

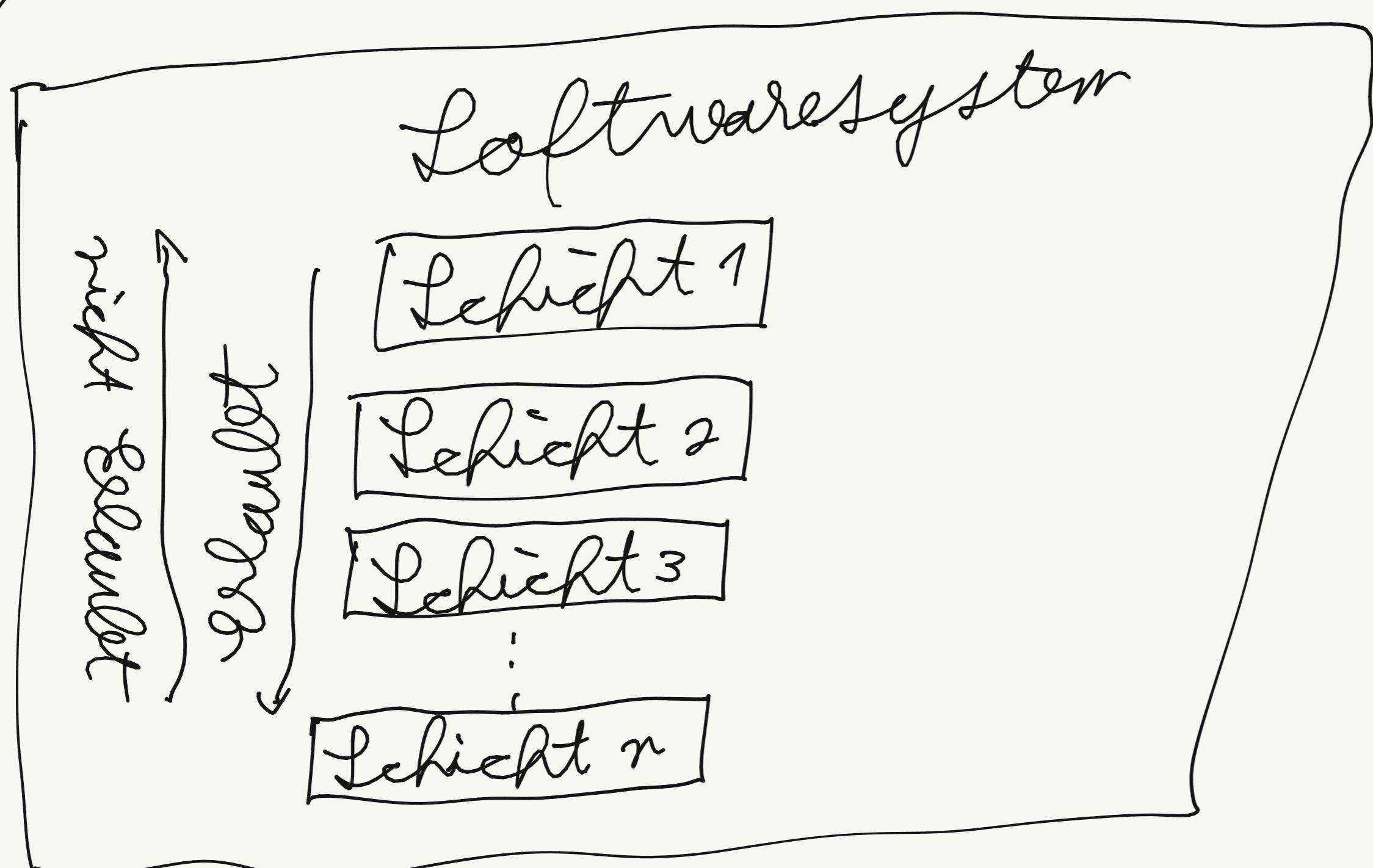
Vorlesung 4

Architekturmuster

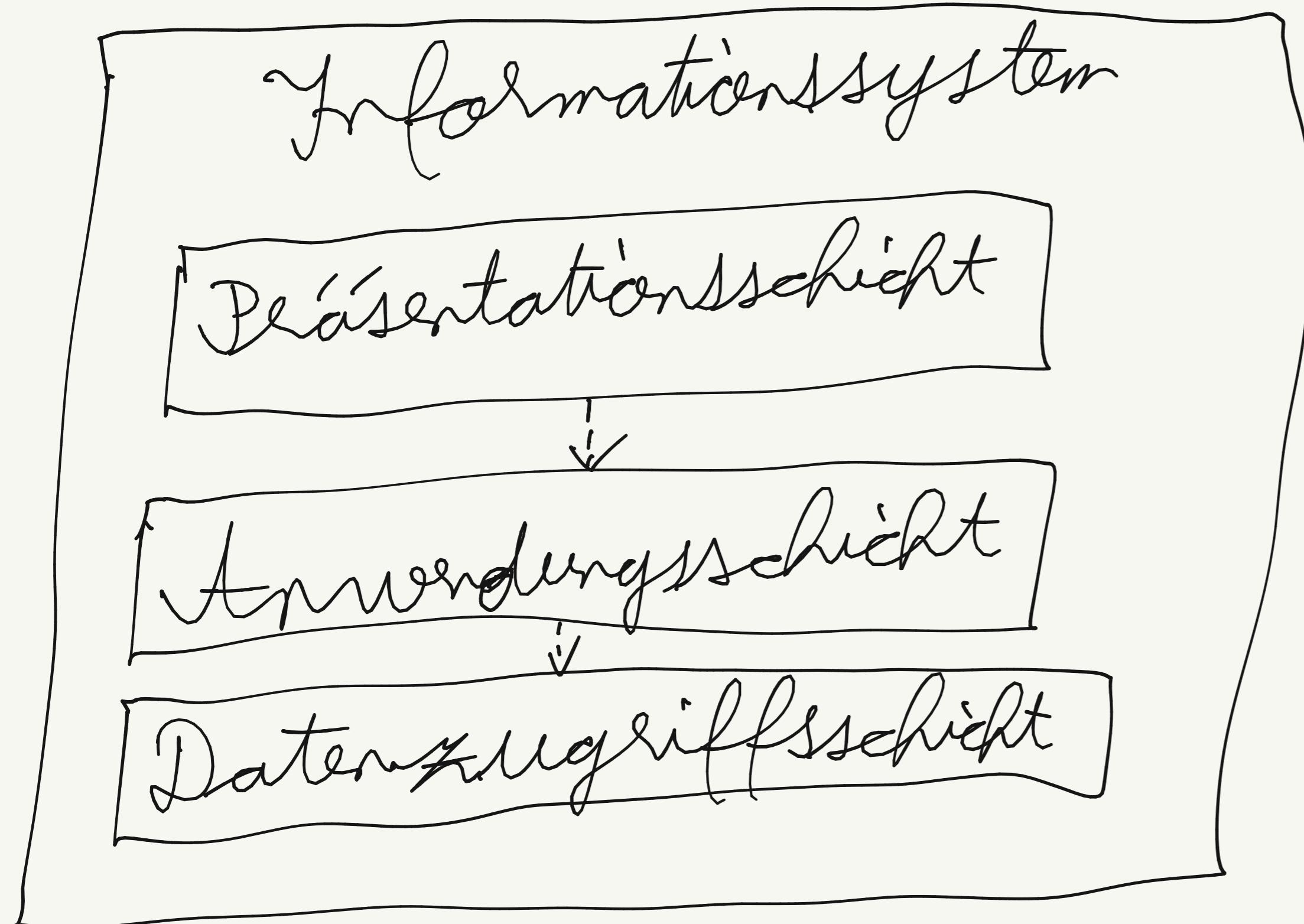
- Bewährte Lösungen für häufig auftretende Probleme beim Architekturentwurf
- Bieter strukturierte Herangehensweise und vordefinierte Konfigurationen, um spezifische funktionale und nicht-funktionale Anforderungen eines Systems zu erfüllen
- Abstrakte Ebene als Designmuster
→ Fokus auf Gesamtstruktur des Systems und das Zusammenspiel von Komponenten

Schichtenarchitektur

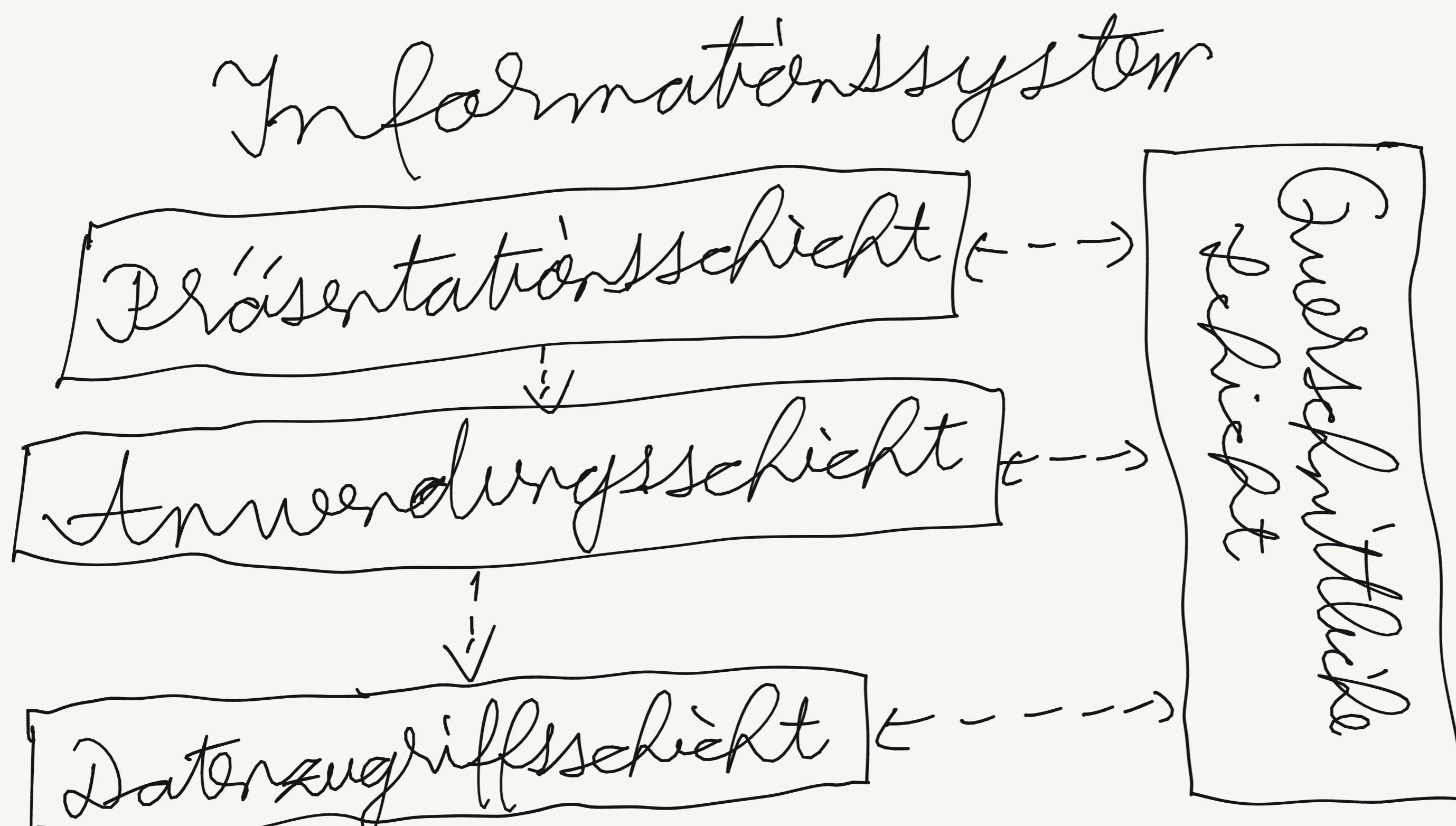
- Das Softwaresystem wird in Schichten strukturiert. Einzelne Aspekte (z.B.: Klassen oder Komponenten) eines Softwaresystems werden jeweils einer Schicht zugeordnet
- Aspekte einer höheren Schicht dürfen ausschließlich Aspekte der gleichen oder einer niedrigeren Schicht verwenden



3 - Schichter-Architektur

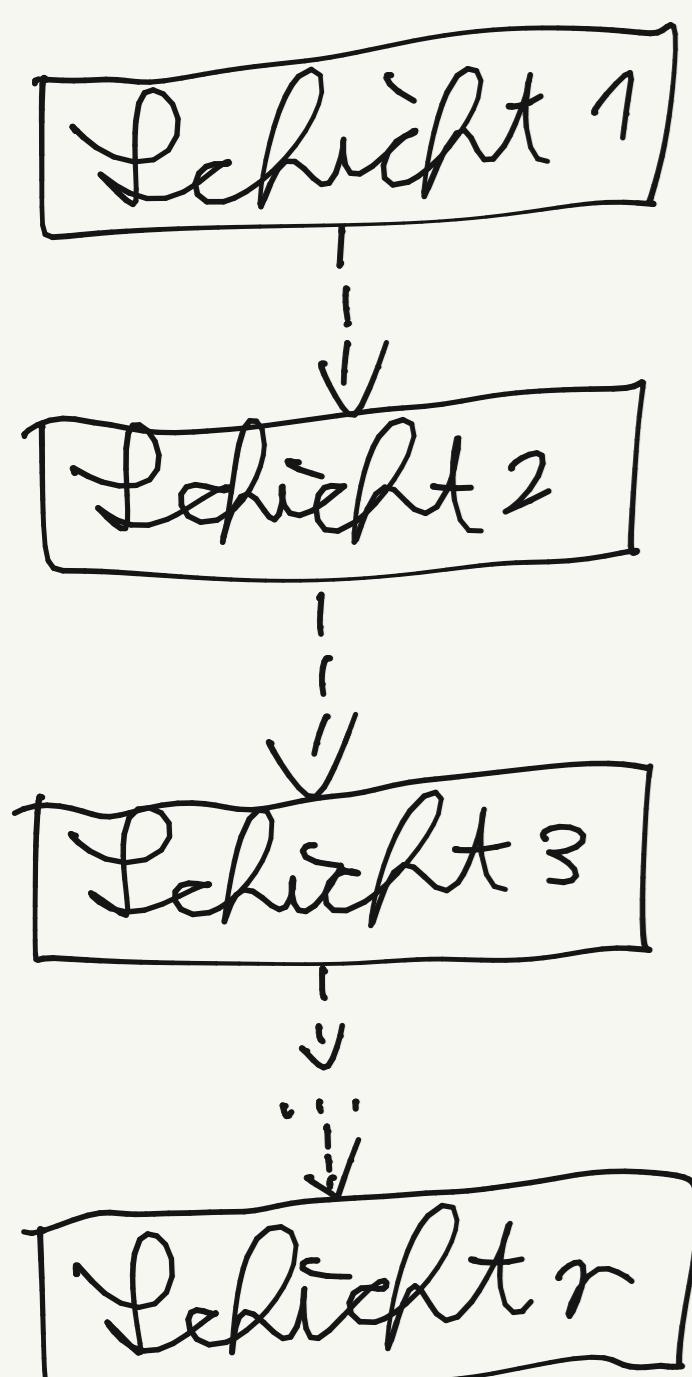


Geschnittliche Schicht



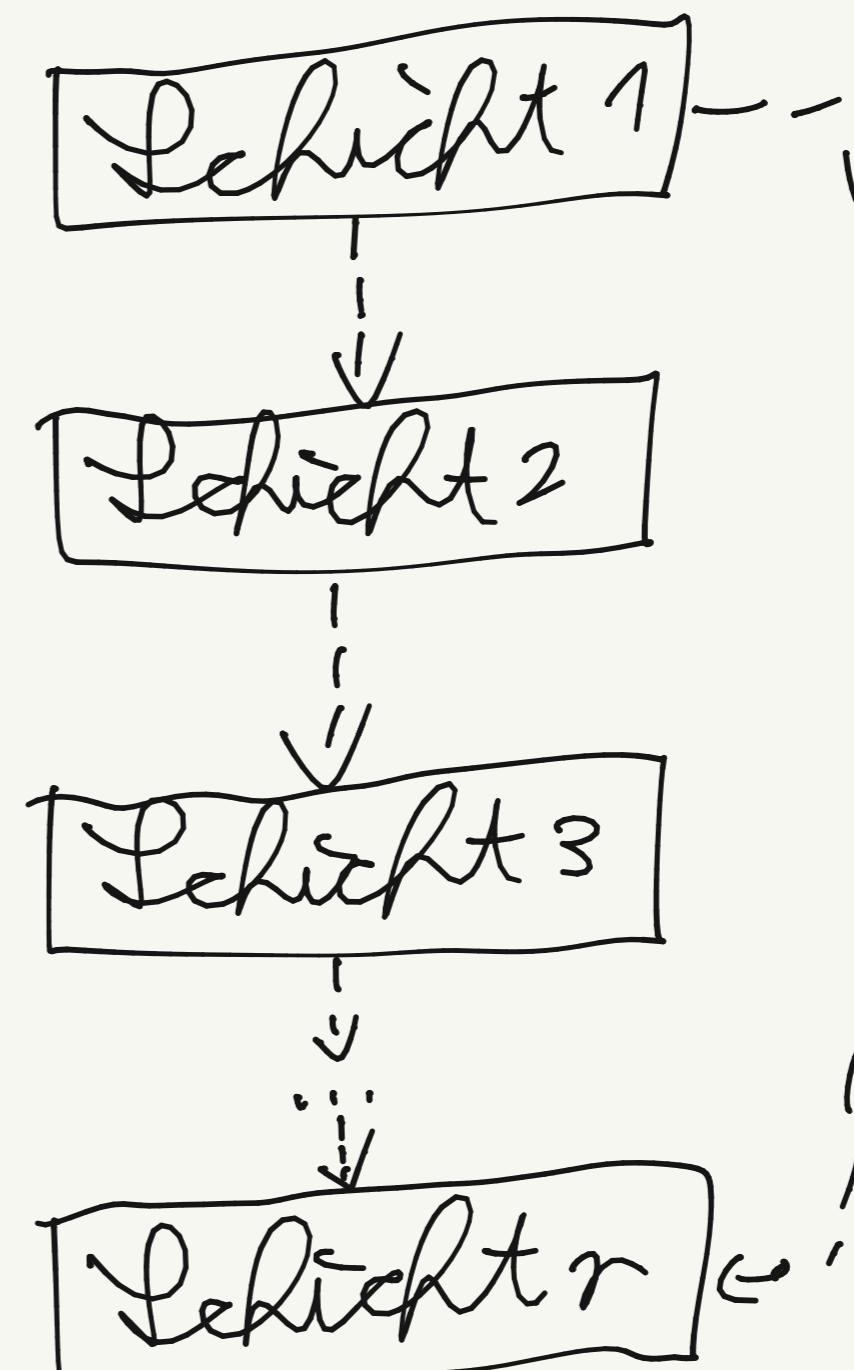
Strikte vs. Nicht-strikte Schichtenarchitektur

strikt



Zugriff ausschließlich
auf die nächst-niedere
Schicht erlaubt

nicht-strikt



Zugriff auf beliebige
untere Schicht erlaubt

nicht-strikt

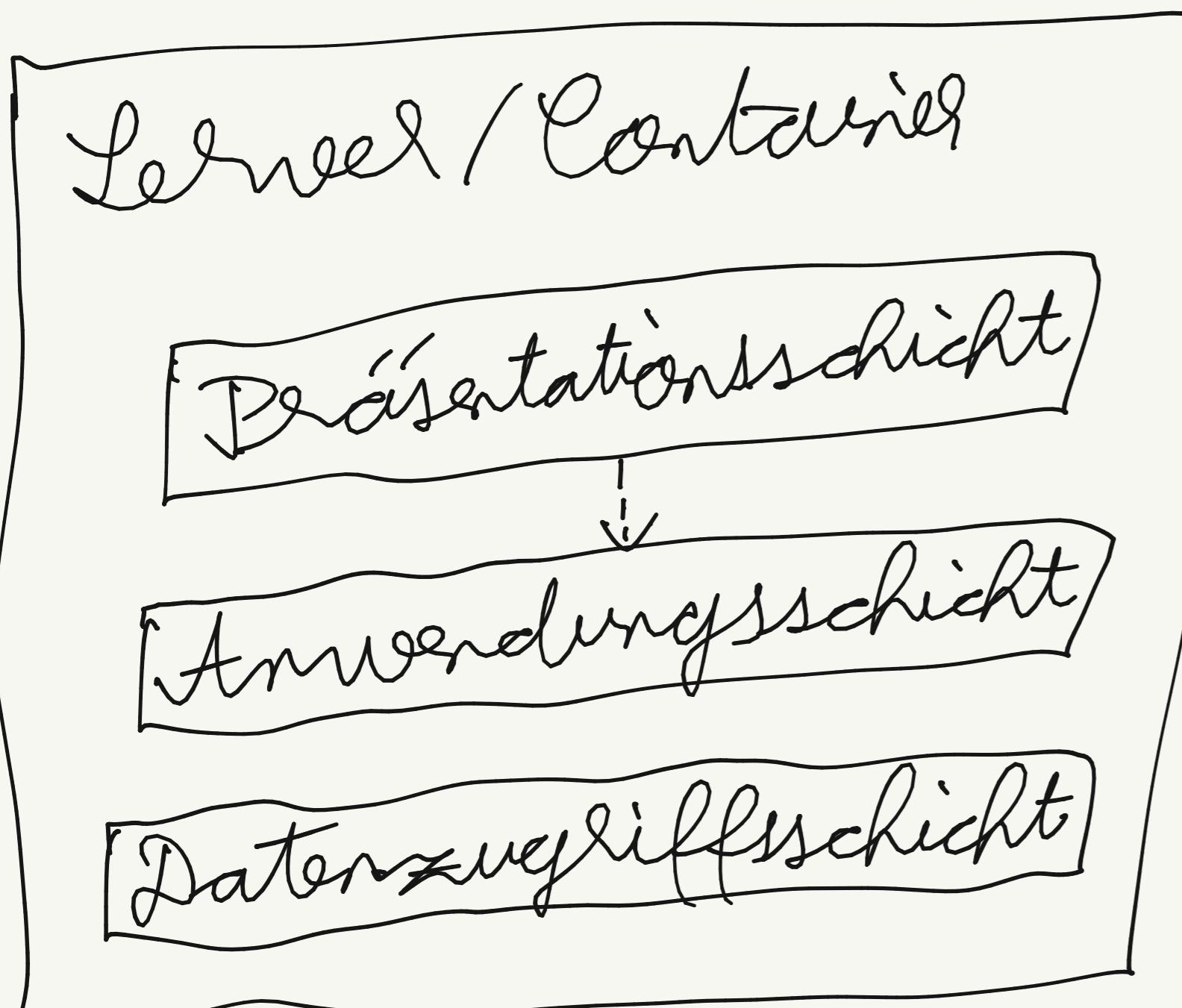
- Bessere Performance, da der Zugriff nicht erst über X Schichten weitergeleitet werden muss
- Dafür eine engere Kopplung als bei der strikten Schichtenarchitektur
- Trade-off zwischen Performance und loser Kopplung notwendig

Einige Möglichkeiten der Realisierung:

- Paketenstruktur
- Komponenten
- Namenskonventionen
- Frameworks und Bibliotheken
- Kombination mit anderer Architekturmustern

Deployment-Varianten

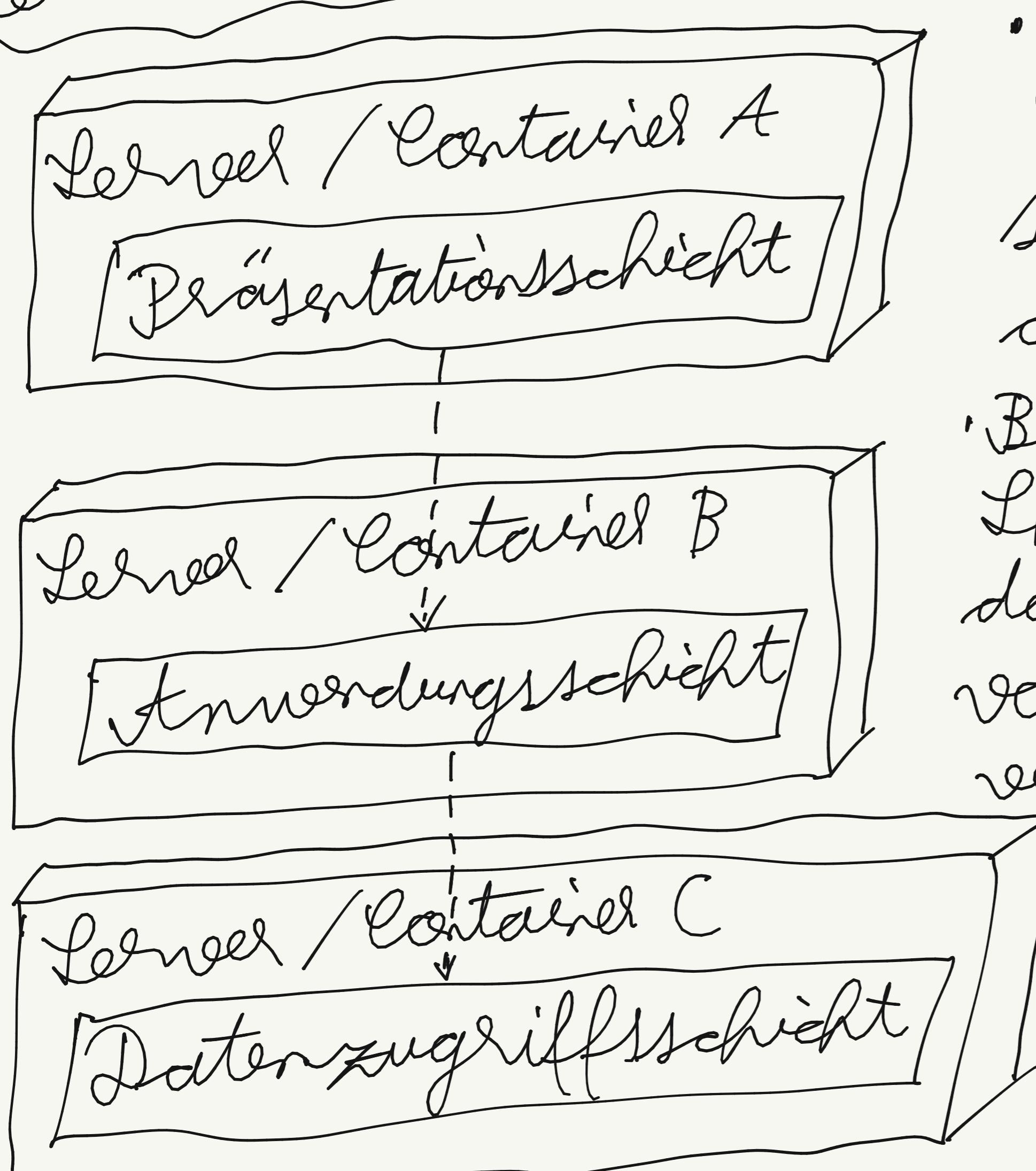
Monolithische Bereitstellung



- Alle Schichten werden gemeinsam auf einer Serverinstanz oder einem Container ausgeführt
- Eignet sich für kleinere Anwendungen mit einfacher Architektur und geringer Anforderungen an Skalierbarkeit

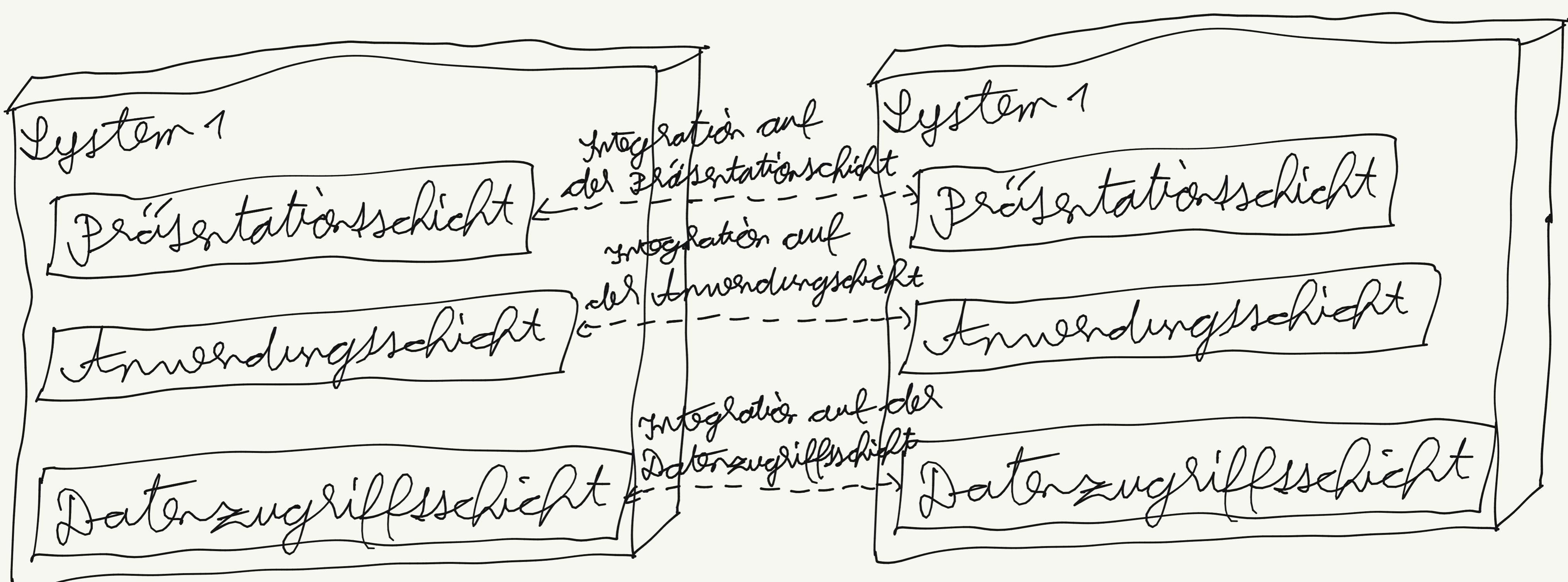
- Debugging und Management der Anwendung oft einfacher als bei verteilter Bereitstellung

Verteilte Bereitstellung



- Alle oder einzelne Schichten werden auf separaten Serverinstanzen oder Containern bereitgestellt
- Bietet eine bessere Skalierbarkeit und Flexibilität, da jede Schicht unabhängig voneinander skaliert und verwaltet werden kann
- Debugging und Verwaltung des Gesamtsystems können schwierig und aufwändig werden

Integration zweier Anwendungen



Präsentationschicht

- Technische Umsetzung z.B. Beispiel mittels iFrames (Inline Frames)
- z.B. Linked To

Anwendungsschicht

- Web-Services und APIs
- Middleware und Message-Broker
- SDKs

Datenzugriffsschicht

- Direkte Datenbankverbindung
- Datenbank-Replikation
- Datenbankschnittstellen und APIs
- Datenbankpools und Datenbankservices

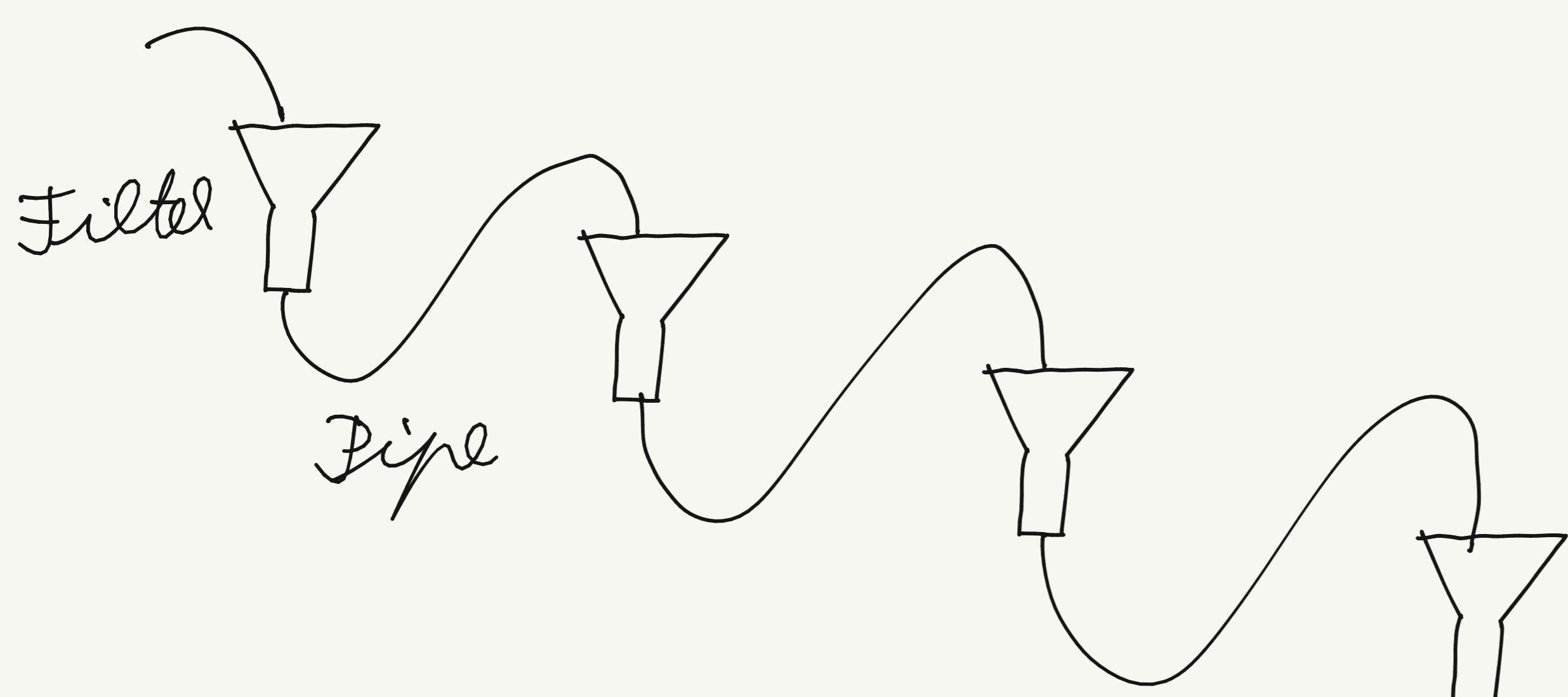
Einige Kritikpunkte und Fazit

- Gefahr von technischer Schulden innerhalb von Schichten
- Breakdown durch Schichten
- Starre Struktur

Trotz dieser Kritikpunkte wird die Schichtenarchitektur nach wie vor häufig in der Softwareentwicklung eingesetzt, da sie eine klare Trennung von Verantwortlichkeiten und eine modulare Struktur bietet, die die Wartung, Skalierung und Weiterentwicklung von Anwendungen erleichtert.

Pipes und Filters

- Basiert auf dem Konzept der Datenverarbeitungspipeline
- Die Architektur besteht aus:
 - Filter: Verarbeitung von Daten
 - Pipes: ermöglichen den Datenfluss zwischen Filtern



Filter

- Jeder Filter verarbeitet also eingehende Daten und gibt sie über eine Pipe an den nächsten Filter weiter
- Verarbeiter kann bedeuten:
 - Herausfiltern/Löschen von Daten
 - Hinzufügen von Daten
 - Verändern von Daten

Eigenschaften der Architektur

- Jeder Filter erfüllt eine spezifische Aufgabe, dadurch Trennung von Verantwortlichkeit
- Jeder Filter kann unabhängig von anderen Filtern entwickelt werden
- Ermöglicht Wiederverwendbarkeit von Filtern
- einfache Konfiguration/Rekonfiguration des Gesamtsystems durch:
 - Hinzufügen/Entfernen von Filtern
 - Änderung der Filter-Reihenfolge

Gemeinsamer Speicher

- Gemeinsamer Speicher / zentrale Datenstruktur nicht zwangsläufig notwendig
 - Jeder Filter gibt in der Regel seine Ausgabe direkt an den nächsten Filter weiter
 - Fordert eine lose Kopplung zwischen Filtern
 - Forderung einer losen Kopplung zwischen Filtern kann notwendig werden, falls:
- Gemeinsamer Speicher kann notwendig werden, falls:
 - Zustandsbehaftete Verarbeitung erforderlich ist
 - Kommunikation über den Datenstrom hinaus notwendig ist
 - Parallelisierung oder Spalierung erforderlich ist

Beispiel für Anwendungsbereiche

- Textverarbeitung
 - Rechtschreibprüfung, Grammatikprüfung)
 - Textformatierungen, Übersetzungen
- Bildverarbeitung
 - Rauschunterdrückung, Konturerkennung)
 - Farbspektren, Objekterkennung
- Audiodatenverarbeitung
 - Rauschunterdrückung, Equalizer,
 - Spracherkennung, Audiokompressor
- Datenanalyse und -transformation
 - ETL-Prozesse (Extract, Transform, Load), Datenbereinigung) - aggregation und -visualisierung

Blackboard-Architektur

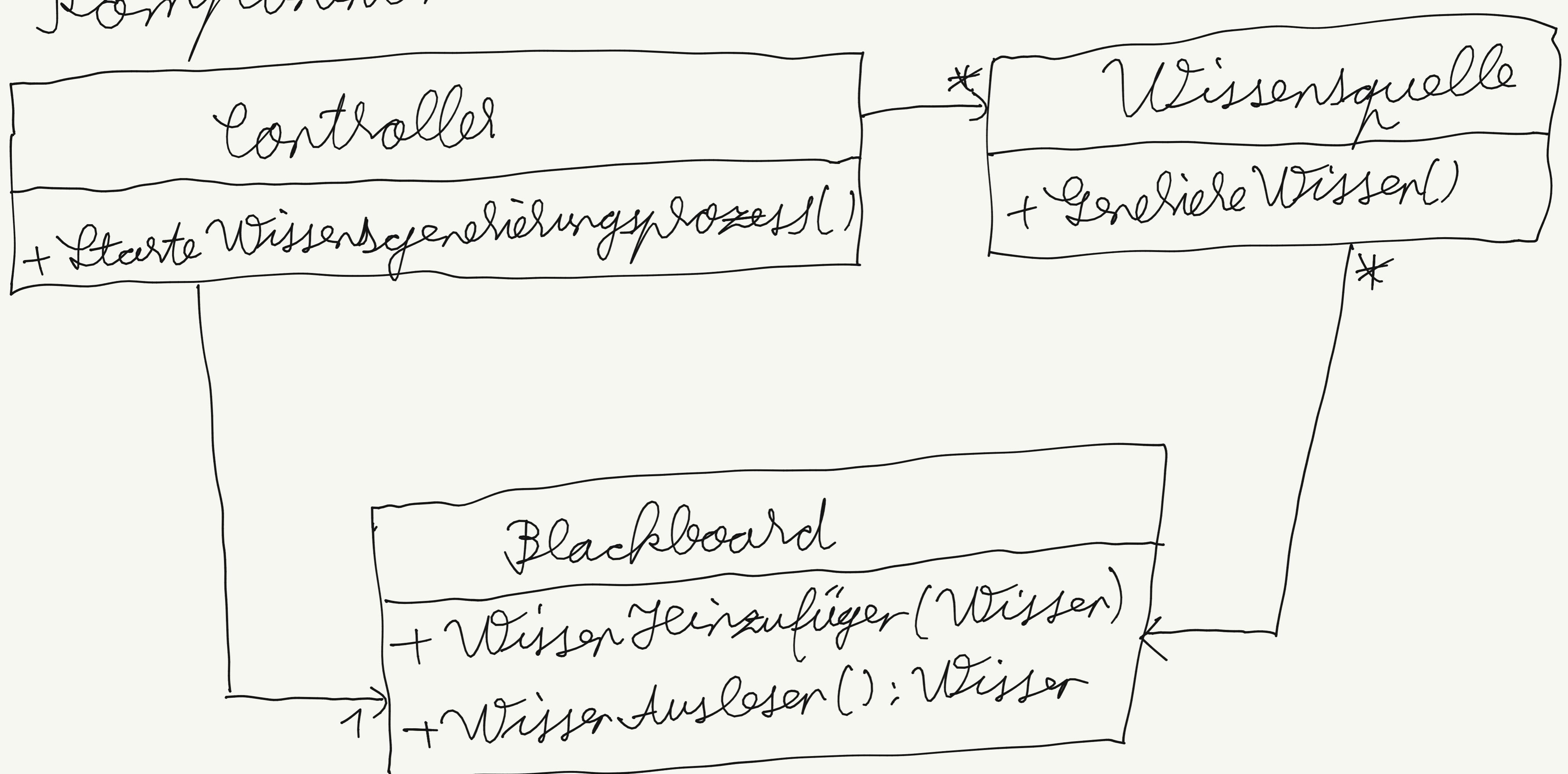
- hilfreich für Probleme, zu denen keine deterministische Lösungsstrategie bekannt ist
Gründe hierfür können zum Beispiel sein,
dass das Problem:
 - Komplexität aufweist, die über die Fähigkeit deterministischer Methoden hinausgeht
 - Unstrukturiert ist und keine klareren Regeln oder Muster aufweist nach denen vorgegangen werden kann
 - Unsicherheit beinhaltet, so dass selbst bei der Anwendung bestarker Methoden das Ergebnis unvorhersehbar sein kann
- Eine Blackboard-Architektur kann hierfür eine Lösung sein
 - ermöglicht flexible und dynamische Vorgehensweise durch Anpassung des Vorgehens an neue, aktuelle Informationen
 - Verschiedene Wissensquellen arbeiten zusammen, um gemeinsam eine Lösung zu finden
 - Die Lösungsentwicklung erfolgt iterativ
⇒ Hypothesen werden generiert, getestet und neue Informationen integriert

Typische Anwendungsfälle:

- Komplexe medizinische Diagnosen
- Wissenschaftliche Forschung und Entdeckung
- Kreatives Problemlösen und Design

→ Anwendung immer dann, wenn ein komplexes Problem gelöst werden muss, das von verschiedener Experten mit unterschiedlicher Fachwissen abgängiger werden kann

Komponenten einer Blackboard-Architektur:



Blackboard:

- Zentraler Datenspeicher
- Speicherort für alle verfügbare Rohdaten, Informationen und Hypothesen
- Ist für alle Beteiligten Komponenten zugänglich
- Zentrale Schnittstelle für die Zusammenarbeit
- Zentrale Schnittstelle zum Lesen und Schreiben
- Bietet Schnittsteller zum Lesen und Schreiben am schwarzen Brett

Wissensquelle

- Eigenständige Komponenten oder Expertensysteme
- Steller spezialisiertes Wissen oder Algorithmen
- für die Lösung eines Problems bereit
- Jede Wissensquelle ist für einen bestimmten Aspekt des Problems zuständig
- Erzeugt, analysiert oder verfeinert Hypothesen
- Interagieren über das Blackboard miteinander
- Signalisiern, unter welchen Bedingungen sie zu einer (Teil-) Lösung beitragen können

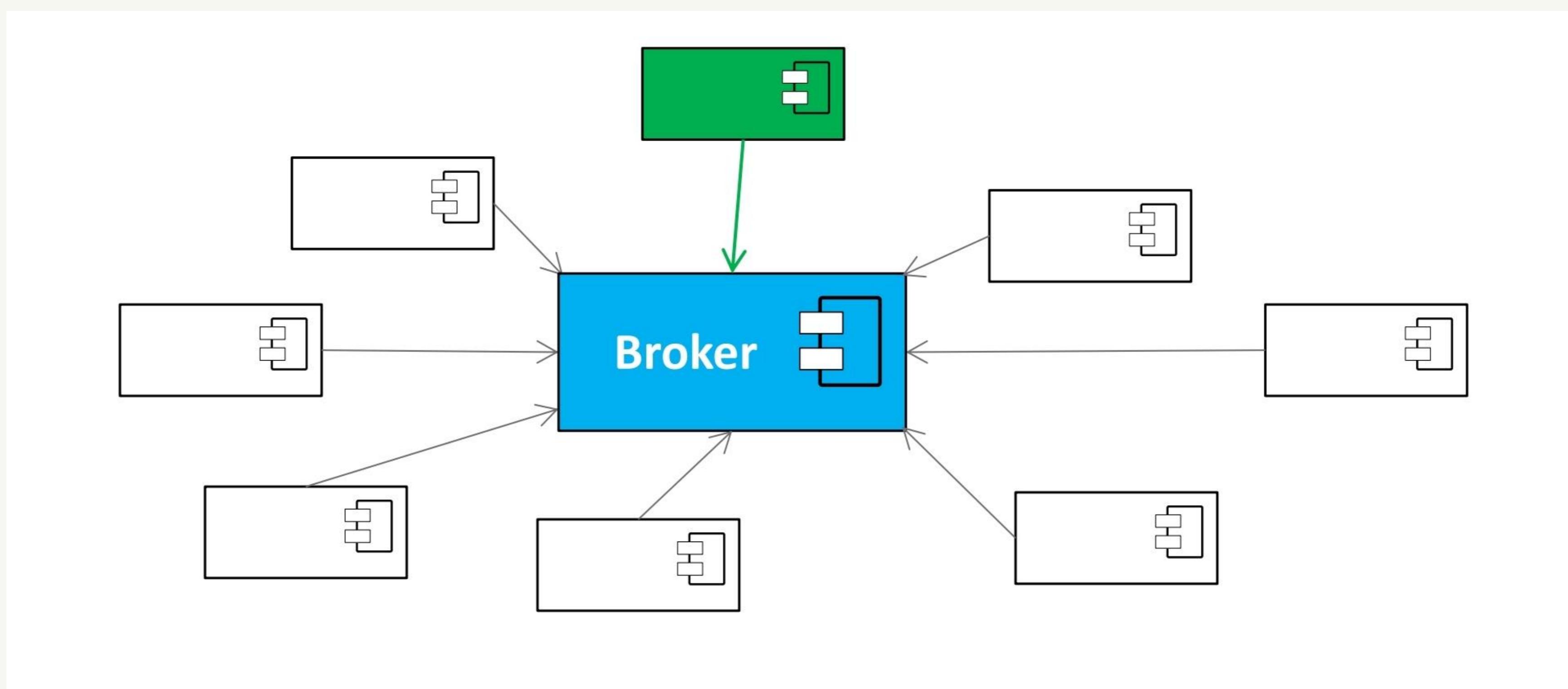
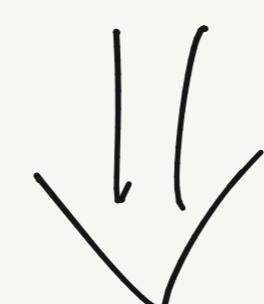
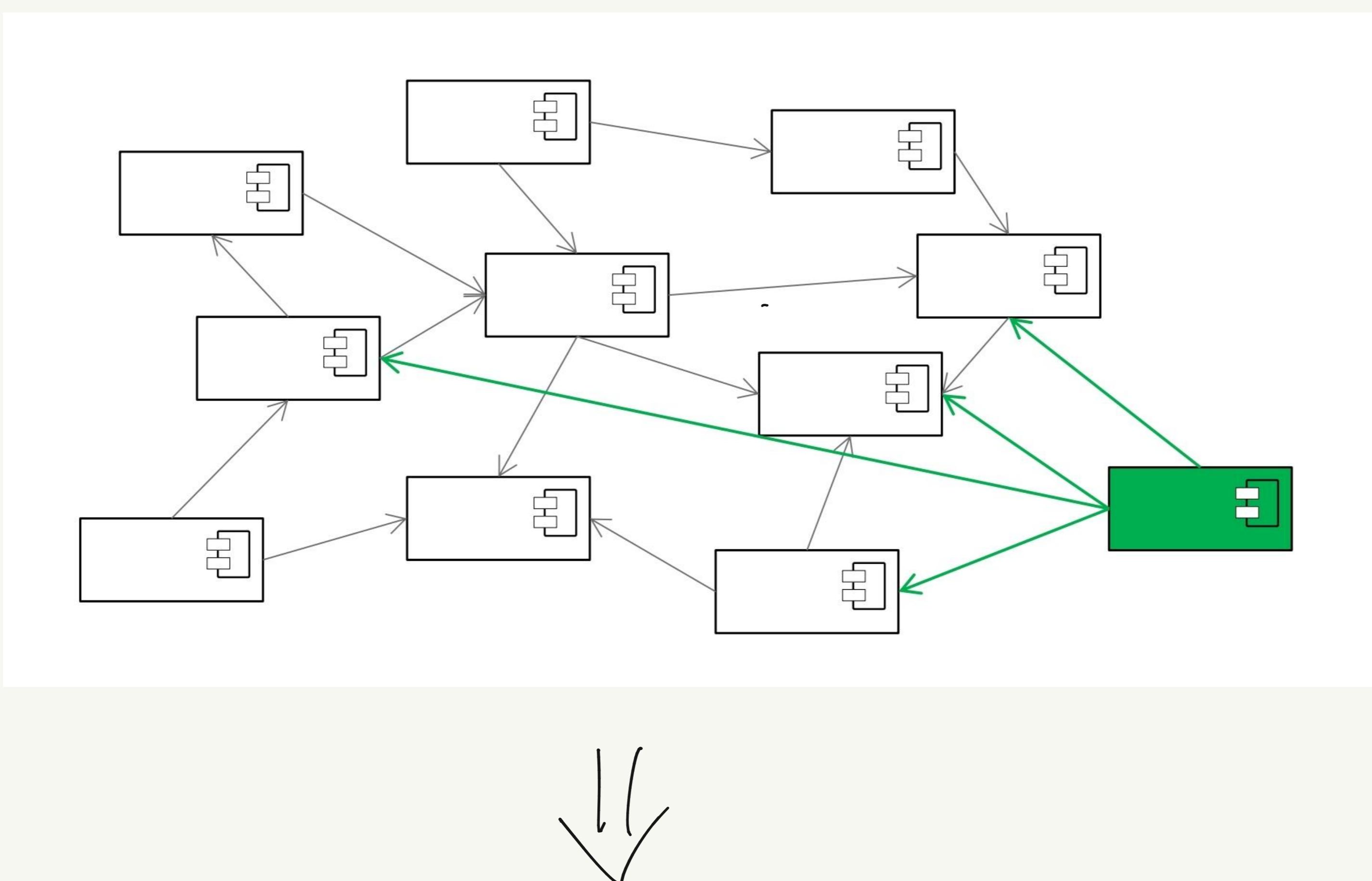
Controller:

- Überwacht und steuert den Ablauf des Lösungsprozesses
- Entscheidet, welche Wissensquellen aktiviert werden sollen, basierend auf:
 - Aktueller Bedingungen des Problems
 - Informationen auf dem Blackboard
 - Bewährter Lösungsstrategien

Broker-Architektur

- Wird in verteilter Systemen verwendet, um die Kommunikation und Interaktion zwischen Komponenten zu erleichtern
- Zentrale Komponente: Broker (auch Mittelsmann / Vermittler)
 - Vermittelt Nachrichten zwischen Komponenten

- Komponenten agieren nicht direkt miteinander, sondern über den Broker



Zentrale Konzepte

- Broker empfängt Nachrichten und leitet sie an Empfänger weiter
- je nach Implementierung kann der Broker Nachrichten:
 - Transformierer
 - Filtern
 - Priorisieren
- Das Architekturmuster fördert die Entkoppelung von Komponenten
- Vorteile bei der Skalierbarkeit
 - Zum Beispiel wird das Hinzufügen von Komponenten vereinfacht
 - Ohne den Brokerarchitektur muss man im schlechtesten Fall n Schnittstellen implementieren ($n = \text{Anzahl der Komponenten des ursprünglichen Systems}$)
 - In einer Broker-Architektur muss mit einer Schnittstelle programmiert werden

Herausforderungen bei der Implementierung des Brokers

- Skalierbarkeit
- Nachrichtenbearbeitung und Durchsatz
- Sicherheit und Datenschutz
- Kommunikationsprotokolle und Interoperabilität
- Komplexität und Wartbarkeit

Frameworks und Technologien

- Apache Kafka
- RabbitMQ
- Apache ActiveMQ
- Mosquitto (MQTT)
- NATS

Message Broker am Beispiel von RabbitMQ

- Open-Source-Messaging-Broker für die Implementierung von zuverlässigen und skalierbaren Nachrichtenübertragungssystemen
- Wurde erstmals 2007 von der Firma Rabbit Technologies Ltd. entwickelt und später von VMware übernommen
- Wird von der RabbitMQ-Community aktiv weiterentwickelt
- Ist Bestandteil vieler Architekturen wie Microservices, Event-Driven Architekturen und Cloud-Anwendungen
- Basiert Advanced Message Queuing Protocol (AMQP) basiert
 - Industriestandard für die Messaging-Kommunikation
- Unterstützt auch Protokolle wie MQTT, STOMP und HTTP

Elemente von Rabbit MQ

- (P) • Produzent (Producer)
 - Eine Komponente die Nachrichten sendet
 - (C) • Konsument (Consumer)
 - Eine Komponente, die Nachrichten empfängt
- } . Körner auf unterschiedliche Rechner ausgeführt werden (der Regelfall)
- . Komponenten/Anwendung können sowohl Produzent als auch Konsument sein

queue
name

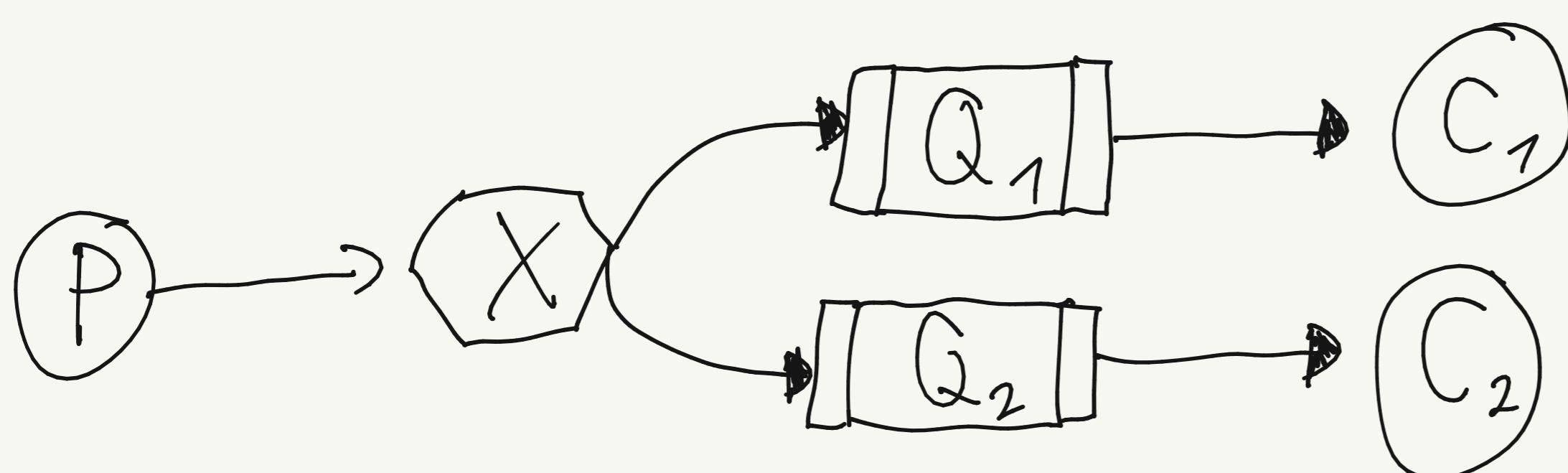
- Warteschlange (Queue)
 - Ein Nachrichtenpuffer, der Nachrichten weiterleitet und speichern kann
- Kanal (Channel)
 - Kommunikationspipeline zwischen einer Anwendung / Komponente und dem Rabbit MQ-Service

Exchange Types

- (X) • Produzenten schicken Nachrichten an einer sogenannten Exchange
- Ein Exchange entscheidet, wie mit eingehender Nachricht weiter verfahren wird
- Unter anderem definiert Rabbit MQ folgende Exchange Types:
 - Fanout Exchange
 - Direct Exchange
 - Topic Exchange

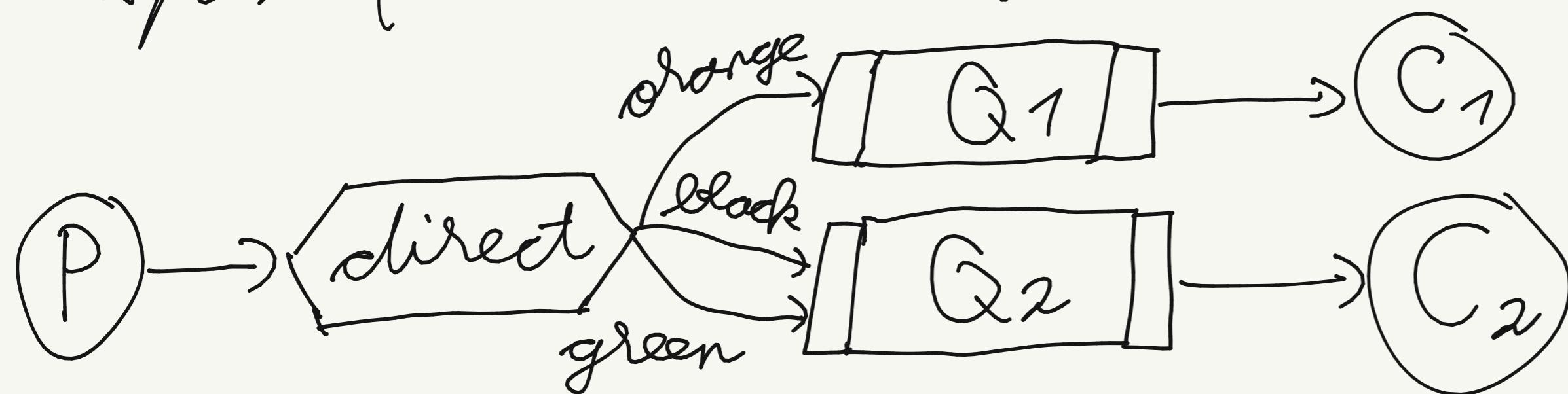
Exchange Type - Fanout Exchange

- Leitet Nachrichten an alle an ihr angeschlossene Warteschlanger weiter
- ermöglicht eine einfache Broadcast- oder Multicast-Verteilung von Nachrichten an alle Warteschlanger, die sich an der Exchange angeschlossen haben
- Nachrichten werden an alle Warteschlange weitergeleitet, die an den Exchange gebunden sind

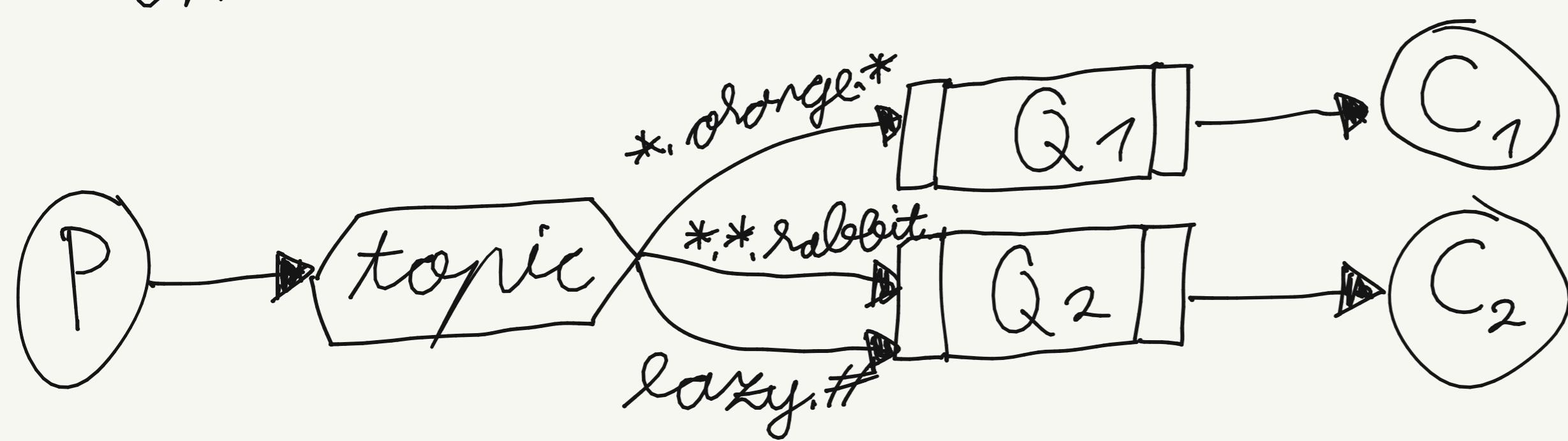


Exchange Type - Direct Exchange

- Leitet Nachrichten basierend auf einem festgelegten Routing-Schlüssel an eine oder mehrere Warteschlanger weiter
- Der Routing-Schlüssel einer Nachricht wird mit einem spezifischer Routing-Schlüssel einer Warteschlange verglichen, und die Nachricht wird an die Warteschlange weitergeleitet, wenn die Schlüssel übereinstimmen
- ermöglicht eine einfache und direkte Zuordnung von Nachrichten zu Warteschlängen, indem ein spezifischer Routing-Schlüssel verwendet wird



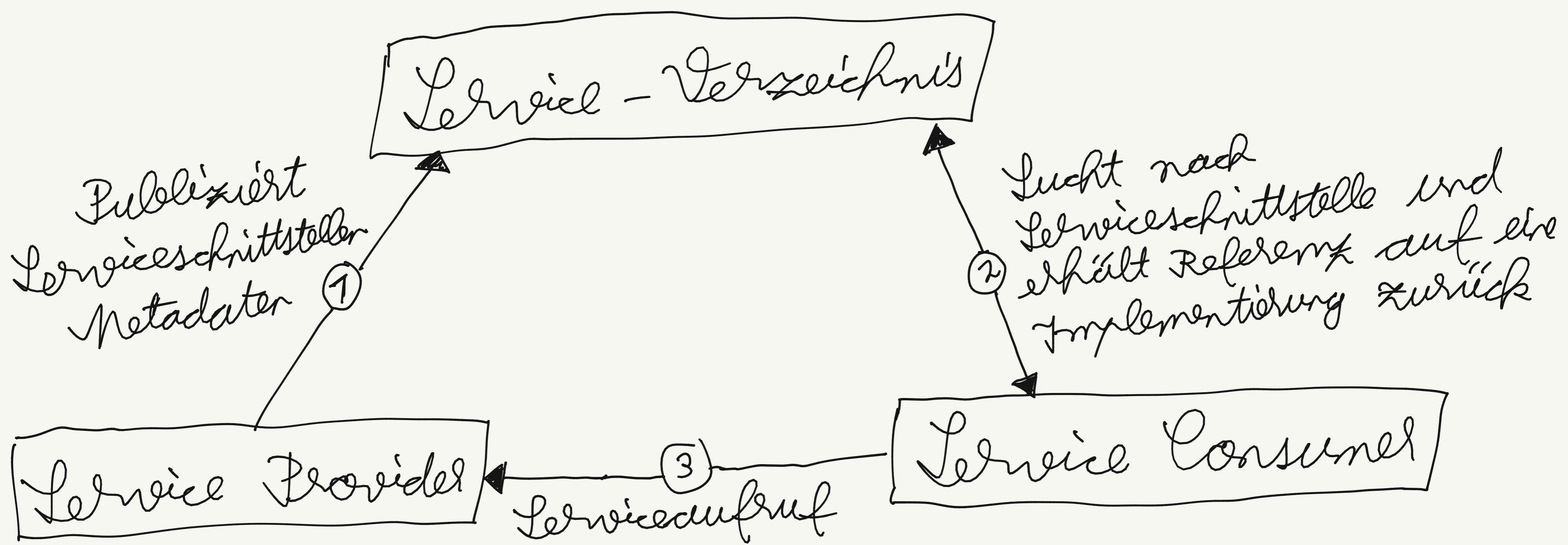
- ## Exchange Type - Topic Exchange
- Leitet Nachrichten basierend auf einem Muster oder Thema an eine oder mehrere Warteschlanger weiter
 - Der Routing-Schlüssel einer Nachricht wird als Thema interpretiert, das aus mehreren Wörtern oder Segmenten besteht, die durch Punkte oder Trennzeichen getrennt sind (z.B.: "weather.usa.newyork")
 - Warteschlanger können mit einem Routing-Muster konfiguriert werden, das dem Thema entspricht, und die Nachricht wird an alle Warteschlanger weitergeleitet, die mit dem entsprechenden Muster übereinstimmen



*. = Platzhalter für genau 1 Wort
= Platzhalter für mehrere Wörter

Serviceorientierte Architektur (SOA)

→ System besteht aus verteilten, wiederverwendbaren, lose gekoppelten und standardisiert zugreifbaren Diensten (Services)



Weitere Eigenschaften:

→ Services sind im Ideallfall Zustandsfrei

→ Idempotent

→ Egal wie oft sie mit dem gleichen Eingabewert aufgerufen werden, sie führen immer zu den gleichen Ergebnissen

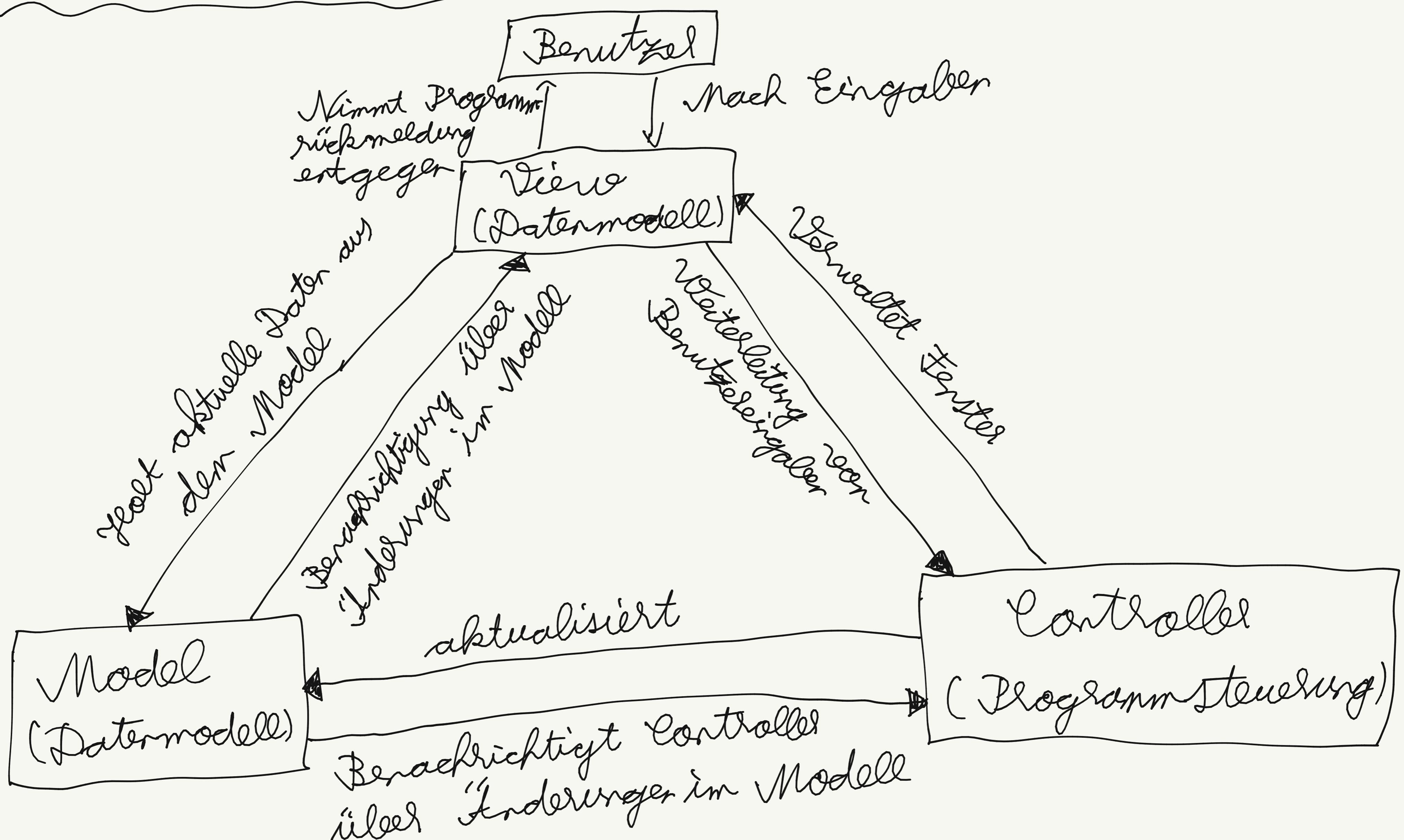
→ Serviceschnittsteller Besitzer Vertragscharakter und leender Servicekonsument oder Serviceanbieter (der Serviceimplementierung)

→ Design-Ley-Contract

→ Serviceanbieter sind austauschbar, sofern die Schnittstellenzusagen eingehalten werden

→ Services sind ortsunabhängig und können jederzeit und vor jedem Ort aus aktiviert werden (Ortstransparenz)

Model - View - Controller



View: nimmt Benutzereingaben entgegen

- Stellt Daten aus dem Modell dar und nimmt Benutzereingaben entgegen
- Meist wird die View über Änderungen im Modell benachrichtigt und holt sich die aktuelle Daten aus dem Modell

Controller:

- Verwaltet die Views und verarbeitet Benutzereingaben
- Zu jeder Präsentation existiert ein Modell
- Entscheidet, welche Daten im Modell geändert werden müssen
- Steuert die Ablöge der anzuziegender Fenster

Modell: darzustellende Daten

- Enthält die darzustellenden Daten
- Ist von Steuerung und Präsentation unabhängig
- Änderungen an relevanten Daten im Modell werden durch das Entwurfsmuster Observer bekannt gemacht

Vorteile:

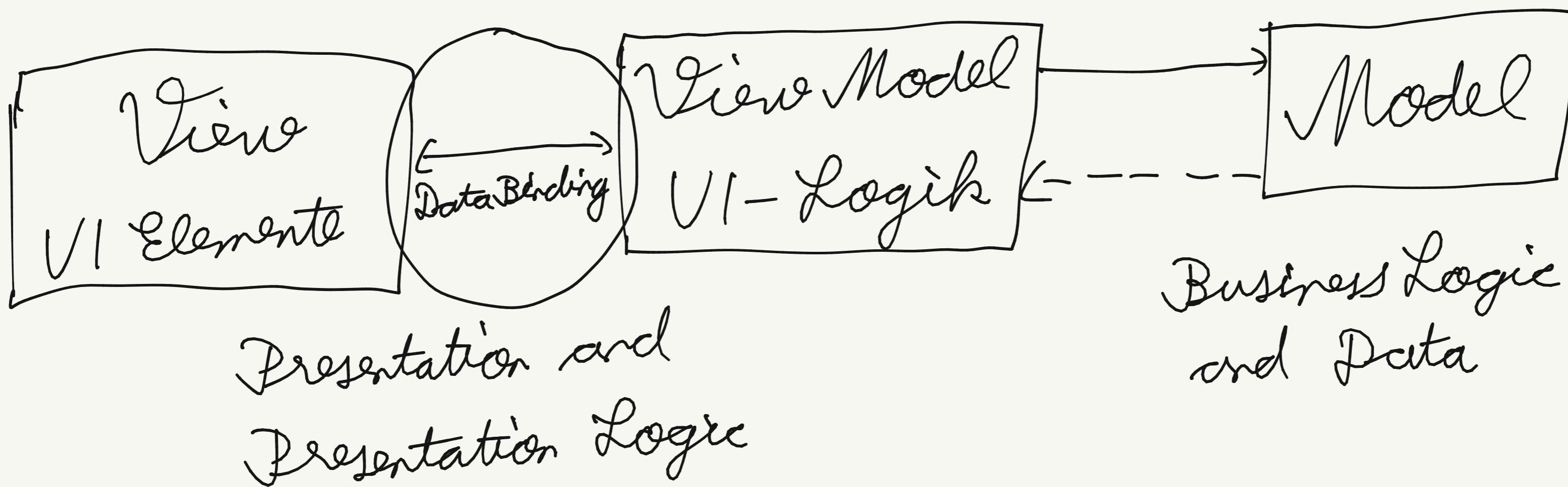
- Trennung von Verantwortlichkeiten
- Wiederverwendbarkeit
- Parallel Entwicklung
- Bessere Testbarkeit

Nachteile:

- Komplexität
- Verständnis und Implementierung
- zusätzlicher Code
- Keine Modularisierung nach fachlichen Gesichtspunkten

Model-View-View Model

- Variante des Model-View-Controller Musters
- Trennung von Darstellung und Logik der Benutzerschnittstelle
- Findet Verwendung in UI-Plattformen wie → Cocoa, Windows Presentation Framework (WPF), JavaFX, ...
- Erlaubt eine Rollentrennung von UI-Designern und Entwicklern



Data Binding

→ Technologiespezifisch, da für das Binding eine entsprechende Infrastruktur benötigt wird

View Model:

- Stellt dem View öffentlich Eigenschaften und Befehle zur Verfügung
- Diese werden von der View an Steuerelemente "geleert"

- Datenbindung ist Schlüsselkonzept
- Einweg-Bindung (One-Way Binding): Daten fließen nur von View Model zum View
- Zweienweg-Bindung (Two-Way Binding): Daten fließen sowohl von View Model zum View als auch umgekehrt
- Einweg-zu-Quelle Bindung (One-Way to Source Binding): Daten fließen nur von der View zum View Model

Vorteile:

- Klare Trennung von Verantwortlichkeiten
 - Trennung UI-Design klassischer Quelle Code
- Verbesserte Testbarkeit
 - Anwendung kann unabhängig von der View getestet werden

Event Sourcing

- Ein Architekturmuster zur Implementierung der Speicherung von Zustandsänderungen in einem System
- Nicht der aktuelle Zustand eines Objektes wird gespeichert, sondern alle Änderungen, die an diesem Objekt auftreten (Ereignisse)
- Der aktuelle Zustand wird durch die sequentielle Wiedergabe der Ereignisse rekonstruiert

Grundprinzipien:

- Ereignisse als Datenquelle
- Event-Store
- Zustandskonstruktion
- Unveränderlichkeit der Daten
- Historische Daten und Zeitreihen

Umsetzungsmöglichkeiten eines Event Stores

→ Relationale Datenbanken

→ NoSQL-Datenbanken

→ Spezielle Event-Store Frameworks, wie z.B.
· Event Store DB

· Rick

→ Cloud-basierte Speicherlösungen

→ Eigenentwickelte Lösungen

Nachteile und Herausforderungen

- Komplexität
 - Event-Logs, Event-Verarbeitung und Zustands-Wiederherstellung
- Skalierbarkeit
 - Große Mengen von Ereignissen kann Herausforderung für Skalierbarkeit werden
- Wartung und Datenbereinigung
 - Verwaltung, Wartung und ggf. Bereinigung großer Mengen von Event-Daten kann anspruchsvoll sein
- Leseröhleme
 - Abfragen des aktuellen Zustands kann langsamer sein als der direkte Zugriff auf einen klassischen Datenbankzustand
- Rückwirkende Änderungen
 - Hinzufügen oder Ändern von Events in der Vergangenheit erfordert viel Vorsicht, da dies Auswirkungen auf den gesamten Systemzustand haben kann
 - Sollte in einer Event-Sourcing-Architektur vermieden werden

Einsatzfelder:

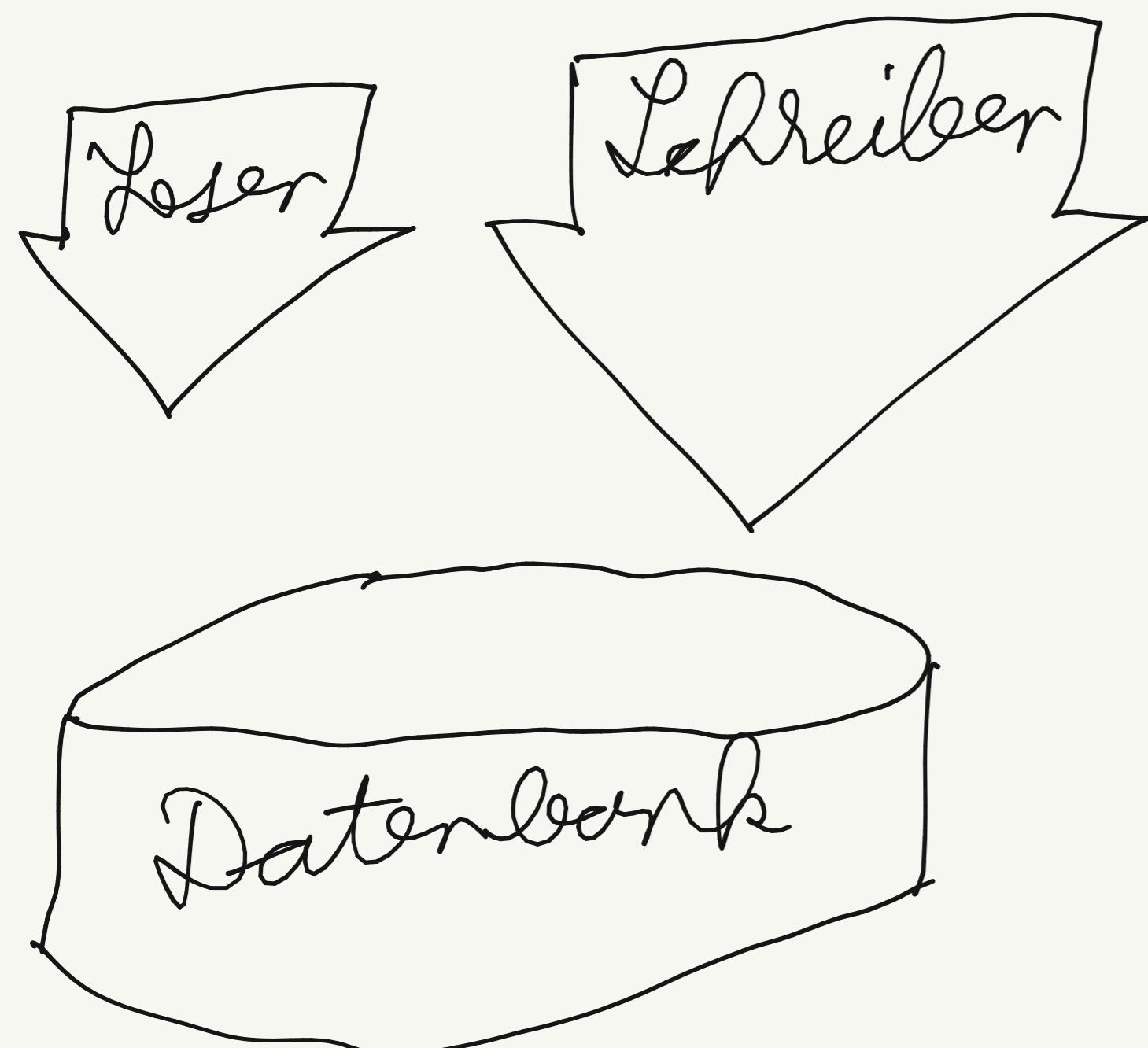
→ Anwendung insbesonders in Bereichen, in denen eine nachvollziehbare Historie von Datenänderungen erforderlich ist

- Finanz- und Bankwesen
- E-Commerce
- Gesundheitswesen
- Logistik und Lieferkettenmanagement
- IoT (Internet of Things)
- Soziale Netzwerke und Community-Plattformen

CQRS -

Command Query Responsibility Segregation

Anwendung



- Grundgedanke
→ Verantwortlichkeiten für Lese und Schreiboperationen in einer Anwendung zu trennen
- Vorteile
 - Optimierte Daten-Schemata für jede Seite
 - einfachere Abfrager
 - Unabhängige Skalierung
- Herausforderung
 - Synchronisation zwischen Schreib- und Lesedatenbank

Kombination mit Event Sourcing

- CQRS und Event Sourcing werden häufig in Kombination verwendet
- Die Schreib-optimierte Datenbank wird hierbei mittels eines Event-Stores realisiert
→ Single source of truth
- Die Lese-optimierte Datenbank realisiert materialisierte Views, häufig in Kombination mit denormalisierte Datenbanken
- Nur so die materialisierten Views verändert werden, kann der Zustand wiederhergestellt werden durch ein erneutes „Abspielen“ aller Events

Wann macht der Einsatz von CQRS Sinn?

Gründe für der Einsatz von CQRS laut Microsoft

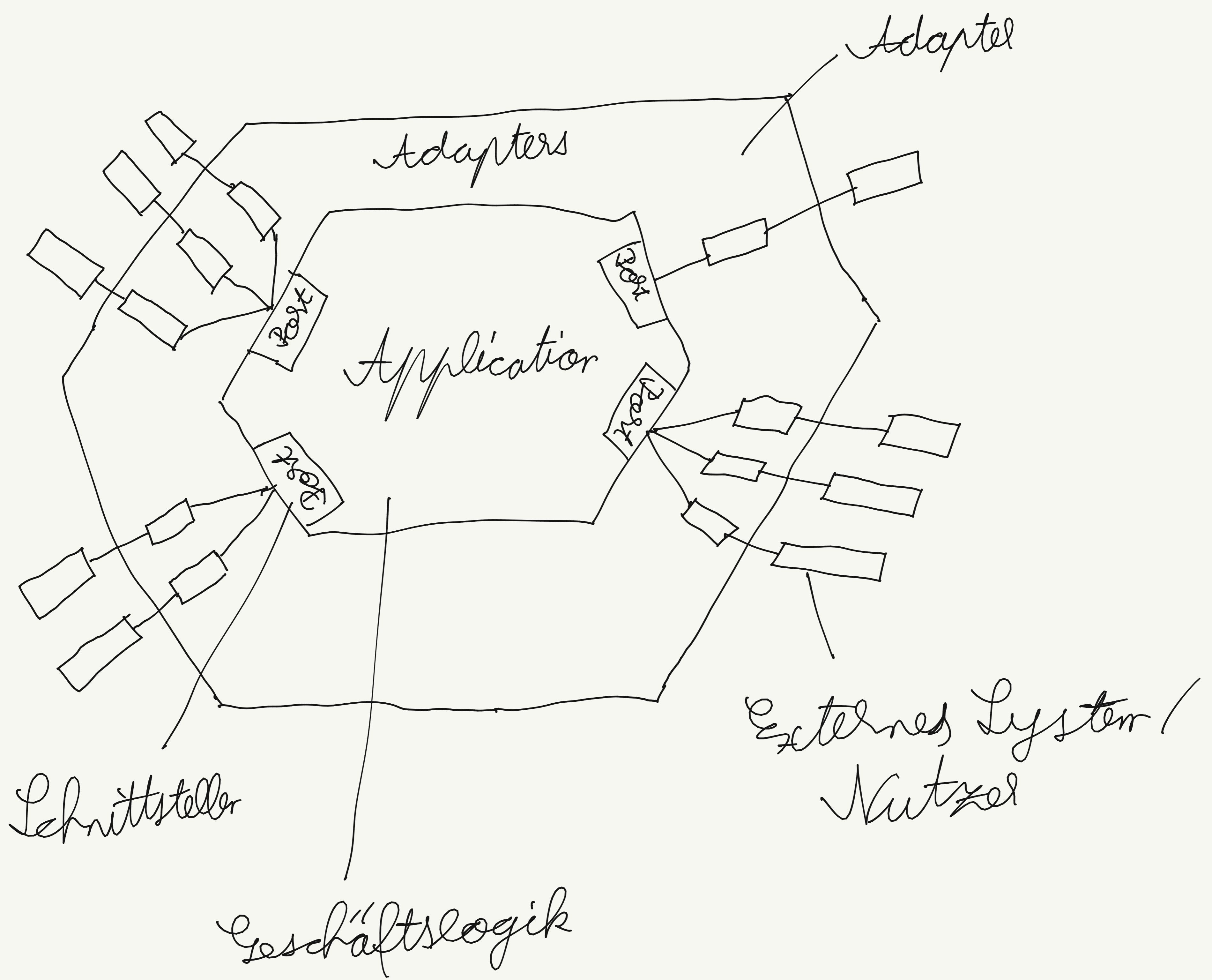
- Sehr großer paralleler Zugriff auf Daten
- Szenarien, in denen ein Team sich dediziert an die Reservate und ein Team sich dediziert an die Schreibseite kümmert
- Szenarien, in denen Geschäftsregeln und das Domänenmodell komplex sind und sich häufig ändern
- Wenn die Anzahl der Lesezugriffe um ein Vielfaches höher ist als die Anzahl der Schreibzugriffe

Das Architekturmuster wird (von Microsoft)

explizit nicht empfohlen, wenn

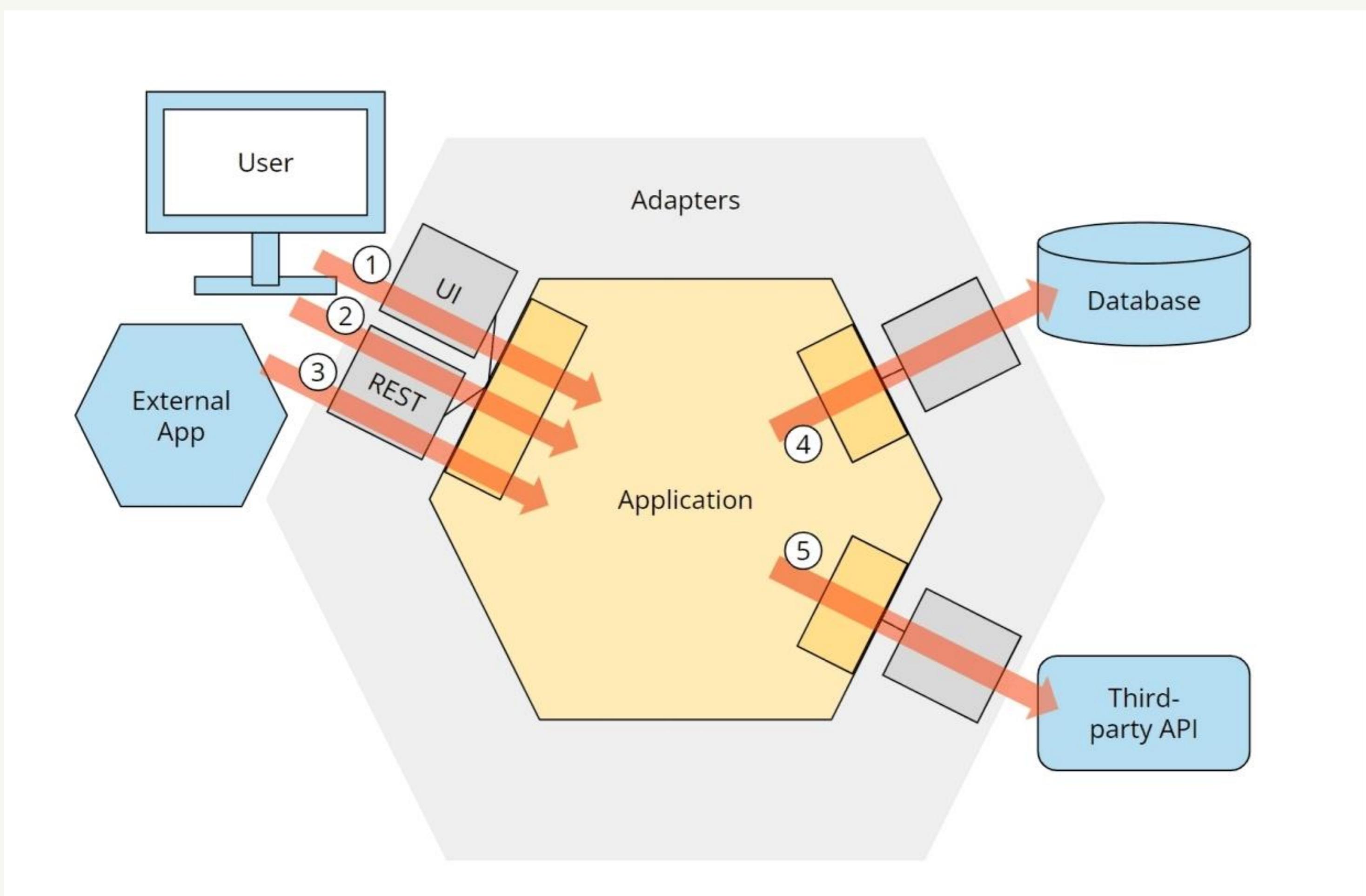
- Die Domäne und die Geschäftsregeln einfach sind
- Wenn einfache CRUD-Operationen eine gute Lösung darstellen

Hexagonale Architektur



Ziele:

- Anwendung soll gleichmäßer von Benutzern, anderen Anwendungen oder automatisierte Tests gesteuert werden können
- Geschäftslogik soll isoliert vor der Datenbank, vor sonstiger Infrastruktur und vor Drittsystemen entwickelt und getestet werden können
- Modularisierung der Infrastruktur soll ohne Anpassungen an die Geschäftslogik möglich sein



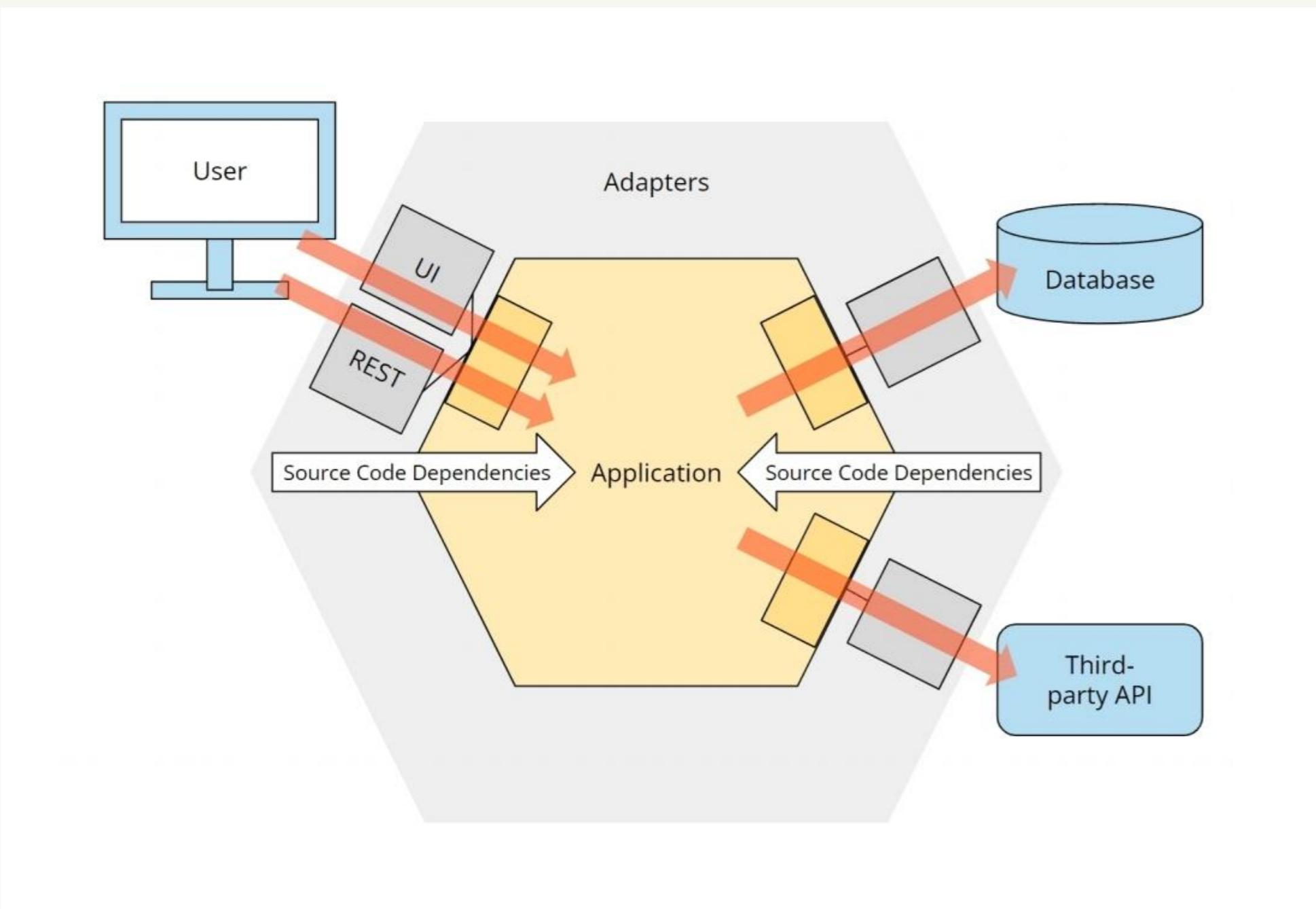
Die Abbildung zeigt eine Beispielhafte Anwendung, die:

1. durch einen User über ein User Interface gesteuert wird
2. durch einen User über ein Rest-API gesteuert wird
3. durch eine externe Anwendung über dieselbe Rest-API gesteuert wird
4. eine Datenbank angesteuert wird
5. eine externe Anwendung gesteuert wird

Der Erfinder der hexagonalen Architektur auf die Frage, ob das Hexagon oder die Zahl „sechs“ eine bestimmte Bedeutung habe:

„Nein“. Er wollte eine Form verwenden, die noch keine verwendet hat. Vierecke werden überall verwendet, und Fünfecke sind schwer zu zeichnen. Also wurde es ein Sechseck.

Abhängigkeiten



Quellcode-Abhängigkeiten sind nur von außen nach innen erlaubt, nicht umgekehrt

Der Erfinder (Alistair Cockburn) auf die Frage: „What do you see inside the Application?“

„I don't care - not my business“

„Wrap your app in an API and put tests around it“.

Microservices

Ursprüngliche Definitionen:

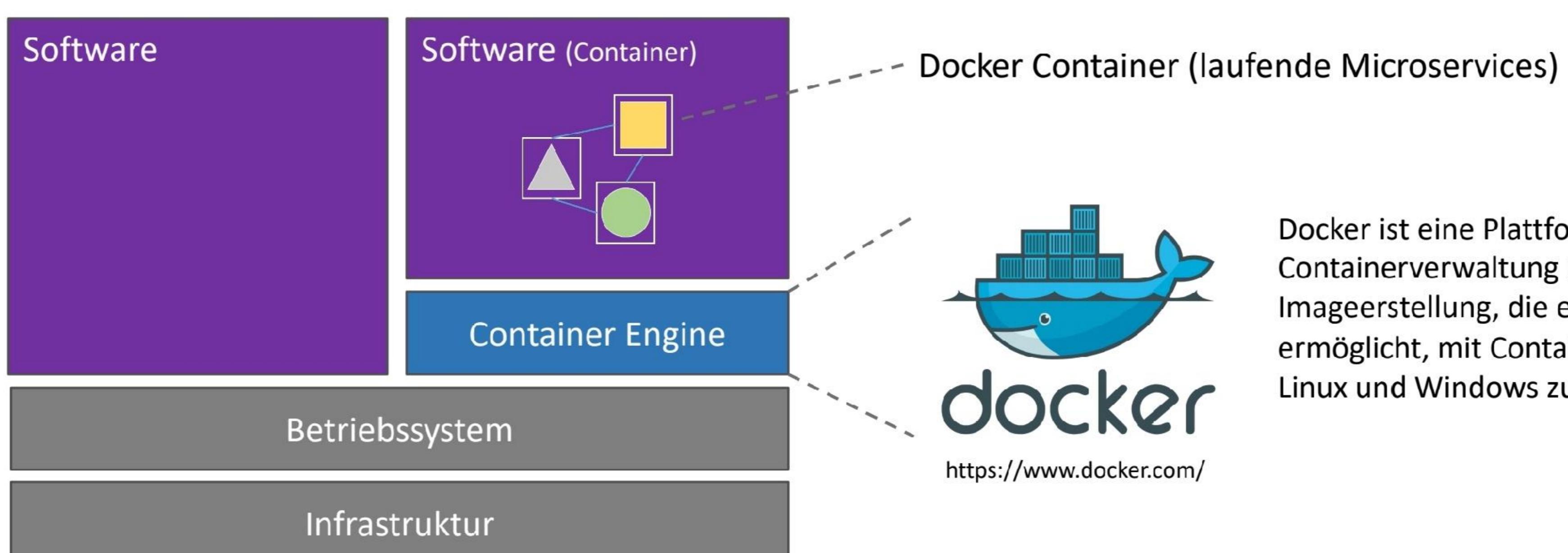
Microservices sind ein Architekturmuster der Informationstechnik, bei dem komplexe Anwendungssoftware aus unabhängiger Prozesser komponiert wird, die untereinander mit sprachunabhängigen Programmierschnittstellen kommunizieren. Die Dienste sind weitgehend entkoppelt und erledigen eine kleine Aufgabe. So ermöglichen sie einen modularen Aufbau von Anwendungssoftware.

In short, the Microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralised management of centralised management of these services, which may be written in different programming languages use different data storage technologies.

Microservices sind keine Technologie, sondern ein Architekturmuster

- Merkmale zusammengefasst
- Unabhängigkeit einzelner Microservices
 - Unabhängige Prozesse
 - Sprachunabhängige Programmierschnittstellen
 - Unabhängige Release-Zyklus
 - Modularer Aufbau
 - Kontinuierliche Bereitstellung und Deployment
 - Fehlerisolation
 - Möglichkeit der Nutzung
 - unterschiedlicher Programmiersprachen je Microservice
 - unterschiedliche Datenhaltungs-Technologien

Container



Docker

Container

→ Die aktive Instanz eines Images. Der Container wird also gerade ausgeführt und ist beschäftigt. Sobald der Container sein Programm ausführt oder mit seinem Auftrag fertig ist, wird der Container automatisch beendet.

Image

→ Eine schreibgeschützte Vorlage, in der ein Container definiert wird. Enthält den Code, der ausgeführt wird, einschließlich aller Definitionen für Bibliotheken und Abhängigkeiten, die der Code benötigt. Aus einem Image können mehrere Container instanziert werden.

Dockerfile

→ Eine Textdatei, welche mit verschiedenen Befehlen ein Image beschreibt. Diese werden bei der Ausführung abgearbeitet. Ein Dockerfile beschreibt einen Container so, dass er auf einem anderen Rechner wieder genau gleich aufgebaut werden kann. Er automatisiert die Erstellung von Images.

Registry

→ Dient der Verwaltung von Images.
Beispiel: Docker Hub

Docker und Kubernetes

Kubernetes ist eine Open-Source-Software zur Unterstützung der Bereitstellung, Skalierung, Verteilung und Verwaltung von Container-Anwendungen. Es unterstützt verschiedene Container-Engines, insbesondere Docker.

Vereinfacht zusammengefasst:

- Ein Kubernetes Cluster besteht aus einem Netzwerk von Nodes:

- Master Node(s)

- Worker Node(s)

- Hier werden die Container ausgeführt
(gekapselt durch sogenannte Pods)

- kubectl

- Tool zur Steuerung des Kubernetes-Cluster Manager

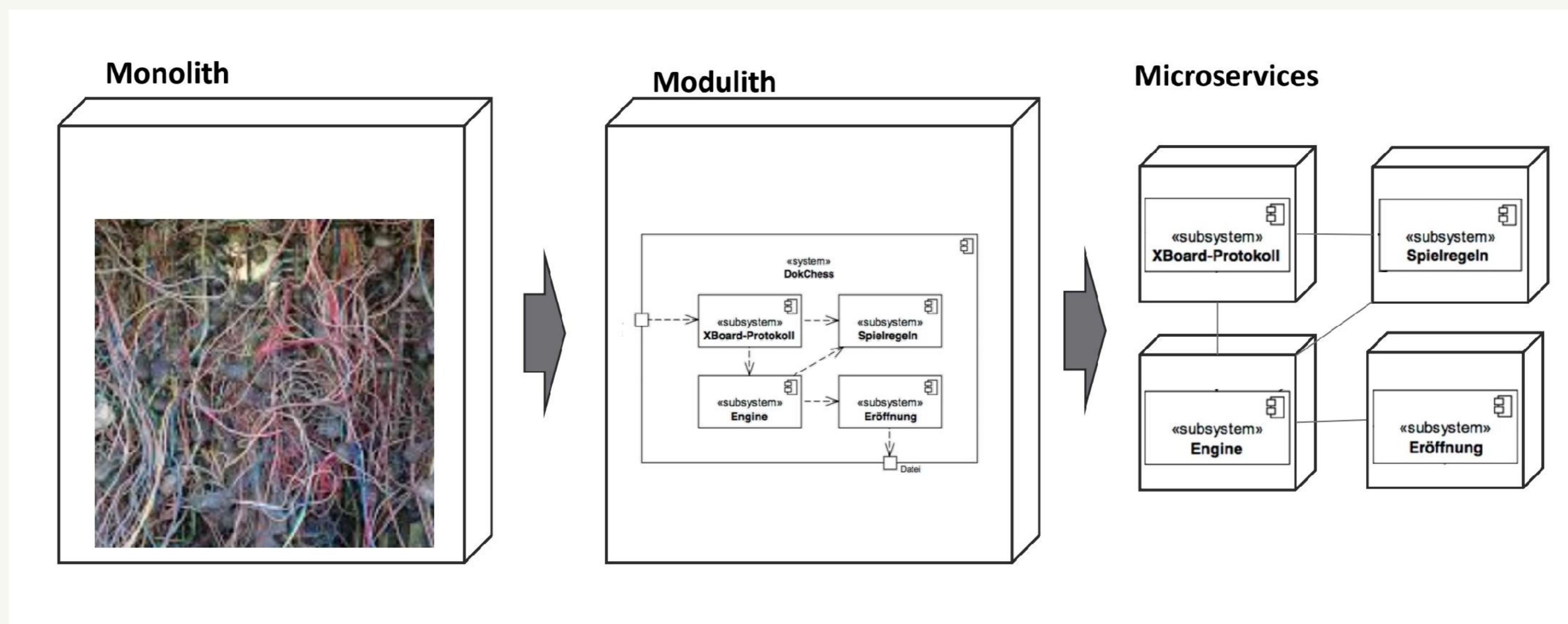
- Hauptzugang zur Konfiguration eines Kubernetes Clusters

Die Installation und der sichere und zuverlässige Betrieb Docker/Kubernetes-Clusters erfordert ein hohes Maß an Expertise und Aufwand.

Kritik an Microservice-Architektur

- Komplexität der Verwaltung
- Inter-Service-Kommunikation
- Entwicklungs- und Deployment-Komplexität
- Sicherheitsbedenken
- Fachwissen und Kultur
- Das „Micro“ in Microservices suggeriert, dass kleine Services anzustreben sind
 - Viele kleine Services erzeugen zusätzliche und ggf. unnötige Komplexität
 - Nicht klar, ob wann ein Service als „klein“ zu bewerten ist
 - Trennung rein nach Anzahl der Codezeilen wenig sinnvoll

Monolith → Modulith → Microservices



Häufig ist ein Modulith einer Microservice-Architektur vorzuziehen

Modulith vs. Microservice

Vorteile eines Modulith gegenüber einem Microservice

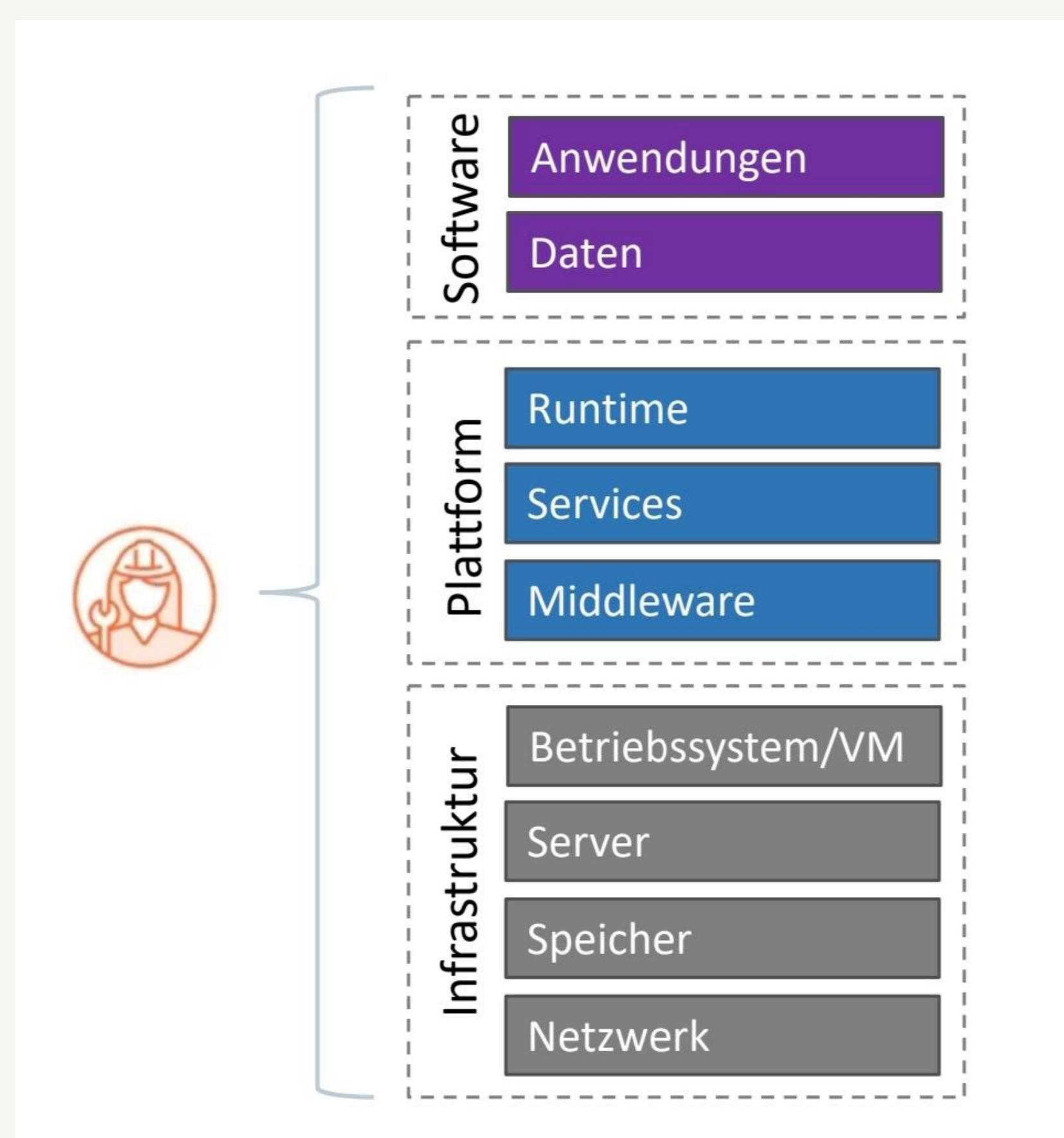
- Geringere Komplexität
- einfache Entwicklungsumgebung
- geringer operationaler Overhead
- Kosteneffizienz
- Einfachere Datenverwaltung

Wann von einem Modulith zu einer Microservice-Architektur wechseln?

- Skalierbarkeit
- unabhängige Entwicklung und Bereitstellung
- Technologische Vielfalt
- Geschäftliche Anforderungen
- Zuverlässigkeit und Fehlerisolation
(siehe auch: Alter von Kopplung - gemeinsam genutzte Infrastruktur)

Cloud - Leistungsmodelle

On-Premises (keine Cloud)



Die Verantwortung für

- der Aufbau und der Betrieb der Infrastruktur
- die Bereitstellung der Entwicklungsumgebung, Komponenten, Laufzeitumgebung
- das管理 der Daten und Anwendung

liegt im eigenen Unternehmen

Infrastruktur as a Service (IaaS)

Infrastruktur as a Service bedeutet:

- Verantwortung für die Infrastruktur wird an der Cloud-Anbieter übertragen
- Wartung und Management
- Betrieb der Ressourcen

Vorteile:

- Sparen von Zeit und Geld in der Infrastruktur
- Flexibilität – eine weitere Infrastruktur kann bei Bedarf unmittelbar angeboten werden

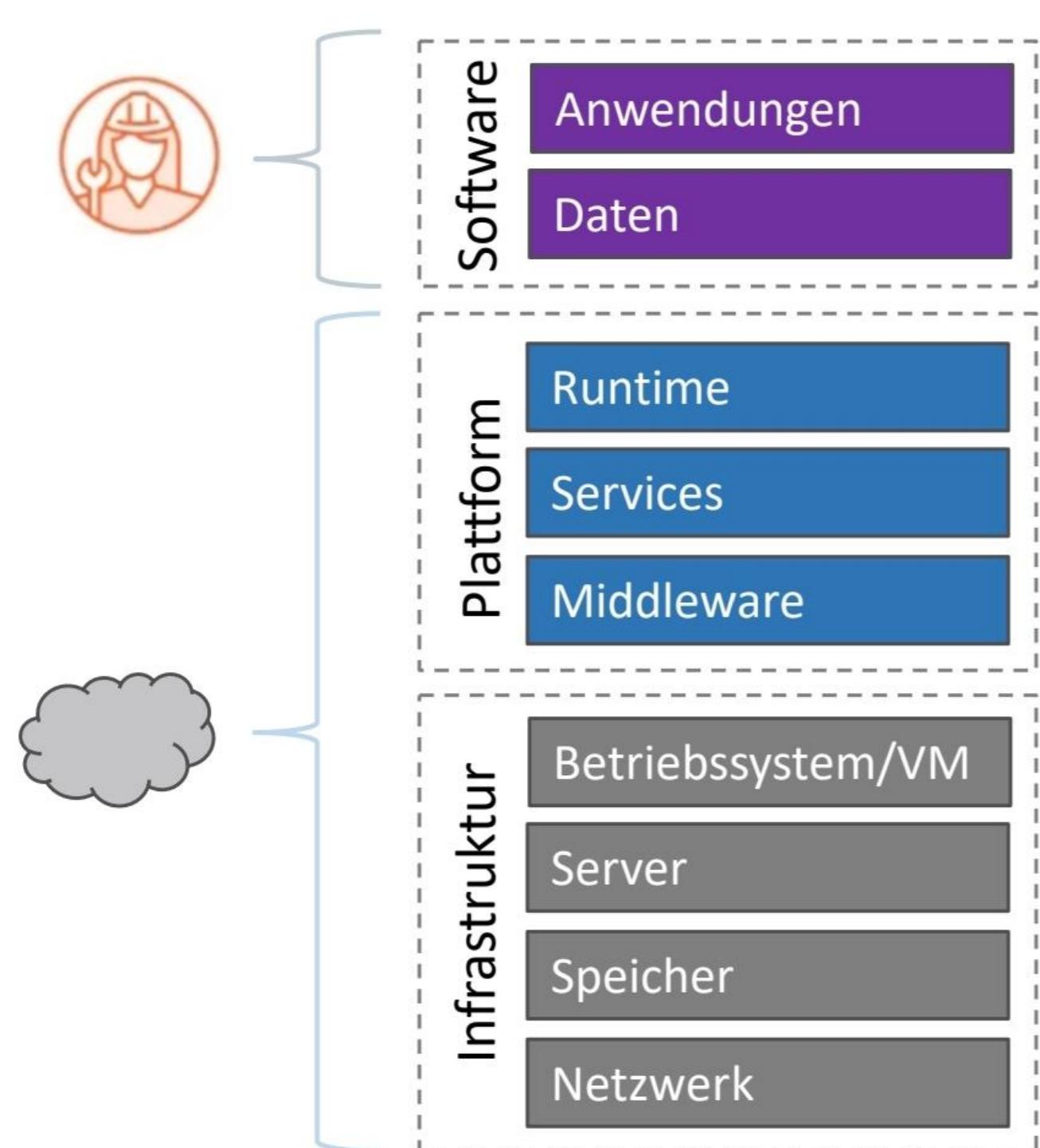
vor Lastspitzen

bestehender Microservice – Umsetzung in der Cloud

→ automatisches Abfangen

→ einfache Migration
Umsetzung in der Cloud

Platform as a Service (PaaS)



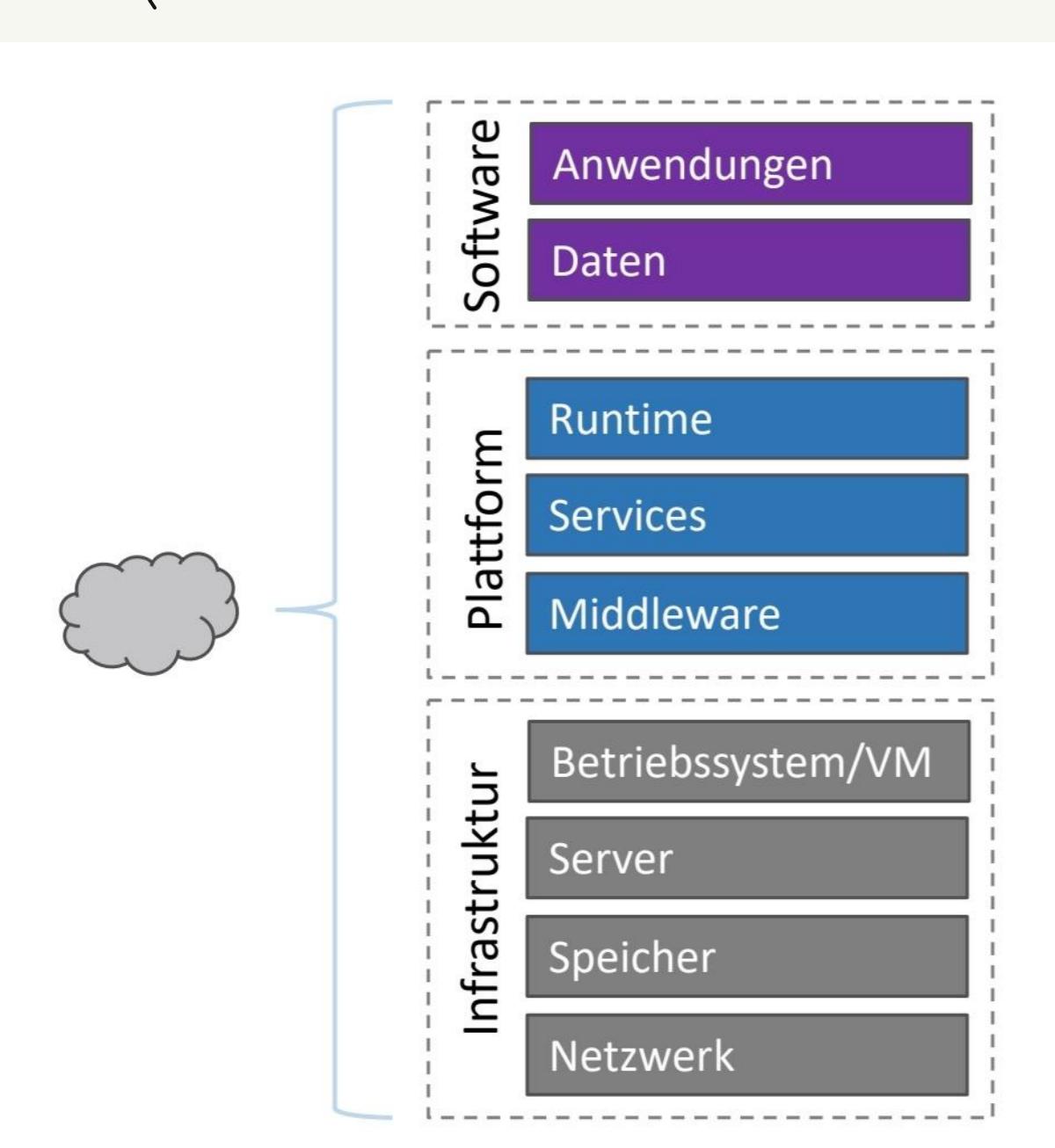
bedeutet:

- Entwicklungswerkzeuge werden durch Cloud-Anbieter zur Verfügung gestellt
- Nutzung von Softwarekomponenten, welche durch Cloud-Anbieter bereitgestellt und betrieben werden
- Nutzung Cloud-native Microservice-Frameworks möglich

Vorteile:

- Aufeinander abgestimmte Tools und Entwicklungsumgebungen
- Flexibilität - weitere Services können bei Bedarf unmittelbar angemietet werden
- kann mit IaaS kombiniert werden
- Fokus auf Auslieferung von Funktionalität an Kunden

Software as a Service (SaaS)



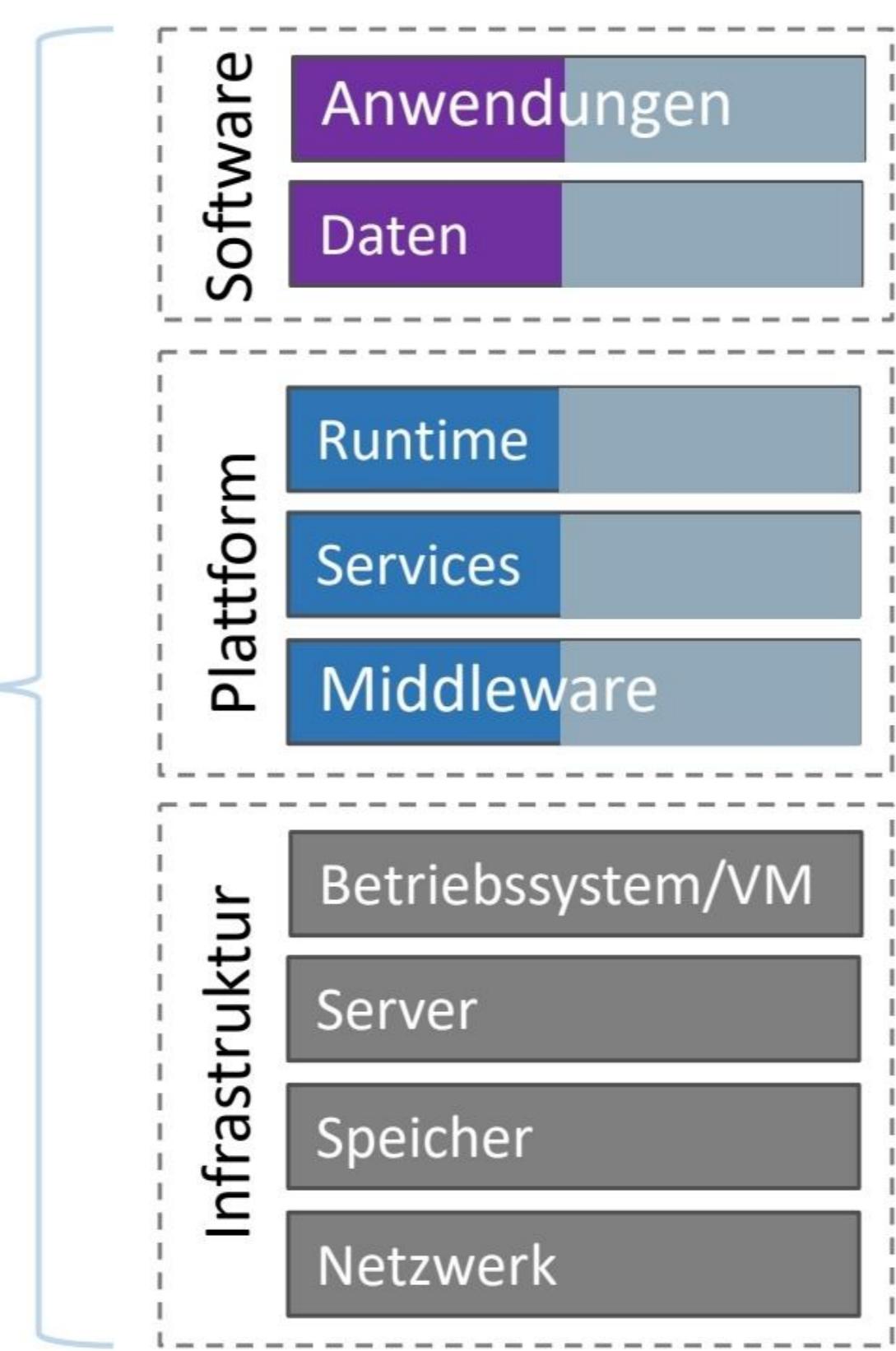
bedeutet:

- Del Cloud stellt ganze Anwendungen bereit
- Del Softwareentwicklung und -betrieb liegt in der Verantwortung des Anbieters
- Software kann bei Bedarf unmittelbar angemietet werden

Vorteile:

- Aufeinander abgestimmte Tools und Entwicklungsumgebungen
- Flexibilität - eine weitere Services können bei Bedarf unmittelbar angemietet werden - Eine auf PaaS und IaaS basierende Software kann SaaS-Anwendungen integrieren
- kann mit IaaS und PaaS kombiniert werden - Eine auf PaaS und Microservices basierende Software kann SaaS-Anwendungen integrieren
- Fokus und Auslieferung von Funktionalität an Kunden

Hybride Ansätze



vorgefertigter Machine in der Cloud

- Betriebe von Microservices (z.B. in Form von Docker-Containern) im eigenen Unternehmen und der Anbindung an Cloud-native Microservice-Angebote

Beispiele:

- Integration von Legacy-Systemen bei gleichzeitiger Nutzung von Standard-Softwarekomponenten aus der Cloud
- Bei Nutzung eigener Infrastruktur für die Datenbehandlung bei gleichzeitiger Nutzung von Learning Algorithmen in

Serverless Computing

- Methoden / Funktionen liegen in der Cloud und werden dort ausgeführt
- Jede Funktion enthält Code und Konfigurationen (Funktionsname, Ressourcenanforderungen)
- Körner beim Aufrufer bestimmt Ereignisse automatisch zur Ausführung gebracht werden
- Infrastruktur wird durch Cloud-Anbieter bereitgestellt
 - Code-Überwachung und Protokollierung
 - Server und Betriebssystemwartung
 - Skalierung / Kapazitätsbereitstellung
 - Sicherheitspatches

Es gibt zahlreiche Anbieter von Serverless Computing wie z.B.: Azure Functions AWS Lambda

→ auch beim sogenannter "Serverless Computing"
gibt es ein Server!

Hauptmerkmale

- ereignisgesteuerte Ausführung
- Automatische Skalierung
- Abrechnung basierend auf Nutzung
- Kurze Lebensdauer der Ausführungsumgebung

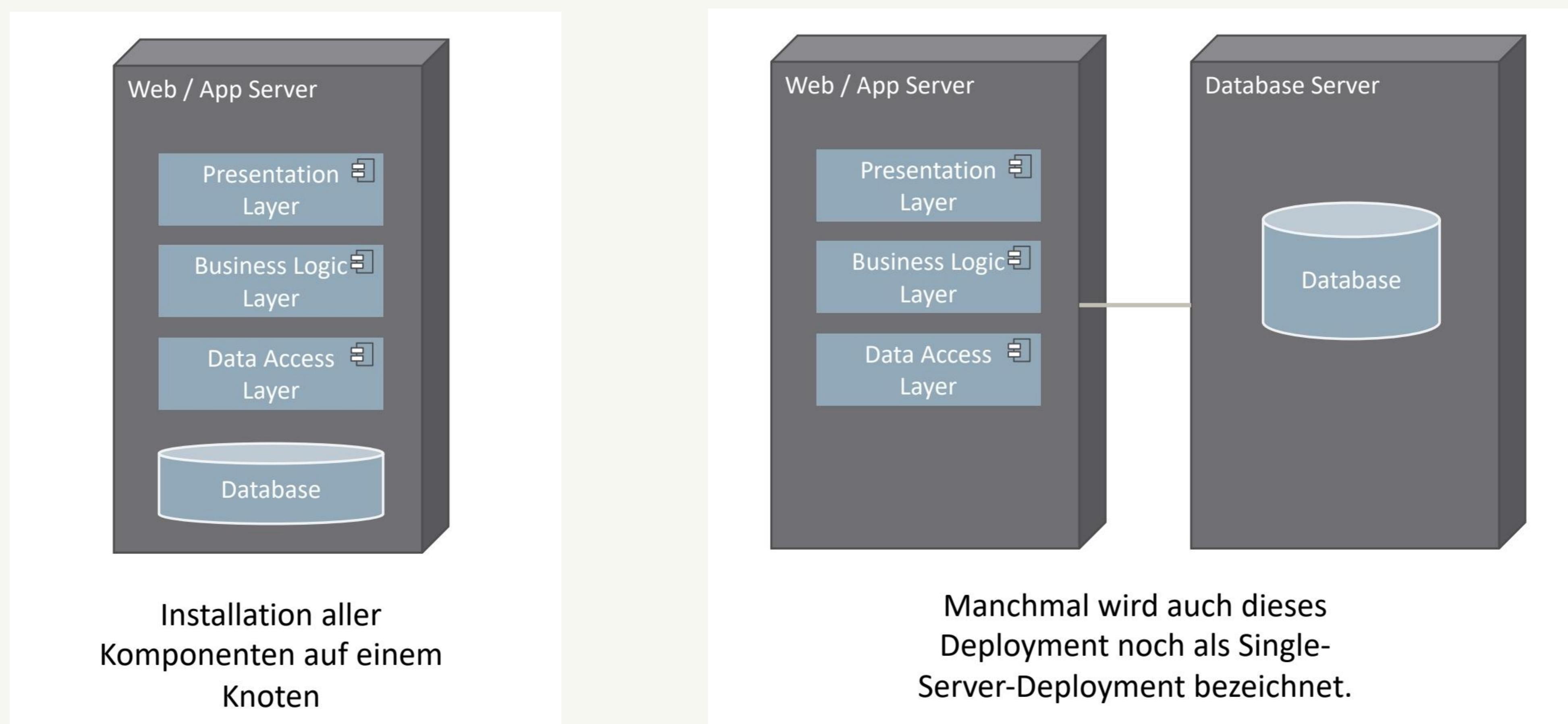
Nachteile

- Kaltstart - Latenz
- State - Management
- Vendor-Lock-in

Deploymentmuster

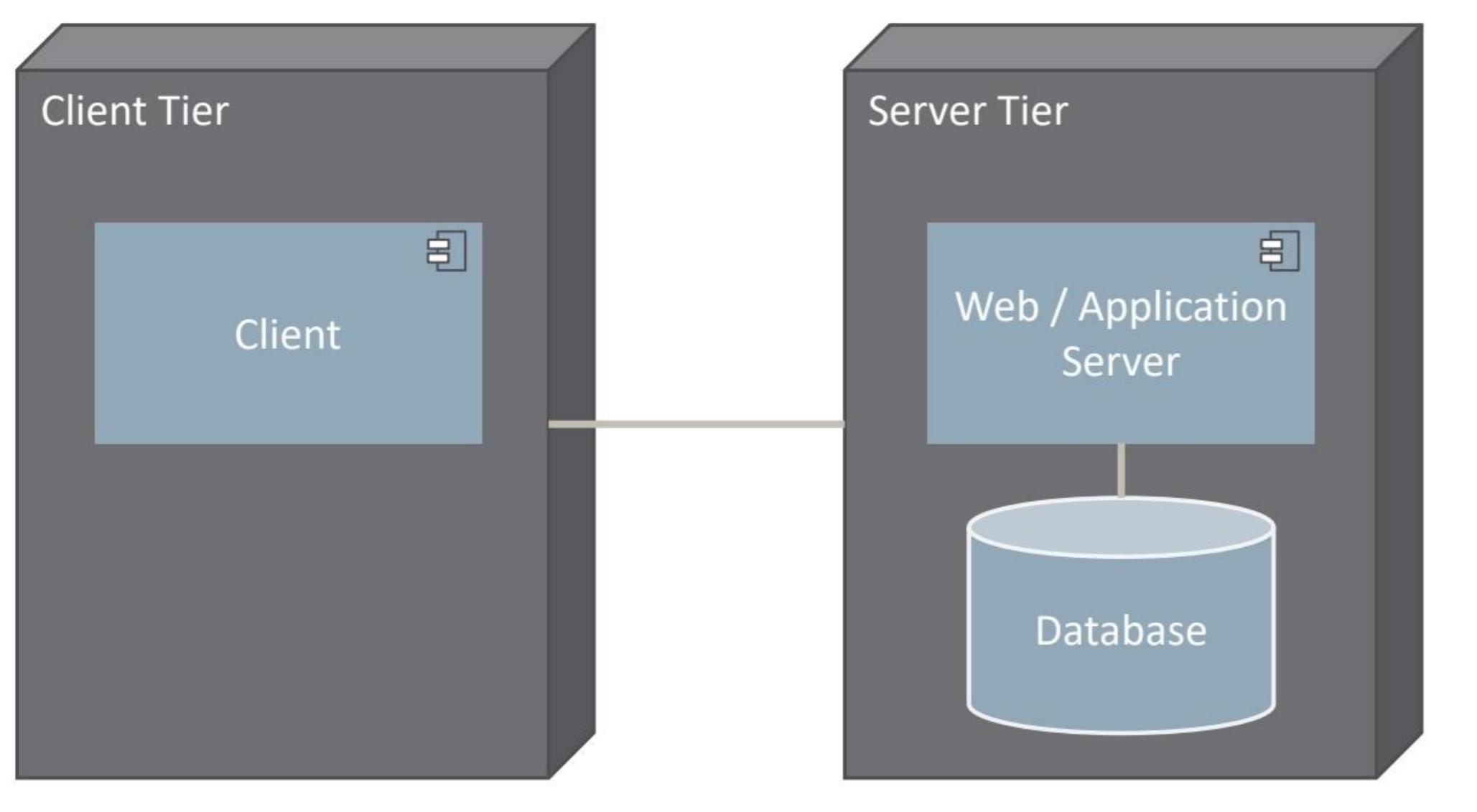
- Benötigtes Vorgehen, um Software von der Entwicklungsumgebung in die Produktionsumgebung zu übertragen und dort zu installieren oder zu aktivieren
- Es können verschiedene Aspekte im Bereitstellungsprozess abgedeckt werden, wie Infrastruktur - Provisionierung
 - Bereitstellungsstrategien
 - Skalierbarkeit und Verfügbarkeit
 - Versionierung und Rollback
 - Überwachung und Logging

Single - Level Deployment



Multi-Tier Deployment

• 2 - Tier - Deployment



• 3 - Tier - Deployment

