

SOFTWARE-ARCHITEKTUR

Maurice Müller

2023-10-11

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

In Projekten

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

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."

BIG BALL OF MUD

Code that does something useful, but without explaining how

— Eric Evans

- Eigenschaften
 - Code verbirgt Intention (was/warum)
 - technische Belange belasten/verunreinigen die Fachlogik
 - Fachlogik ist über die (gesamte) Anwendung verteilt
- Resultat
 - selbst kleine Anpassungen sind nur mit hohem Aufwand (und hohem Risiko!) umsetzbar

BIG SMALL BALLS OF MUD

- es lohnt sich nicht immer alles korrekt zu designen
- manchmal ist Geschwindigkeit wichtiger als Qualität
- wichtig ist, dass der *Small Ball of Mud* isoliert vom Rest bleibt und eine saubere Schnittstelle hat
- Microservices

- 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."

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

ARCHITEKTUR UNTERSTÜTZT DEN ÜBERGANG VON DER ANALYSE ZUR REALISIERUNG

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

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?

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

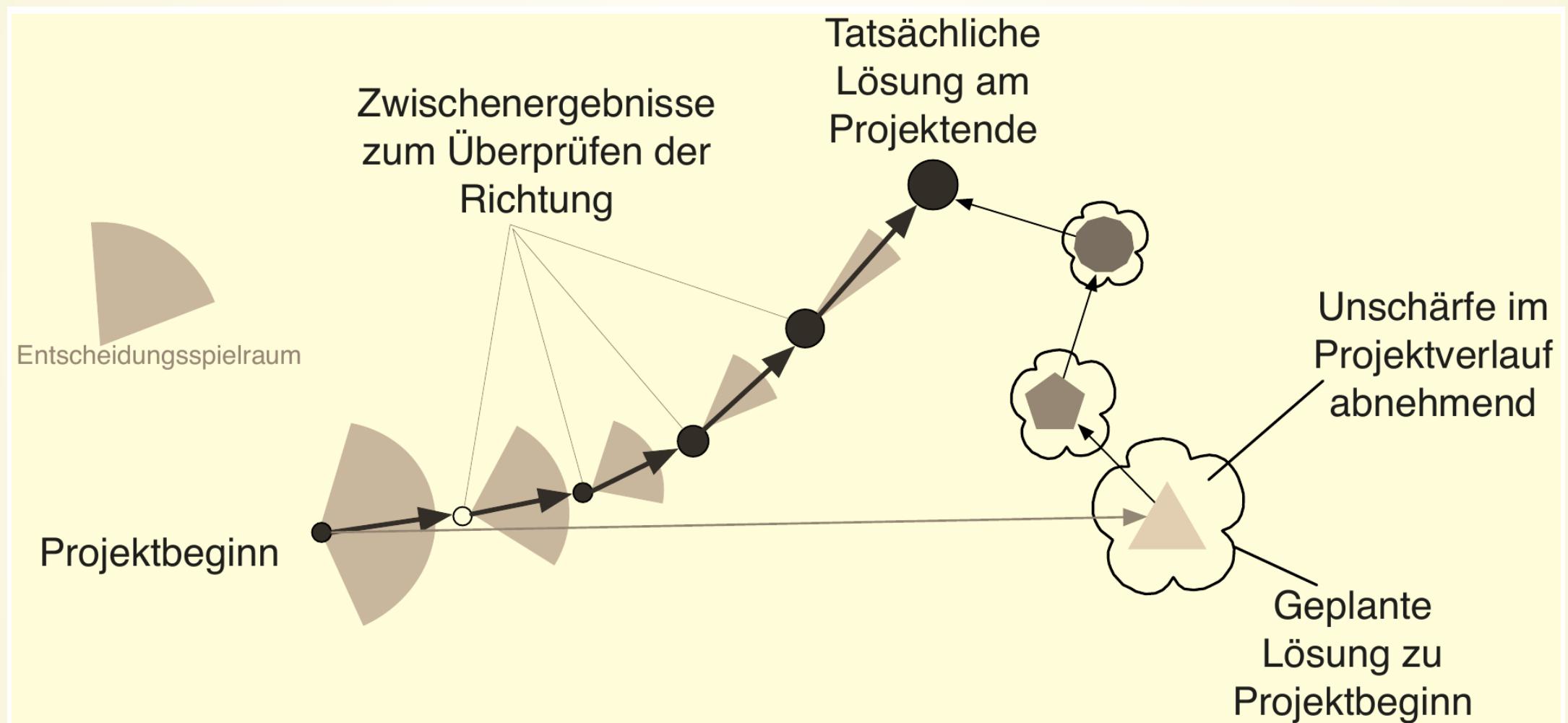
PROBLEM: MOVING TARGET

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

KONKRETES VORGEHEN

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

ANFORDERUNGEN & RANDBEDINGUNGEN KLÄREN

Immer zu erst nach vorhandenen Lösungen suchen!

- 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**

- 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

- explizit statt implizit
 - implizite Annahmen vermeiden und Entscheidungen / Entwurf explizit festhalten
 - z.B. fachliche Zusammenhänge zwischen Domänenobjekten, 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

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

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

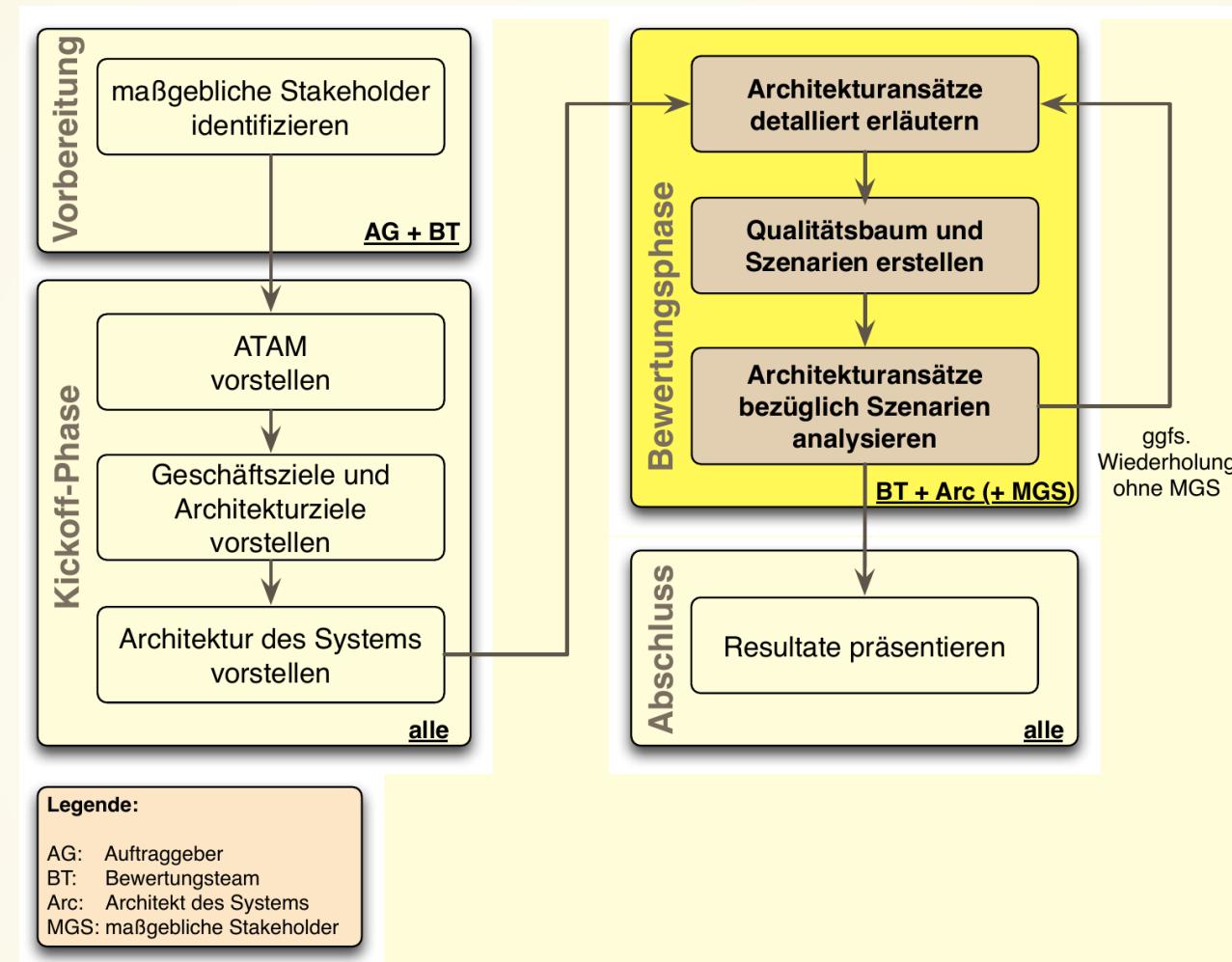
Ziele – Metriken

Soll/Ist – Zahlen

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

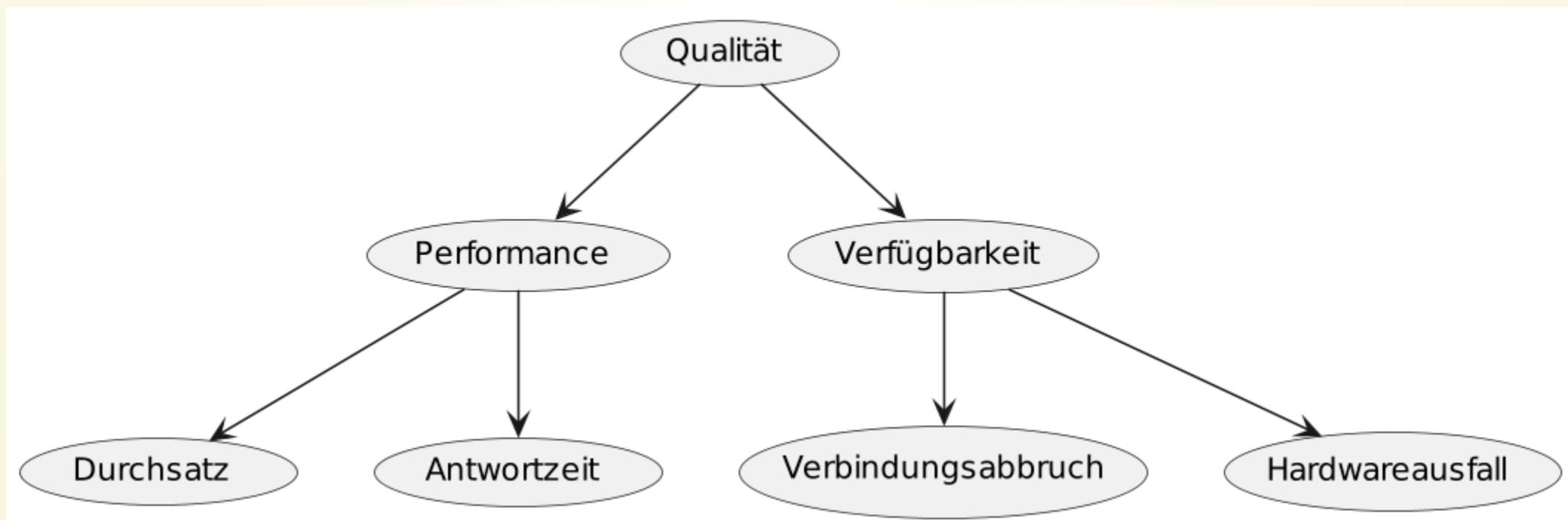
- Idealfall: Auftraggeber stellt geschäftliche (Qualitäts-)Ziele vor
- geschäftlicher Kontext & Einordnung in größere Systeme / Prozesse
- häufig sitzen erstmals alle maßgeblichen Stakeholder an einem Tisch

ARCHITEKTUR VORSTELLEN

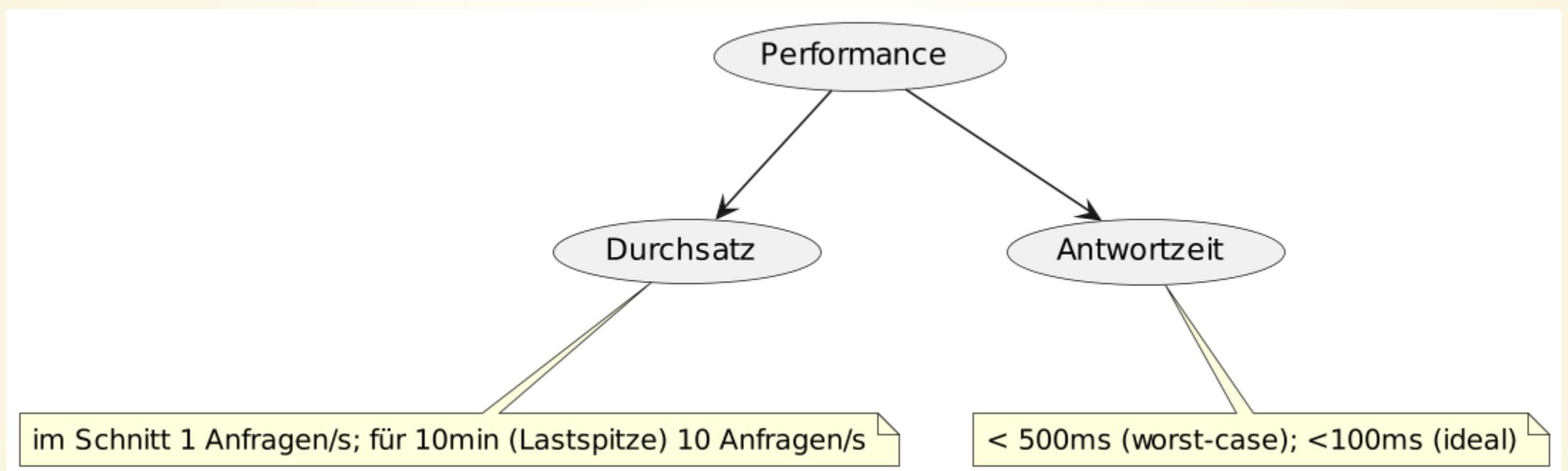
- hohe Abstraktion → nur die obersten Bausteine erläutern
- Kontext (d.h. Nachbarsysteme) einbeziehen

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

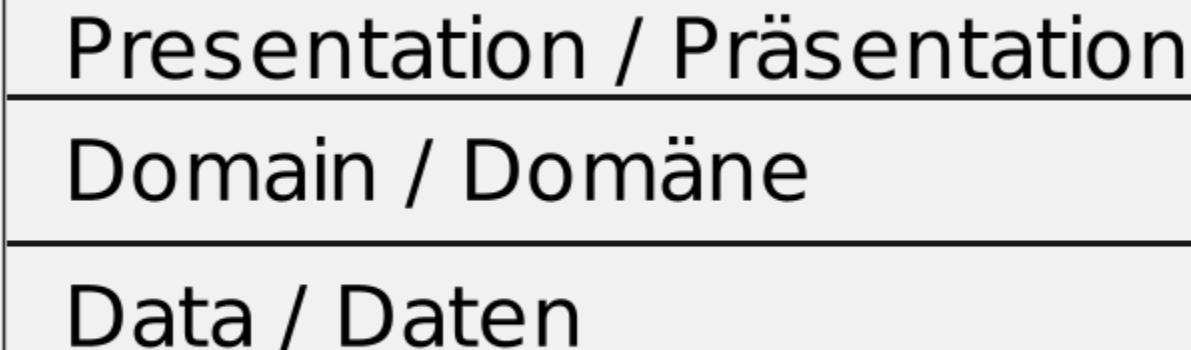
- Ergebnisse der Analyse präsentieren
- ggf. Entscheidungen / Maßnahmen treffen, um Architektur anzupassen

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** → **Separation 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, ...

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

slides-only::begin[] Wie würde das Schichtenmodell aussehen?

slides-only::end[]

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.

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

- die Anfrage und die Rückgabe muss durch alle Schichten durch
- auch durch die Domain-Schicht, die nur die Rechenlogik enthält
 - das macht wenig Sinn
- hoher Implementierungsaufwand

Schichtmodell

- Schichten dürfen direkt auf jede unterliegende Schicht zugreifen
 - z.B. könnte die Service-Schicht direkt auf die Daten-Schicht zugreifen
- geringerer Implementierungsaufwand
- Domain-Logik wird nicht verunreinigt

Generelle Vor-/Nachteile

Reines Schichtmodell

- (+) alle Schichten sind richtig voneinander isoliert
- (+) Änderungen in einer Schicht betrifft nur die direkt darunter liegende Schicht
- (-) hoher Implementierungs- und Wartungsaufwand
- (-) schwer durchschaubare implizite Abhängigkeiten zwischen Schichten
- (-) Verunreinigung zwischenliegender Schichten

Schichtmodell

- (+) erleichterter Implementierungs- und Wartungsaufwand, da direkt auf die benötigte Schicht zugegriffen werden kann
- (+) klarere Abhängigkeiten zwischen Schichten
- (-) Schichten haben mehr als eine andere Schicht als Abhängigkeit

HEXAGONALE ARCHITEKTUR

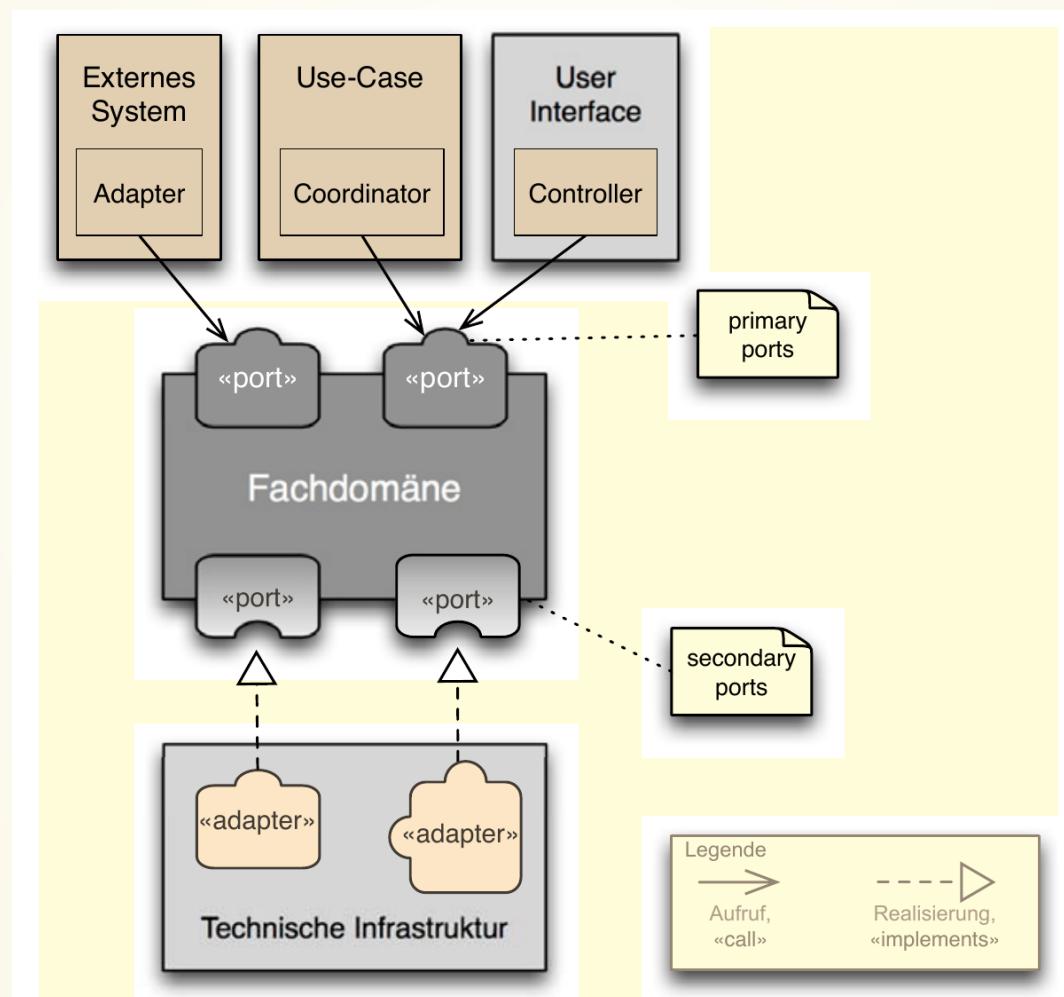
Ports & Adapters, Onion-Architecture

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

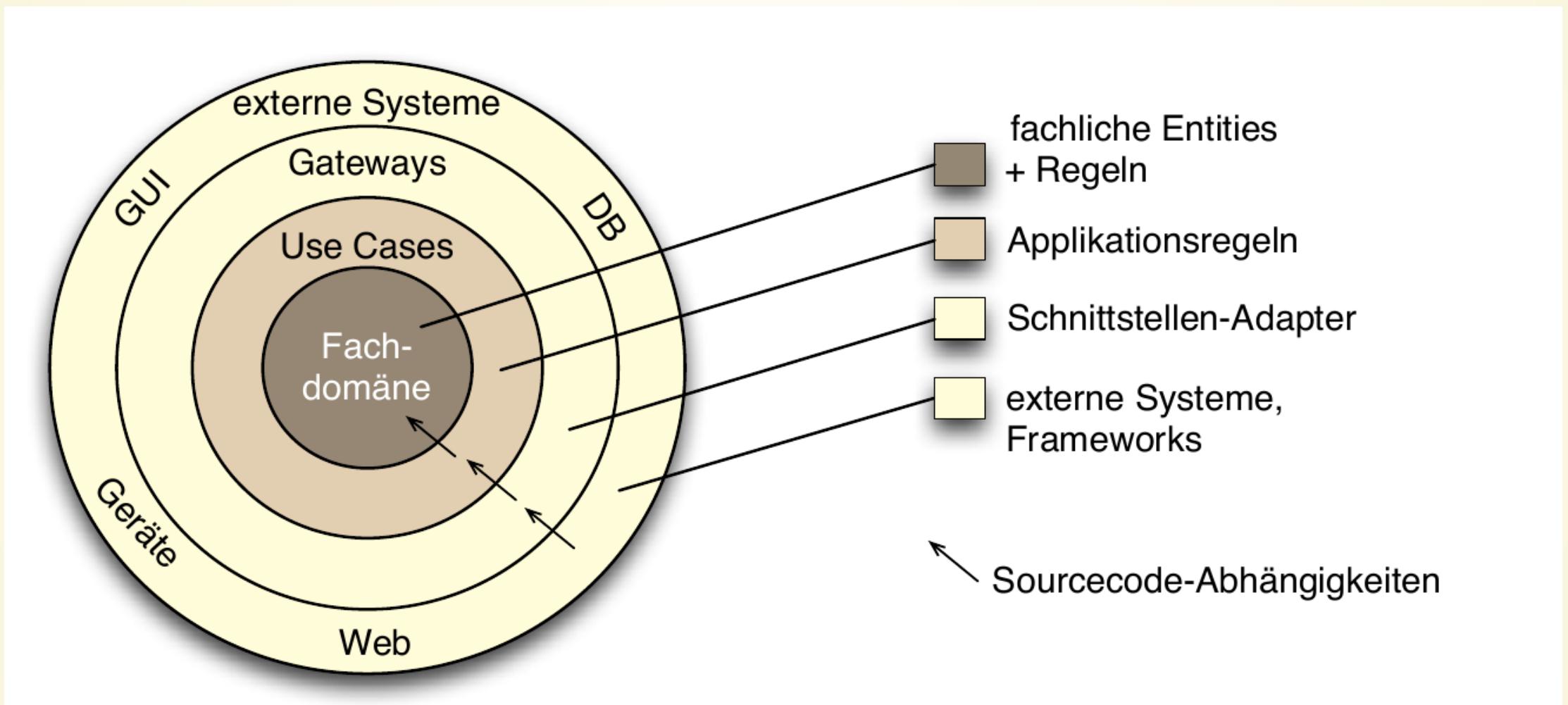
ursprüngliches Modell von Alistair Cockburn



Quelle: Effektive Software Architekturen von Gernot Starke

- **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

SCHALENMODEL



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

HEXAGONALE ARCHITEKTUR: BEISPIEL TASCHENRECHNER

ZUORDNUNG KOMPONENTE IN DAS SCHALENMODELL

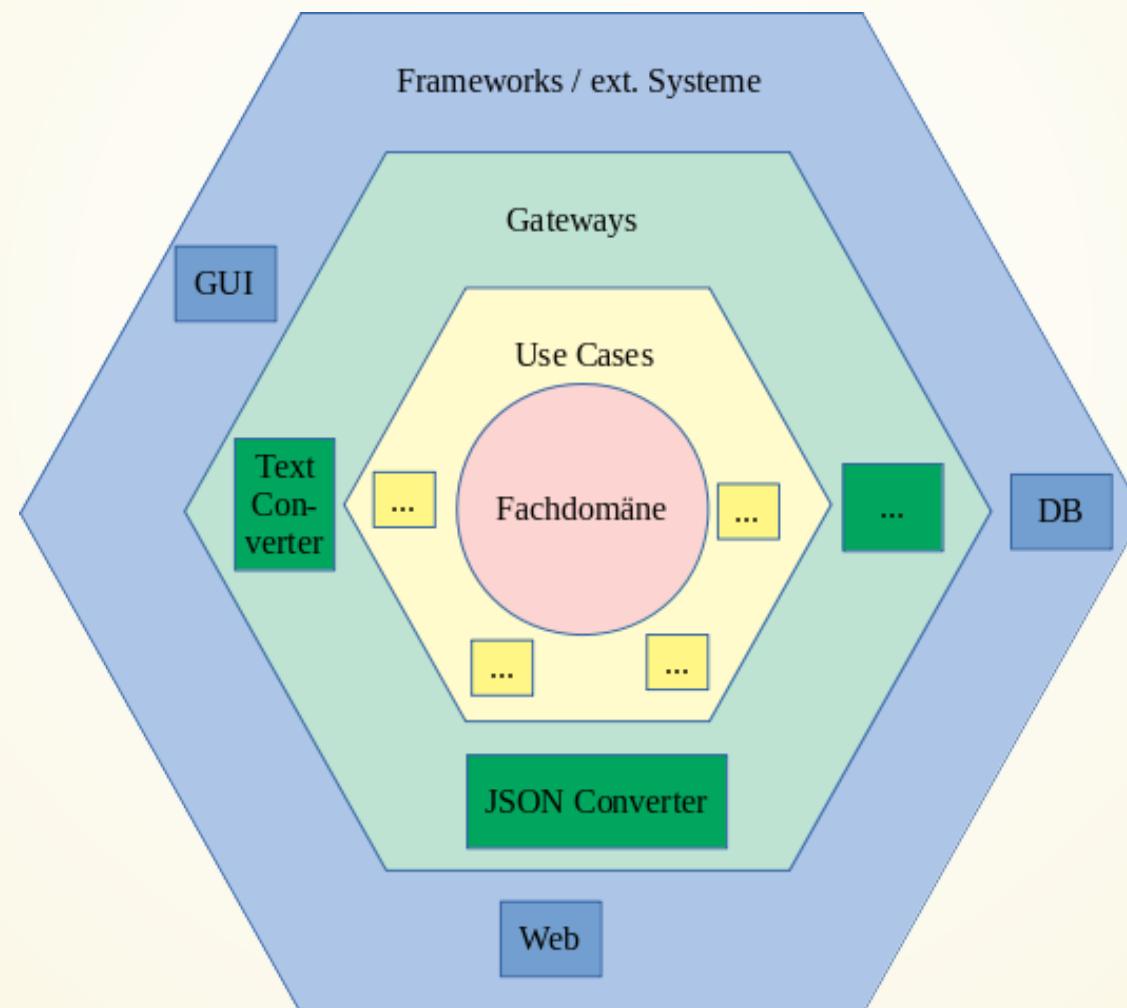
- **Fachdomäne:** Rechenoperationen und Zahlenobjekte
- **Use Cases:** parallele Verarbeitung
- **Gateway**
 - JSON \leftrightarrow Objekte für die HTTP API
 - Text \leftrightarrow Objekte für die CLI
 - DTO \leftrightarrow Objekte für die Datenbank
- **Frameworks/ext. Systeme:** HTTP-Server, Terminal-Anbindung,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

ALTERNATIVE DARSTELLUNG DES SCHALENMODELLS

Darstellung als geschachtelte Hexagons



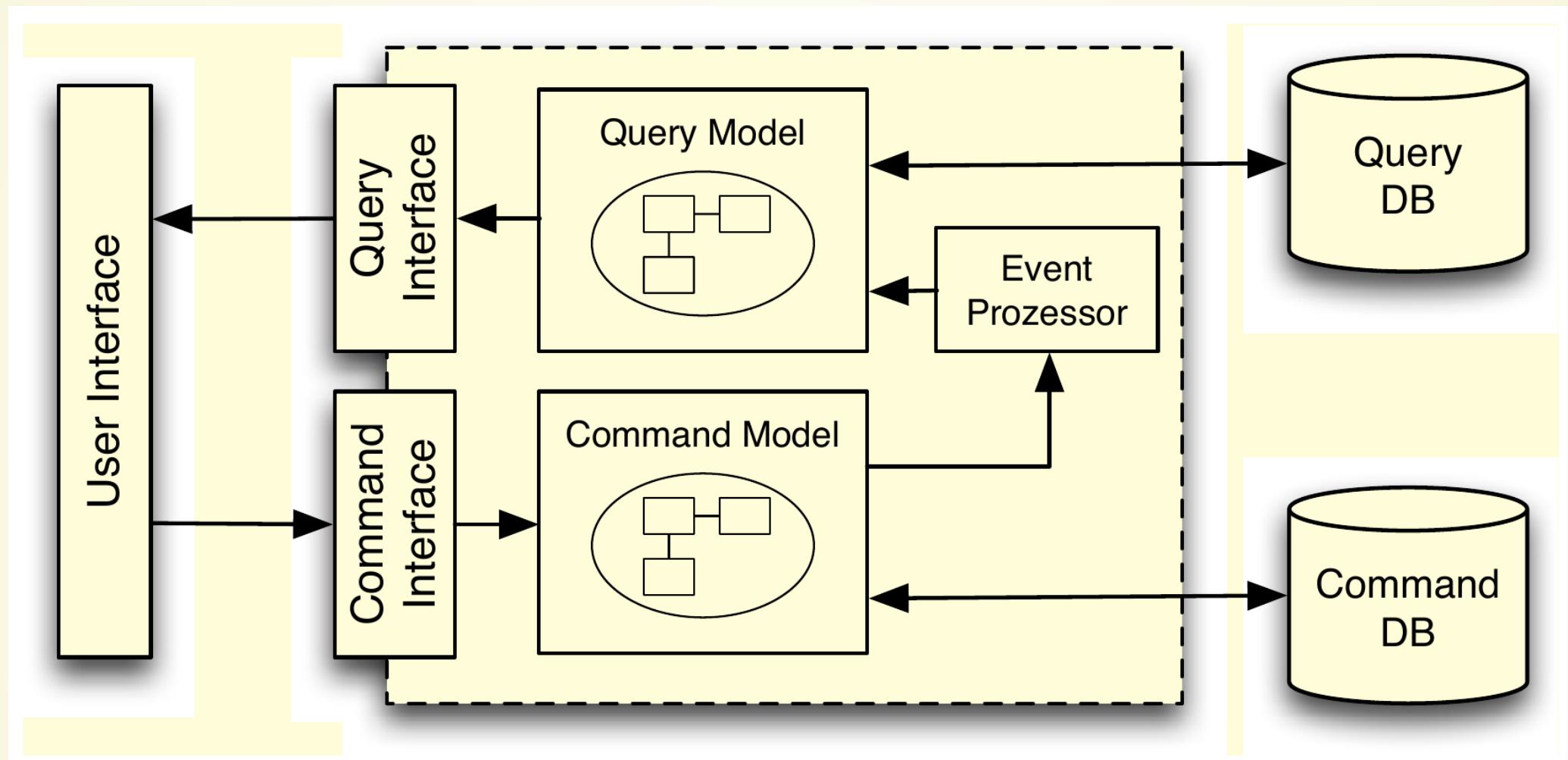
CQRS

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.



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: 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 peformante Abfragen notwendig sind
- (o) eher ungeignet als Gesamtarchitektur → eher geeignet für einzelne Komponente

SOA

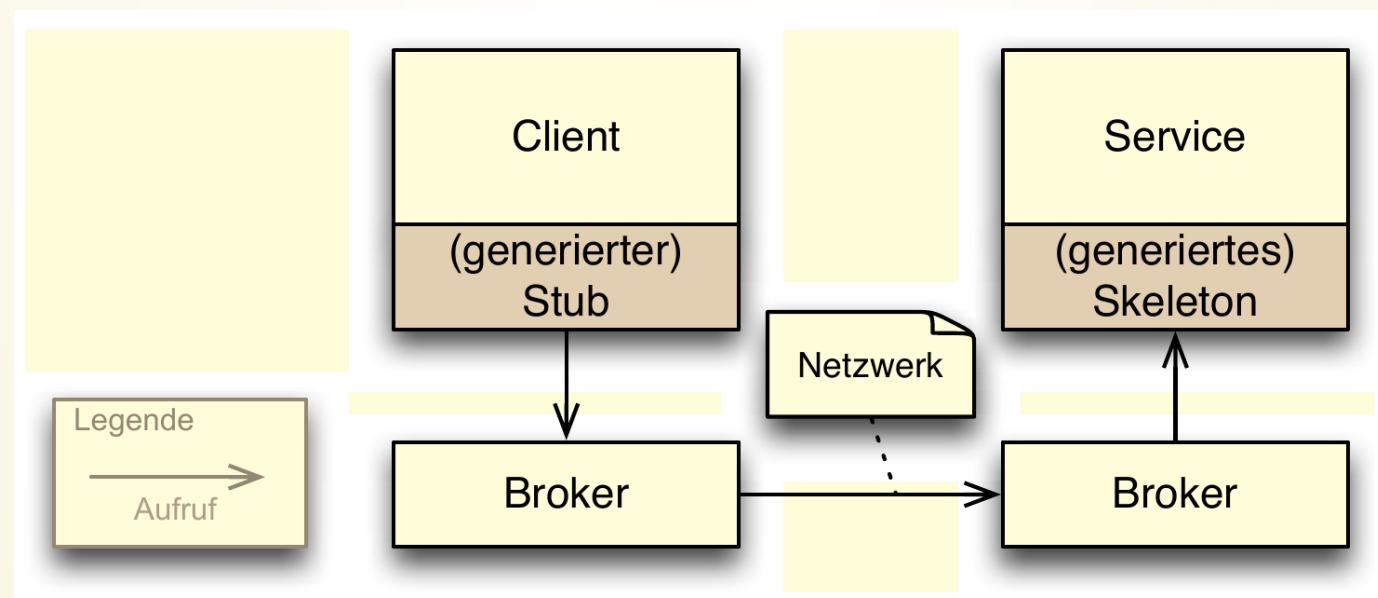
Service-oriented architecture

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

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

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.

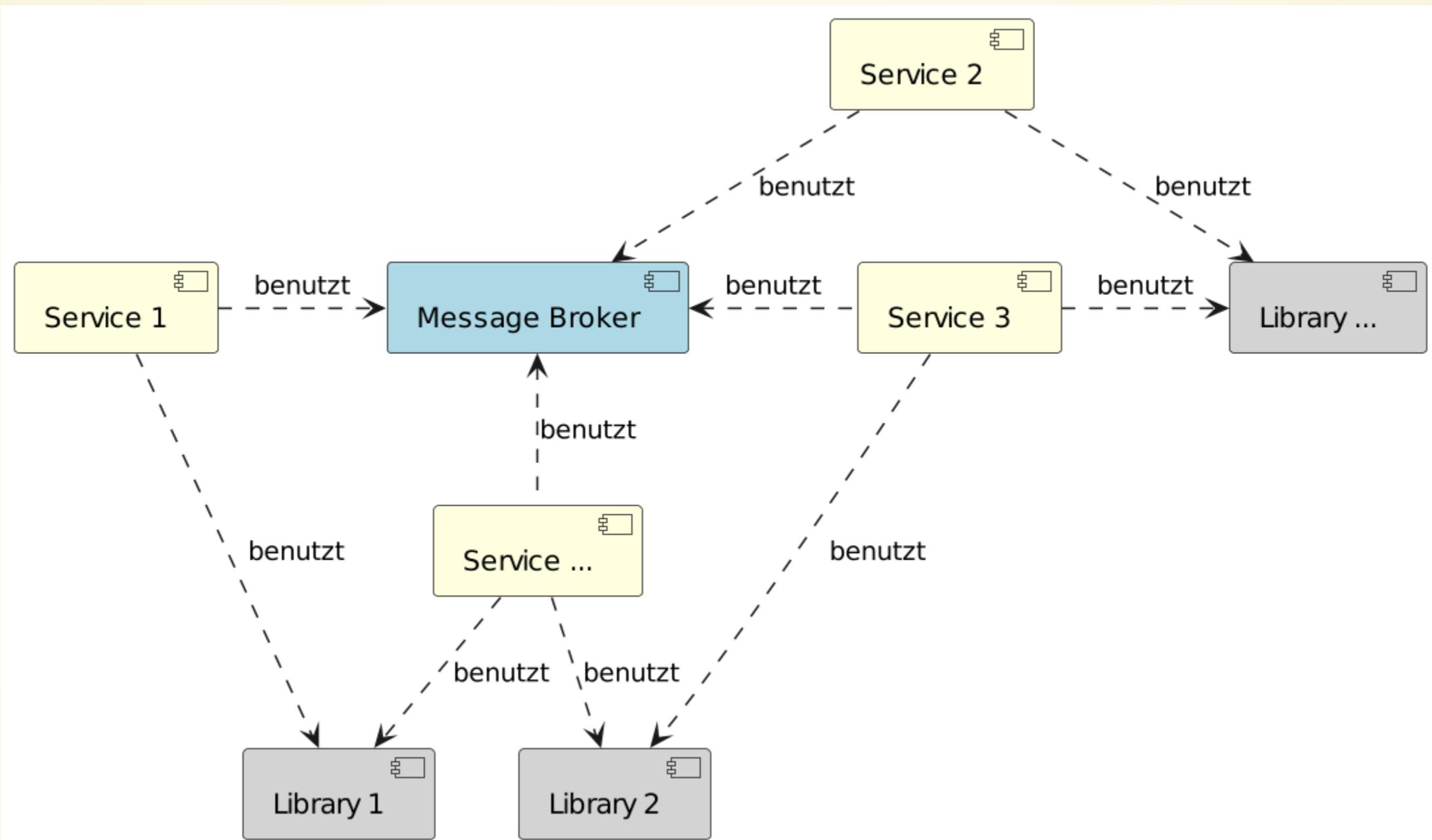
→ Die Qualität eines Monoliths hängt von der Qualität der internen Architektur ab.

MONOLITH: VOR- UND NACHTEILE

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

MICROSERVICES

- 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

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

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 → zuerst 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 $\leftarrow \rightarrow$ 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 → 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

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

