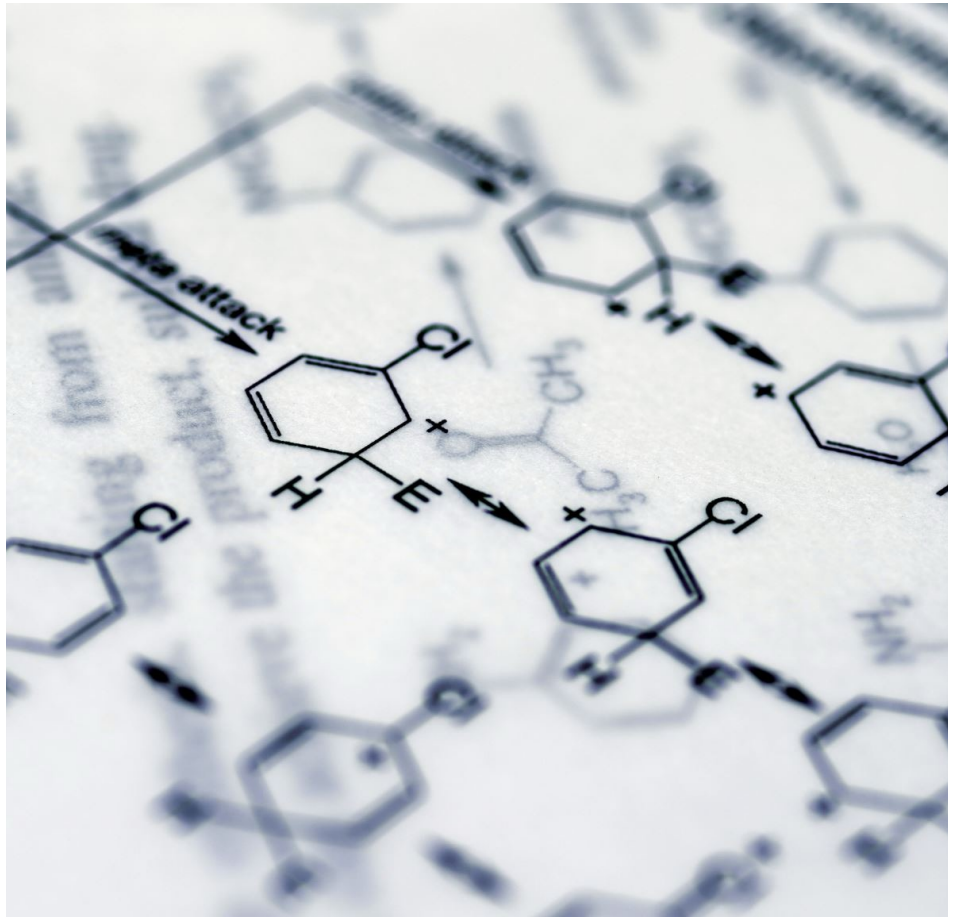
Themenbereiche der Vorlesung



Themenbereiche

A – Grundlegende Konzepte

B – Dokumentation und Kommunikation

C – Entwurf von Softwarearchitekturen

D – Architekturmuster

E – Qualität von Softwarearchitekturen

Architekturmuster

Themenbereich D

Inhalte

- Schichtenarchitektur
- Pipes & Filters
- Blackboard-Architektur
- Broker-Architektur
- Service-Oriented Architecture (SOA)
- MVC & MVVM
- Event Sourcing
- CQRS
- Hexagonale Architektur
- Microservices
- Cloud-Servicemodelle
- Serverless Computing
- Deploymentmuster

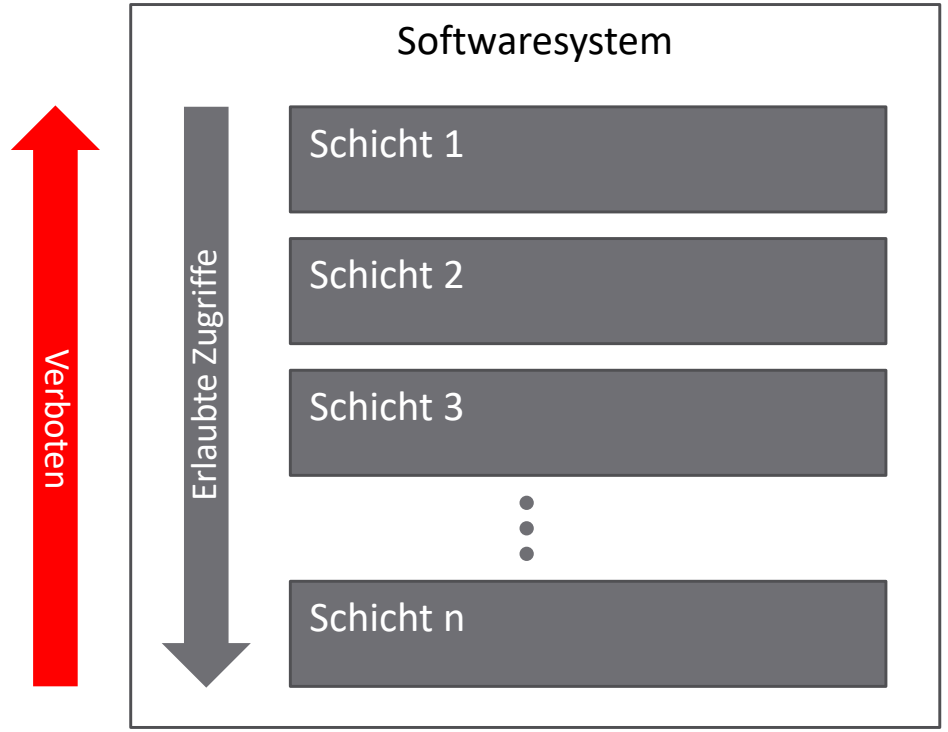
Architekturmuster

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

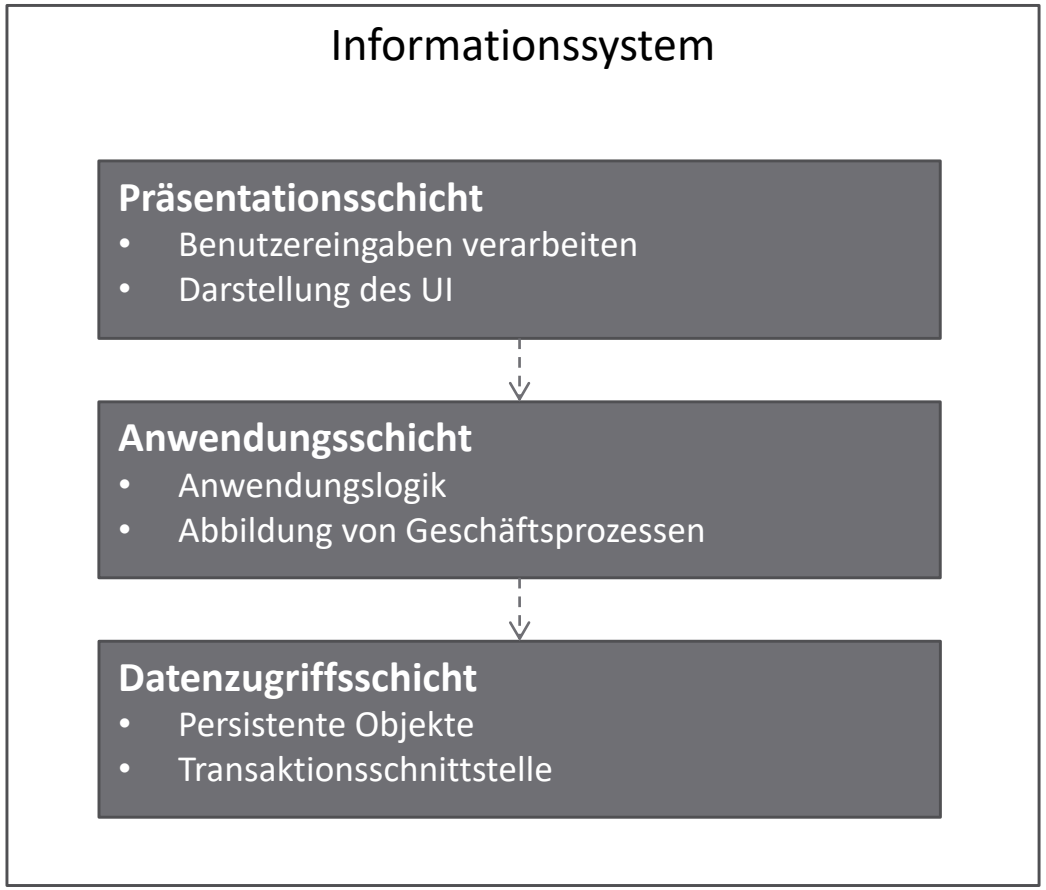
Schichtenarchitektur

Schichtenarchitektur

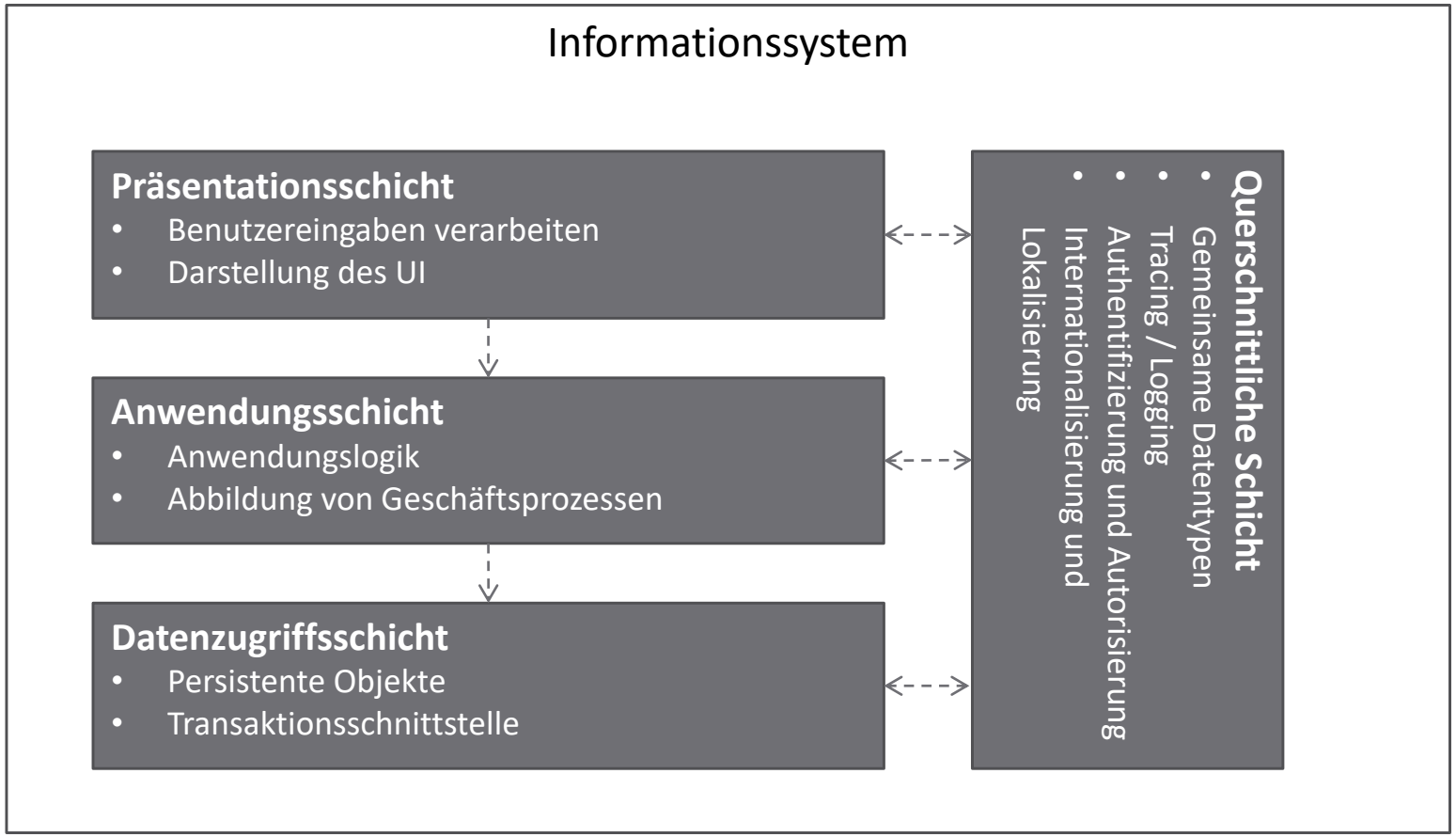
- Das Softwaresystem wird in Schichten strukturiert. Einzelne Aspekte (z.B. Klassen oder Komponenten) eines Softwaresystems werden jeweils einer Schicht zugeordnet.
- Aspekte einer höheren Schicht dürfen ausschließlich Aspekte der gleichen oder einer niedrigeren Schicht verwenden.



3-Schichten-Architektur

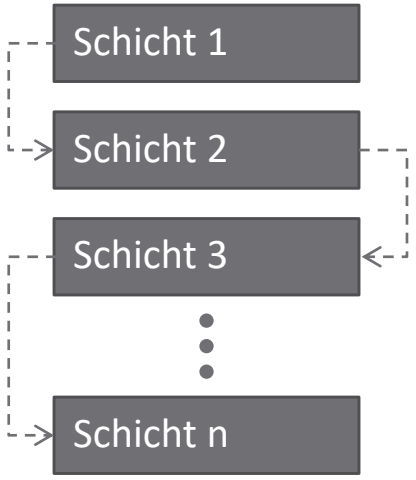


Querschnittliche Schicht



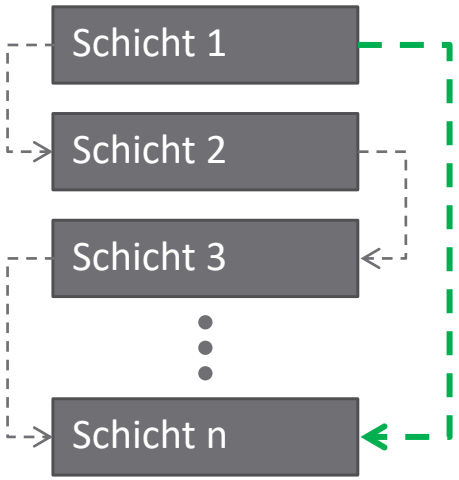
Strikte vs. Nicht-strikte Schichtenarchitektur

strikt



Zugriff ausschließlich auf die nächst-tiefere Schicht erlaubt.

nicht-strikt



Zugriff auf beliebige untere Schichten erlaubt.

- 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

- Paketstruktur
- Komponenten
- Namenskonventionen
- Frameworks und Bibliotheken
- In Kombination mit anderen Architekturmustern

Beispielimplementierung in C#

Bücheranwendung mittels Schichtenarchitektur

Beispielimplementierung in C# Presentation Layer

```
public class CPresentationLayer {

    3 Verweise
    private IApplicationLayer applicationLayer;

    1 Verweis
    public CPresentationLayer(IApplicationLayer applicationLayer)
    {
        this.applicationLayer = applicationLayer;
    }

    1 Verweis
    public void DisplayAllBooks() {
        Console.WriteLine("All Books:");
        foreach (var book in applicationLayer.GetAllBooks())
        {
            Console.WriteLine($"{book.Id}: {book.Title} by {book.Author} ({book.Year})");
        }
    }

    0 Verweise
    public void DisplayBookDetails(int bookId)
    {
        Book book = applicationLayer.GetBookById(bookId);
        if (book != null)
        {
            Console.WriteLine($"Book Details for Book ID {bookId}:");
            Console.WriteLine($"Title: {book.Title}");
            Console.WriteLine($"Author: {book.Author}");
            Console.WriteLine($"Year: {book.Year}");
        }
        else
        {
            Console.WriteLine($"Book with ID {bookId} not found.");
        }
    }
}
```

Beispielimplementierung in C# Application Layer

```
1   public interface IApplicationLayer {
        2 Verweise
2       public IEnumerable<Book> GetAllBooks();
3
        2 Verweise
4       public Book GetBookById(int id);
5   }
```

```
2   public class CApplicationLayer : IApplicationLayer
3   {
        3 Verweise
4       private IDataAccessLayer dataAccessLayer;
5
        1 Verweis
6       public CApplicationLayer(IDataAccessLayer dataAccessLayer)
7       {
8           this.dataAccessLayer = dataAccessLayer;
9       }
10
        2 Verweise
11      public IEnumerable<Book> GetAllBooks()
12      {
13          return dataAccessLayer.GetAllBooks();
14      }
15
        2 Verweise
16      public Book GetBookById(int id)
17      {
18          return dataAccessLayer.GetBookById(id);
19      }
20  }
```

Beispielimplementierung in C#
Data Access Layer

```
2   public interface IDataAccessLayer {
        2 Verweise
3       public IEnumerable<Book> GetAllBooks();
4
        2 Verweise
5       public Book GetBookById(int id);
6   }
```

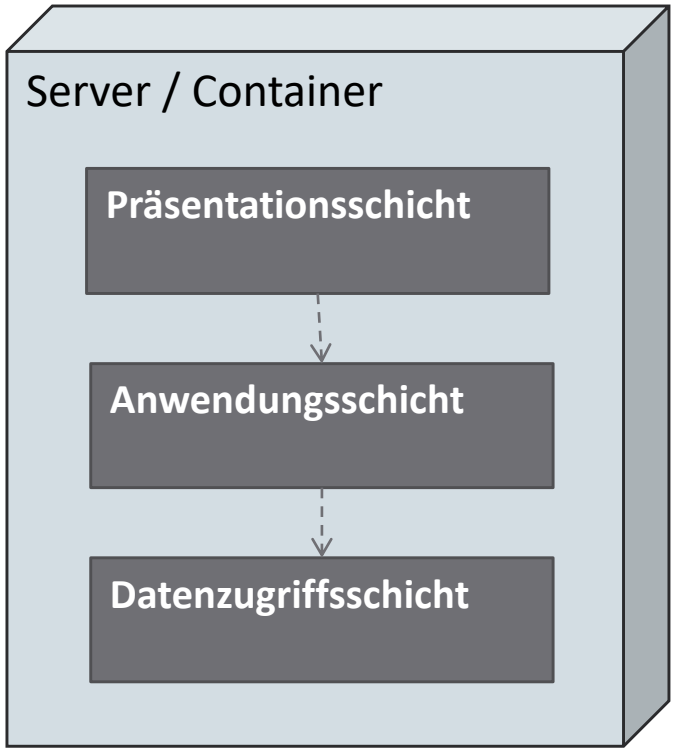
```
3   public class CDataAccessLayer : IDataAccessLayer {
4
        3 Verweise
5       private List<Book> books;
6
        1 Verweis
7       public CDataAccessLayer()
8       {
9           books = new List<Book> {
10              new Book { Id = 1, Title = "Clean Code", Author = "Robert C. Martin", Year = 2008 },
11              new Book { Id = 2, Title = "Design Patterns", Author = "Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides", Year = 1994 },
12              new Book { Id = 3, Title = "Domain-Driven Design", Author = "Eric Evans", Year = 2003 }
13          };
14      }
15
        2 Verweise
16      public IEnumerable<Book> GetAllBooks() {
17          return books;
18      }
19
        2 Verweise
20      public Book GetBookById(int id) {
21          return books.Find(book => book.Id == id);
22      }
23  }
```

Beispielimplementierung in C# Konfiguration & Programm

```
1   public class LayeredArchitectureConfigurator {
        1 Verweis
2       public CPresentationLayer ConfigureSystem() {
3           var dataAccessLayer = new CDataAccessLayer();
4           var applicationLayer = new CApplicationLayer(dataAccessLayer);
5           var presentationLayer = new CPresentationLayer(applicationLayer);
6
7           return presentationLayer;
8       }
9   }
```

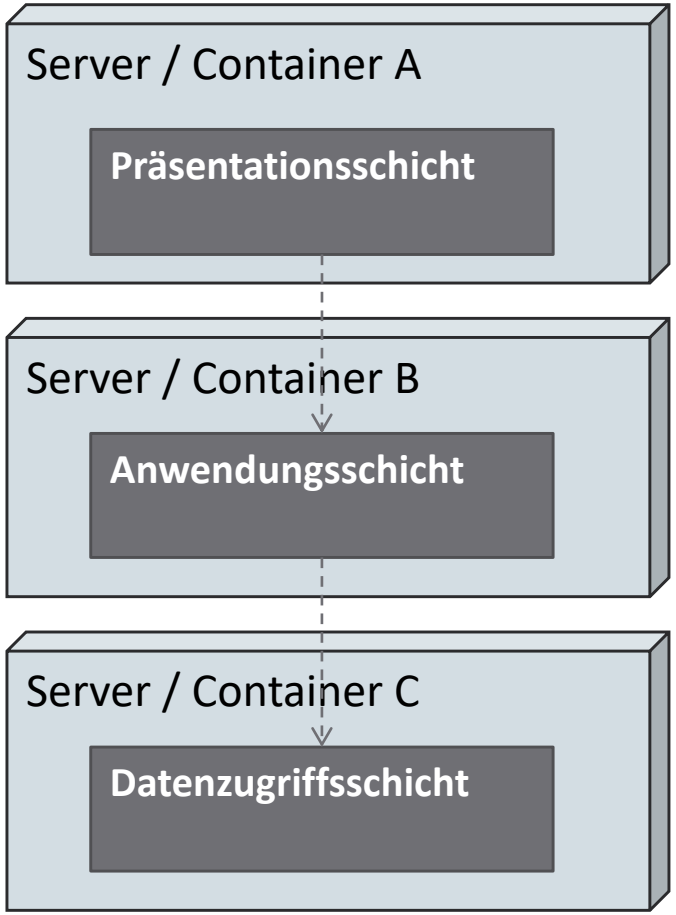
```
3   class Program
4   {
        0 Verweise
5       static void Main(string[] args)
6       {
7           var bookManagement = new LayeredArchitectureConfigurator().ConfigureSystem();
8           bookManagement.DisplayAllBooks();
9       }
10  }
```

Deployment-Varianten
Monolithische Bereitstellung



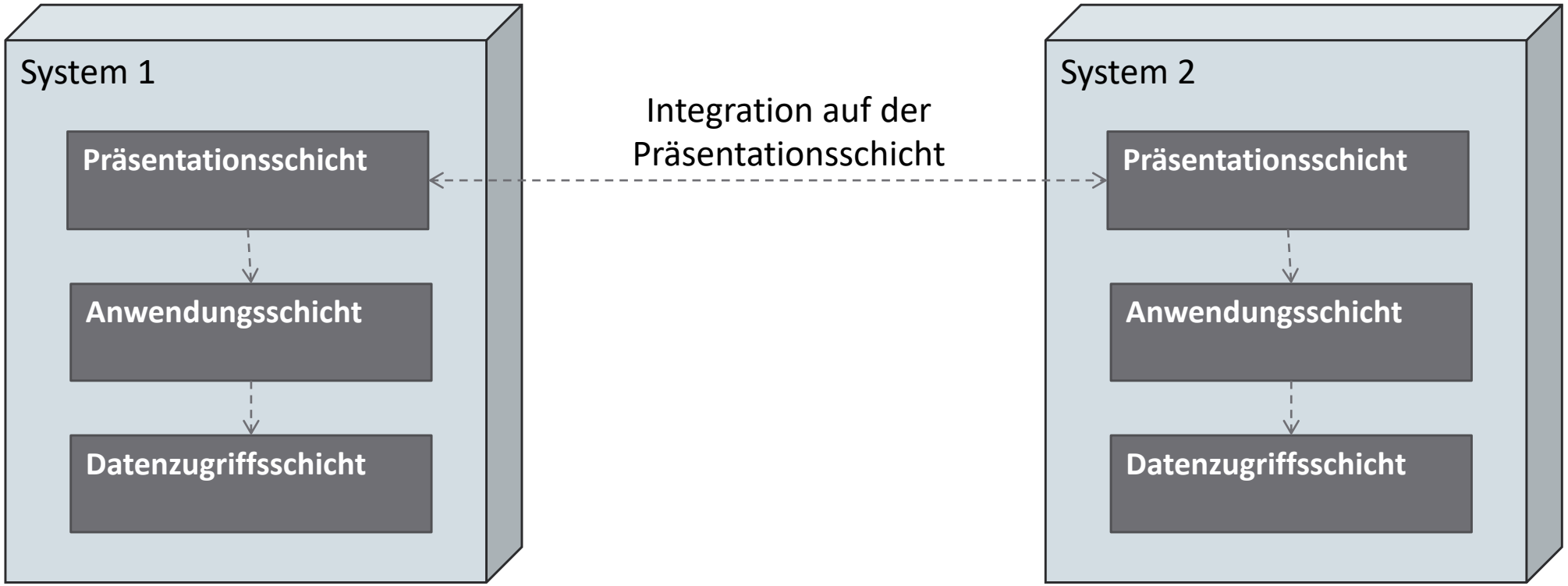
- Alle Schichten werden gemeinsam auf einer Serverinstanz oder einem Container ausgeführt (On-Premises oder in der Cloud)
- Eignet sich gut für kleinere Anwendungen mit einfacher Architektur und geringen Anforderungen an Skalierbarkeit
- Debugging und Management der Anwendung oft einfacher als bei verteilter Bereitstellung

Deployment-Varianten
Verteilte Bereitstellung

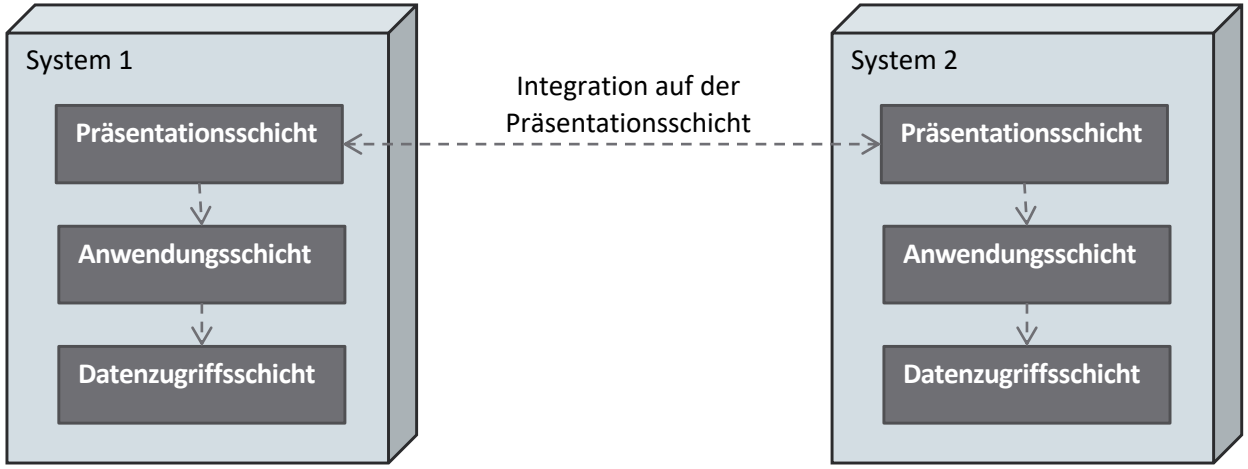


- Alle oder einzelne Schichten werden auf separaten Serverinstanzen oder Containern bereitgestellt (On-Premises oder in der Cloud)
- Bietet eine bessere Skalierbarkeit und Flexibilität, da jede Schicht unabhängig voneinander skaliert und verwaltet werden kann.
- Debugging und Verwaltung des Gesamtsystems können schwieriger auf aufwändiger werden.

Integration zweier Anwendungen auf Ebene der **Präsentationsschicht**

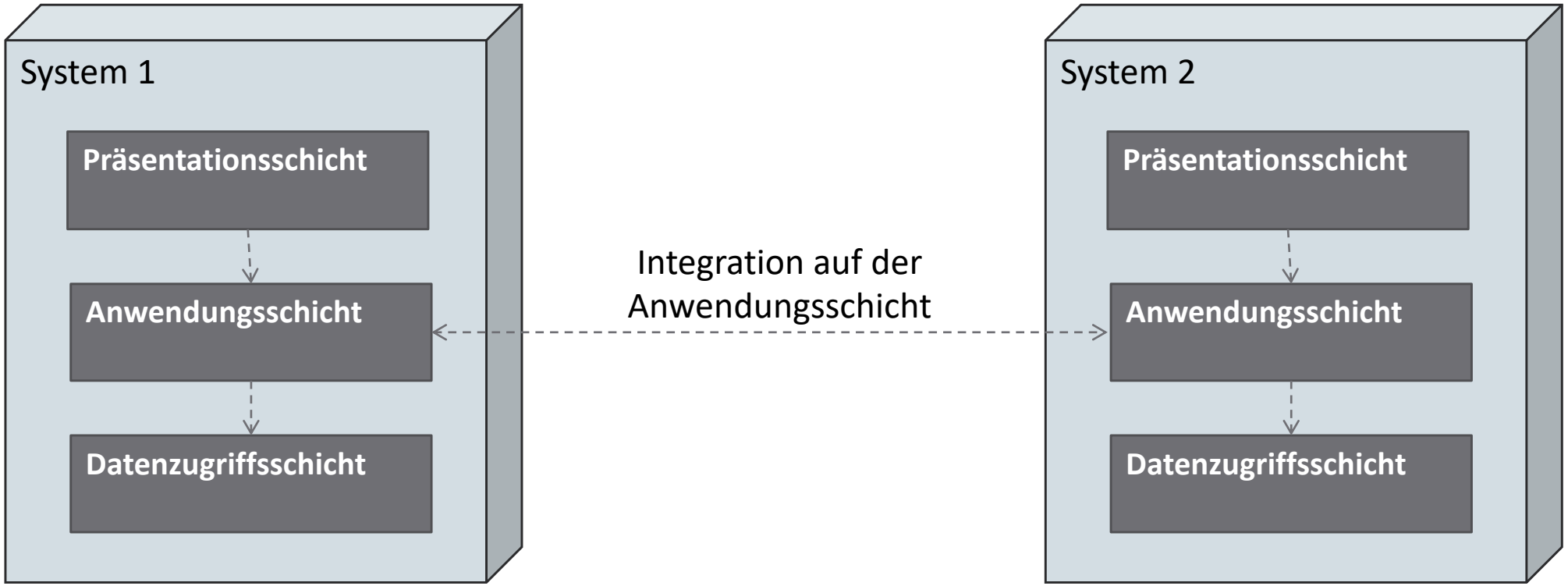


Integration zweier Anwendungen auf Ebene der **Präsentationsschicht**

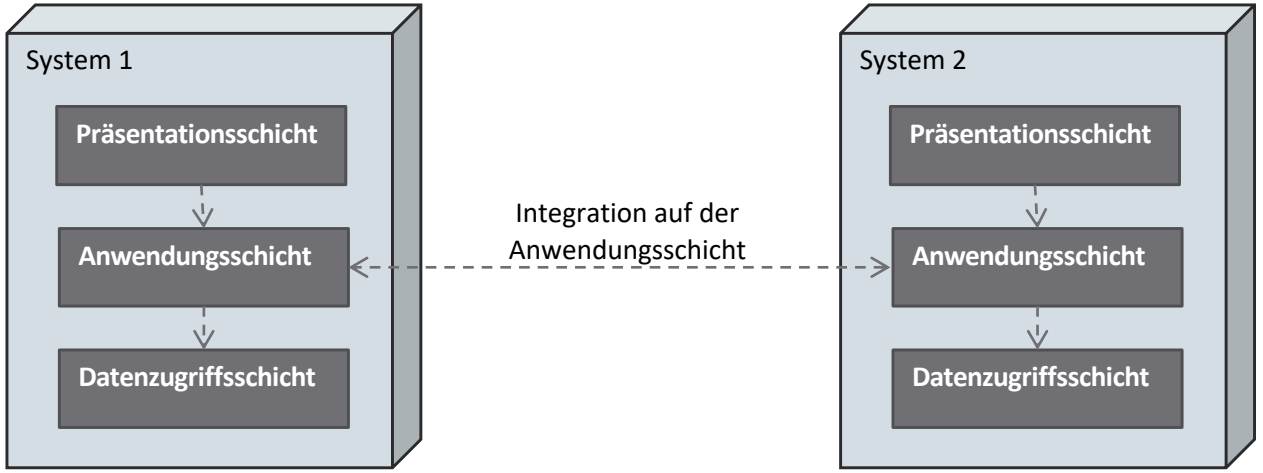


- Technische Umsetzung zum Beispiel mittels IFrames (Inline Frames)
- Beispiel aus einer realen Anwendung: LinkedIn
  - Bietet die Möglichkeit, Inhalte wie Videos, Präsentationen oder Artikel direkt in LinkedIn-Beiträge einzubetten
  - Nutzer können auf diese Beiträge direkt auf der Plattform zugreifen, ohne die Plattform zu verlassen

Integration zweier Anwendungen auf Ebene der **Anwendungsschicht**

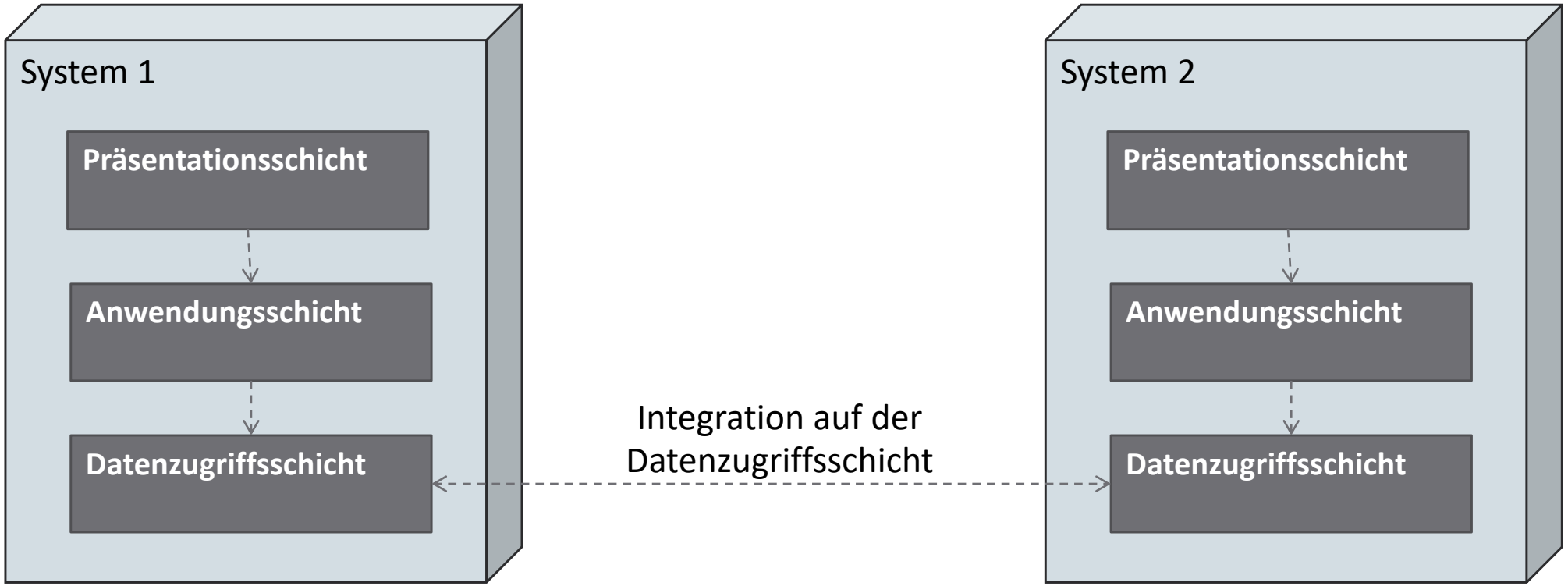


Integration zweier Anwendungen auf Ebene der **Anwendungsschicht**

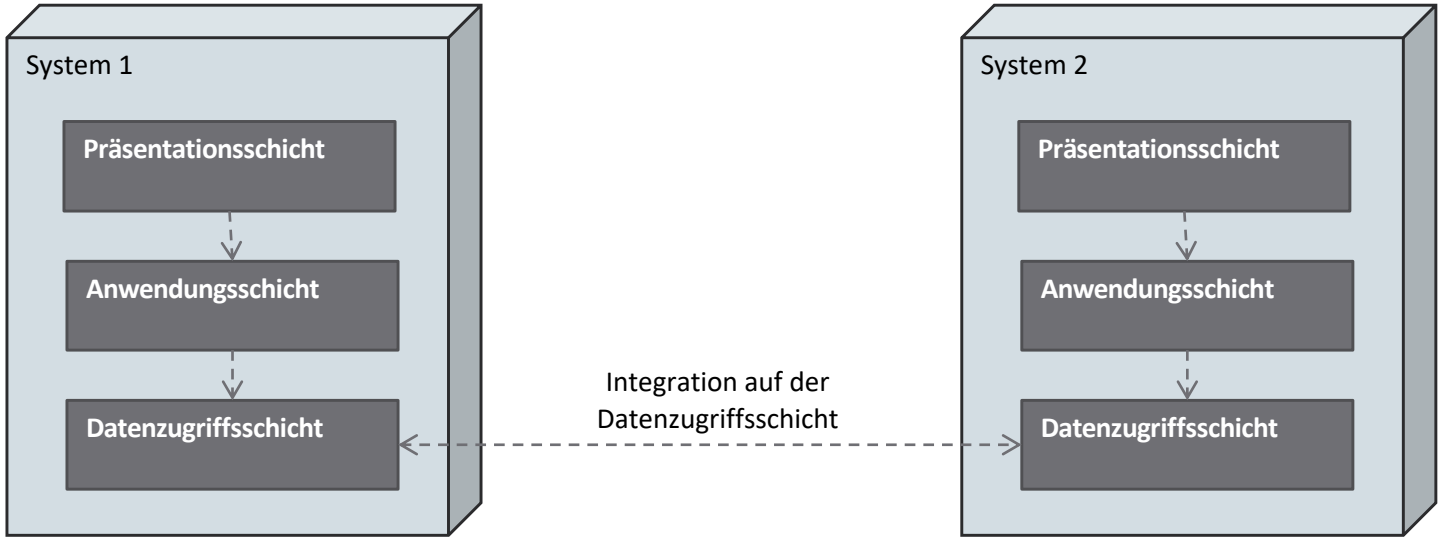


- Web-Services und APIs
- Middleware und Message-Broker
- SDKs (Software Development Kits)

Integration zweier Anwendungen auf Ebene der **Datenzugriffsschicht**



Integration zweier Anwendungen auf Ebene der **Datenzugriffsschicht**



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

Einige Kritikpunkte und Fazit

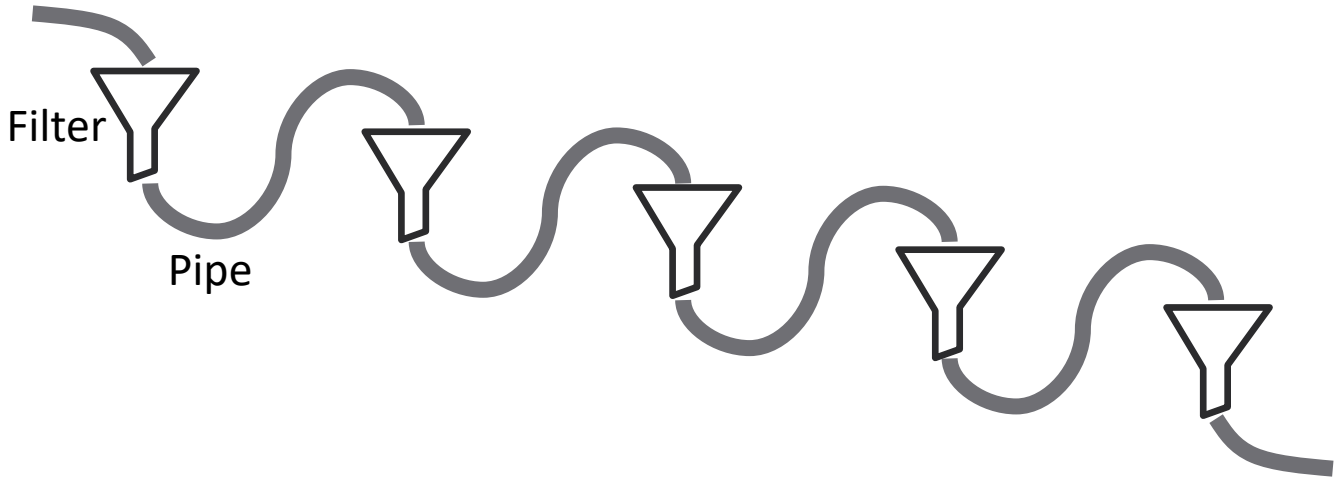
- Gefahr von technischen Schulden innerhalb von Schichten
- Overhead durch Schichten
- Starre Struktur

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 & Filters

Pipes-und-Filters

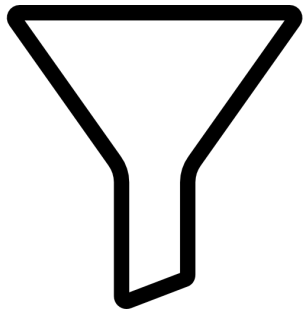
- Basiert auf dem Konzept der Datenverarbeitungspipeline
- Die Architektur besteht aus
  - Filter: Verarbeitung von Daten
  - Pipes: Ermöglichen den Datenfluss zwischen Filter



Filter

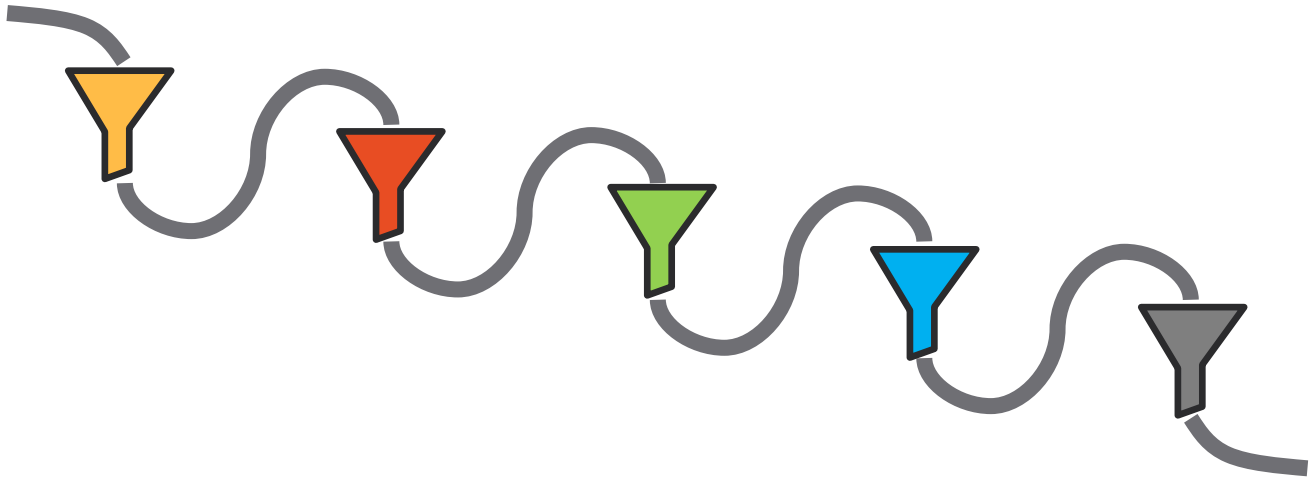
- Jeder Filter verarbeitet also eingehende Daten und gibt sie über eine Pipe an den nächsten Filter weiter.

- Verarbeiten kann bedeuten
  - Herausfiltern/Löschen von Daten
  - Hinzufügen von Daten
  - Verändern von Daten



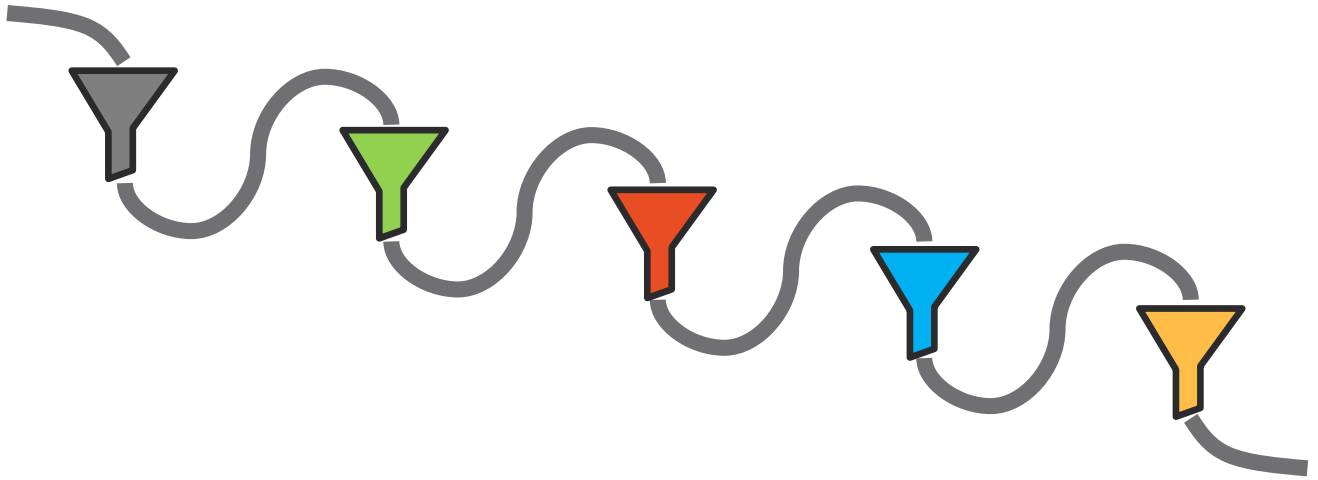
Eigenschaften der Architektur

- Jeder Filter erfüllt eine spezifische Aufgabe, dadurch Trennung von Verantwortlichkeiten
- 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



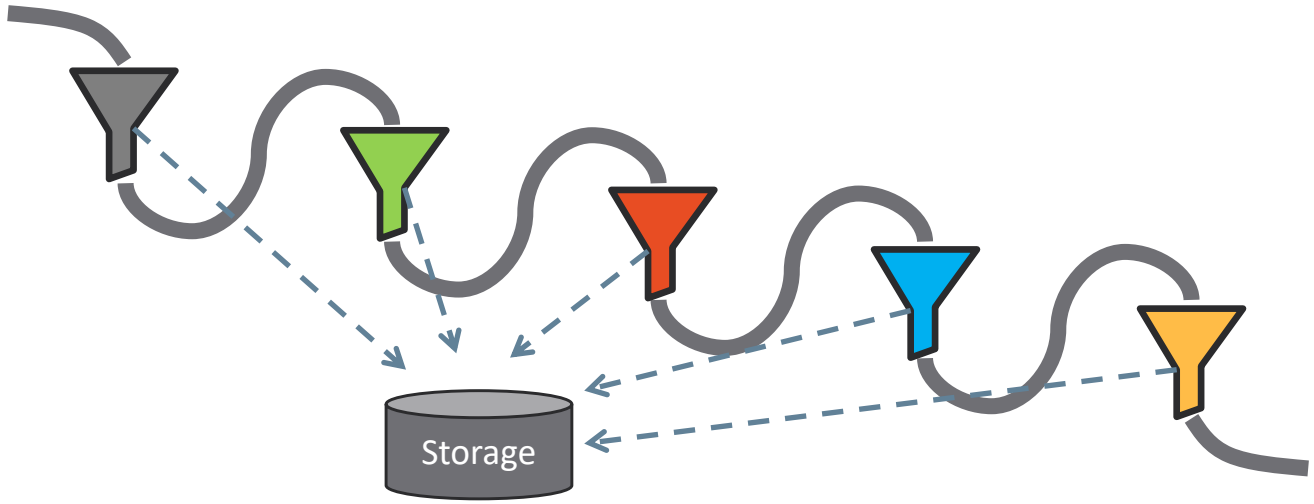
Eigenschaften der Architektur

- Jeder Filter erfüllt eine spezifische Aufgabe, dadurch Trennung von Verantwortlichkeiten
- 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
  - Fördert eine lose Kopplung zwischen Filtern
- Gemeinsamer Speicher kann notwendig werden, falls
  - Zustandsbehaftete Verarbeitung erforderlich ist
  - Kommunikation über den Datenstrom hinaus notwendig ist (z.B. Statusinformationen)
  - Parallelisierung oder Skalierung erforderlich ist



Beispiele für Anwendungsgebiete

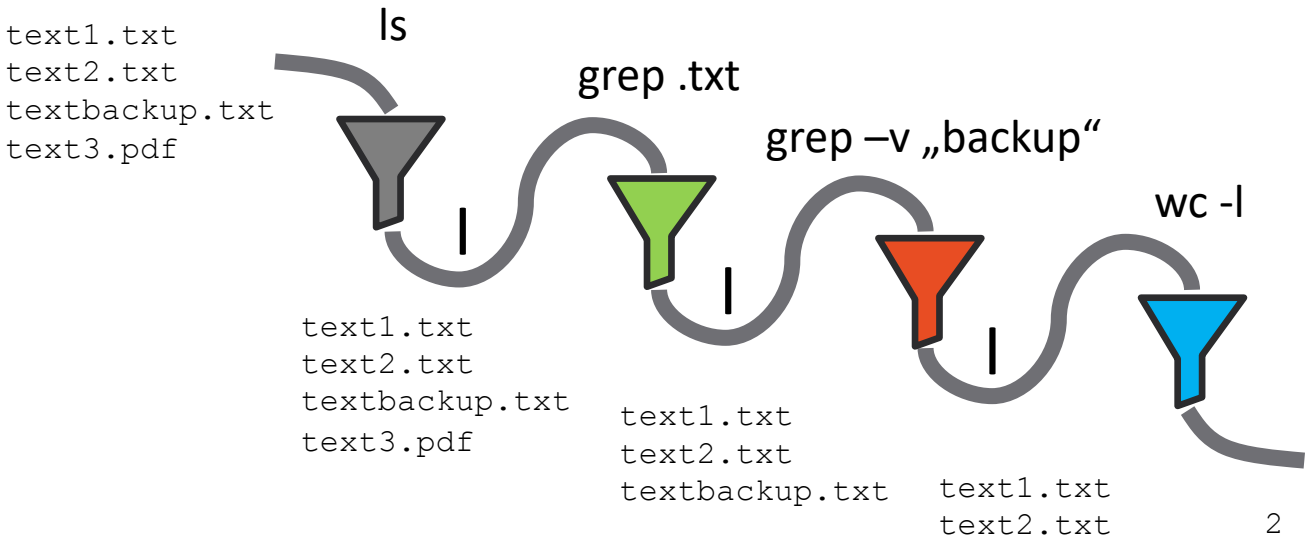
- Textverarbeitung
  - Rechtschreibprüfung, Grammatikprüfung, Textformatierungen, Übersetzungen
- Bildverarbeitung
  - Rauschunterdrückung, Kantenerkennung, Farbkorrektur, Objekterkennung
- Audiodatenverarbeitung
  - Rauschunterdrückung, Equalizer, Spracherkennung, Audiokompression
- Datenanalyse und –transformation
  - ETL-Prozesse (Extract, Transform, Load), Datenbereinigung, -aggregation und -visualisierung

Pipes und Filters in der Linux-Shell

Alle Dateien im aktuellen Verzeichnis	Dateien mit der Endung .txt	Entfernen von Dateien mit „backup“ im Namen aus der Liste	Zählt die Anzahl der Zeilen in der Ausgabe

```
ls | grep .txt | grep –v „backup“ | wc -l
```

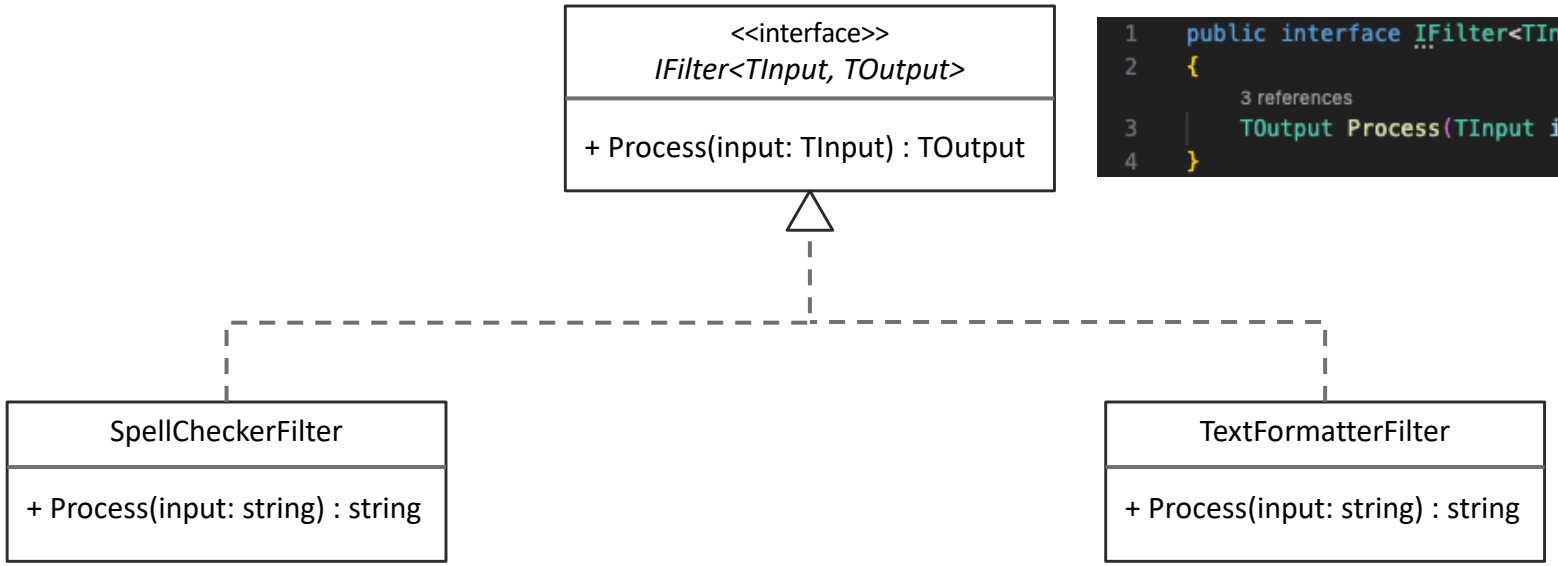
Pipes und Filters in der Linux-Shell



Beispiel-Implementierung in C#

Textverarbeitung mittels Pipes and Filters Architektur

Beispiel-Implementierung in C# Filter




```
1  public interface IFilter<TInput, TOutput>
2  {
       3 references
3      TOutput Process(TInput input);
4  }
```

```
1  // Filter Implementierung für Rechtschreibprüfung
   2 references
2  public class SpellCheckerFilter : IFilter<string, string>
3  {
       2 references
4      public string Process(string input)
5      {
6          // Beispiel: Sehr sehr einfache Rechtschreibprüfung...
7          return input.Replace("Tihs", "This").Replace("writting", "writing");
8      }
9  }
```

```
1  // Filter Implementierung für Textformatierung
   2 references
2  public class TextFormatterFilter : IFilter<string, string>
3  {
       2 references
4      public string Process(string input)
5      {
6          // Beispiel: Text in Großbuchstaben umwandeln
7          return input.ToUpper();
8      }
9  }
```

Beispiel-Implementierung in C# Pipeline

```
using System.Collections.Generic;

// Pipeline-Klasse zur Verwaltung der Filters
2 references | You, 17 hours ago | 1 author (You)
public class Pipeline<TInput, TOutput>
{
    2 references
    private List<IFilter<TInput, TOutput>> filters = new List<IFilter<TInput, TOutput>>();

    2 references
    public void AddFilter(IFilter<TInput, TOutput> filter)
    {
        filters.Add(filter);
    }

    1 reference
    public TOutput Process(TInput input)
    {
        var output = default(TOutput);
        foreach (var filter in filters)
        {
            output = filter.Process(input);
            input = (TInput)Convert.ChangeType(output, typeof(TInput));
        }
        return output;
    }
}
```

Pipeline<IInput, TOutput>
+ AddFilter(Filter<TInput, TOutput> filter) + Process(TInput: input) : TOutput

Beispiel-Implementierung in C# Programm

```
0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        // Erstellen der Filter
        TextFormatterFilter textFormatter = new TextFormatterFilter();
        SpellCheckerFilter spellChecker = new SpellCheckerFilter();

        // Erstellen der Pipeline und Hinzufügen der Filter
        Pipeline<string, string> pipeline = new Pipeline<string, string>();

        pipeline.AddFilter(spellChecker);
        pipeline.AddFilter(textFormatter);

        // Beispieltext
        string inputText = "Tihs is an example of text writting in C#.";

        // Text durch die Pipeline senden
        string outputText = pipeline.Process(inputText);

        // Ausgabe des bearbeiteten Textes
        Console.WriteLine("Originaltext: " + inputText);
        Console.WriteLine("Bearbeiteter Text: " + outputText);
    }
}
```

Ausgabe

Originaltext: Tihs is an example of text writting in C#.
Bearbeiteter Text: TIHS IS AN EXAMPLE OF TEXT WRITTING IN C#.

Blackboard-Architektur

Blackboard-Architektur

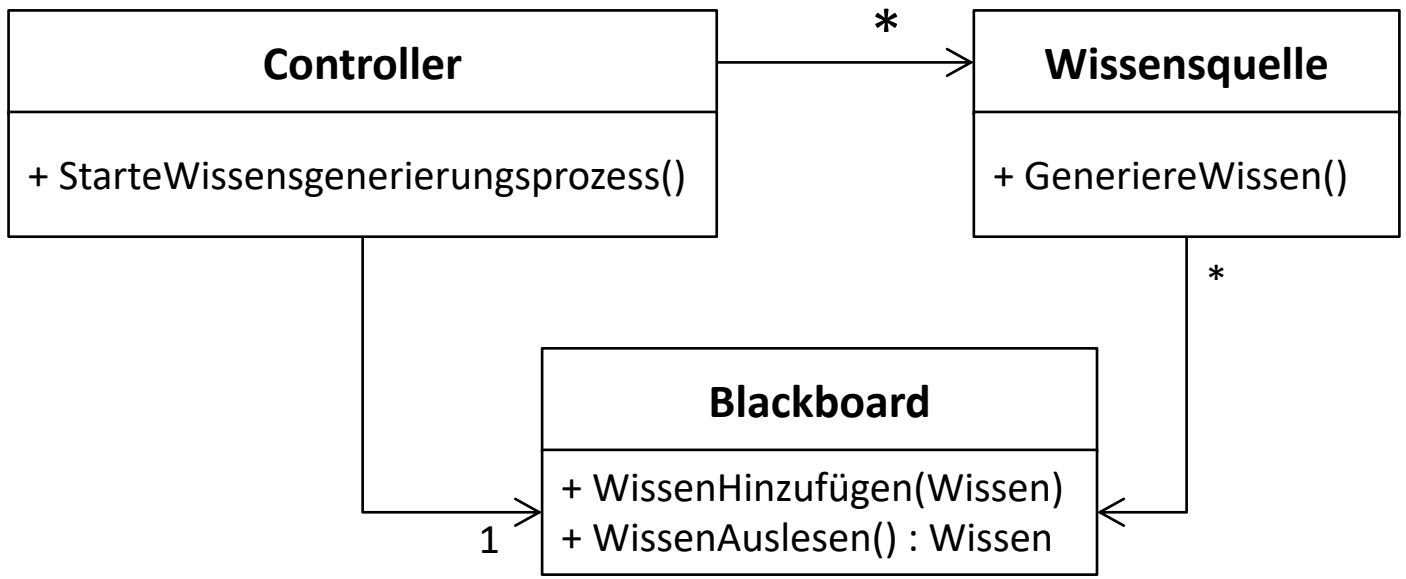
- Die Blackboard-Architektur ist hilfreich für Probleme, zu denen keine deterministische Lösungsstrategie bekannt ist.
- Gründe hierfür können zum Beispiel sein, dass das Problem
  - Komplexität aufweist, die über die Fähigkeit deterministischer Methoden hinausgeht.
  - Unstrukturiert ist und keine klaren Regeln oder Muster aufweist, nach denen vorgegangen werden kann.
  - Unsicherheit beinhaltet, so dass selbst bei der Anwendung bekannter Methoden das Ergebnis unvorhersehbar sein kann.
- Eine Blackboard-Architektur kann hierfür eine Lösung sein
  - Ermöglicht flexible und dynamische Herangehensweise 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

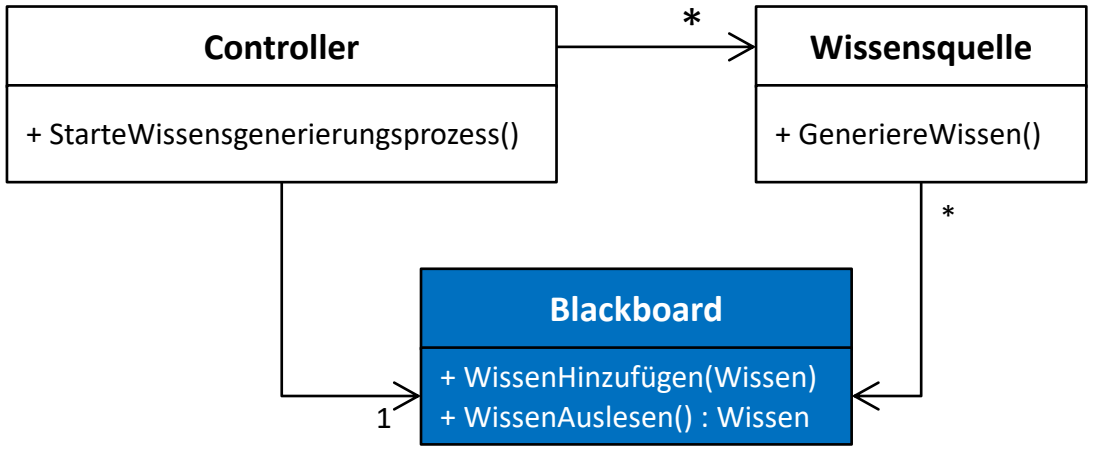
- Komplexe medizinische Diagnosen
- Wissenschaftliche Forschung und Entdeckung
- Kreatives Problemlösen und Design

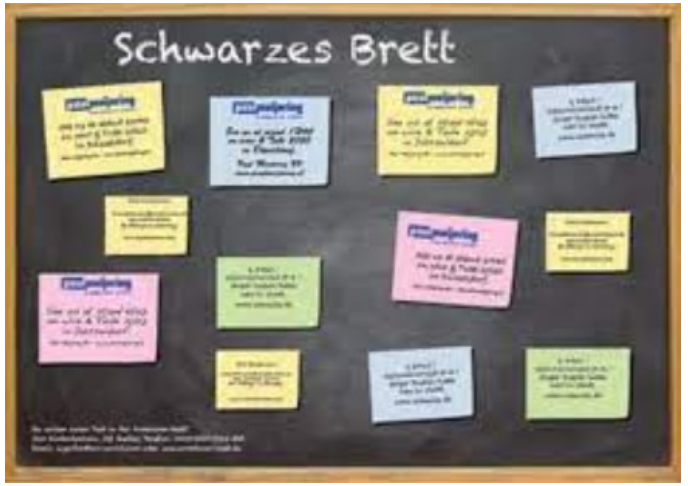
- Anwendung immer dann, wenn ein komplexes Problem gelöst werden muss, das von verschiedenen Experten mit unterschiedlichem Fachwissen angegangen werden kann.

Komponenten einer Blackboard-Architektur



Blackboard

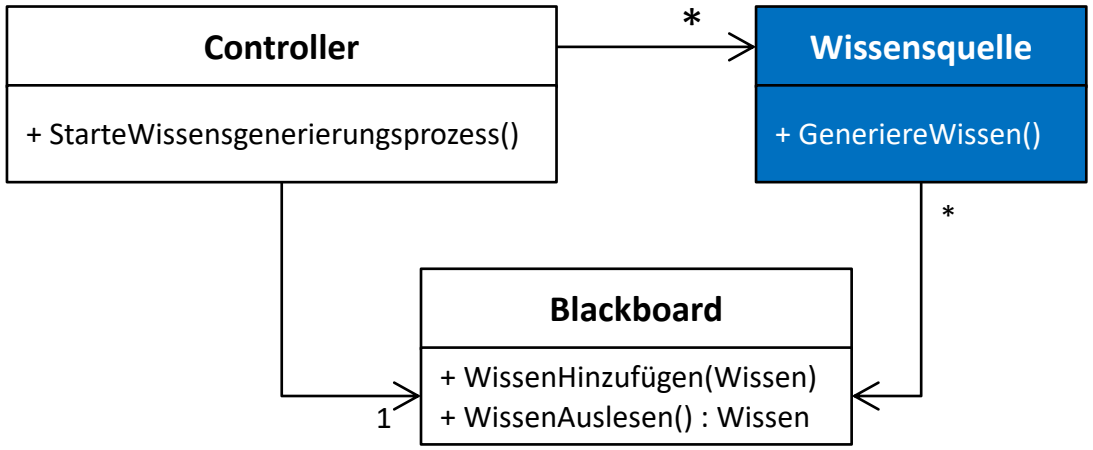




Blackboard (Schwarzes Brett)

- Zentraler Datenspeicher
- Speicherort für alle verfügbaren Rohdaten, Informationen und Hypothesen
- Ist für alle beteiligten Komponenten zugänglich
- Zentrale Schnittstelle für die Zusammenarbeit
- Bietet Schnittstellen zum Lesen und Schreiben ans schwarze Brett

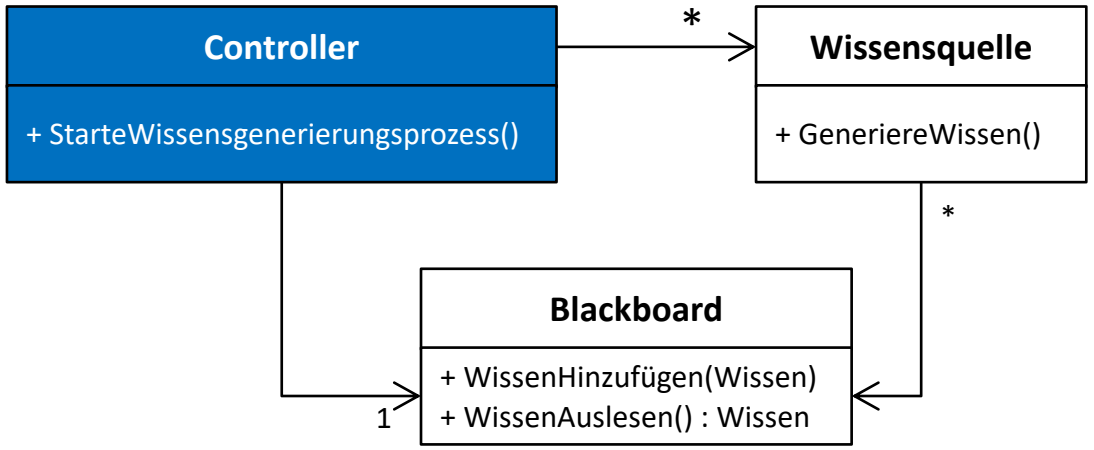
Wissensquellen



Wissensquellen

- Eigenständige Komponenten oder Expertensysteme
- Stellen 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
- Signalisieren, unter welchen Bedingungen sie zu einer (Teil-) Lösung beitragen können

Controller



Controller

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

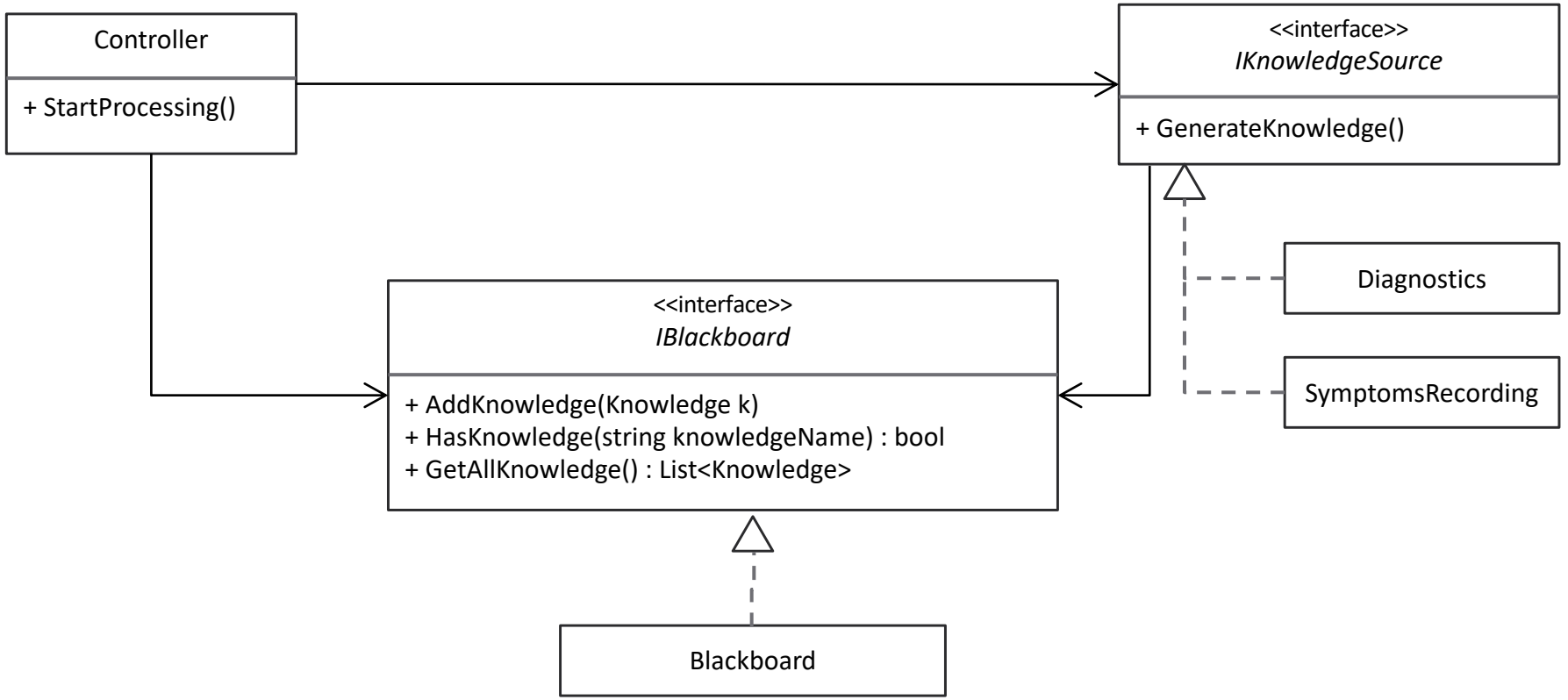
Beispiel: Medizinisches Diagnosesystem

- Blackboard
  - Zentrale Wissensbasis, die von allen Modulen gemeinsam genutzt wird.
  - Speichert Informationen über:
    - Patienten, Symptome, diagnostische Tests, mögliche Krankheiten, Behandlungsmöglichkeiten
- Experten / Wissensquellen
  - Symptomerkennung, Laborergebnisse (z.B. Bluttests), Bilderzeugung (Röntgen, MRT, …), Bildanalyse, Anamnese, Entscheidungsmodul
  - Alle Wissensquellen können wiederum externe Systeme nutzen
- Controller
  - Sorgt für eine geregelte Zusammenarbeit zwischen Modulen
  - Kann zum Beispiel Prioritäten vergeben

Beispiel-Implementierung in C#

Erstellen medizinischer Diagnosen mittels Blackboard Architektur

Beispiel-Implementierung in C#



Beispiel-Implementierung in C# Knowledge Source / Diagnostics

```
1   public interface IKnowledgeSource {
        3 Verweise
2       public void GenerateKnowledge();
3   }
```

```
    2 Verweise | You, vor 1 Stunde | 1 author (You)
3   public class Diagnostics : IKnowledgeSource
4   {
        5 Verweise
5       private IBlackboard blackboard;
6
        1 Verweis
7       public Diagnostics(IBlackboard blackboard)
8       {
9           this.blackboard = blackboard;
10      }
11
        2 Verweise
12      public void GenerateKnowledge()
13      {
14          if (blackboard.GetAllKnowledge().Exists(s => s.Name == "Fieber")
15              && blackboard.GetAllKnowledge().Exists(s => s.Name == "Husten")
16              && !blackboard.GetAllKnowledge().Exists(s => s.Name == "Erkältung"))
17          {
18              blackboard.AddKnowledge(new Knowledge("Erkältung", "Eine häufige Infektionskrankheit der Atemwege."));
19          }
20      }
21  }
```

Beispiel-Implementierung in C# Knowledge Source / SymptomsRecording

```
3   public class SymptomsRecording : IKnowledgeSource
4   {
        7 Verweise
5       private IBlackboard blackboard;
6
        1 Verweis
7       public SymptomsRecording(IBlackboard blackboard)
8       {
9           this.blackboard = blackboard;
10      }
11
        2 Verweise
12      public void GenerateKnowledge()
13      {
14          if(!blackboard.HasKnowledge("Fieber")) {
15              blackboard.AddKnowledge(new Knowledge("Fieber", "Erhöhte Körpertemperatur"));
16          }
17
18          if(!blackboard.HasKnowledge("Husten")) {
19              blackboard.AddKnowledge(new Knowledge("Husten", "Trockener Husten"));
20          }
21
22          if(!blackboard.HasKnowledge("Schnupfen")) {
23              blackboard.AddKnowledge(new Knowledge("Schnupfen", "Laufende Nase"));
24          }
25      }
26  }
```

Beispiel-Implementierung in C# Blackboard

```
3       public interface IBlackboard {
            5 Verweise
4           public void AddKnowledge(Knowledge knowledge);
5
            4 Verweise
6           bool HasKnowledge(string knowledgeName);
7
            5 Verweise
8           public List<Knowledge> GetAllKnowledge();
9       }
10  }
```

```
3   public class Blackboard : IBlackboard
4   {
        3 Verweise
5       private List<Knowledge> knowledges = new List<Knowledge>();
6
        5 Verweise
7       public void AddKnowledge(Knowledge knowledge)
8       {
9           this.knowledges.Add(knowledge);
10      }
11
        5 Verweise
12      public List<Knowledge> GetAllKnowledge()
13      {
14          return this.knowledges;
15      }
16
        4 Verweise
17      public bool HasKnowledge(string knowledgeName)
18      {
19          return knowledges.Any(k => k.Name == knowledgeName);
20      }
21  }
```

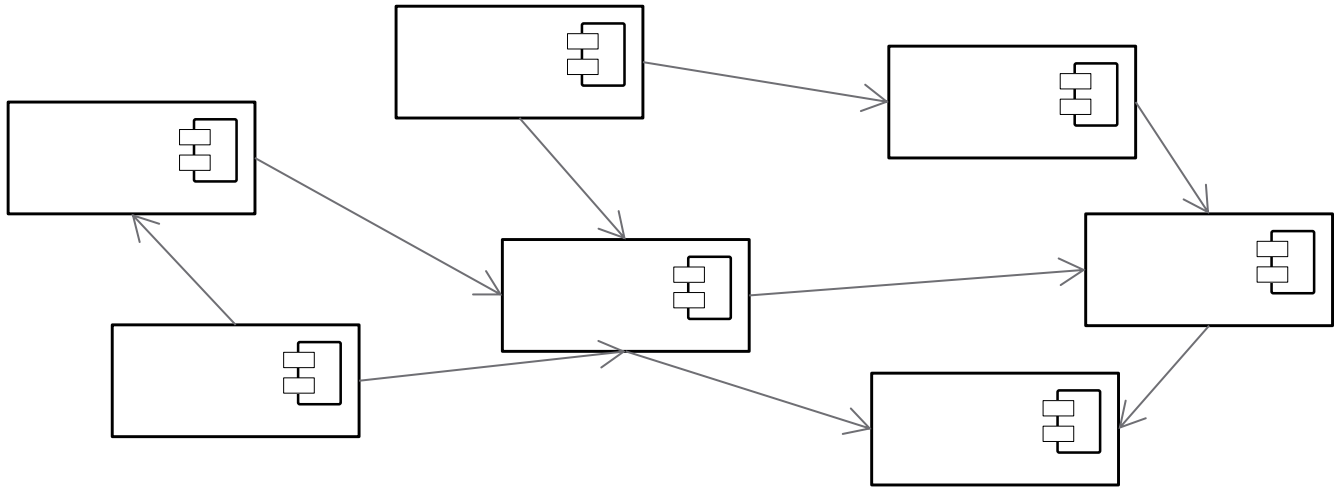
Beispiel-Implementierung in C# Controller

```
3   public class Controller {
        2 Verweise
4       private List<IKnowledgeSource> knowledgeSources;
        1 Verweis
5       private IBlackboard blackboard;
6
        1 Verweis
7       public Controller(IBlackboard blackboard, List<IKnowledgeSource> knowledgeSources)
8       {
9           this.knowledgeSources = knowledgeSources;
10          this.blackboard = blackboard;
11      }
12
        1 Verweis
13      public void StartProcessing()
14      {
15          for(int i = 0; i < 5; i++) {
16              AskEachKnowledgeSourceForNewKnowledge();
17          }
18      }
19
        1 Verweis
20      private void AskEachKnowledgeSourceForNewKnowledge() {
21          foreach (var knowledgeSource in knowledgeSources)
22          {
23              knowledgeSource.GenerateKnowledge();
24          }
25      }
26  }
```

Broker-Architektur

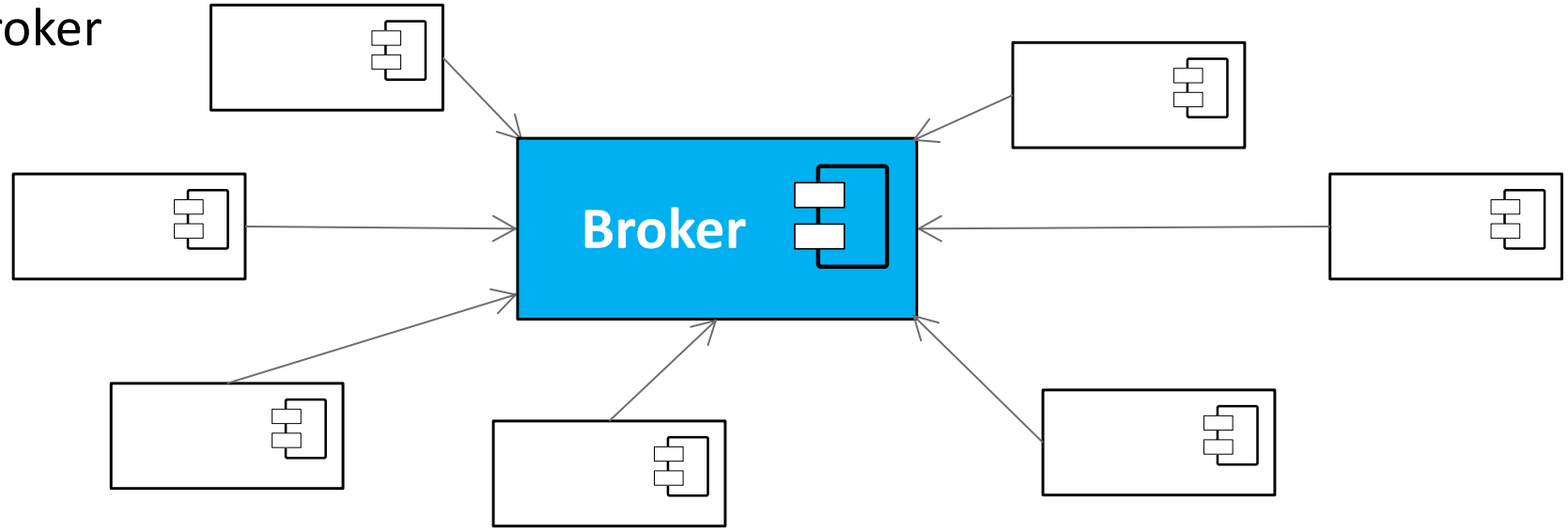
Broker Architektur

- Wird in verteilten 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



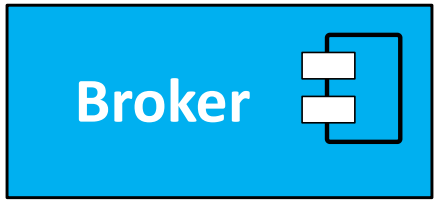
Broker Architektur

- Wird in verteilten 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

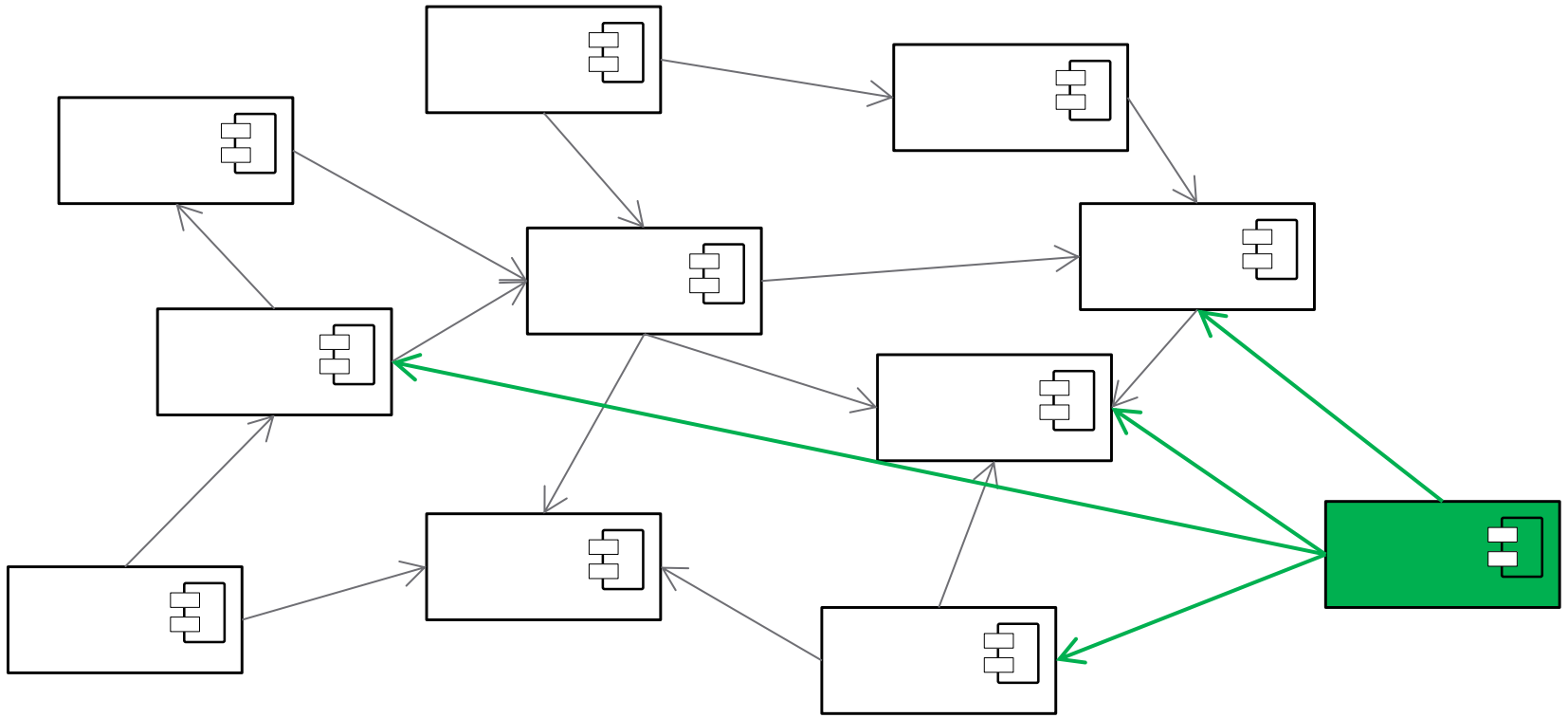


Zentrale Konzepte

- Broker empfängt Nachrichten und leitet sie an Empfänger weiter
- Je nach Implementierung kann der Broker Nachrichten
  - Transformieren
  - Filtern
  - Priorisieren
- Das Architekturmuster fördert die Entkopplung von Komponenten
- Vorteile bei der Skalierbarkeit
  - Zum Beispiel wird das Hinzufügen von Komponenten vereinfacht

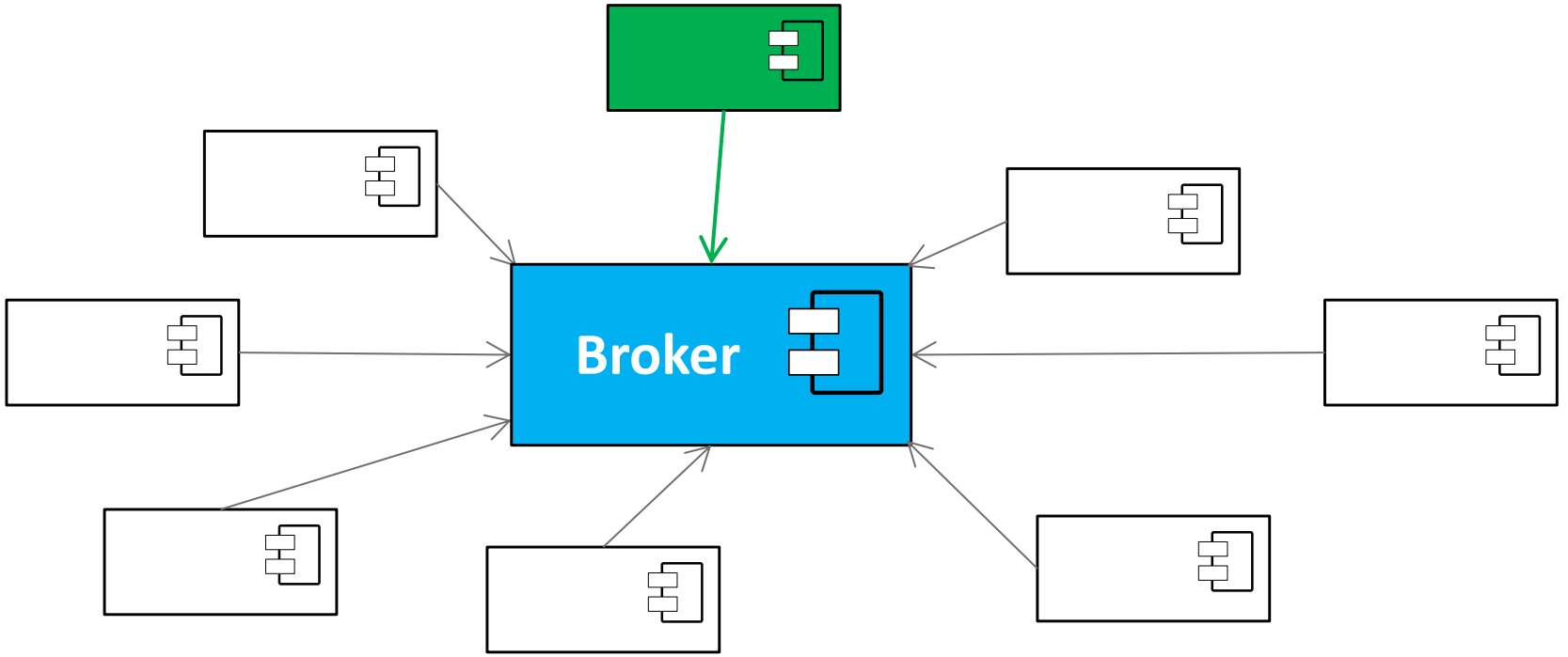


Vorteil Skalierbarkeit



Im schlechtesten Fall müssten n Schnittstellen implementiert werden!
(n = Anzahl der Komponenten des ursprünglichen Systems)

Vorteil Skalierbarkeit



In einer Broker-Architektur muss nur eine Schnittstelle programmiert werden.

Herausforderungen bei der Implementierung des Brokers

- Skalierbarkeit
- Nachrichtenverarbeitung 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
  - https://www.rabbitmq.com/
- 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
  - https://www.rabbitmq.com/tutorials/amqp-concepts
- Unterstützt auch Protokolle wie MQTT, STOMP und HTTP

Elemente von RabbitMQ

- Produzent (Producer)
  - Eine Komponente, die Nachrichten sendet

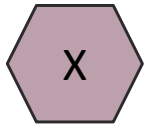
- Konsument (Consumer)
  - Eine Komponente, die Nachrichten empfängt

- Können auf unterschiedlichen Rechnern ausgeführt werden (ist in der Praxis der Regelfall)
- Komponenten/Anwendungen können sowohl Produzent als auch Konsument sein

- Warteschlange (Queue)
  - Ein Nachrichtenpuffer, der Nachrichten weiterleitet und auch speichern kann

- Kanal (Channel)
  - Kommunikationspipeline zwischen einer Anwendung/Komponente und dem RabbitMQ-Server

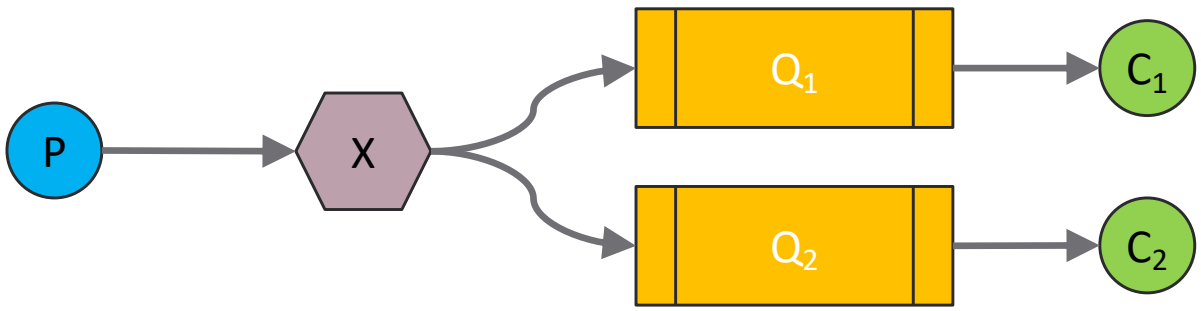
Exchange Types



- Produzenten schicken Nachrichten an einen sogenannten *Exchange*
- Ein Exchange entscheidet, wie mit eingehenden Nachrichten weiter verfahren wird.
- Unter anderem definiert RabbitMQ folgende *Exchange Types*
  - Fanout Exchange
  - Direct Exchange
  - Topic Exchange

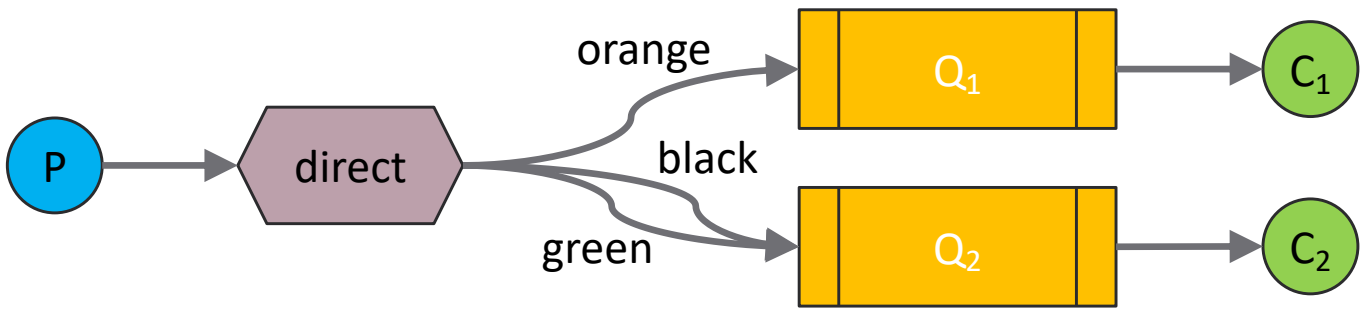
Exchange Type – Fanout Exchange

- Leitet Nachrichten an alle an ihn angeschlossene Warteschlangen weiter.
- Ermöglicht eine einfache Broadcast- oder Multicast-Verteilung von Nachrichten an alle Warteschlangen, die sich an den Exchange angeschlossen haben.
- Nachrichten werden an alle Warteschlangen weitergeleitet, die an den Exchange gebunden sind.



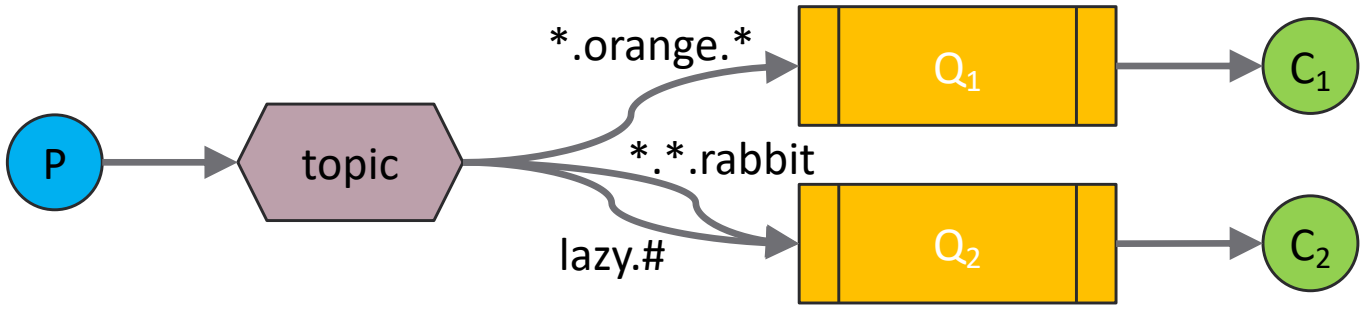
Exchange Type – Direct Exchange

- Leitet Nachrichten basierend auf einem festgelegten Routing-Schlüssel an eine oder mehrere Warteschlangen weiter.
- Der Routing-Schlüssel einer Nachricht wird mit einem spezifischen 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 Warteschlangen, indem ein spezifischer Routing-Schlüssel verwendet wird.



Exchange Type – Topic Exchange

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



*: Platzhalter für genau 1 Wort
#: Platzhalter für mehrere Wörter

Beispiel-Implementierung in C# mit RabbitMQ

Heimautomatisierung mit RabbitMQ unter Verwendung von Topics

```
using System.Text;
using RabbitMQ.Client;

var factory = new ConnectionFactory { HostName = "localhost" };

using var connection = factory.CreateConnection();
using var channel = connection.CreateModel();

channel.ExchangeDeclare(exchange: "home_automation", type: ExchangeType.Topic);

var routingKey = (args.Length > 0) ? args[0] : "anonymous.info";

var message = (args.Length > 1)
              ? string.Join(" ", args.Skip(1).ToArray())
              : "Hello World!";

var body = Encoding.UTF8.GetBytes(message);

channel.BasicPublish(exchange: "home_automation",
                     routingKey: routingKey,
                     basicProperties: null,
                     body: body);

Console.WriteLine($" [x] Sent '{routingKey}':'{message}'");
```

Sender

Source Code → Receiver

```
using System.Text;
using RabbitMQ.Client;
using RabbitMQ.Client.Events;

var factory = new ConnectionFactory { HostName = "localhost" };

using var connection = factory.CreateConnection();
using var channel = connection.CreateModel();

channel.ExchangeDeclare(exchange: "home_automation", type: ExchangeType.Topic);

var queueName = channel.QueueDeclare().QueueName;

if (args.Length < 1)
{
    Environment.ExitCode = 1;
    return;
}

foreach (var bindingKey in args)
{
    channel.QueueBind(queue: queueName,
                      exchange: "home_automation",
                      routingKey: bindingKey);
}

Console.WriteLine(" [*] Waiting for messages. To exit press CTRL+C");

var consumer = new EventingBasicConsumer(channel);
consumer.Received += (model, ea) =>
{
    var body = ea.Body.ToArray();
    var message = Encoding.UTF8.GetString(body);
    var routingKey = ea.RoutingKey;
    Console.WriteLine($" [x] Received '{routingKey}':'{message}'");
};

channel.BasicConsume(queue: queueName,
                     autoAck: true,
                     consumer: consumer);

Console.WriteLine(" Press [enter] to exit.");
Console.ReadLine();
```

```
using System.Text;
using RabbitMQ.Client;
using RabbitMQ.Client.Events;

var factory = new ConnectionFactory { HostName = "localhost" };

using var connection = factory.CreateConnection();
using var channel = connection.CreateModel();

channel.ExchangeDeclare(exchange: "home_automation", type: ExchangeType.Topic);

var queueName = channel.QueueDeclare().QueueName;

if (args.Length < 1) {
    Environment.ExitCode = 1;
    return;
}

foreach (var bindingKey in args) {
    channel.QueueBind(queue: queueName,
                      exchange: "home_automation",
                      routingKey: bindingKey);
}

Console.WriteLine(" [*] Waiting for messages. To exit press CTRL+C");

var consumer = new EventingBasicConsumer(channel);
consumer.Received += (model, ea) =>
{
    var body = ea.Body.ToArray();
    var message = Encoding.UTF8.GetString(body);
    var routingKey = ea.RoutingKey;
    Console.WriteLine($" [x] Received '{routingKey}':'{message}'");
};

channel.BasicConsume(queue: queueName,
                     autoAck: true,
                     consumer: consumer);

Console.WriteLine(" Press [enter] to exit.");
Console.ReadLine();
```

Aufbau eines Kanals zum RabbitMQ Server

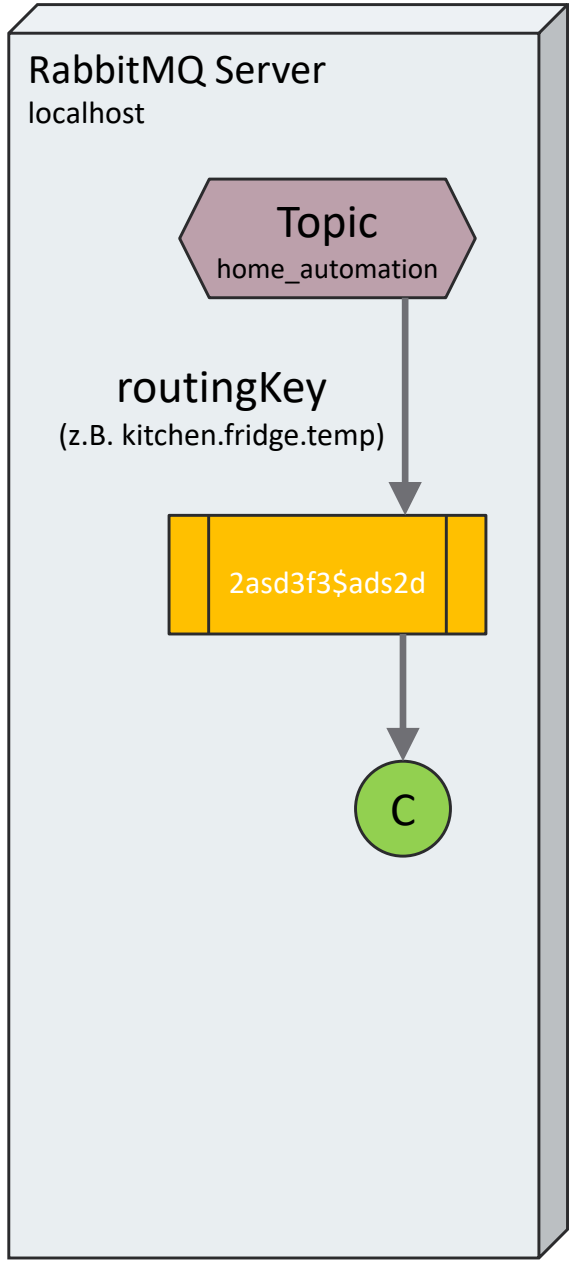
Topic-Exchange erzeugen

Flüchtige, automatisch löschende Warteschlange mit generiertem Namen

„Binding“: Verbindungen zwischen *Exchange* und Warteschlange aufbauen mit jeweiligem routingKey

Erzeugen des Consumers und Implementierung einer Callback-Funktion

Registrieren des Consumers bei der Warteschlange



```
using System.Text;
using RabbitMQ.Client;

var factory = new ConnectionFactory { HostName = "localhost" };

using var connection = factory.CreateConnection();
using var channel = connection.CreateModel();

channel.ExchangeDeclare(exchange: "home_automation", type: ExchangeType.Topic);

var routingKey = (args.Length > 0) ? args[0] : "anonymous.info";

var message = (args.Length > 1)
              ? string.Join(" ", args.Skip(1).ToArray())
              : "Hello World!";

var body = Encoding.UTF8.GetBytes(message);

channel.BasicPublish(exchange: "home_automation",
                     routingKey: routingKey,
                     basicProperties: null,
                     body: body);

Console.WriteLine($" [x] Sent '{routingKey}':'{message}'");
```

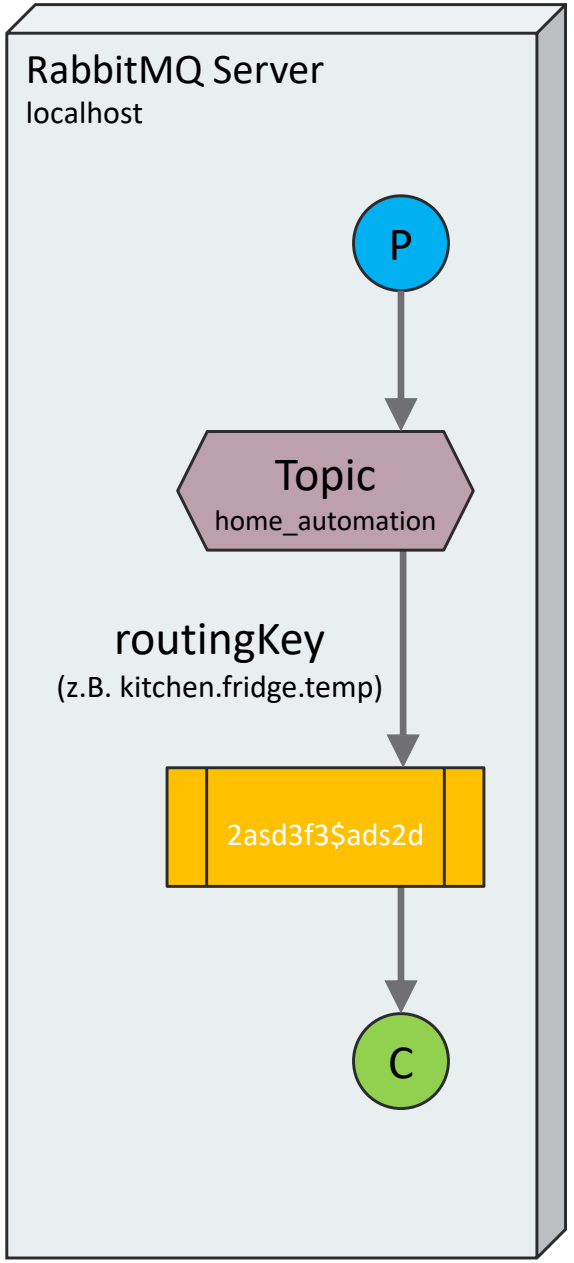
Aufbau eines Kanals zum RabbitMQ Server

Topic-Exchange erzeugen (falls noch nicht vorhanden)

routingKey für das Versenden definieren

Zusammensetzen der Nachricht

Verschicken der Nachricht über den *Exchange* home_automation



Versenden und Verschicken von Nachrichten

Receiver receives everything

```
> dotnet run "#"
```

```
> [x] Received
'kitchen.fridge.temperature':'5'
```

```
> [x] Received
'living_room.*.temperature':'28'
```

Receiver receives all kitchen events

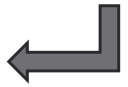
```
> dotnet run "kitchen.*.*"
```

```
> [x] Received
'kitchen.fridge.temperature':'5'
```

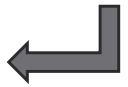
```
>
```

Sender

```
> dotnet run
"kitchen.fridge.temperature" "5"
```



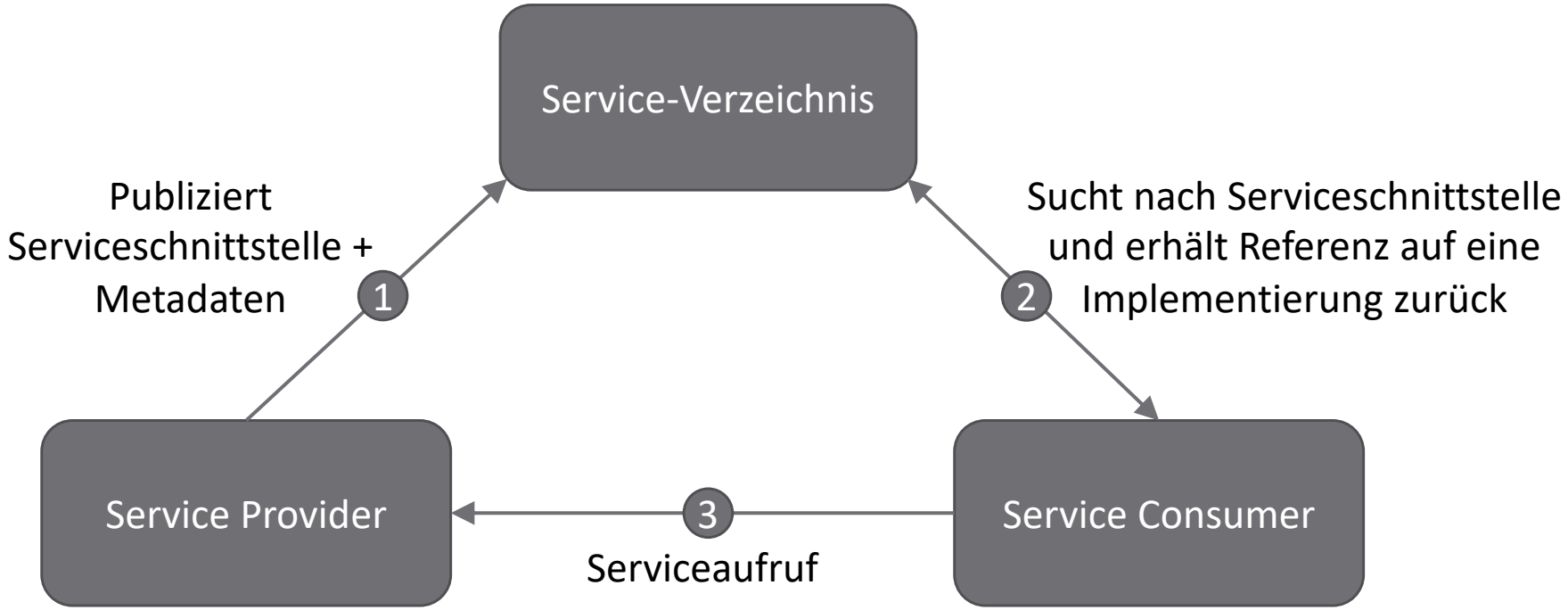
```
> dotnet run
"living_room.*.temperature" "28"
```



Serviceorientiere Architekturen (SOA)

Serviceorientierte Architekturen (SOA)

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

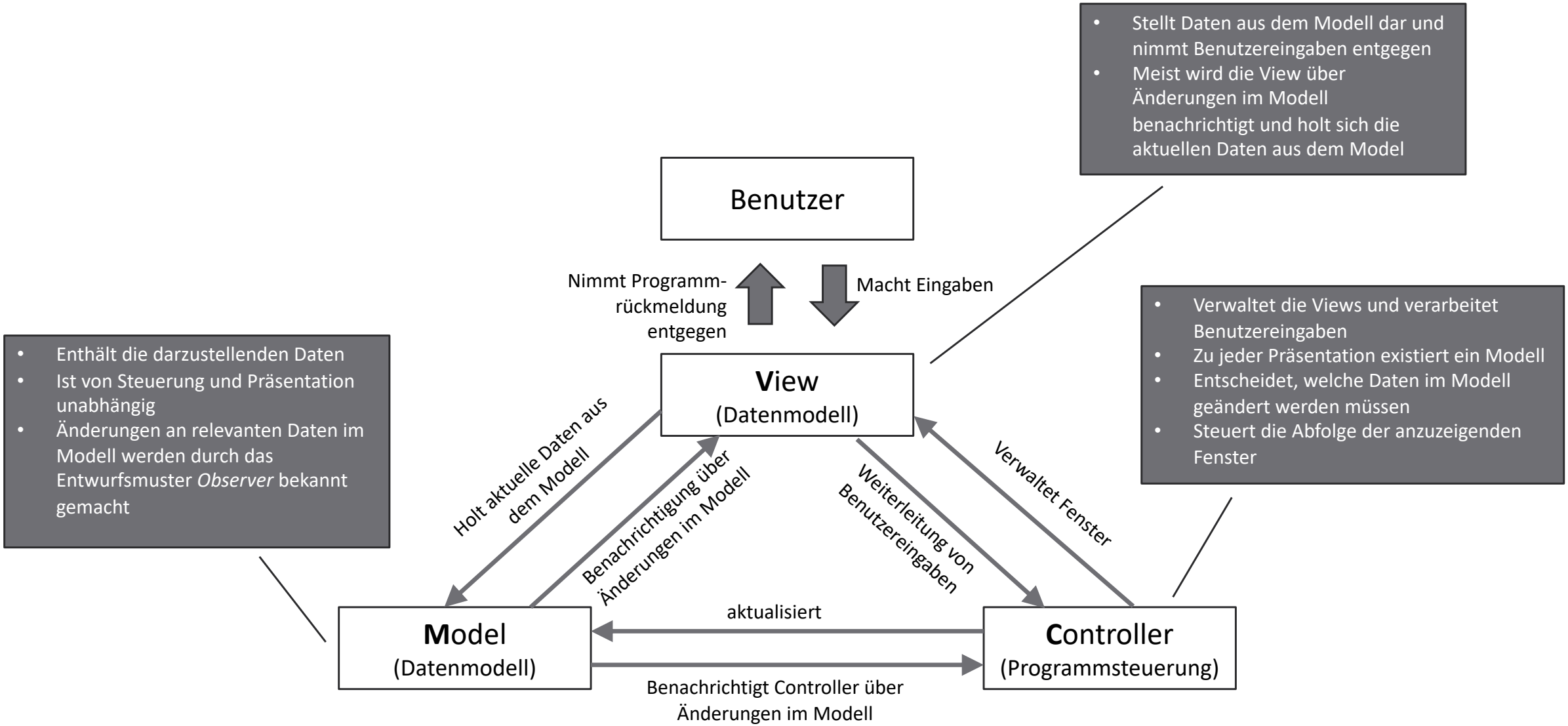


Serviceorientierte Architekturen (SOA)

- Weitere Eigenschaften
  - Services sind im Idealfall zustandsfrei
  - Idempotent
    - Egal wie oft man sie mit den gleichen Eingabedaten aufruft, sie führen immer zu den gleichen Ergebnissen.
  - Serviceschnittstellen besitzen Vertragscharakter und binden Servicekonsumenten an Serviceanbieter (der Serviceimplementierung).
    - Design-by-Contract
  - Serviceanbieter sind austauschbar, sofern die Schnittstellenzusagen eingehalten werden.
  - Services sind ortsunabhängig und können jederzeit und von jedem Ort aus aktiviert werden (Ortstransparenz).

Model-View-Controller
&
Model-View-ViewModel

Model-View-Controller



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

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

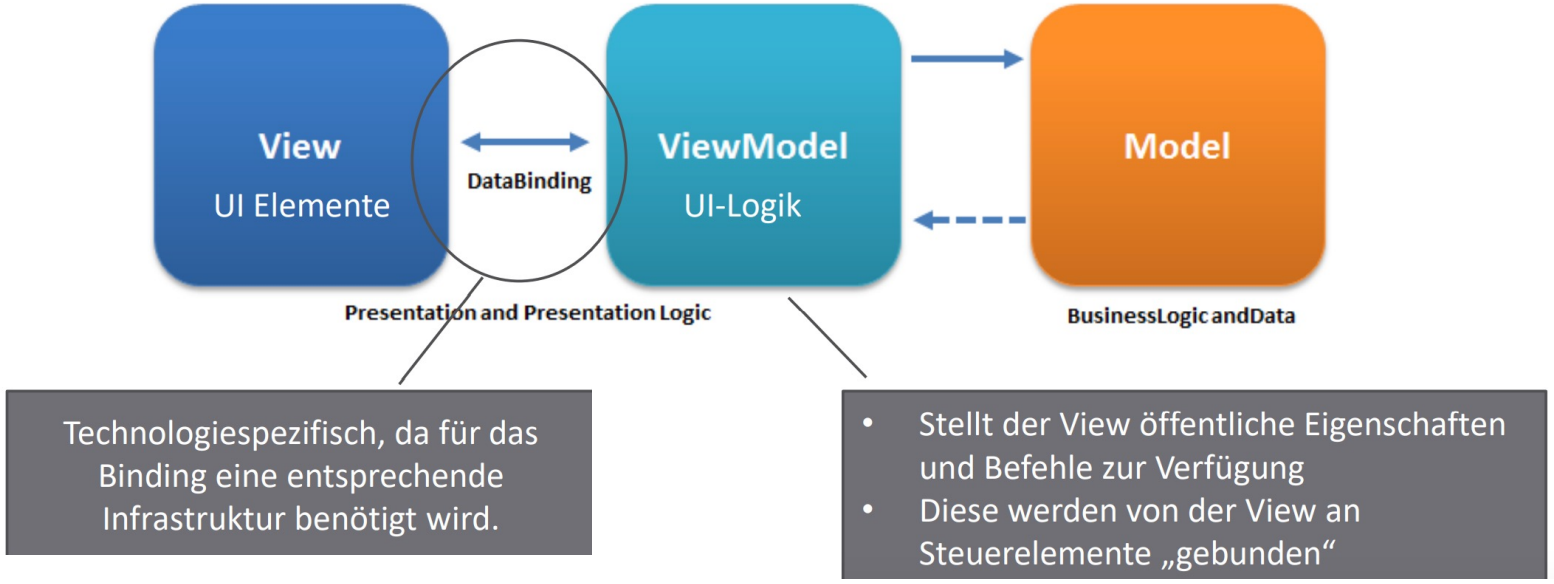
- Verwaltet die Views und verarbeitet Benutzereingaben
- Zu jeder Präsentation existiert ein Modell
- Entscheidet, welche Daten im Modell geändert werden müssen
- Steuert die Abfolge der anzuzeigenden Fenster

Vor- und Nachteile

- Vorteile
  - Trennung von Verantwortlichkeiten
  - Wiederverwendbarkeit
  - Parallele Entwicklung
  - Bessere Testbarkeit
- Nachteile
  - Komplexität
  - Verständnis und Implementierung
  - Zusätzlicher Code
  - Keine Modularisierung nach fachlichen Gesichtspunkten

Model-View-ViewModel

- 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



Model-View-ViewModel

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

- Vorteile
  - Klare Trennung von Verantwortlichkeiten
    - Trennung UI-Design und klassischem Source Code
  - Verbesserte Testbarkeit
    - Anwendung kann unabhängig von der View getestet werden

Model-View-ViewModel

Model

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}
```

View

```
<Window x:Class="MVVMExample.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <StackPanel>
            <TextBox Text="{Binding Name, Mode=TwoWay}" Width="200" Margin="10"/>
            <TextBox Text="{Binding Age, Mode=TwoWay}" Width="200" Margin="10"/>
        </StackPanel>
    </Grid>
</Window>
```

ViewModel

```
public class PersonViewModel : INotifyPropertyChanged
{
    private Person _person;

    public PersonViewModel()
    {
        _person = new Person();
    }

    public string Name
    {
        get { return _person.Name; }
        set
        {
            if (_person.Name != value)
            {
                _person.Name = value;
                OnPropertyChanged();
            }
        }
    }

    public int Age
    {
        get { return _person.Age; }
        set
        {
            if (_person.Age != value)
            {
                _person.Age = value;
                OnPropertyChanged();
            }
        }
    }
```

Event Sourcing

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 sequenzielle Wiedergabe der Ereignisse rekonstruiert

Grundprinzipien

Ereignisse Als Datenquelle

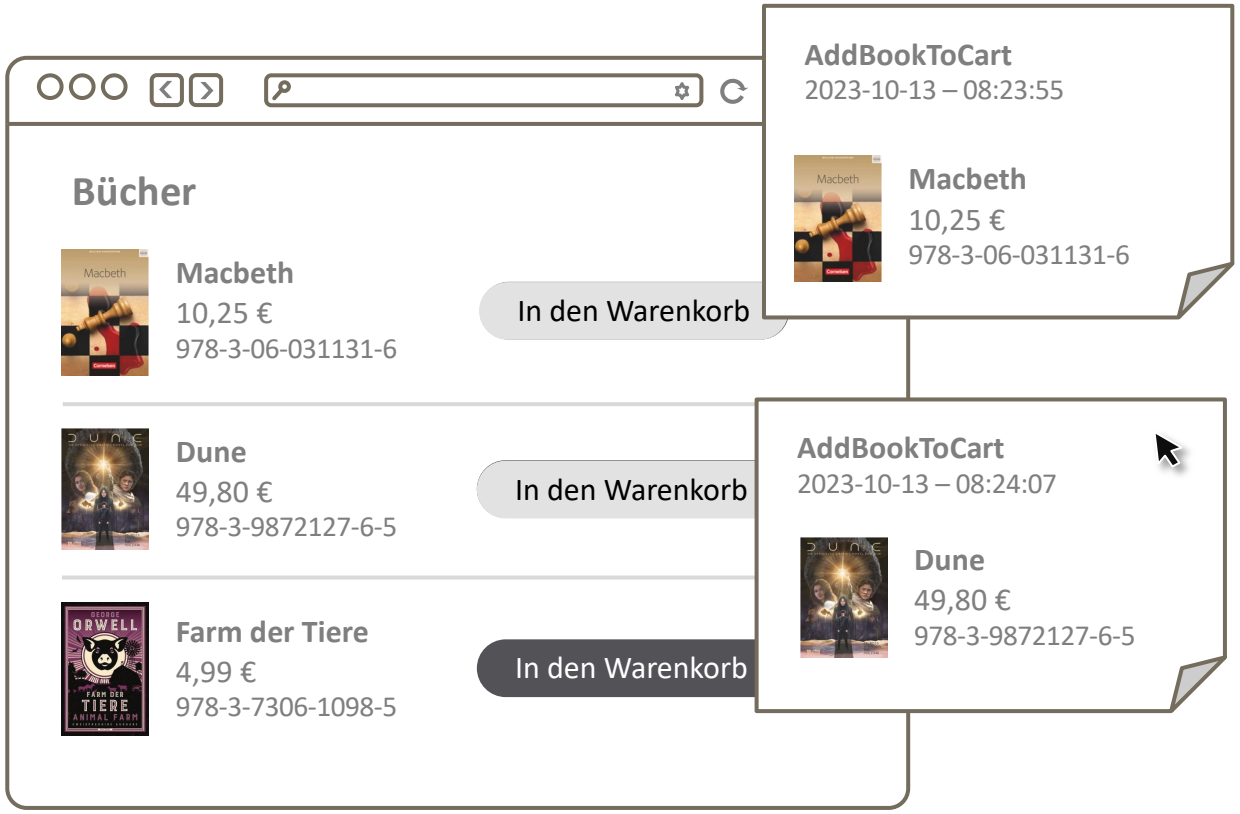
Event-Store

Zustandsrekonstruktion

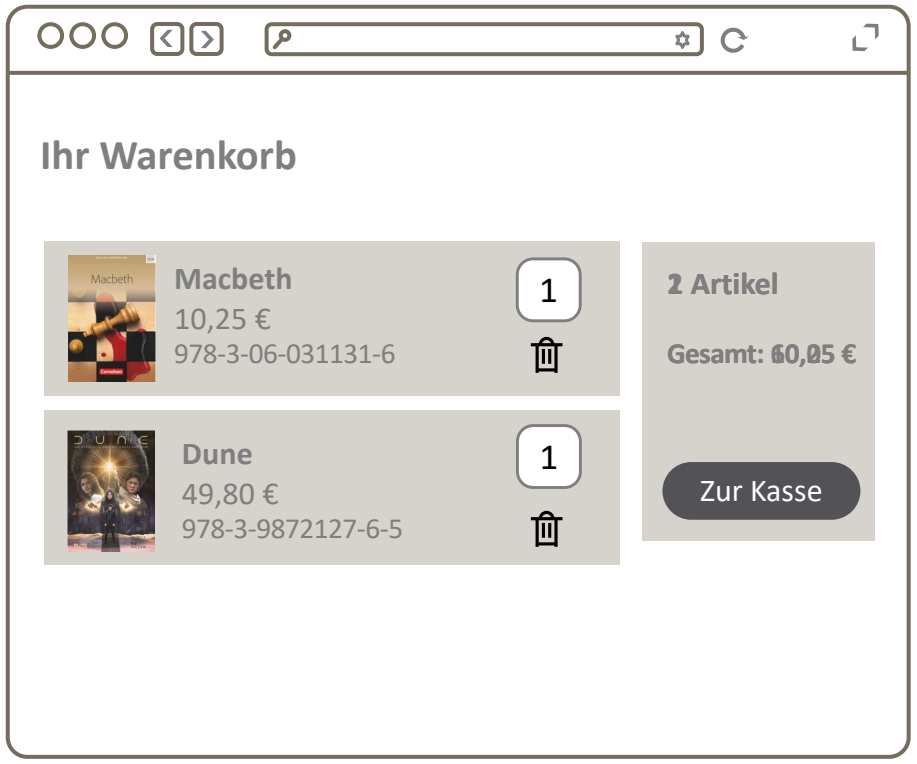
Unveränderlichkeit der Daten

Historische Daten und Zeitreisen

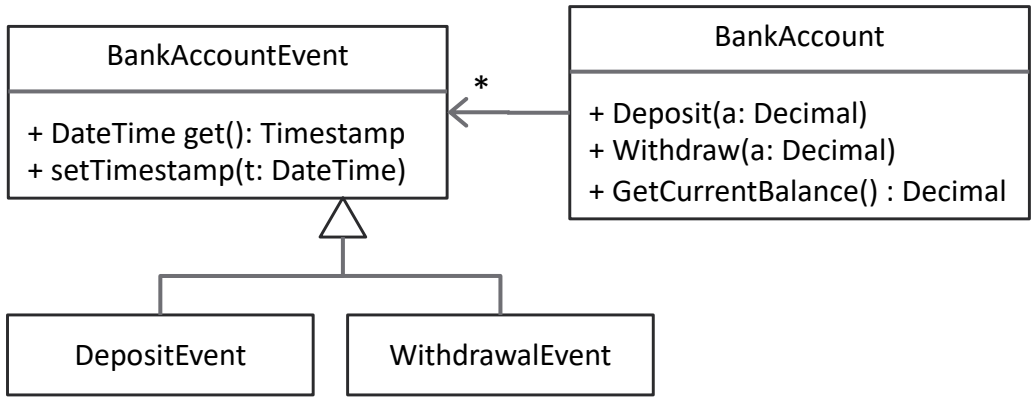
Beispiel Online-Shop



Beispiel Online-Shop



Beispiel-Implementierung in C# Bankkonto




```
public class BankAccountEvent {
    2 Verweise
    public DateTime Timestamp {get; set; }
    4 Verweise
    public decimal Amount { get; set; }
}
```

```
public class DepositEvent : BankAccountEvent {

}
```

```
public class WithdrawalEvent : BankAccountEvent {

}
```

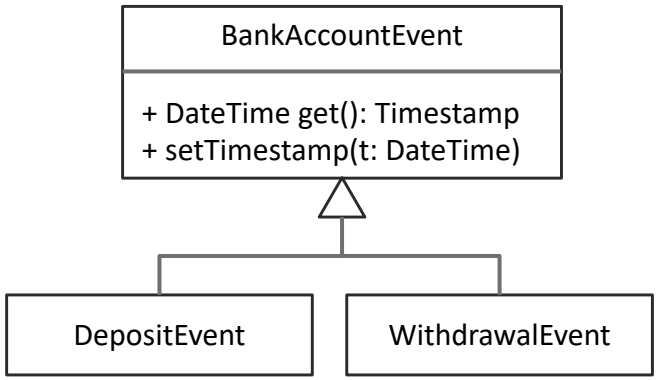
```
public class BankAccount {
    3 Verweise
    private readonly List<BankAccountEvent> events = new List<BankAccountEvent>();

    // Methode zum Einzahlen
    2 Verweise
    public void Deposit(decimal amount) {
        var depositEvent = new DepositEvent { Timestamp = DateTime.UtcNow, Amount = amount };
        events.Add(depositEvent);
    }

    // Methode zum Abheben
    1 Verweis
    public void Withdraw(decimal amount) {
        var withdrawalEvent = new WithdrawalEvent { Timestamp = DateTime.UtcNow, Amount = amount };
        events.Add(withdrawalEvent);
    }

    // Methode zur Berechnung des aktuellen Kontostands aus den Ereignissen
    1 Verweis
    public decimal GetCurrentBalance() {
        decimal balance = 0;
        foreach (var ev in events) {
            if (ev is DepositEvent depositEvent) {
                balance += depositEvent.Amount;
            } else if (ev is WithdrawalEvent withdrawalEvent) {
                balance -= withdrawalEvent.Amount;
            }
        }
        return balance;
    }
}
```

Beispiel-Implementierung in C# Bankkonto




```
1   public class BankAccountEvent {
        2 Verweise
2       public DateTime Timestamp {get; set; }
        4 Verweise
3       public decimal Amount { get; set; }
4   }
```

```
1   public class DepositEvent : BankAccountEvent {
2
3   }
```

```
1   public class WithdrawalEvent : BankAccountEvent {
2
3   }
```

```
3   class Program
4   {
        0 Verweise
5       static void Main(string[] args)
6       {
7           // Erstelle ein Bankkonto
8           var account = new BankAccount();
9
10          // Einzahlung von $100
11          account.Deposit(100);
12
13          // Abhebung von $50
14          account.Withdraw(50);
15
16          // Einzahlung von $200
17          account.Deposit(200);
18
19          // Abfrage des aktuellen Kontostands
20          decimal balance = account.GetCurrentBalance();
21          Console.WriteLine($"Current balance: ${balance}");
22      }
23  }
```

Umsetzungsmöglichkeiten eines Event Stores

- Relationale Datenbanken
- NoSQL-Datenbanken
- Spezielle Event-Store Frameworks, wie z.B.:
  - EventStoreDB (https://www.eventstore.com/eventstoredb)
  - Riak (https://docs.riak.com/index.html)
- Cloud-basierte Speicherlösungen
- Eigenentwickelte Lösungen

Nachteile & Herausforderungen

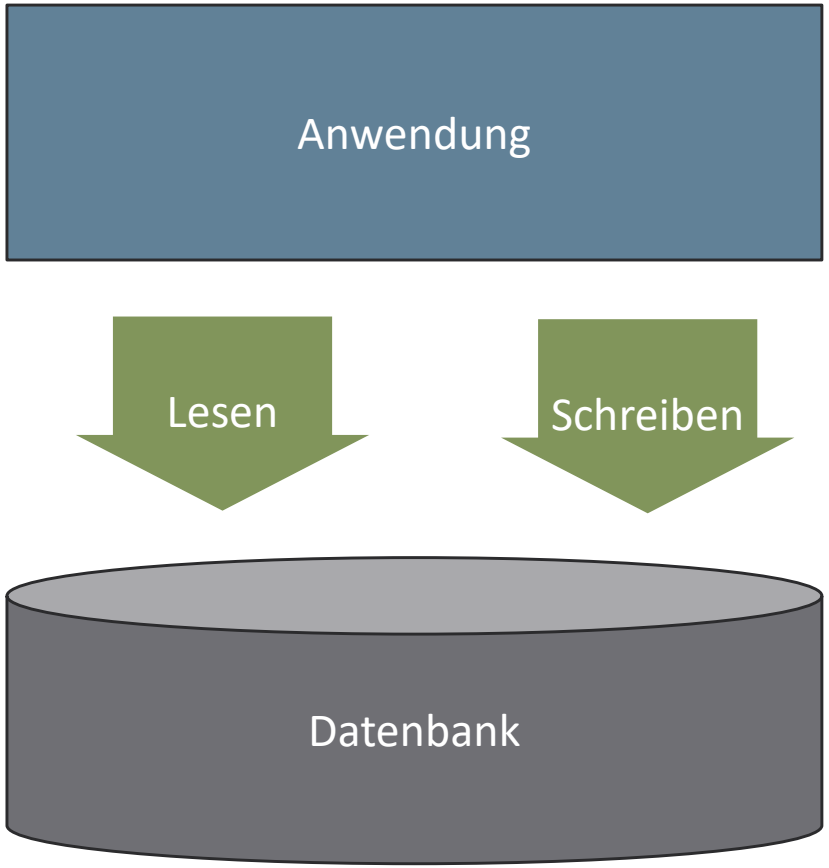
Komplexität	Skalierbarkeit	Wartung und Datenbereinigung	Leseprobleme	Rückwirkende Änderungen
• Event-Logs, Event-Verarbeitung und Zustands-Wiederherstellung	• Große Mengen von Ereignissen kann Herausforderung für Skalierbarkeit werden	• Verwaltung, Wartung und ggf. Bereinigung großer Mengen von Event-Daten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

Einsatzfelder

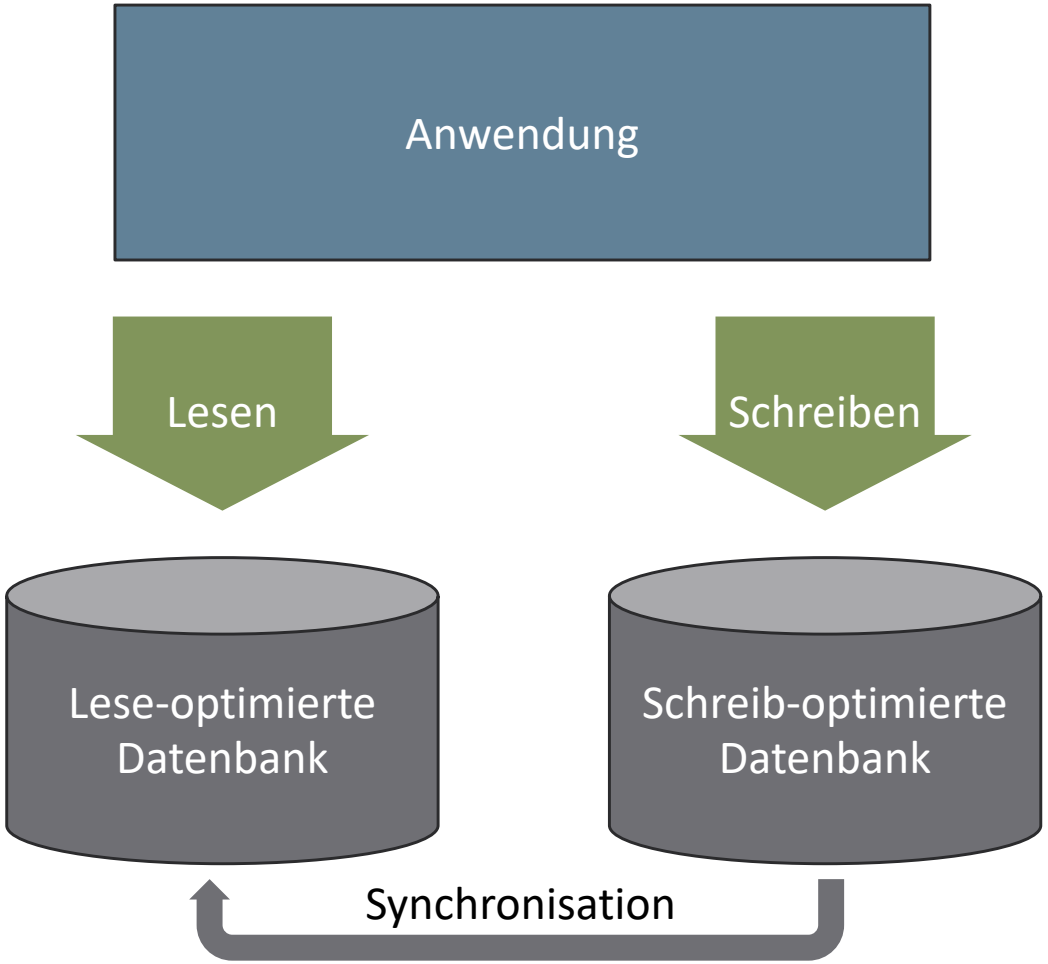
- Anwendung insbesondere in Bereichen, in denen eine nachvollziehbare Historie von Datenänderungen erforderlich ist
- Finanz- und Bankenwesen
- E-Commerce
- Gesundheitswesen
- Logistik und Lieferkettenmanagement
- IoT (Internet of Things)
- Soziale Netzwerke und Community-Plattformen

CQRS

CQRS
Command Query Responsibility Segregation



CQRS
Command Query Responsibility Segregation



- Grundgedanke
  - Verantwortlichkeiten für Lese- und Schreiboperationen in einer Anwendung zu trennen
- Vorteile
  - Optimierte Daten-Schemata für jede Seite
  - Einfachere Abfragen
  - 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 denormalisierten Datenbanken
- Müssen 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 den Einsatz von CQRS laut Microsoft

- Sehr großer paralleler Zugriff auf Daten
- Szenarien, in denen ein Team sich dediziert um die Leseseite und ein Team sich dediziert um 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

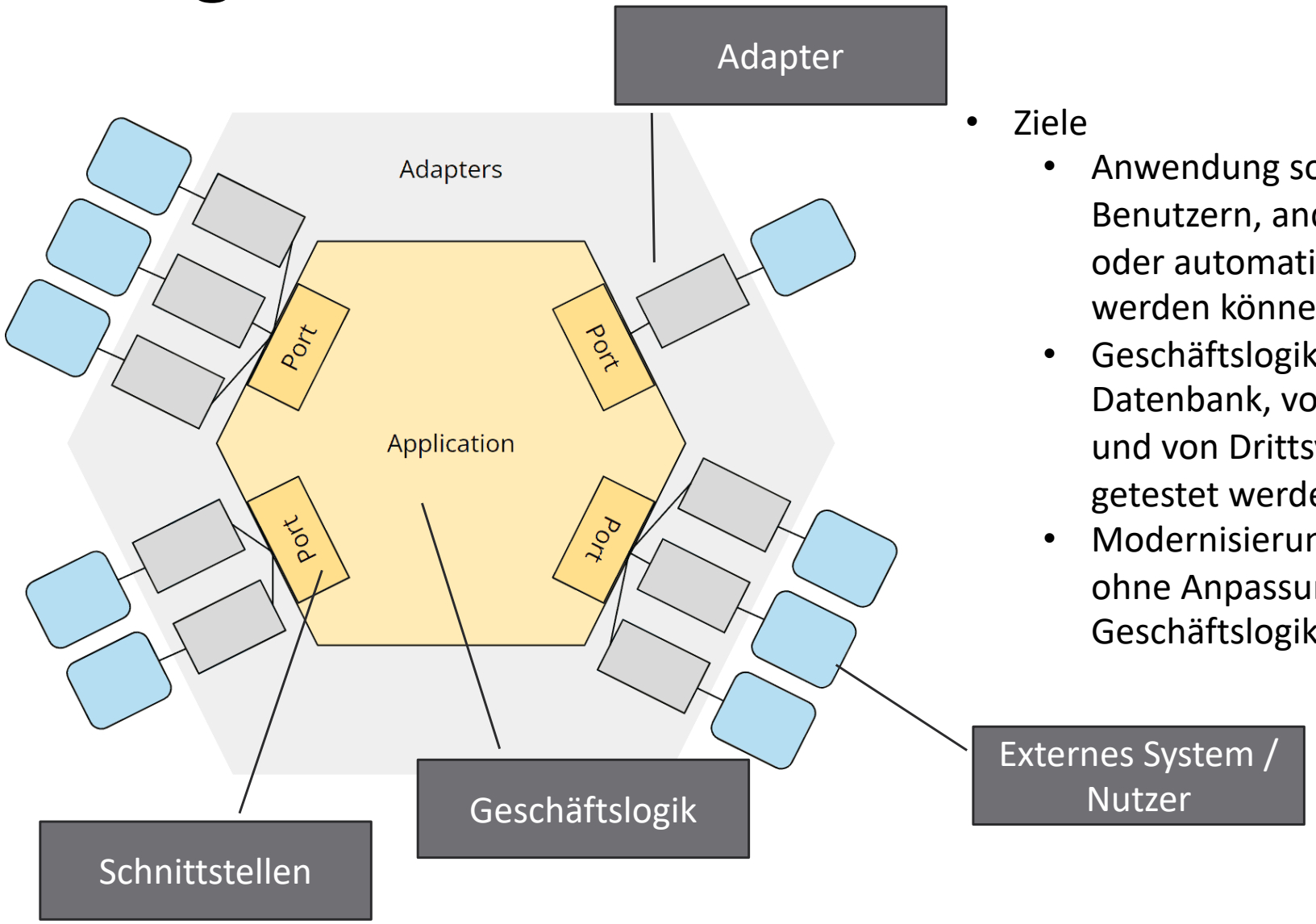
Das Architekturmuster wird (von Microsoft) explizit nicht empfohlen, wenn

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

Quelle: Microsoft

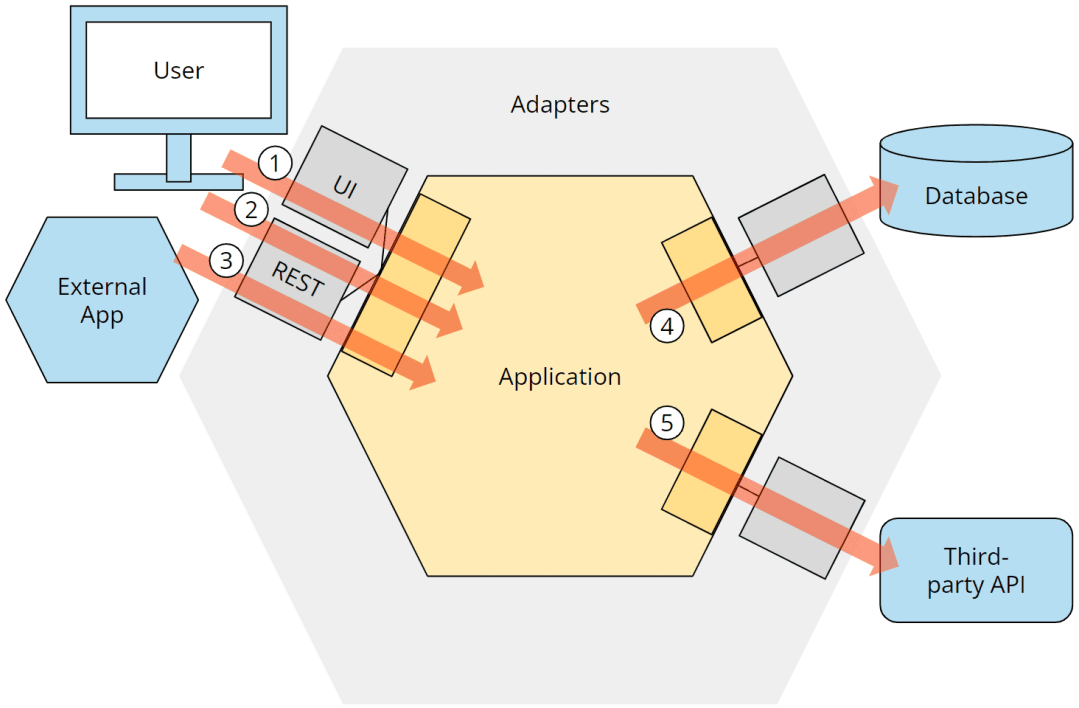
Hexagonale Architektur

Hexagonale Architektur



- Ziele
  - Anwendung soll gleichermaßen von Benutzern, anderen Anwendungen oder automatisierten Tests gesteuert werden können
  - Geschäftslogik soll isoliert von der Datenbank, von sonstiger Infrastruktur und von Drittsystemen entwickelt und getestet werden können
  - Modernisierung der Infrastruktur soll ohne Anpassungen an der Geschäftslogik möglich sein

Hexagonale Architektur



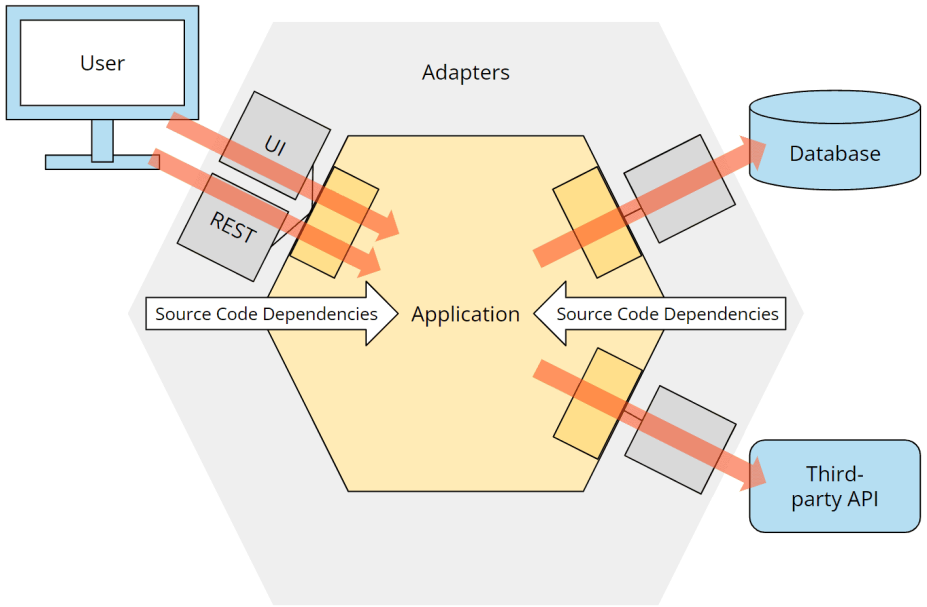
Die Abbildung zeigt eine beispielhafte Anwendung, die

1. durch einen User über ein User Interface gesteuert wird,
2. durch einen User über eine REST-API gesteuert wird,
3. durch eine externe Anwendung über dieselbe REST-API gesteuert wird,
4. eine Datenbank ansteuert und
5. eine externe Anwendung ansteuert

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 keiner 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 Efinder (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 **unabhängigen Prozessen** 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.

https://de.wikipedia.org/wiki/Microservices

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 centralized management of these services, which **may be written in different programming languages** and **use different data storage technologies.**

Martin Fowler (https://martinfowler.com/articles/microservices.html)

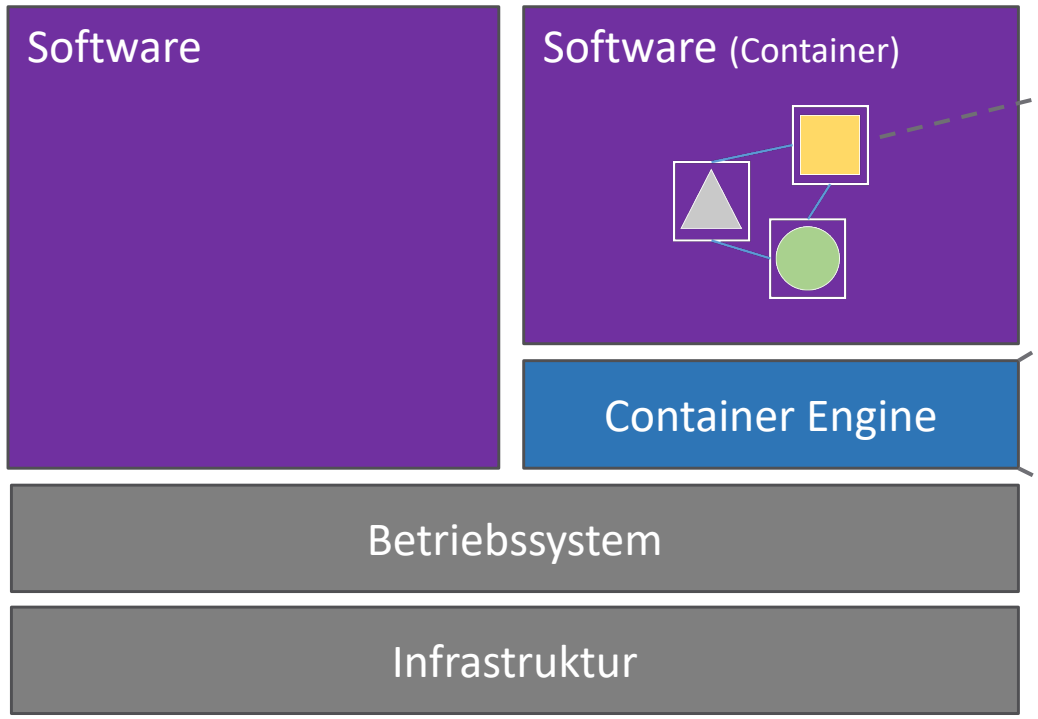
Microservices sind keine Technologie, sondern ein Architekturmuster.

Merkmale Zusammengefasst

- Unabhängigkeit einzelner Microservices
  - Unabhängige Prozesse
  - Sprachunabhängige Programmierschnittstellen
  - Unabhängige Release-Zyklen
- Modularer Aufbau
- Kontinuierliche Bereitstellung und Deployment
- Fehlerisolierung
- Möglichkeit der Nutzung
  - unterschiedlicher Programmiersprachen je Microservice
  - unterschiedlicher Datenhaltungs-Technologien

Container Engine

Container



Docker Container (laufende Microservices)

docker

https://www.docker.com/

Docker ist eine Plattform für die Containerverwaltung und Imageerstellung, die es ermöglicht, mit Containern unter Linux und Windows zu arbeiten.

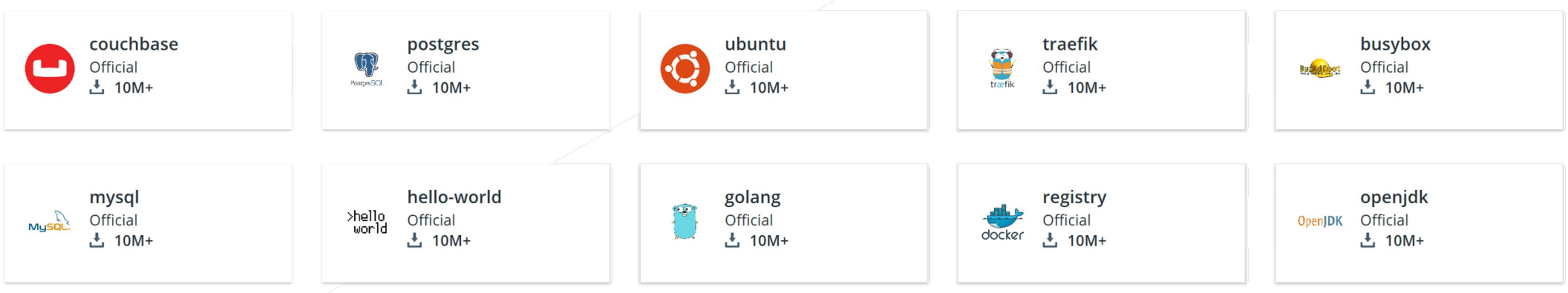
Docker

Container
Die aktive Instanz eines Images. Der Container wird also gerade ausgeführt und ist beschäftigt. Sobald der Container kein 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 instanziiert werden.

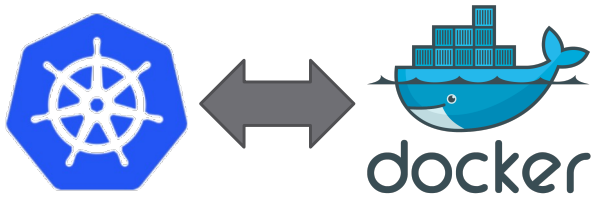
Dockerfile
Eine Textdatei, welche mit verschiedenen Befehlen ein Image beschreibt. Diese werden bei der Ausführung abgearbeitet. Ein Dockerfile beschreibt einen Container so, dass er auf einem anderen Rechner wieder genau gleich aufgebaut werden kann. Er automatisiert die Erstellung von Images.

Registry
Dient der Verwaltung von Images. Beispiel: Docker Hub (https://www.docker.com/products/docker-hub)



Docker & 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.



<u>Vereinfacht zusammengefasst:</u>

- Eine Kubernetes Cluster besteht aus einem Netzwerk von *Nodes*
  - Master Node(s)
  - Worker Nodes
    - Hier werden die Container (z.B. Docker-Container) ausgeführt (gekapselt durch sogenannte *Pods*)
- `kubectl`
  - Tool zur Steuerung des Kubernetes Cluster Managers
  - 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.

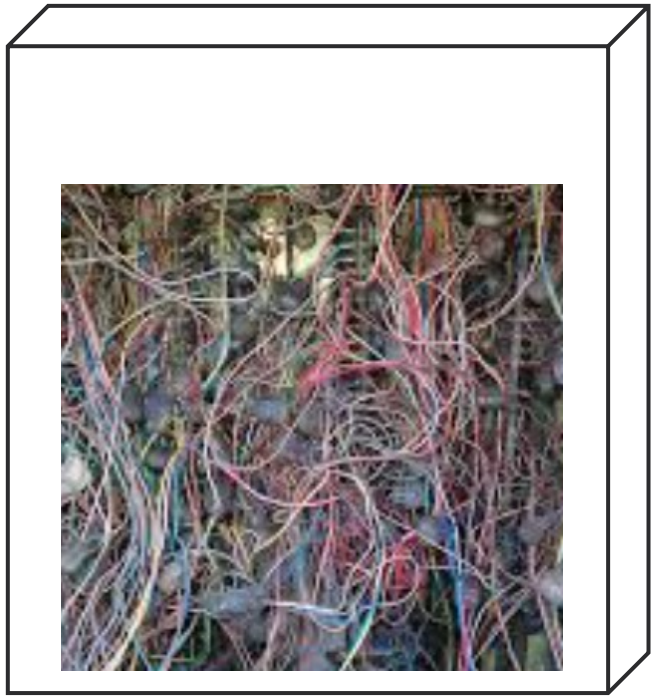
Kritik an Microservice-Architekturen

An der ursprünglichen Definition gibt es vermehrt Kritik:

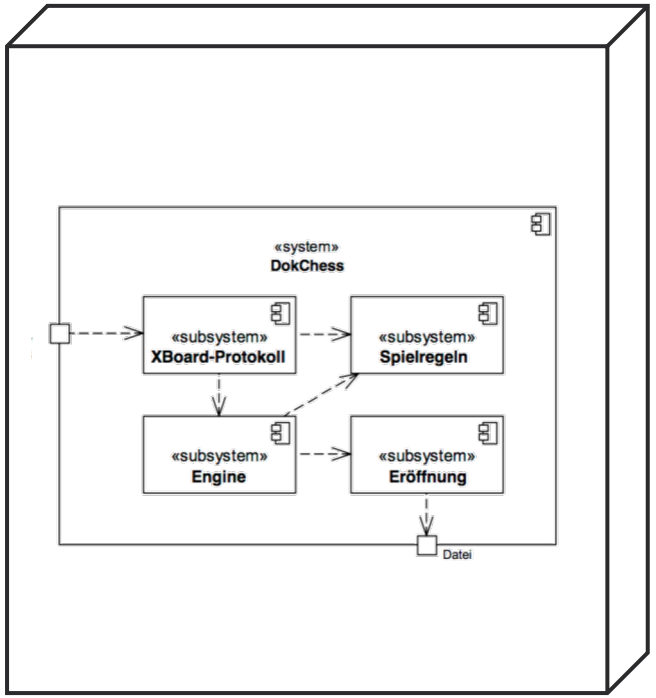
- Komplexität der Verwaltung
- Inter-Service-Kommunikation
- Entwicklungs- und Deployment-Komplexität
- Sicherheitsbedenken
- Fachwissen und Kultur
- Das „Micro“ in Microservices suggeriert, dass kleine Services anzustreben sind
  - Viele kleine Services erzeugen zusätzliche und ggf. unnötige Komplexität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

Monolith -> Modulith -> Microservices

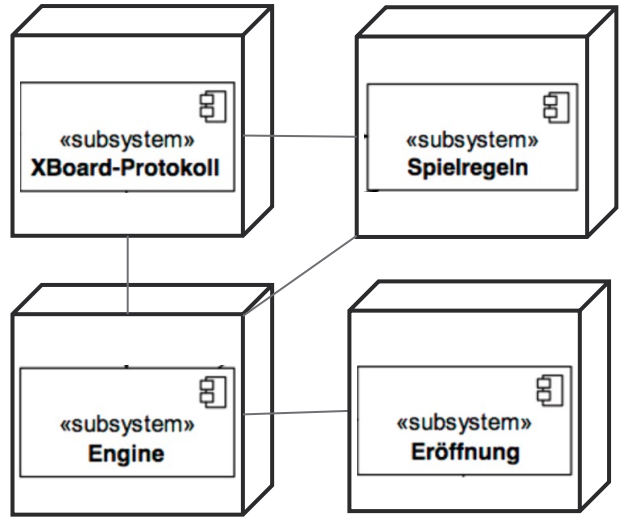
Monolith



Modulith



Microservices



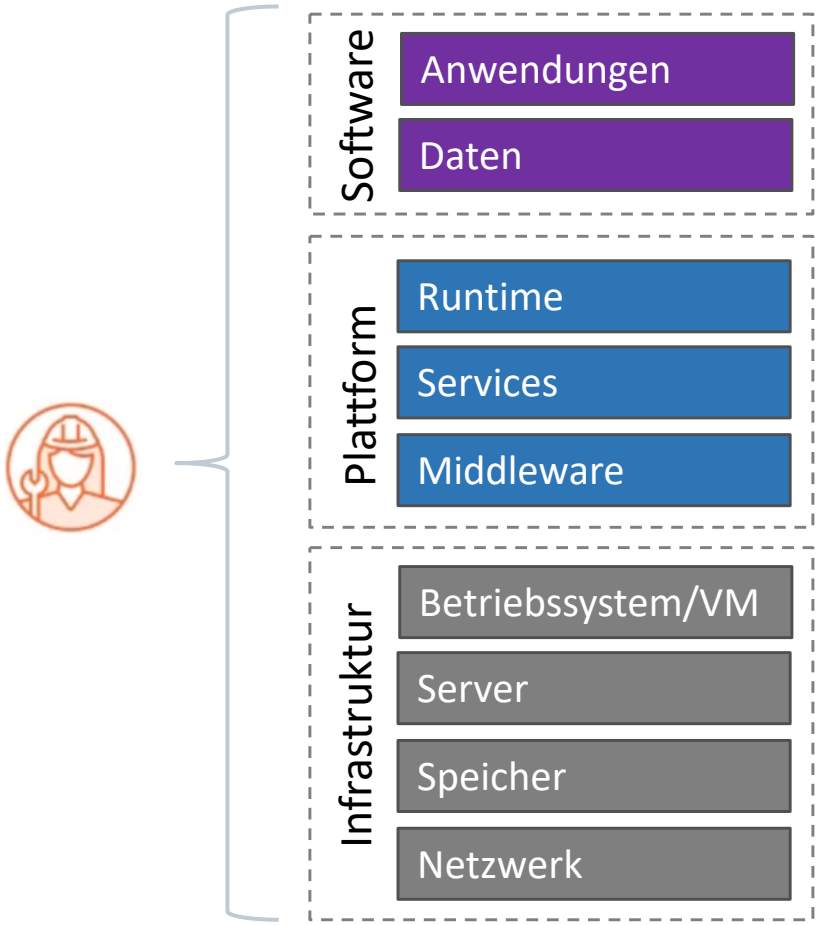
Häufig ist ein Modulith einer Microservice-Architektur vorzuziehen

Modulith vs. Microservice

- Vorteile eines Modulithen gegenüber einem Microservice
  - Geringere Komplexität
  - Einfache Entwicklungsumgebung
  - Geringerer operationaler Overhead
  - Kosteneffizienz
  - Einfachere Datenverwaltung
- Wann von einem Modulithen zu einer Microservice-Architektur wechseln?
  - Skalierbarkeit
  - Unabhängige Entwicklung und Bereitstellung
  - Technologische Vielfalt
  - Geschäftliche Anforderungen
  - Zuverlässigkeit und Fehlerisolierung
    - (siehe auch: Arten der Kopplung – gemeinsam genutzte Infrastruktur)

Cloud-Servicemodelle

On-Premises (keine Cloud)



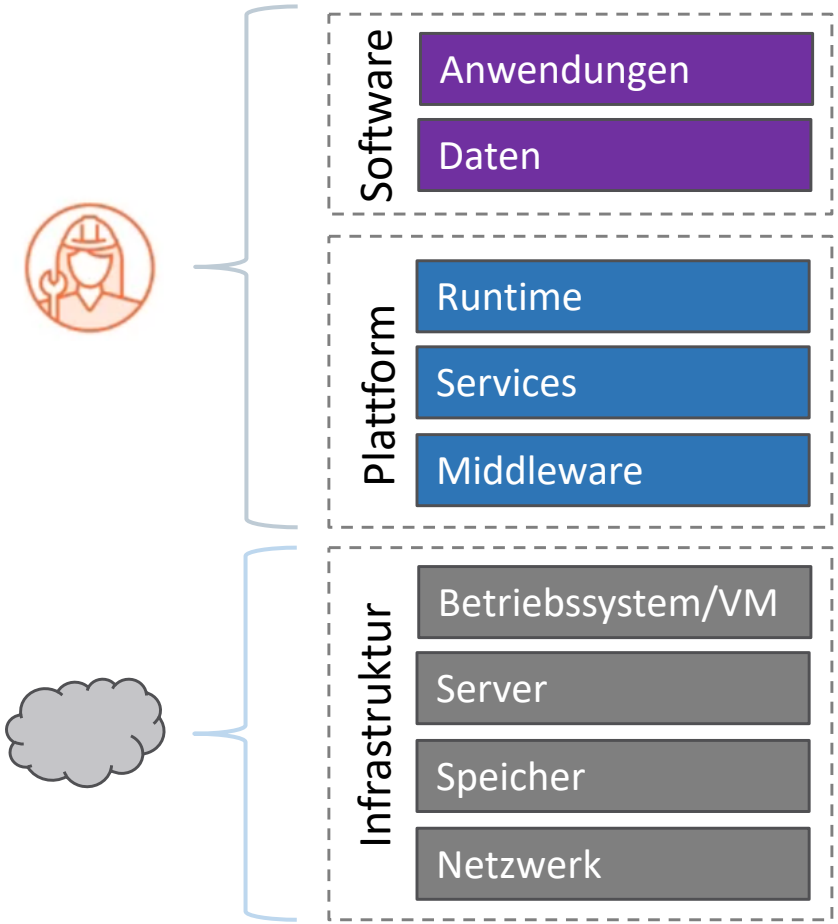
On Premises

Die Verantwortung für

- den Aufbau und den Betrieb der Infrastruktur,
- die Bereitstellung der Entwicklungsumgebung, Komponenten, Laufzeitumgebungen (z.B. JRE, OSGi, ...),
- das managen der Daten und Anwendungen

liegt im eigenen Unternehmen.

Infrastructure as a Service (IaaS)



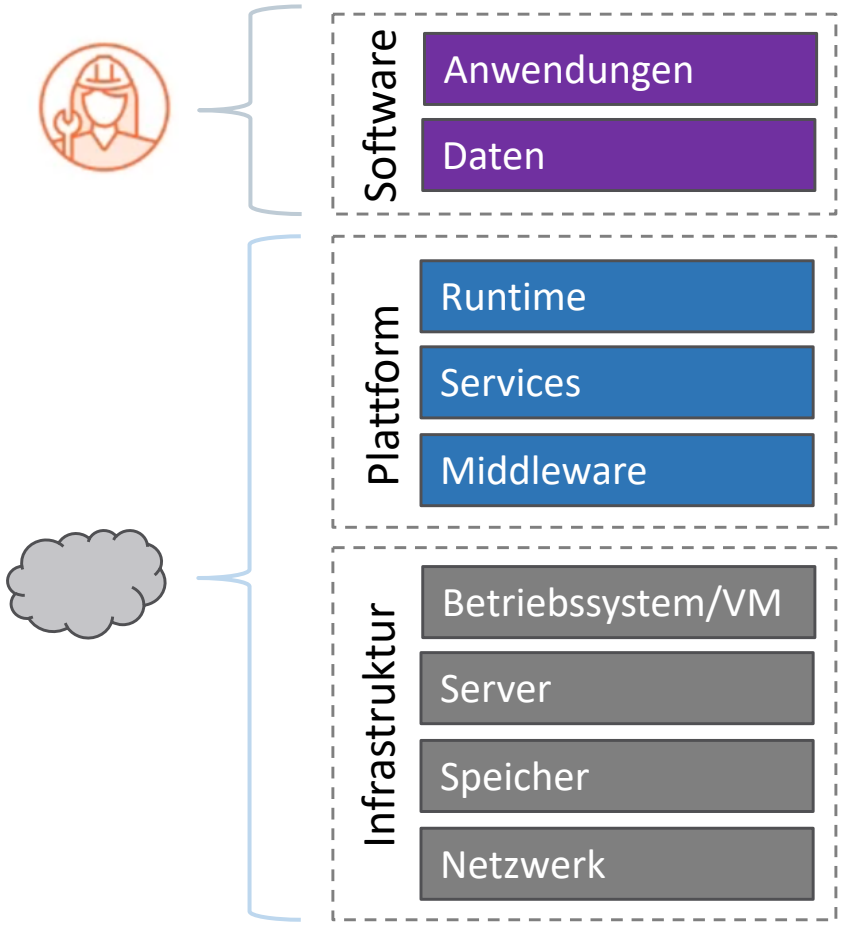
Infrastructure as a Service (IaaS) bedeutet:

- Verantwortung für die Infrastruktur wird an den 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 angemietet werden
- Automatisches Abfangen von Lastspitzen
- Einfache Migration bestehender Microservice-Umsetzungen in die Cloud

Platform as a Service (PaaS)



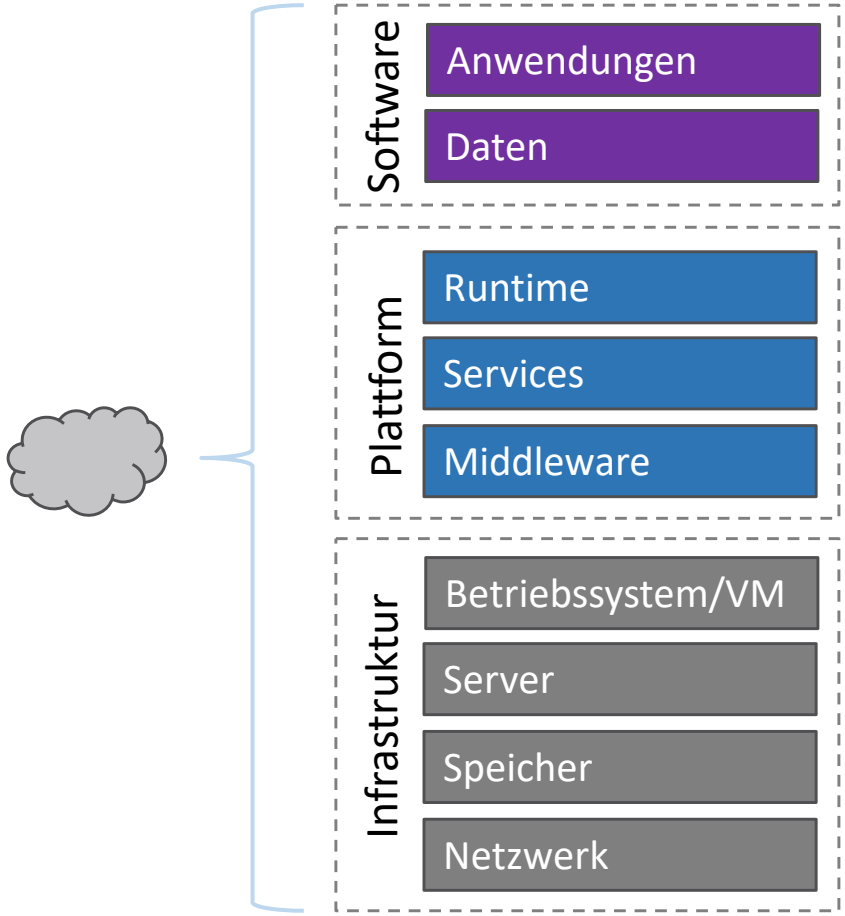
Platform as a Service (PaaS) bedeutet:

- Entwicklungswerkzeuge werden durch Cloud-Anbieter zur Verfügung gestellt
- Nutzung von Softwarekomponenten, welche durch Cloud-Anbieter bereitgestellt und betrieben werden
- Nutzung Cloud-nativer 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)



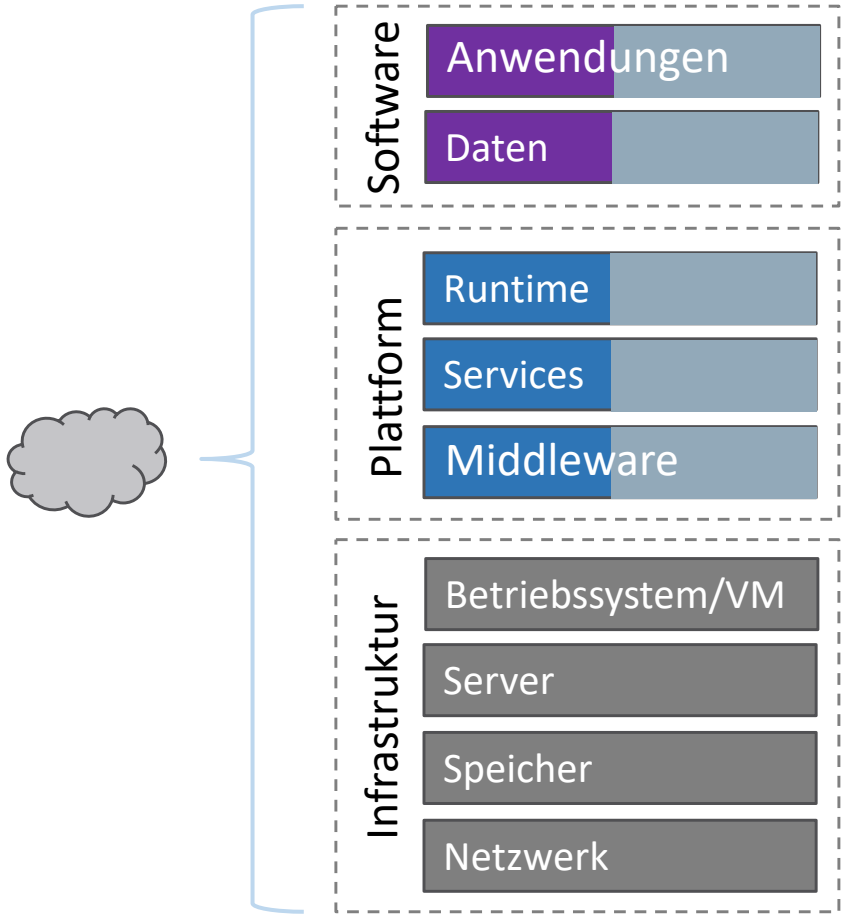
Software as a Service (SaaS) bedeutet:

- Der Cloud-Anbieter stellt ganze Anwendungen bereit
- Softwareentwicklung- und betrieb liegen 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
- Kann mit IaaS und PaaS kombiniert werden – Eine auf PaaS und auf Microservices basierende Software kann SaaS-Anwendungen integrieren
- Fokus auf Auslieferung von Funktionalität an Kunden

Hybride Ansätze



Hybride Ansätze

Beispiele

- Integration von Legacy-Systemen bei gleichzeitiger Nutzung von Standard-Softwarekomponenten aus der Cloud
- Nutzung eigener Infrastruktur für die Datenhaltung bei gleichzeitiger Nutzung von vorgefertigten Machine Learning Algorithmen in der Cloud
- Betrieb von Microservices (z.B. in Form von Docker-Containern) im eigenen Unternehmen und deren Anbindung an Cloud-native Microservice-Angebote
- …

Serverless Computing

Serverless Computing

- Methoden/Funktionen liegen in der Cloud und werden dort ausgeführt.
- Jede Funktion enthält Code und Konfigurationsinformationen (Funktionsname, Ressourcenanforderungen)
- Können beim Auftreten bestimmter 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. Z.B.:

Serverless Computing

Auch beim sogenannten „Serverless Computing“ gibt es Server!

Hauptmerkmale

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

Nachteile

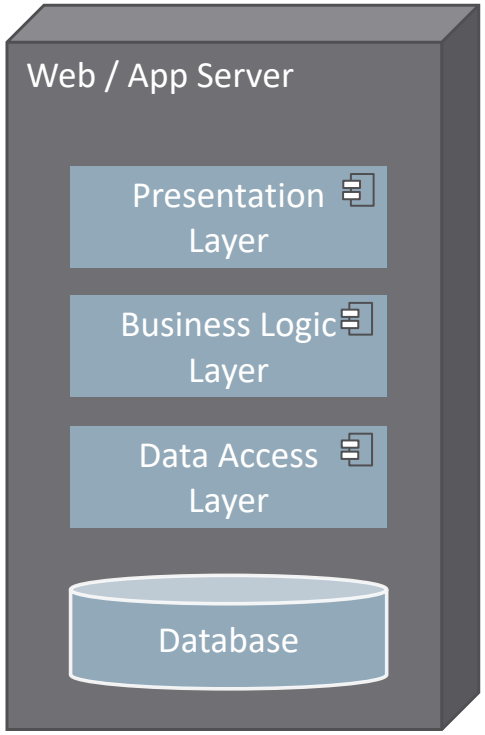
- Kaltstart-Latenz
- State-Management
- Vendor-Lock-in

Deploymentmuster

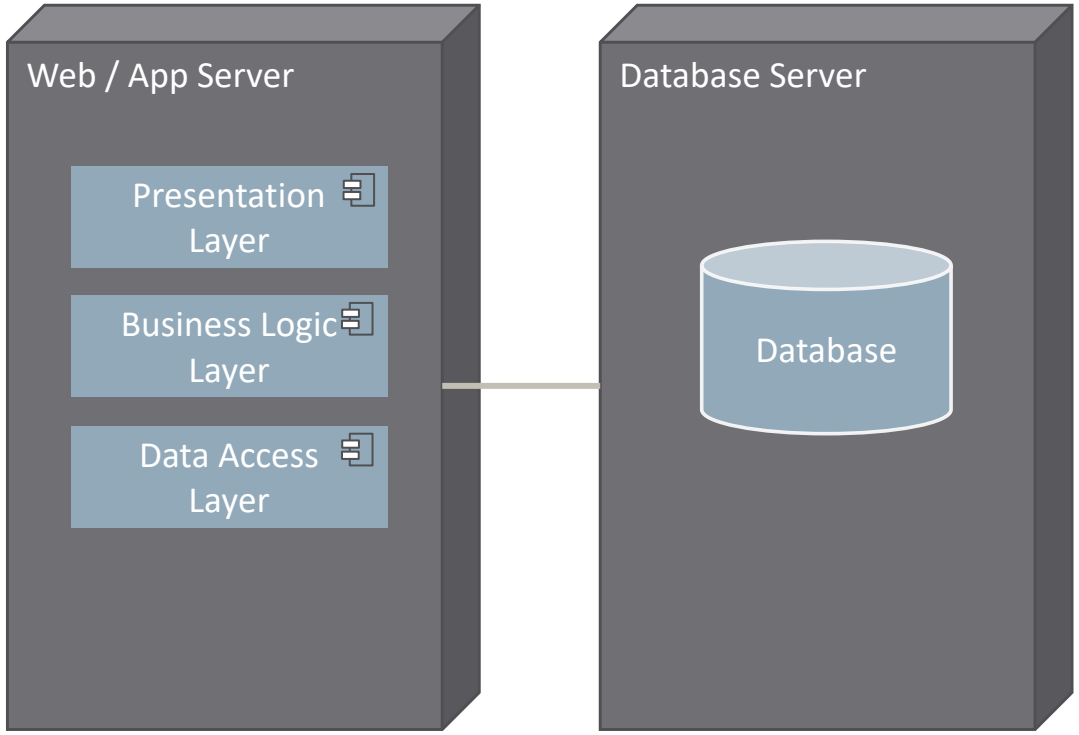
Deploymentmuster

- Bewährtes 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-Server Deployment



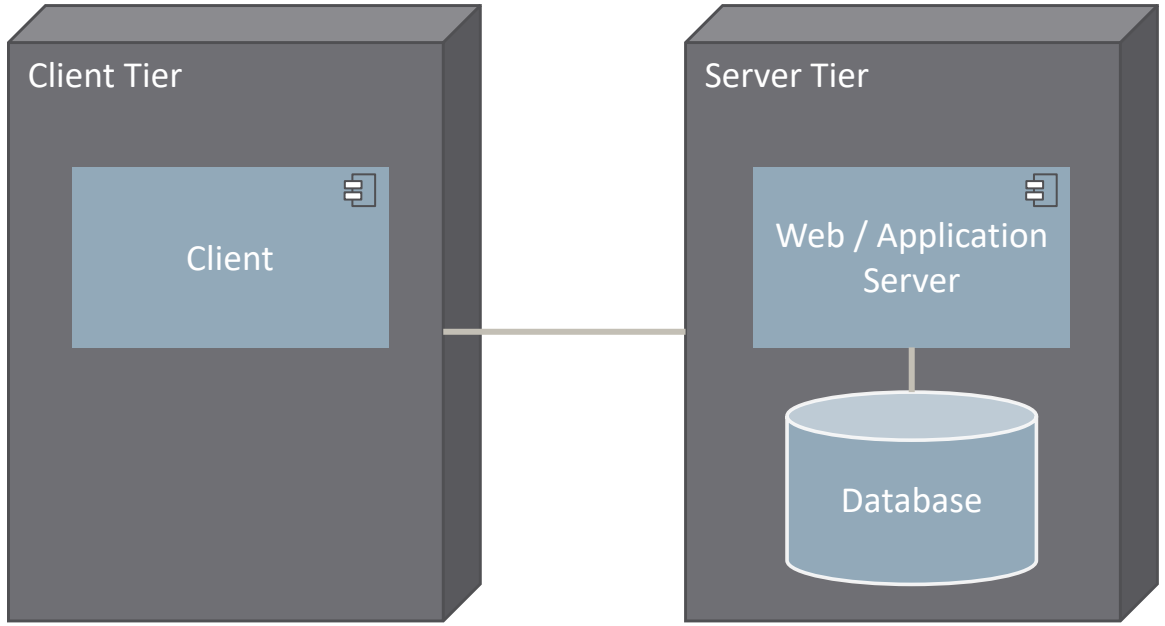
Installation aller Komponenten auf einem Knoten



Manchmal wird auch dieses Deployment noch als Single-Server-Deployment bezeichnet.

Multi-Server Deployment

2-Tier-Deployment



3-Tier-Deployment