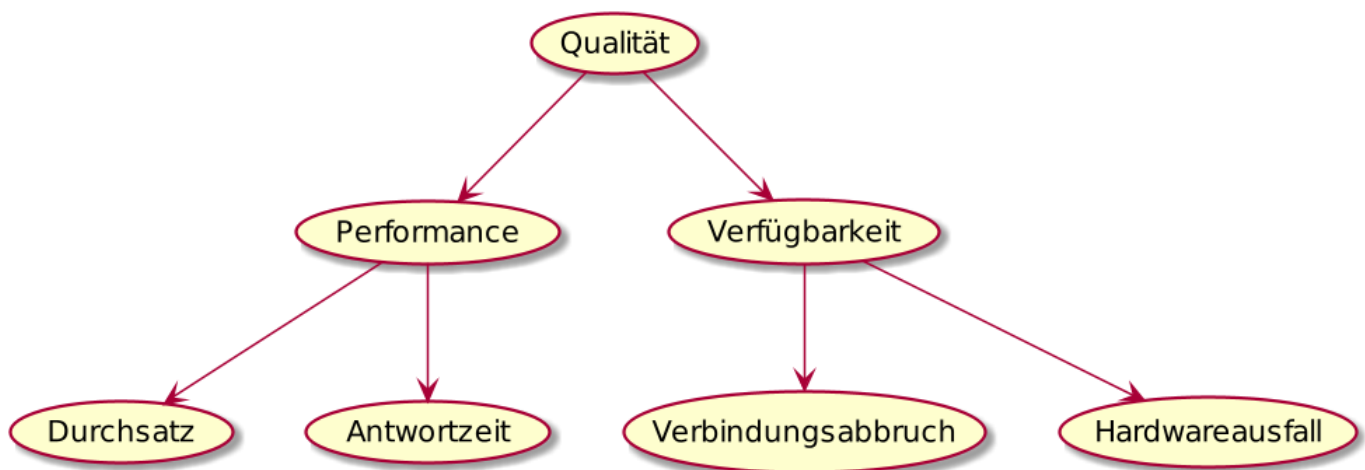
- hohe Abstraktion → nur die obersten Bausteine erläutern
- Kontext (d.h. Nachbarsysteme) einbeziehen

## Architekturansätze erläutern

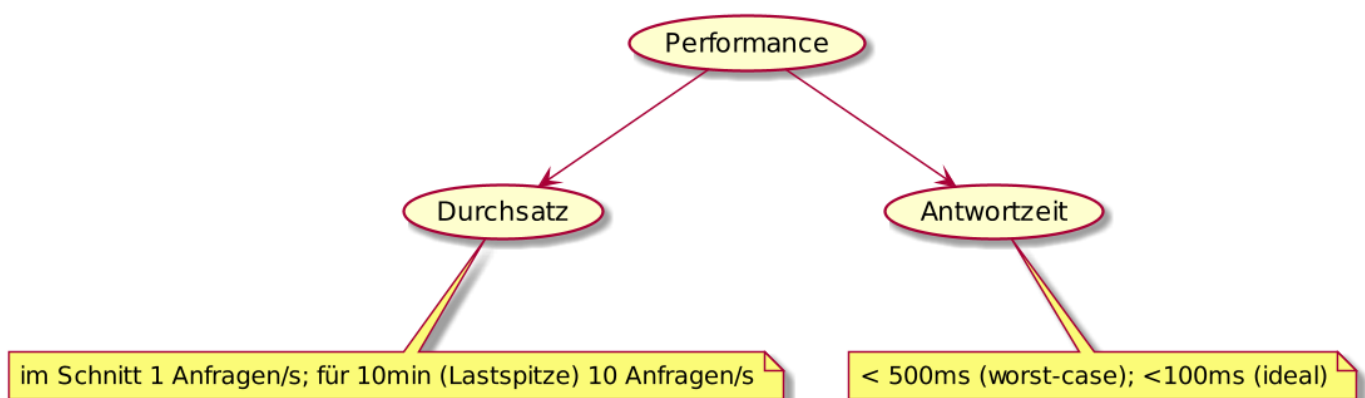
- in Bezug zu Geschäftsziele

## Qualitätsbaum und -szenarien

- Brainstorming und anschließendes Gruppieren & Priorisieren



- Qualitätsszenarien konkretisieren die Qualitätsziele



## Analyse

- Kernaufgabe → ggf. weitere Rücksprachen / Iterationen nötig
- anhand der Szenarien die Architektur(entscheidungen) bewerten
- Kompromisse beschreiben

- mögliche Risiken aufdecken
- was kann/wird auf jeden Fall erreicht

## Präsentation

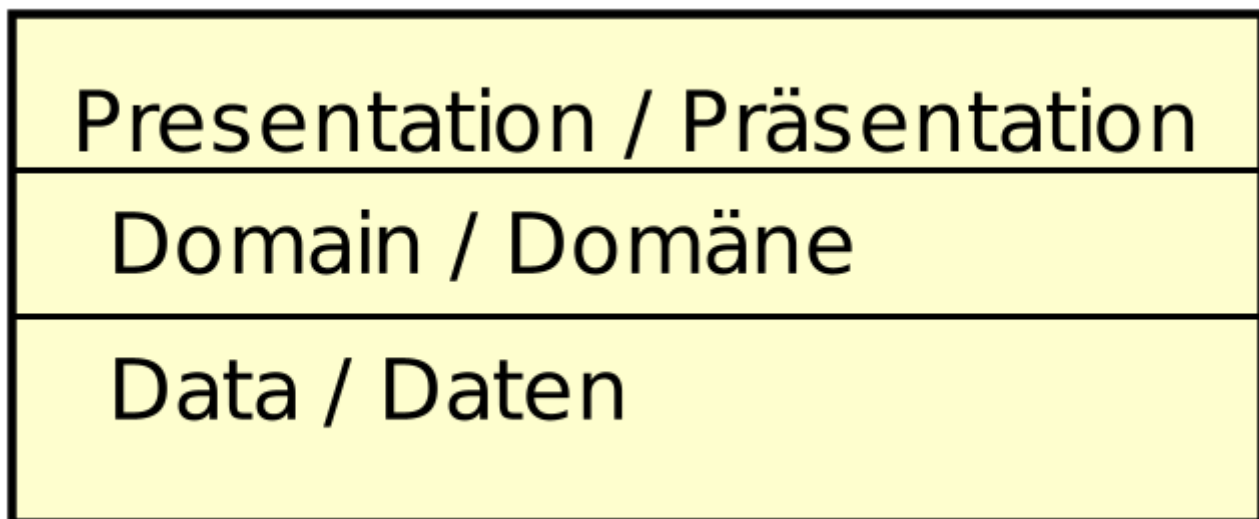
- Ergebnisse der Analyse präsentieren
- ggf. Entscheidungen / Maßnahmen treffen, um Architektur anzupassen

# Schichtenmodell

## Layering

- unterschiedliche Schichten decken einen unterschiedlichen Aufgabenbereich ab
- eine Schicht hängt nur von den Schichten unter ihr ab, niemals von den Schichten über ihr
  - d.h. Änderungen in einer Schichten können Änderungen in den Schichten über ihr zur Folge haben
- eine Schicht ist unabhängig von den Schichten über ihr
  - d.h. Änderungen in einer Schichten sollten keine Änderungen in den unterliegenden Schichten zur Folge haben
- führt zu **SoC** → **Seperation of Concerns**
- *reines* Schichtenmodell: eine Schicht kennt ausschließlich die direkte Schicht unter ihr

## Einfaches Schichtmodell



- **Presentation:** Benutzerinteraktion, UI, CLI, Batch API, HTTP API, ...
- **Domain:** Geschäftslogik (business logic), die *eigentliche* Aufgabe
- **Data source:** Datenbanken, Messaging-Systeme, Transaktionen, ...



## NOTE

- Presentation: stellt eigene Services für andere bereit
- Data source: stellt andere Services für die eigene Applikation bereit

# Einfaches Schichtmodell: Beispiel Taschenrechner

- einfache CLI, die die Zahlen und Rechenoperationen nacheinander einzeln einliest
- Ergebnis soll am Ende ausgegeben werden
- die letzten 10 Ergebnisse sollen gespeichert werden und abrufbar sein

Presentation: CLI

Domain: Rechenlogik

Data Source: Datenbank mit letzten 10 Ergebnissen

**Neue Anforderung: Batch API hinzufügen → Einlesen & Auswerten von Dateien mit Rechenausdrücken**

*(Batch API hat nichts mit Windows Batch Dateien zu tun)*

Presentation: CLI, Batch API

Domain: Rechenlogik

Data Source: Datenbank mit letzten 10 Ergebnissen

**Neue Anforderung: um die Leistung zu steigern, sollen die Rechenausdrücke der Batch API parallel ausgeführt werden.**

## Wo kann die Nebenläufigkeit implementiert werden?

- in der Batch API
  - (+) es ist da, wo es Sinn macht
  - (o) kann nicht wiederverwendet werden
- in der Rechenlogik
  - (+) alle überliegenden Schichten profitieren
  - (-) Nebenläufigkeit ist eine technische Eigenschaft und hat (normalerweise) nichts in der Domain zu suchen

- in einer neuen Komponente
  - (+) kann wiederverwendet werden
  - (o) neue Abstraktion

**Neue Anforderung: HTTP API → HTTP Clients sollen sowohl nacheinander einzelne Zahlen und Rechenoperation sowie Batch-Operationen schicken können.**

## Service-Schicht

Presentation: CLI, Batch API, HTTP API
Services: Controller für Nebenläufigkeit
Domain: Rechenlogik
Data Source: Datenbank mit letzten 10 Ergebnissen

Die Service-Schicht orchestriert Domain-Objekte ohne selbst Teil der Domain zu sein - z.B. um bestimmte Use Cases umzusetzen.

**NOTE** | Weiteres Beispiel für die Service-Schicht: Event Listener

## Reines Schichtmodell VS Schichtmodell

gegeben:

Presentation: CLI, Batch API, HTTP API
Services: Controller für Nebenläufigkeit
Domain: Rechenlogik
Data Source: Datenbank mit letzten 10 Ergebnissen

Aufgabe: Die letzten beiden Rechenergebnisse sollen abrufbar sein.

Reines Schichtmodell	Schichtmodell
----------------------	---------------



<ul style="list-style-type: none"> <li>• die Anfrage und die Rückgabe muss durch alle Schichten durch</li> <li>• auch durch die Domain-Schicht, die nur die Rechenlogik enthält <ul style="list-style-type: none"> <li>◦ das macht wenig Sinn</li> </ul> </li> <li>• hoher Implementierungsaufwand</li> </ul>	<ul style="list-style-type: none"> <li>• Schichten dürfen direkt auf jede unterliegende Schicht zugreifen <ul style="list-style-type: none"> <li>◦ z.B. könnte die Service-Schicht direkt auf die Daten-Schicht zugreifen</li> </ul> </li> <li>• geringerer Implementierungsaufwand</li> <li>• Domain-Logik wird nicht verunreinigt</li> </ul>
---	--

### Generelle Vor-/Nachteile

Reines Schichtmodell	Schichtmodell
<ul style="list-style-type: none"> <li>• (+) alle Schichten sind richtig voneinander isoliert</li> <li>• (+) Änderungen in einer Schicht betrifft nur die direkt drunter liegende Schicht</li> <li>• (-) hoher Implementierungs- und Wartungsaufwand</li> <li>• (-) schwer durchschaubare implizite Abhängigkeiten zwischen Schichten</li> <li>• (-) Verunreinigung zwischenliegender Schichten</li> </ul>	<ul style="list-style-type: none"> <li>• (+) erleichterter Implementierungs- und Wartungsaufwand, da direkt auf die benötigte Schicht zugegriffen werden kann</li> <li>• (+) klarere Abhängigkeiten zwischen Schichten</li> <li>• (-) Schichten haben mehr als eine andere Schicht als Abhängigkeit</li> </ul>

# Hexagonale Architektur

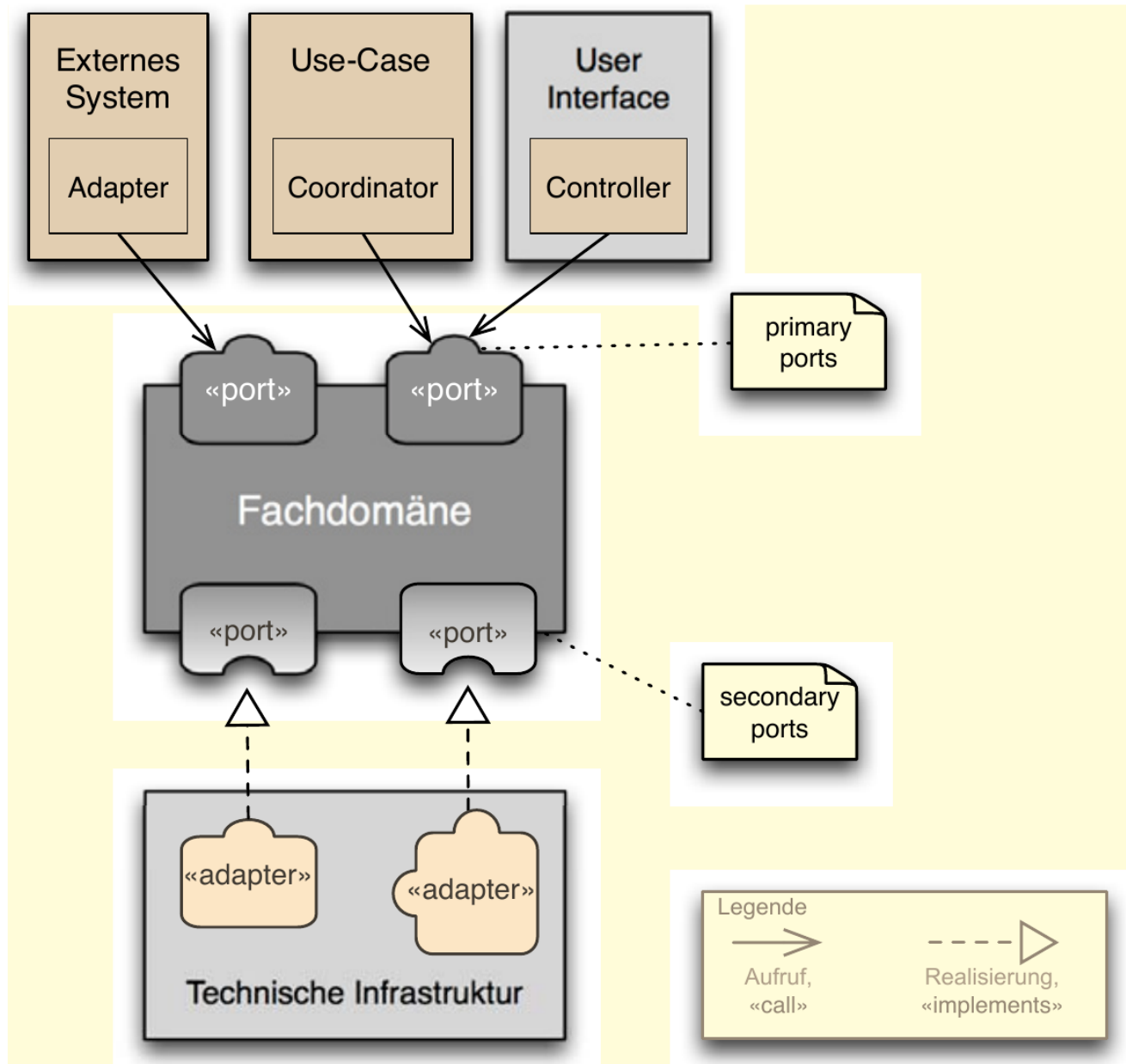
Ports & Adapters, Onion-Architecture

Zwiebelarchitektur, Schalenmodell

- Kern jeder Applikation ist die Fachdomäne
- Kern hat **keine** Abhängigkeiten nach außen
- Abhängigkeiten zeigen immer nach innen
  - d.h., die äußeren Schalen benutzen die inneren Schalen (**nie** umgekehrt)
- Fachdomäne kapselt fachliche Entitäten / Funktionen / Datenstrukturen und Regeln

## Ports & Adapters

ursprüngliches Modell von Alistair Cockburn



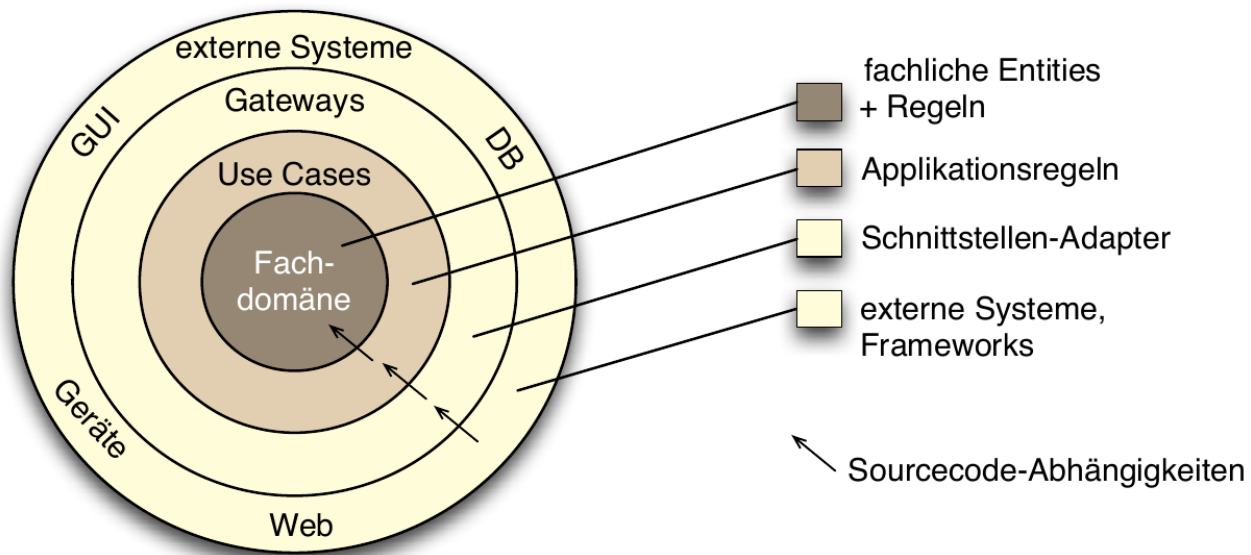
Quelle: Effektive Software Architekturen von Gernot Starke

#### NOTE

Alistair Cockburn ist ein amerikanischer Informatiker und Vordenker / Mitbegründer der agilen Software-Entwicklung.

- **Primäre Ports:** Domain API - direkter Aufruf von Funktionalität der Fachdomäne
- **Sekundäre Ports:** Schnittstellen, die implementiert werden müssen; u.a. von der Infrastruktur (z.B. Persistenz)
- **Adapter:** es gibt **primäre** und **sekundäre** Adapter, die entsprechend die primären bzw. sekundären Ports nutzen / implementieren und somit zwei unterschiedliche Schnittstellen verbinden

## Schalenmodell



Quelle: Effektive Software Architekturen von Gernot Starke

- **Use Cases:** Implementierung der Anwendungsfälle des Systems (u.a. Regeln, Objekte, Daten die sich auf d. Applikationszustand/verhalten beziehen)
- **Gateways (Schnittstellen-Adapter):** Konvertierung von Datenformaten oder fachlichen Schnittstellen zu Datenformaten oder Schnittstellen der Infrastruktur (z.B. Datenbank-Formate, UI-Controller, ...)
- **Frameworks/ext. Systeme:** Datenbanken, UI, ... ; werden im Normalfall nicht selbst angepasst (sondern nur mit s.g. *Glue Code* angebunden)
- innerhalb einer Schale können unterschiedliche (voneinander isolierte) Komponenten liegen

#### NOTE

*Unterschied zwischen den Schalen **Gateway** und **ext. Systeme**\*: im Gateway wird z.B. ein Fachdomänen-Objekt in JSON konvertiert - mit Hilfe von Glue Code innerhalb der Schale \*ext. Systeme wird dieses JSON z.B. in eine Dokumentendatenbank abgelegt*

## Hexagonale Architektur: Beispiel Taschenrechner

- einfache CLI, die die Zahlen und Rechenoperationen nacheinander einzeln einliest
- Ergebnis soll am Ende ausgegeben werden
- die letzten 10 Ergebnisse sollen gespeichert werden und abrufbar sein
- Batch API: Einlesen & Auswerten von Dateien mit Rechenausdrücken
  - um die Leistung zu steigern, sollen die Rechenausdrücke der Batch API parallel ausgeführt werden
- HTTP API: HTTP Clients sollen sowohl nacheinander einzelne Zahlen und Rechenoperation sowie Batch-Operationen schicken können

# Zuordnung Komponente in das Schalenmodell

- **Fachdomäne:** Rechenoperationen und Zahlenobjekte
- **Use Cases:** parallele Verarbeitung
- **Gateway**
  - JSON  $\leftrightarrow$  Objekte für die HTTP API
  - Text  $\leftrightarrow$  Objekte für die CLI
  - DTO  $\leftrightarrow$  Objekte für die Datenbank
- **Frameworks/ext. Systeme:** HTTP-Server, Terminal-Anbindung, Datenbank

## Vor-/Nachteile Schalenmodell

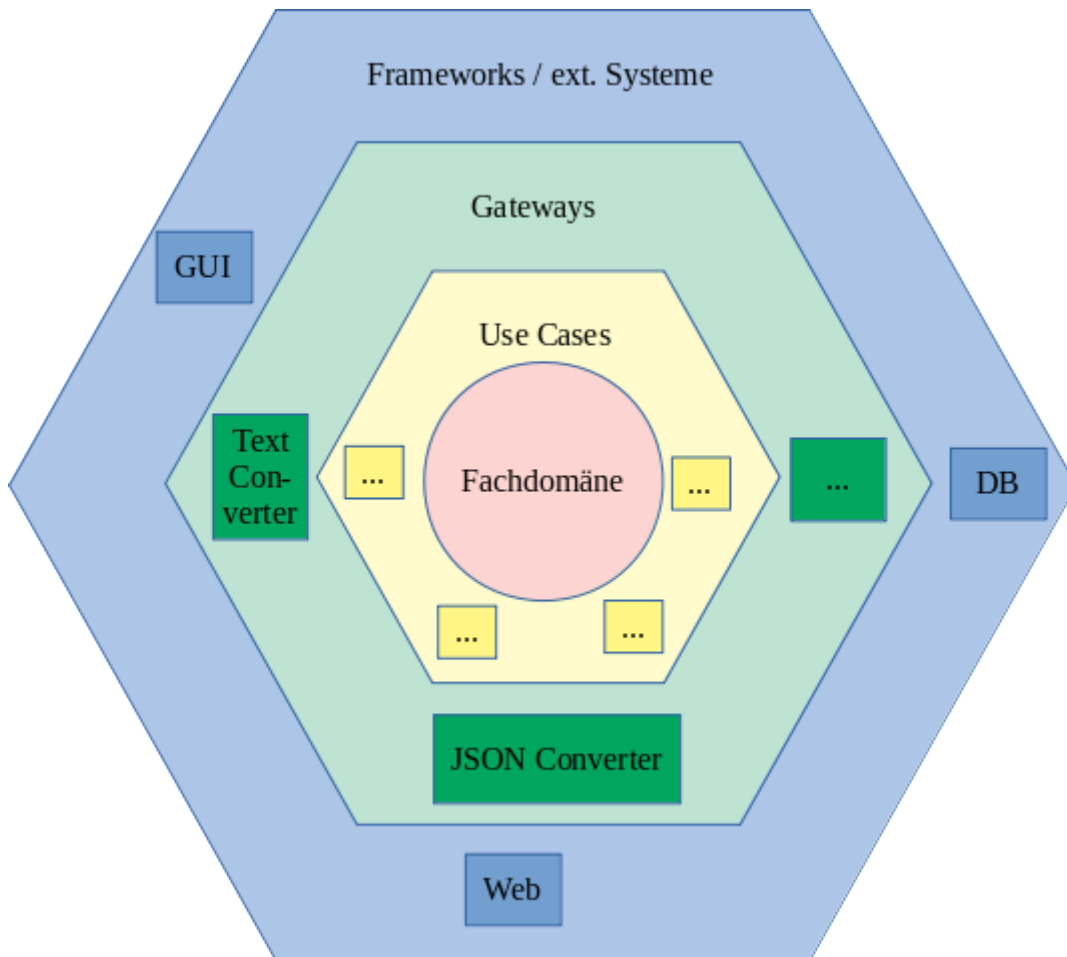
- (+) Fachdomäne ist technologieunabhängig
  - wird z.B. nicht beeinflusst, wenn sich die Infrastruktur ändert
- (+) erhöhte Wartbarkeit und vereinfachte Änderbarkeit durch Modularisierung
- (+) leichter Austausch einzelner Komponente (z.B. Austausch der Datenbank)
- (o) Umsetzung des Schalenmodells in bestehenden Systemen fast nicht möglich (da häufig die Modularisierung fehlt)

### NOTE

Das Schalenmodell ist ähnlich wie das Schichtenmodell. Einer der größten Unterschiede ist, dass die Fachdomäne im Kern ist und die Technologie in der äußersten Schicht. Somit wird konsequent die Infrastruktur vom Rest getrennt und eine Abhängigkeit von außen nach innen kann umgesetzt werden.

## Alternative Darstellung des Schalenmodells

Darstellung als geschachtelte Hexagons



# CQRS

Command Query Responsibility Segregation

hervorgegangen aus CQS - Command Query Separation

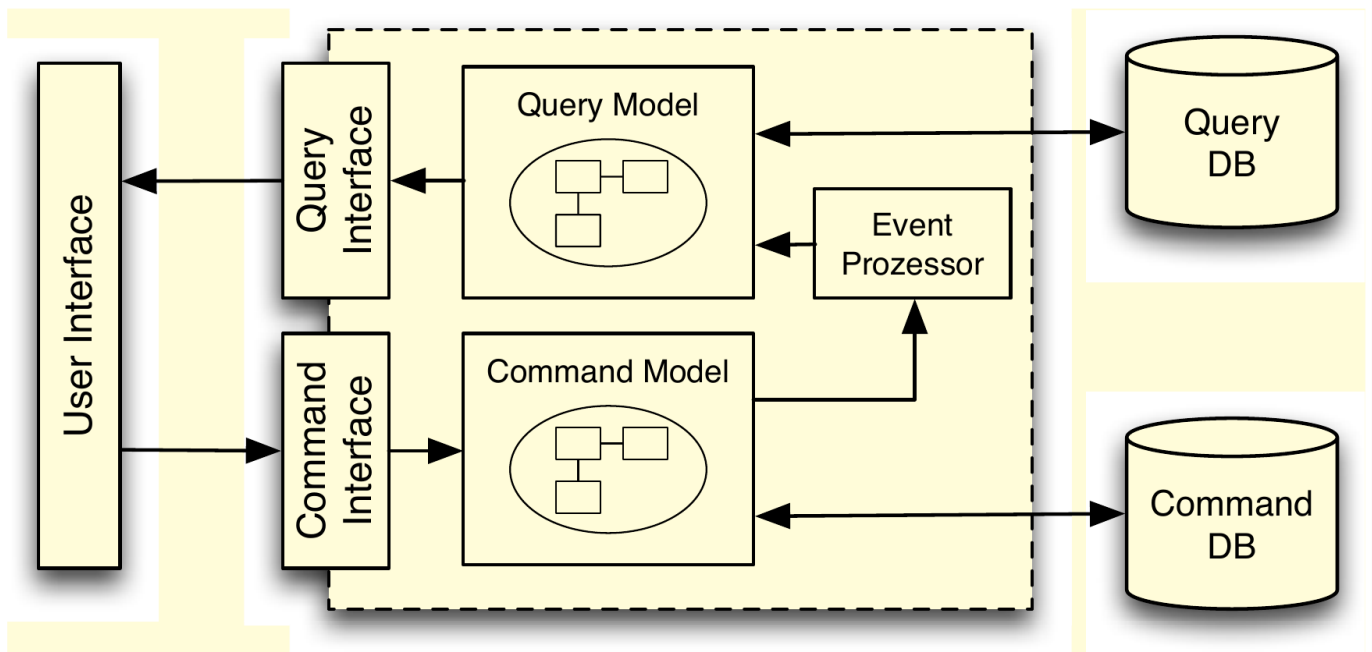
- Unterscheidung von Funktionalität / Komponenten in Commands und Queries
- **Command:** ändert den Zustand
- **Query:** gibt einen Zustand zurück
- **nie** macht eine Funktionalität / Komponente beides

*Eine Frage (Query) sollte nicht die Antwort (Zustand) ändern. Das Ändern einer Tatsache (Zustand) sollte keine Antwort zur Folge haben.*

## NOTE

In den 1980er Jahren hat Bertrand Meyer schon diese Unterscheidung bei Objekt-Methoden gemacht (was jetzt als CQS bekannt ist).

Es ist ein natürliches Prinzip, da in der analogen zwischenmenschlichen Welt normalerweise nie beides in einem auftritt.



Quelle: Effektive Software Architekturen von Gernot Starke

- im Extremfall kann es zwei unterschiedliche DBMSes für Queries und Commands geben
  - DBMSes können entsprechend der unterschiedlichen Anforderungen ausgewählt und optimiert werden
  - funktioniert nur, wenn inkonsistente Datensätze kein Problem darstellen (was relativ häufig der Fall ist)
- der Command-Teil muss den Query-Teil über Änderungen informieren - z.B. über Events
  - im Bild dargestellt als Event-Prozessor
- Erweiterung: **Event-Sourcing**
  - nur die Events werden gespeichert (inklusive Zeitstempel) auf der Command-Seite und Snapshots des aktuellen Zustands auf der Query-Seite
  - so lassen sich beliebige Zustände wiederherstellen / nachvollziehen

## CQRS: Beispiel Taschenrechner

- einfache CLI, die die Zahlen und Rechenoperationen nacheinander einzeln einliest
- Ergebnis soll am Ende ausgegeben werden
- die letzten 10 Ergebnisse sollen gespeichert werden und abrufbar sein
- Batch API: Einlesen & Auswerten von Dateien mit Rechenausdrücken
  - um die Leistung zu steigern, sollen die Rechenausdrücke der Batch API parallel ausgeführt werden
- HTTP API: HTTP Clients sollen sowohl nacheinander einzelne Zahlen und Rechenoperation sowie Batch-Operationen schicken können

CQRS macht für den Taschenrechner keinen Sinn.

# CQRS: Vor-/Nachteile

- (+) sehr performante Ausführung von Lese- und Schreibzugriffen möglich
- (-) Synchronisation zwischen Query-Model und Command-Model problematisch
- (-) Inkonsistenz zwischen Query-Model und Command-Model
- (o) lohnt sich im Normalfall nur, wenn sehr performante Abfragen notwendig sind
- (o) eher ungeeignet als Gesamtarchitektur → eher geeignet für einzelne Komponente

## NOTE

CQRS löst ein sehr spezielles Problem: Optimierung von Lese- und Schreibzugriffen. Daher eignet es sich eher nicht als Gesamtarchitektur, sondern als Architektur von Komponenten, die dies benötigen.

# SOA

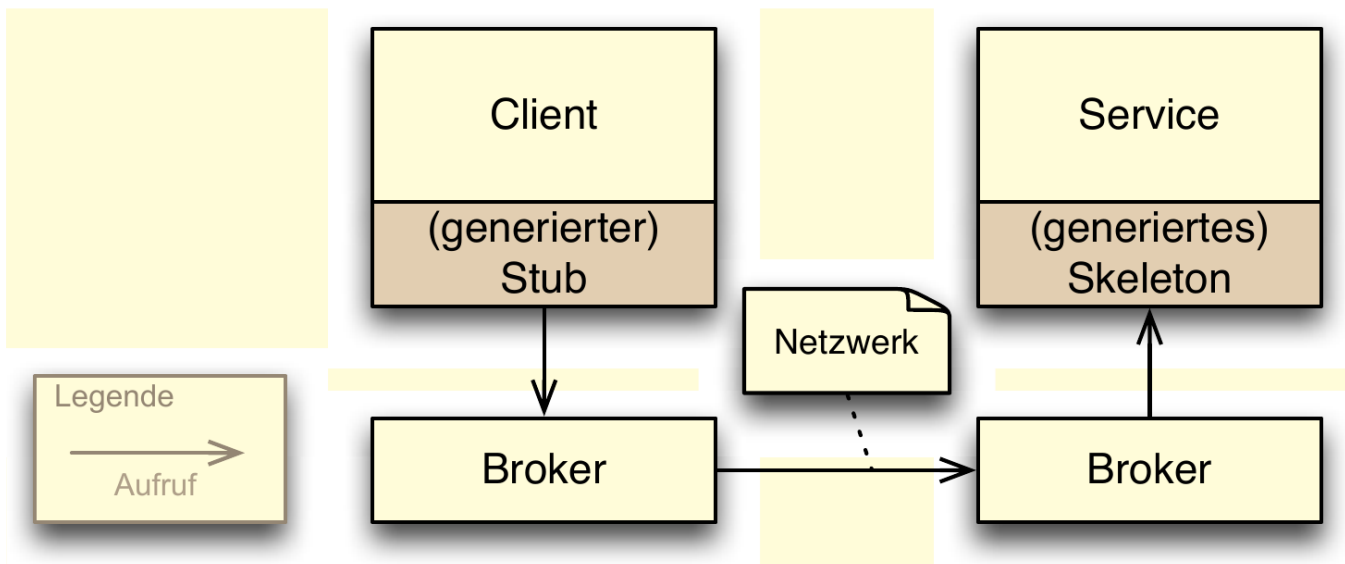
Service-oriented architecture

Serviceorientierte Architektur

- es gibt keine einheitliche Definition
- hat die Ursprünge u.a. in CORBA und ESBs
- ganz allgemein: Sammlung von Services, die (miteinander) Aufgaben erledigen
- bei SOA geht es im Normalfall um bestehende Applikationen, die miteinander kommunizieren müssen / sollen

# Geschichte

- isolierte Applikationen sollten miteinander kommunizieren
- per CORBA (Common Object Request Broker Architecture) erstmals eine standardisierte Schnittstelle
  - Aufrufe auf einem Objekt wurden transparent in einer anderen Applikation ausgeführt



Quelle: Effektive Software Architekturen von Gernot Starke

- CORBA
  - (+) technologieunabhängig
  - (+) einfacher Aufruf von anderen Services
  - (o) transparenter Aufruf → im Code ist nicht ersichtlich, ob es ein entfernter oder ein lokaler Aufruf ist ⇒ Performance-Killer möglich
  - (-) erschwertes Refactoring (Stubs und Skelette müssen neu generiert und implementiert werden)
- Nachrichten lösten CORBA ab
  - (+) technologieunabhängig
  - (+) einfache Implementierung
  - (-) Nachrichtentransformation schwierig

#### NOTE

Nachrichtentransformation ist schwierig, Beispiel: \* 3 Services sollen miteinander kommunizieren \* alle Services haben eigene nicht-kompatible Nachrichten \* wenn Service 1 und Service 2 an Service 3 eine Nachricht schickt, müssen entweder Service 1 und Service 2 wissen, wie die Nachricht zu Typ 3 transformiert wird oder Service 3 muss wissen, wie er die Nachrichten von Typ 1 und 2 in seinen Typ transformiert \*\* in jedem Fall müssen die Services Informationen über sich haben → dies führt zu einer stärkeren Kopplung \* werden neue Services integriert, geht das Spiel von vorne los

## ESBs

- Nachrichtenaustausch wurde durch eine richtige Middleware ersetzt
- ESB - Enterprise Service Bus
  - (+) technologieunabhängig
  - (+) Transformationen erledigt der ESB



- (+) Integration neuer Services vereinfacht
- (-) hoher Wartungsaufwand
- (-) tendiert zu hoher Komplexität

## SOA: Beispiel Taschenrechner

- einzelner Service pro Rechenoperation
- einzelner Service pro Schnittstelle nach außen (CLI, HTTP)
- ein Service für die Persistenz
- ESB benötigt man nur, wenn die Services unterschiedliche Nachrichtenformate benutzen

## SOA: Vor- und Nachteile

- (+) technologieunabhängig
- (+) Skalierung auf Serviceebene möglich
- (-) Transformationen / Middleware tendiert dazu, schlecht wartbar zu sein
- (o) ESB macht nur Sinn, wenn vorhandene inkompatible Services vernetzt werden

### NOTE

SOA - sowie hier vorgestellt - ähnelt Microservices sehr stark. Der Hauptunterschied ist, dass man bei SOA normalerweise keinen Einfluss auf die Dienste hat. Im Gegensatz zu Microservices, bei denen man alle Services von Grund auf entwickelt.

Daher benötigt SOA normalerweise den ESB und Microservices nicht.

Achtung: Manche SOA-Vertreter lehnen den Vergleich mit Microservices strikt ab. Dies liegt an den verschiedenen Auffassungen zu SOA.

## Monolith



- alles in einer Applikation
- häufig negativ konnotiert, weil
  - ohne Architektur historisch gewachsen
    - alles hängt von allem ab
    - duplizierter Code
    - schlecht wartbar
  - schlechtere Skalierung

Ein *Monolith* lässt alle Freiheiten, was die interne Architektur angeht.

→ Die Qualität eines Monoliths hängt von der Qualität der internen Architektur ab.

## Monolith: Vor- und Nachteile

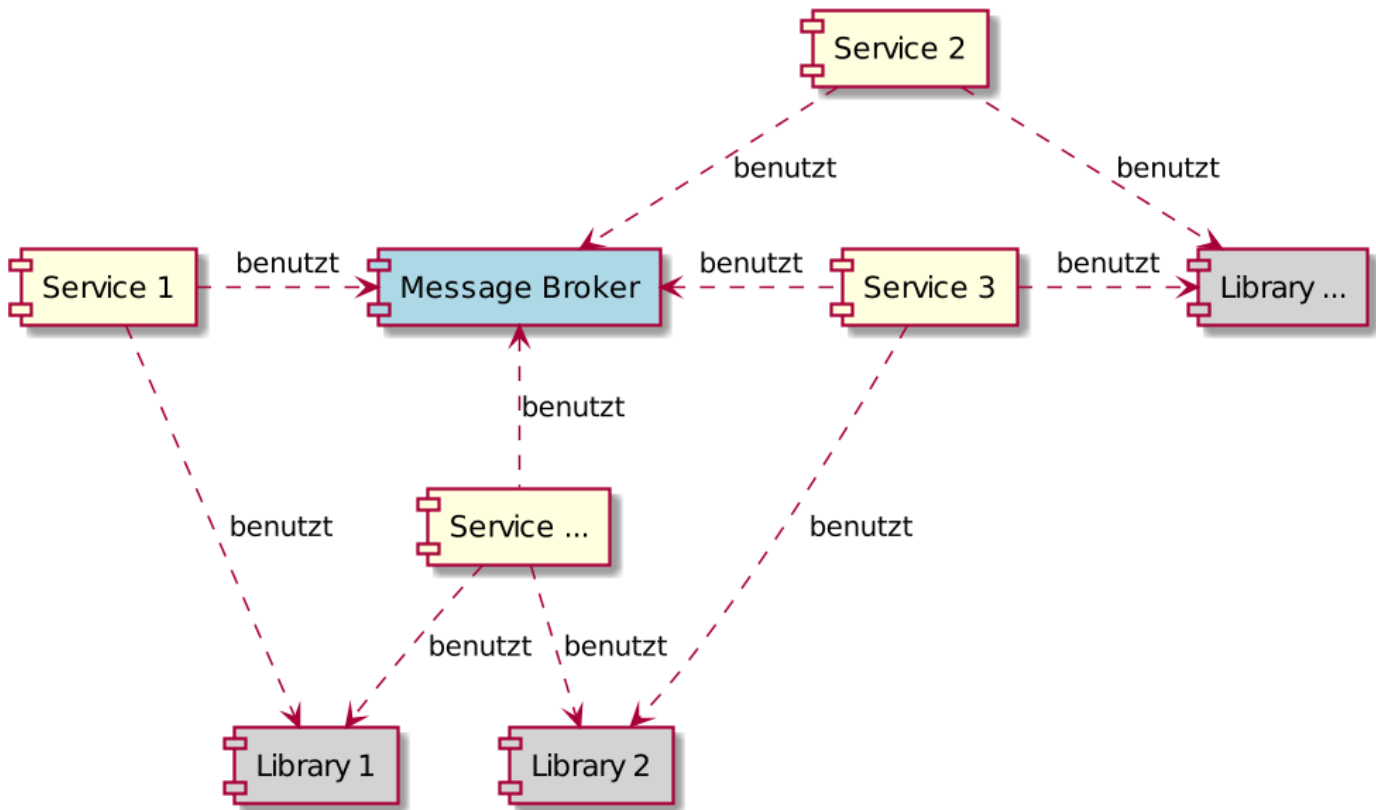
- (+) eine Code-Base
- (+) Releases sind in sich kompatibel
- (-) schlechtere Skalierung

## Microservices

## NOTE

Microservice-Architektur ist gerade ein Hype aber keine Wunderwaffe. Wie immer gilt: nur einsetzen, wenn es Sinn macht, und ggf. sich nur die Rosinen rauspicken.

- es gibt keine einheitliche Definition
- hat die Ursprünge in Web Services
- ganz allgemein: Sammlung von Services, die (miteinander) Aufgaben erledigen
- im Normalfall ist eine größere Applikation in kleinere Applikationen / Services zerlegt („System of Systems“)



## Kommunikation zwischen Microservices

- Event-getrieben
  - z.B. mit einem Message Broker
  - (+) entkoppelte Kommunikation
  - (+) höherwertige Funktionalität möglich (Queues, Broadcast, ...)
  - (-) Single Point of Failure
- direkt
  - z.B. HTTP, RPC, ...
  - (-) sehr starke Kopplung
  - (-) keine Standardlösung für höherwertige Funktionalität wie Broadcasting, ...

# Architektur-Sichten

- Sicht: Domänenarchitektur
  - fachliche Blöcke (Domänen, Subdomänen, Komponente) und deren Zusammenspiel
  - jede Komponente bildet genau eine Fachlichkeit ab und übernimmt alle Aspekte
  - jede Komponente ist autonom
- Sicht: Makroarchitektur
  - system- und serviceübergreifende Entscheidungen
  - z.B. Monitoring, Kommunikation (RESTful HTTP, AMQP, ...), Deployment, ...
  - gibt nur das Minimum vor, um den Services maximale Freiheit zu lassen
- Sicht: Mikroarchitektur
  - Architektur des einzelnen Services
  - abhängig von Domänenarchitektur und Makroarchitektur
  - kann pro Service unterschiedlich sein

## NOTE

Durch unterschiedliche Architektur-Sichten wird klarer, was für Auswirkungen eine Änderung in einem Microservice hat: lokal oder global → Grenzen werden klarer.

## Microservices: Beispiel Taschenrechner

- einzelner Service pro Rechenoperation
- einzelner Service pro Schnittstelle nach außen (CLI, HTTP)
- ein Service für die Persistenz
- eine Library für die Domainobjekte
- eine Library für die Kommunikation
- Message Broker für erhöhte Entkopplung zwischen den Services

## Vorgehensweisen

- Monolith first
  - M. Fowler → zu erst komponentenbasierter Monolith und bei Bedarf einzelne Komponente als Microservices extrahieren
- Macroservices first
  - mit größeren Services anfangen und bei Bedarf weiter zerlegen
- direkt Microservices

# Microservices: Vor- und Nachteile

- (+) schaffen klare Grenzen (zwischen Services, Fachlichkeiten und Technologien)
- (+) erleichtern Skalierung einzelner Funktionalitäten
- (+) erleichtern eine Mischung aus unterschiedlichen und gegensätzlichen Technologien
- (o) Dezentralisierung der Funktionalität und der Technologie
- (-) zu große Fragmentierung möglich
- (-) schwer zu durchschauende Abhängigkeiten
- (-) u.U. mehrfache Implementierung gleicher technologischer Komponenten

## Technische Schulden

Vom Umgang mit Schulden in der Software-Entwicklung

### Definition: technische Schulden

- essentielle Komplexität  $\leftrightarrow$  ungewollte (accidental) Komplexität
- Mehraufwand, der nicht sein muss
- z.B. ein Feature würde normalerweise 3 Tage zur Implementierung benötigen; durch die ungewollte Komplexität (z.B. Spaghetti-Code) dauert es aber 5 Tage
- technische Schulden verursachen „Zinsen“  $\rightarrow$  Mehraufwand

### Umgang

- keine Schulden (bewusst) aufnehmen  $\rightarrow$  das passiert von alleine
- Schulden wenn möglich zurückzahlen, d.h. Code säubern, Tests schreiben, etc.
  - Refactoring / Reviews fest einplanen
  - spätestens verbessern, wenn man damit arbeitet (Pfadfinder-Regel)
- Schulden immer isolieren und nach außen saubere Schnittstellen anbieten

#### NOTE

Schulden sammeln sich von alleine an, weil man z.B. zu Beginn noch nicht wusste, wie man etwas am Besten lösen kann.

Wenn es keine saubere Schnittstelle zwischen den Schulden und dem Rest-System gibt, verbreiten sich die Schulden sehr schnell im Gesamt-System.

## Broken-Windows-Theorie



- Korrelation zwischen Verfall von Stadtgebieten und Kriminalität
- wenn ein Fenster kaputt ist, ist die Hemmschwelle niedriger, ein weiteres kaputt zu machen → es scheint sich eh niemand darum zu kümmern
- kaputte Fenster verursachen weitere kaputte Fenster



## Broken-Windows-Theorie in der Entwicklung

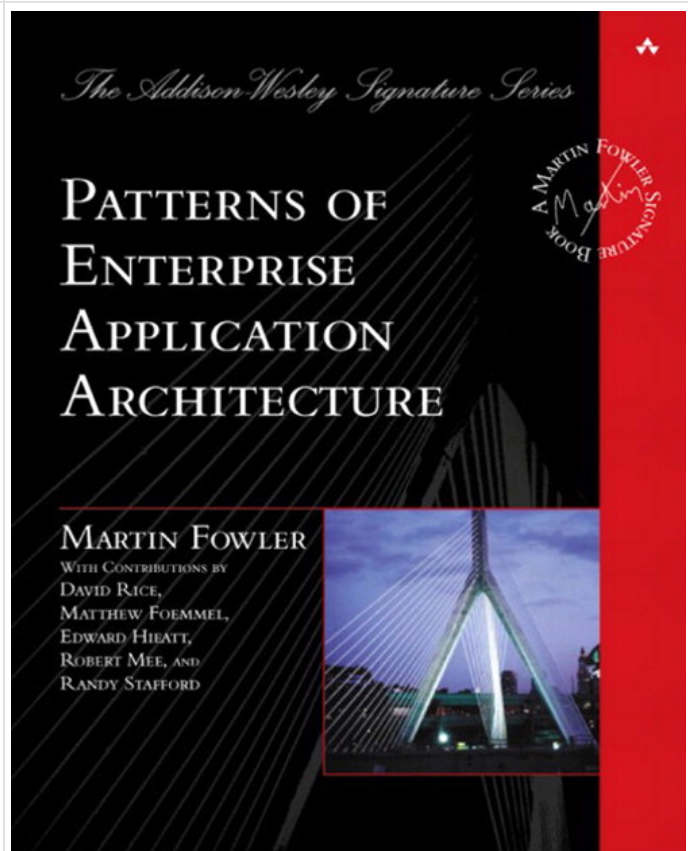
- Schulden / schlechter Code sorgt für weitere Schulden / schlechter Code
- niemand scheint sich um den Code / die Architektur zu kümmern → warum sollte man es als einzelner tun?
- Mentalität verändert sich
- dem gegenüber: wenn alles perfekt ist, traut sich niemand, etwas kaputt zu machen

## Literatur

Effektive Software Architekturen von Gernot Starke



Patterns of Enterprise Application Architecture von Martin Fowler



#### Weiterführende Links:

#### NOTE

- Überblick: <https://herbertograca.com/2017/07/03/the-software-architecture-chronicles/>
- Anti-Pattern: <http://www.laputan.org/pub/foote/mud.pdf>
- Technische Schuld: <https://martinfowler.com/bliki/TechnicalDebt.html>
- Zwiebel-Architektur: <https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/>
- Monolith first: <https://martinfowler.com/bliki/MonolithFirst.html>