



# **Software Engineering I**

## **Kapitel 5: System Design (Grundlagen von Software-Architekturen)**

**Vorlesung für den Studiengang Bachelor Wirtschaftsinformatik**

**Wintersemester 2017 / 2018**

Prof. Dr. Sascha Alda  
([sascha.alda@h-brs.de](mailto:sascha.alda@h-brs.de))

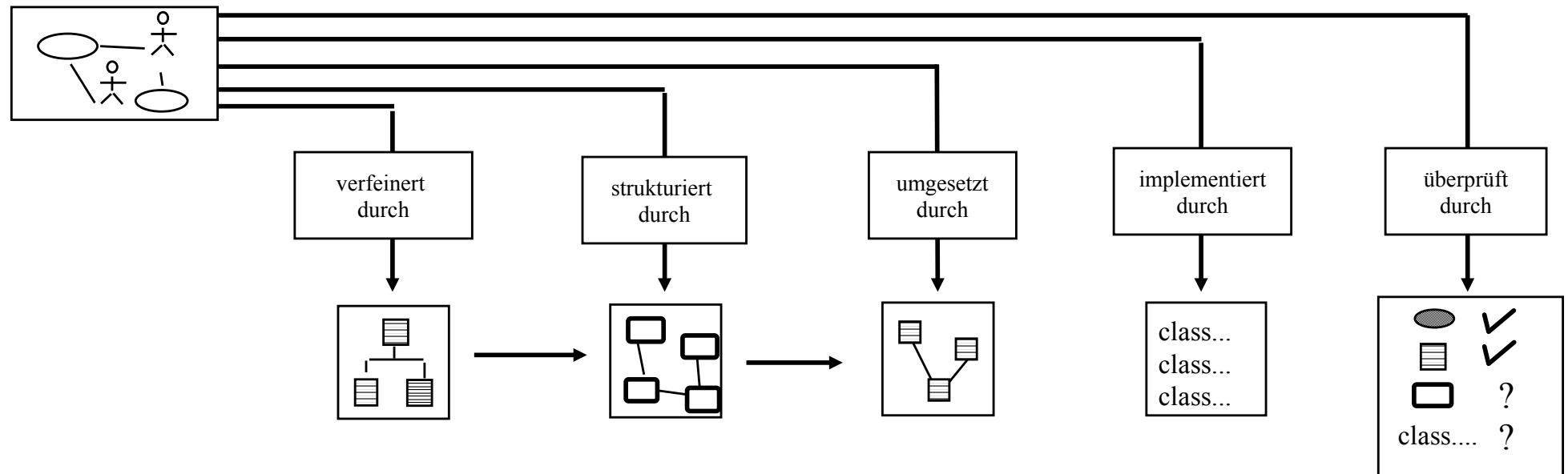
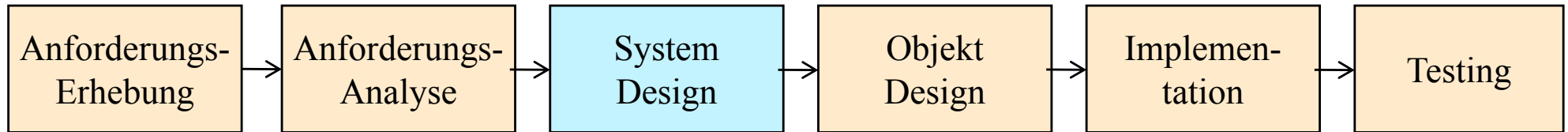


Kapitel	Thema	
1	Einführung ins Software Engineering	✓
2	Software-Prozessmodelle im Software Engineering	✓
3	Modellierung und Erhebung von Anforderungen	✓
4	Objektorientierte Analyse von Anforderungen	✓
<b>5</b>	<b>System-Design (Grundlagen von Software-Architekturen)</b>	
6	Objektorientiertes Design (Grundlagen und Entwurfsmuster)	
7	Testgetriebene Entwicklung mit JUnit	
8	Software-Wartung (Refactoring von Source Code)	
	Gastvortrag Alireza Farnoudi, FA. Congstar, Köln „Software-Entwicklung mit Scrum - ein Erfahrungsbericht“ Januar 2018.	

**Anmerkung: ein Kapitel erstreckt sich über 1-2 Vorlesungen**



# Generischer Software-Prozess nach Bruegge



Anwendungs-fälle  
(Use Case)

Analyse-Modell  
(Application Domain Objects)

**Software-Architektur**  
(Software Architecture)

Design-Modell  
(Solution Domain Objects)

Source Code

Testfälle  
(Test Cases)



## Kapitel 5: System Design (Grundlagen von Software-Architekturen)

1	Wiederholung und Motivation	✓
2	<b>Definition und Eigenschaften einer Software-Architektur</b>	
3	Modellierung von Software-Architekturen mit UML	
4	Grundlegende Architekturmuster	
5	Exkurs: Abbildung von Klassen auf eine Architektur	
6	Zusammenfassung und Ausblick	



# Was ist eine Software-Architektur?

- Eine **Software-Architektur** beschreibt die Dekomposition (Zerlegung) eines einzelnen Software-Systems.



- **Software-Architekturen** orientieren sich an Software-Architekturstile (auch **Architekturmuster** genannt), die Richtlinien und Vorgaben für einzelne Typen (Arten) von Software-Architekturen machen (→ siehe Abschnitt 4)



# Zusammenfassende Definition Software-Architektur

---

- Eine Software-Architektur beschreibt die Dekomposition eines Software-Systems, die den Vorgaben eines zugehörigen Architekturstils genügt. Die Architektur-Beschreibung umfasst folgende Bestandteile:
  - die grundlegenden **Architektur-Elemente und ihre Schnittstellen**
  - die **Interaktionsbeziehungen** zwischen den Architektur-Elementen
  - die **Architektur-Direktive** der Gesamtarchitektur
  - die charakterisierenden **Kennzahlen** der Gesamtarchitektur

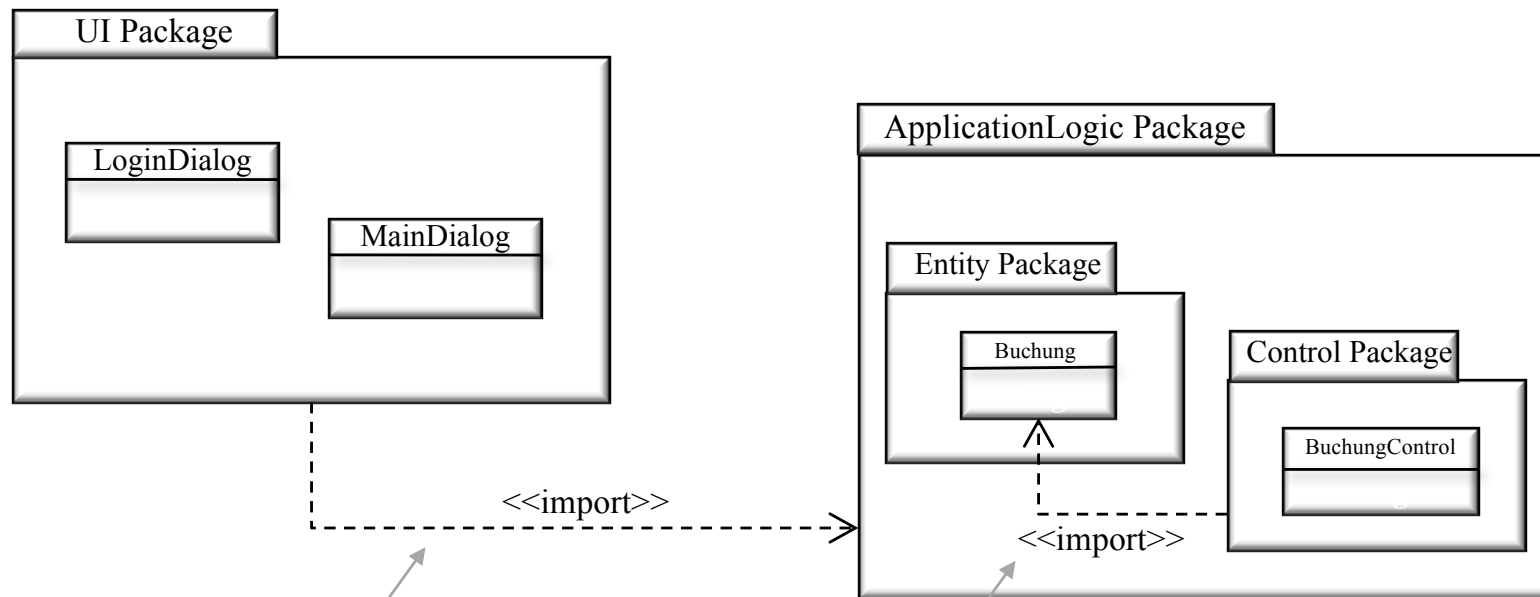


Vorlesung SOA [Alda, 2010]

- Aufgabe des System-Design: Abbildung des Analysemodells (Klassen...) auf eine Software-Architektur



- Die **Architektur-Elemente** beschreiben die atomaren (Klassen) und komplexen Bestandteile (Subsysteme) einer Software-Architektur
- Architektur-Elemente können hierarchisch angeordnet sein
- Modellierung mit Hilfe eines **Paketdiagramms (Package Diagram)** der UML 2.5:
  - Gute Quelle: (Rupp, 2012), Kapitel 7\*



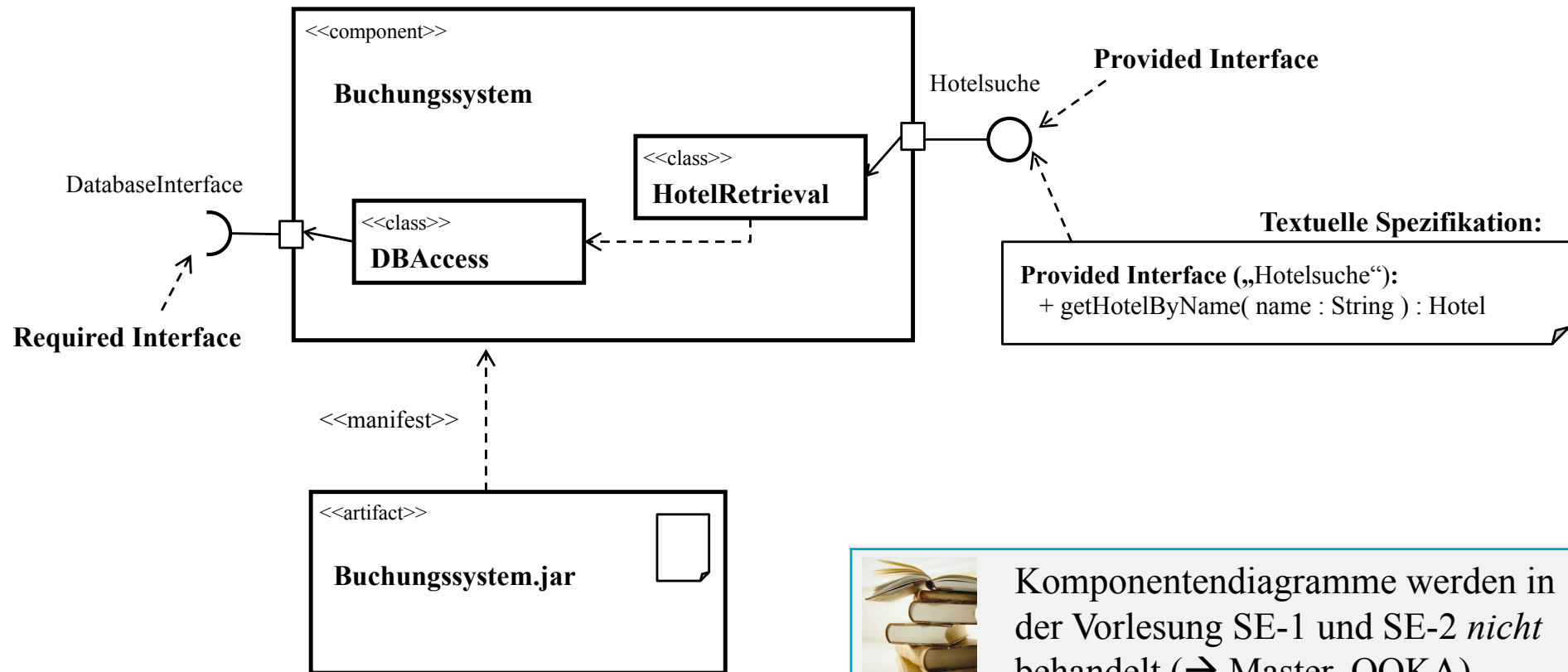
**Package Import (Default)**


**Element Import**

\* Andere Beziehungsvarianten möglich,  
hier ausgelassen, vgl. (Rupp, 2012)



- Der Zugriff auf die Funktionen (insbesondere) der Subsysteme erfolgt über explizite **Interfaces** (dt.: Schnittstellen)
- Explizite Modellierung mit Hilfe von **Komponentendiagramm** der UML 2.5:



 Komponentendiagramme werden in der Vorlesung SE-1 und SE-2 *nicht* behandelt (→ Master, OOKA)





# Architektur-Elemente – Implementierung in Java

- Die Architektur-Elemente einer Architektur lassen sich auf bestehende Programmiersprache-Konzepte (meist) direkt anwenden. Beispiel **Java**:

```
package org.softwareforen.Buchungssystem;

public class HotelRetrieval implements Hotelsuche {

    public Hotel getHotelByName ( String Name ) throws BadInput {
        // Some Sample Code

        return address;
    }

    boolean checkName( String Name ){
        return true;
    }
    ...

    private class DBAccess{
        // Some Sample Code
    }
}
```

**Package (= Subsystem)**

**Klasse (= Modul)**

**Öffentliches Interface des Subsystems (Vorgabe durch Interface)**

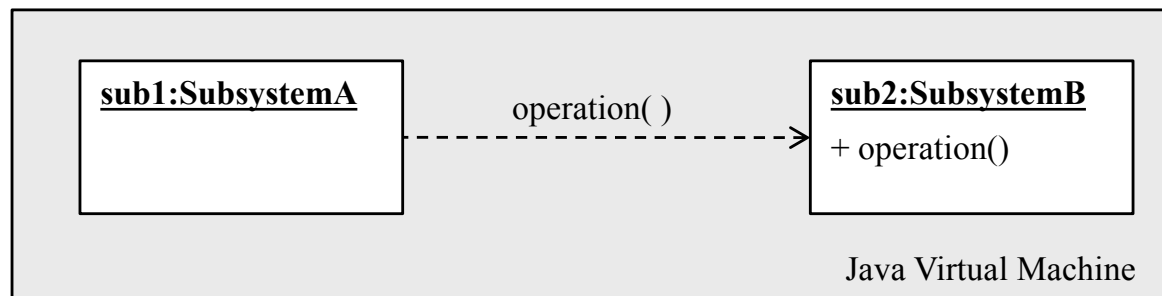
**Methode nur innerhalb des Packages sichtbar**

**Internal Class (= Sub-Modul), außerhalb nicht sichtbar!**

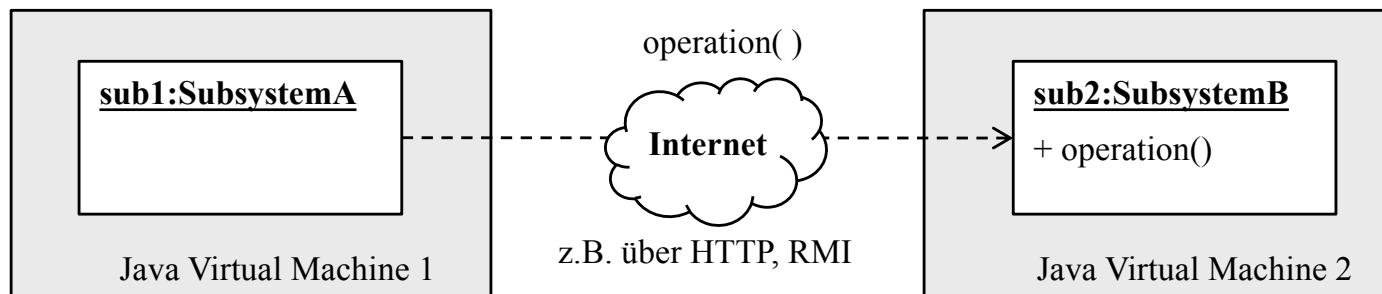


# Interaktionsbeziehungen – Mögliche Ausprägungen

- Lokale Interaktionsbeziehungen:
  - Lokale Methodenaufrufe zwischen zwei Subsystemen innerhalb eines Prozesses (z.B. in einer Java Virtual Machine)

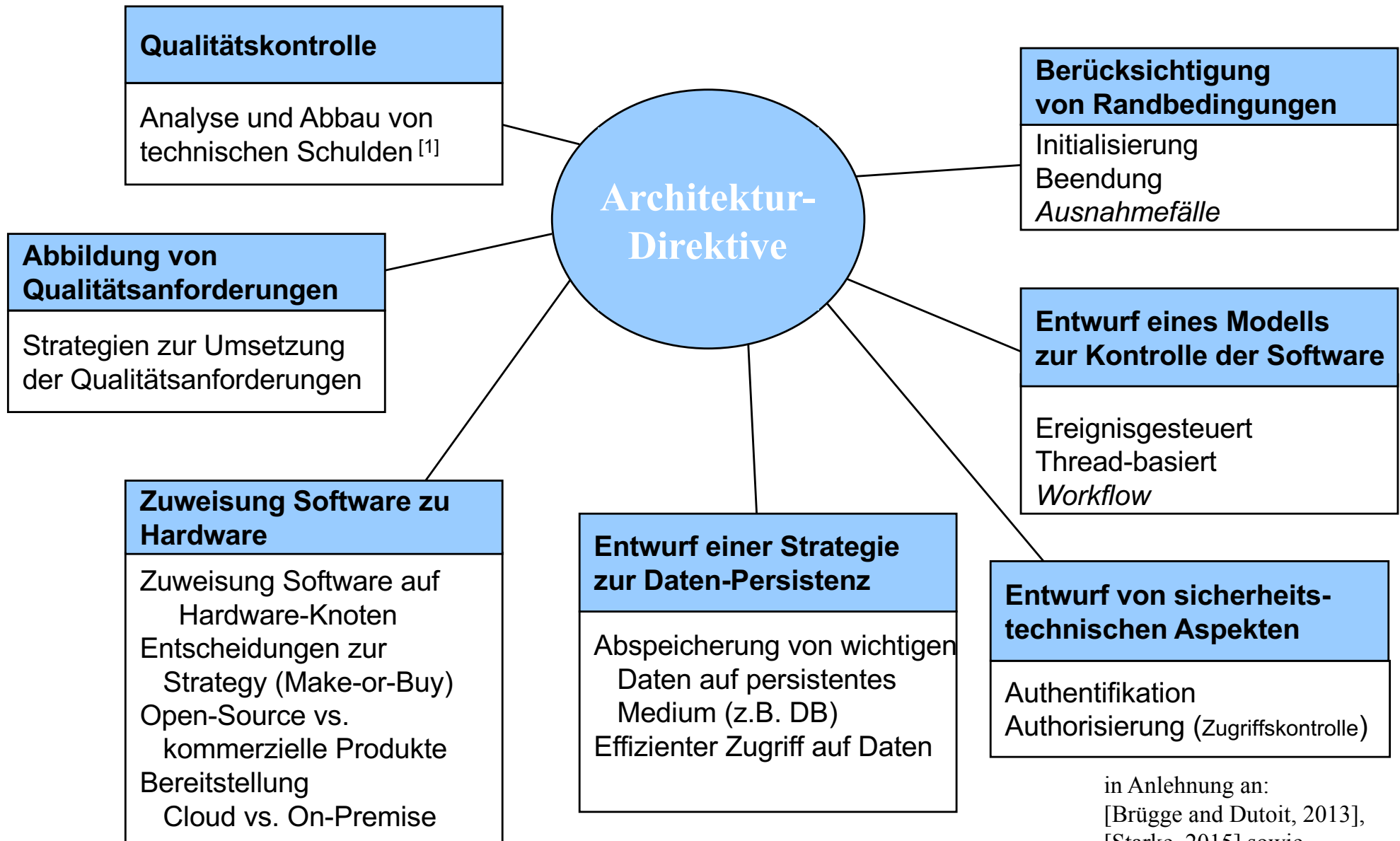


- Verteilte Interaktionsbeziehungen (verteilte Software-Architekturen)
  - Zugriff auf Methoden *entfernter* Subsysteme über ein Netzwerk





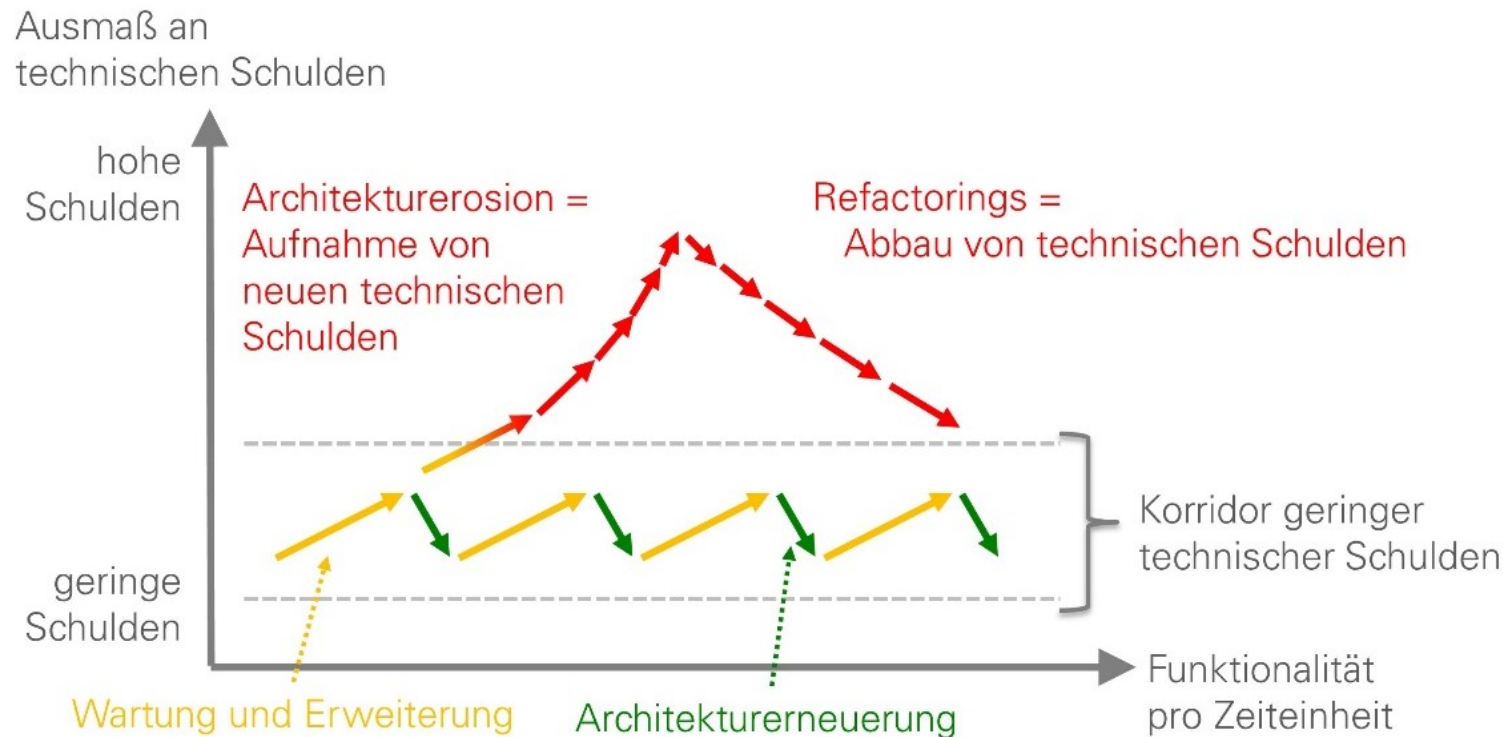
# Architektur-Direktive – Weitere Entwurfsaufgaben



in Anlehnung an:  
[Brügge and Dutoit, 2013],  
[Starke, 2015] sowie  
[1] [Lilienthal, 2016]



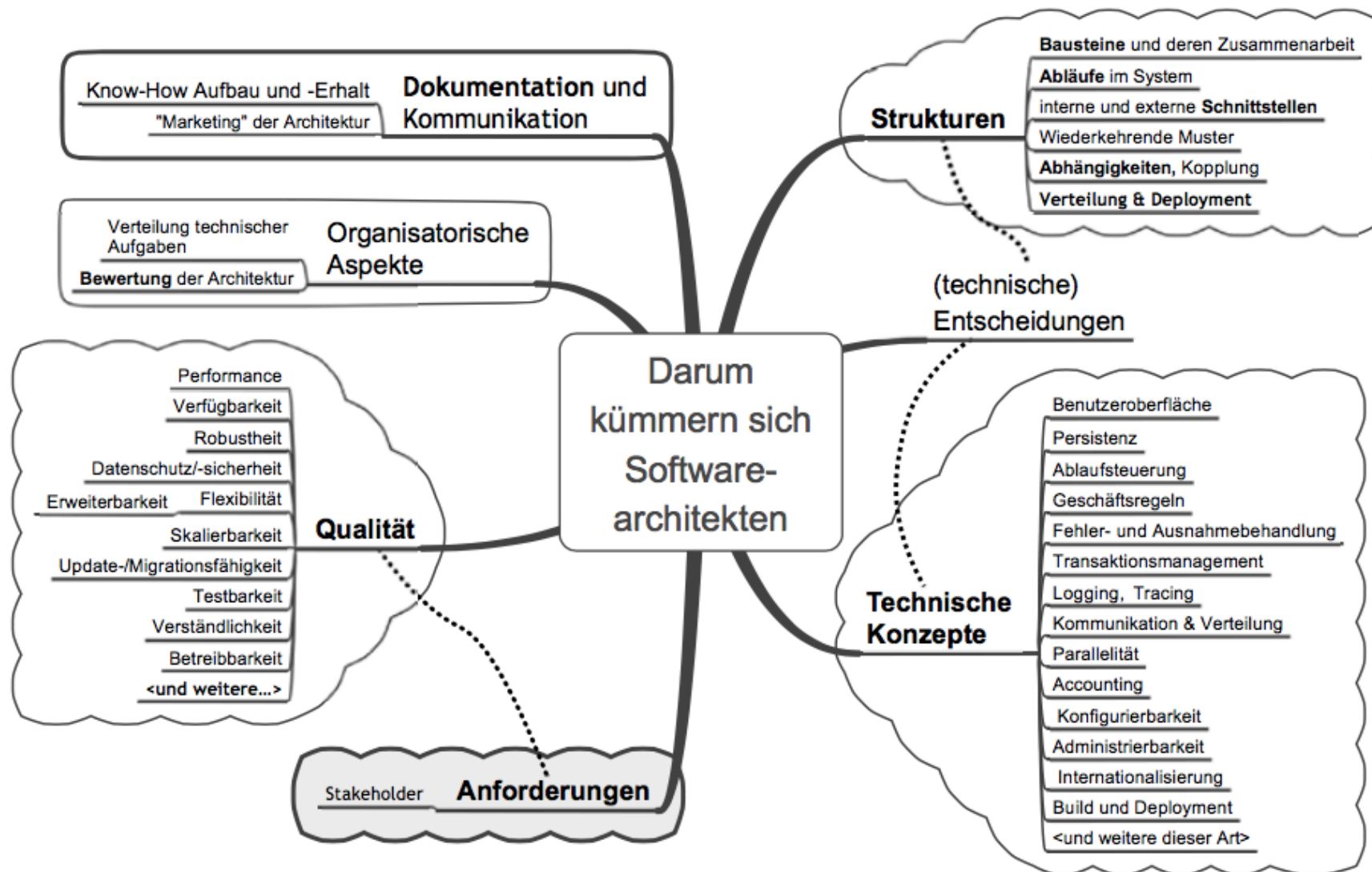
# Technische Schulden (Lilienthal, 2016)



- **Technische Schulden** sind Qualitätseinbußen, die entstehen, wenn bewusst oder unbewusst falsche oder suboptimale technische Entscheidungen getroffen werden
- Übliche Ausprägungen: Implementationsschulden (z.B. Code-Smells), **Architekturschulden**, Testschulden oder Dokumentationsschulden



# Darum kümmern sich Software-Architekten



Was sind typische Kompetenzen eines Software-Architekten?

(Starke, 2015), S. 26



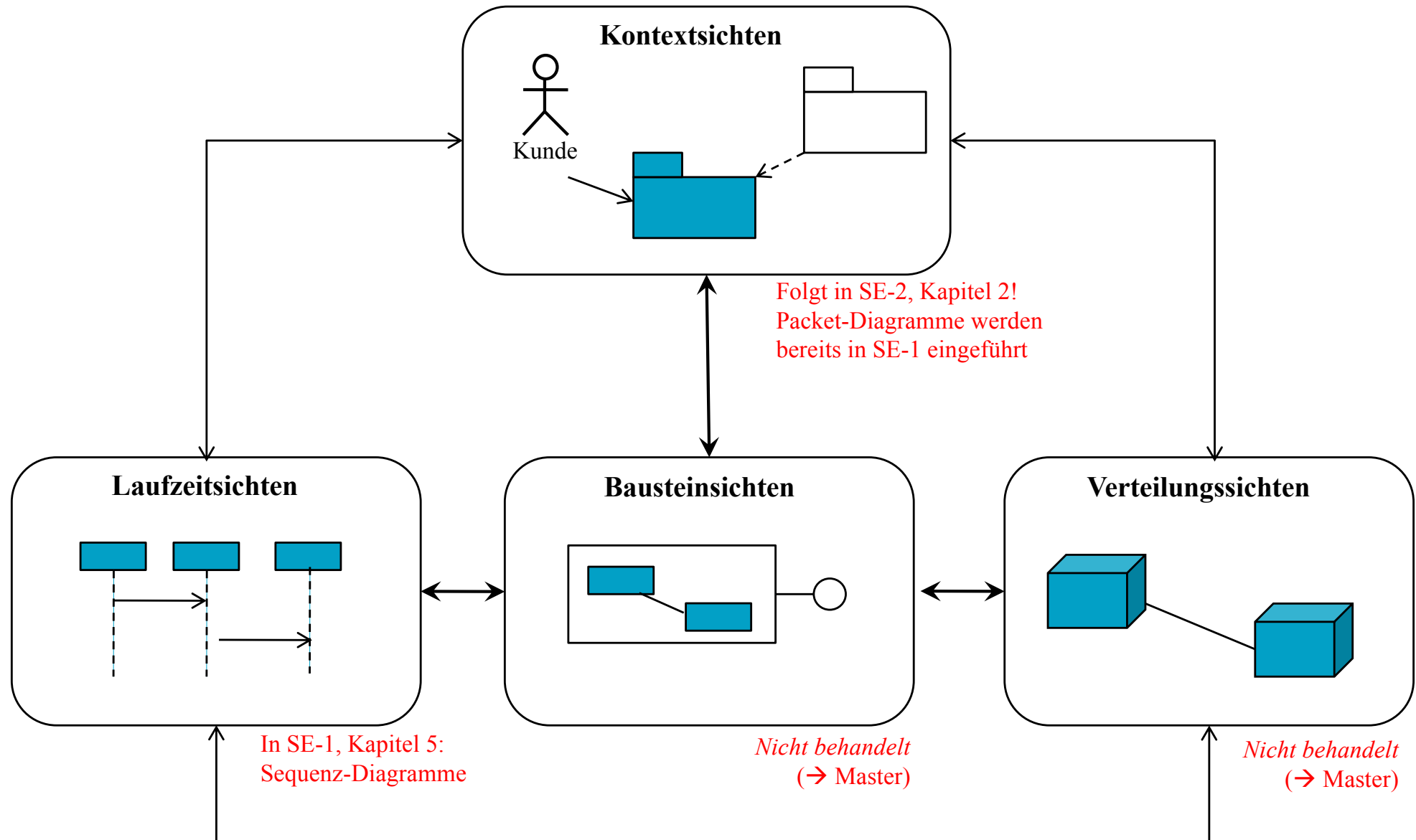
## Kapitel 5: System Design (Grundlagen von Software-Architekturen)

1	Wiederholung und Motivation	✓
2	Definition und Eigenschaften einer Software-Architektur	✓
<b>3</b>	<b>Modellierung von Software-Architekturen mit UML</b>	
4	Grundlegende Architekturmuster	
5	Exkurs: Abbildung von Klassen auf eine Architektur	
6	Zusammenfassung und Ausblick	



- Software-Architekten müssen ihre Entwurfsentscheidungen für eine Software-Architektur an verschiedene Stakeholder kommunizieren
- Eine Diskussion auf Basis von Source-Code ist nahezu unmöglich.
- Die **Modellierung von abstrahierenden** Modellen (UML...) ist unumgänglich gerade bei komplexen Software-Architekturen.
- Aber: Eine einzelne Darstellung kann meist die Komplexität und Vielschichtigkeit einer Architektur nicht vermitteln
- Notwendig: Einführung von verschiedenen Sichten

# Das 4-Sichten-Modell von Starke (Starke, 2015)







# Das 4-Sichten-Modell: Laufzeitsicht

---

## Inhalt:

- Die Laufzeitsicht beschreibt, welche Bestandteile des Systems zur Laufzeit existieren und wie sie zusammenwirken
- Welche Instanzen von Architekturbausteinen gibt es zur Laufzeit und wie interagieren diese miteinander?
- Darstellung der Interaktionen auch mit Akteuren

## Zielgruppe:

- Entwickler, Administratoren (z.B. Betreiber von externen Systemen)

## Darstellung:

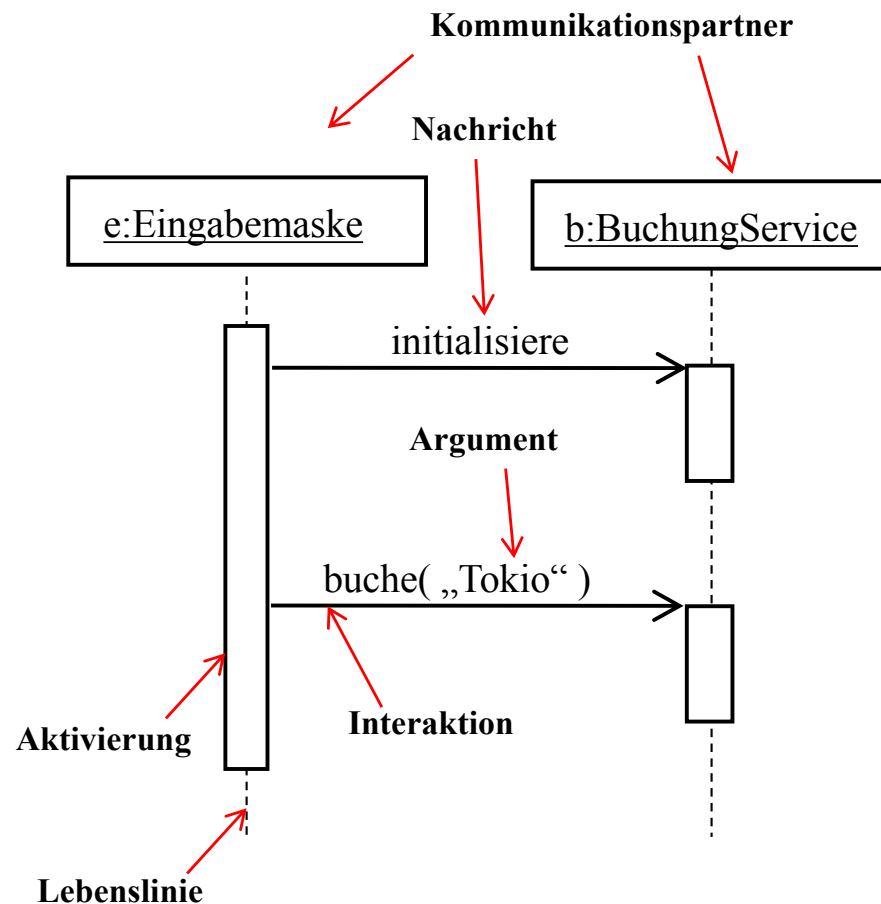
- UML **Sequenzdiagramme**, Aktivitätsdiagramme



- Sequenzdiagramme modellieren den dynamischen (zeitlichen) Ablauf eines objektorientierten Systems
- Interaktionen modelliert anhand einer vertikalen Zeitachse und horizontalen Interaktionen zwischen den Kommunikationspartnern
- Fokus auf die Abfolge der Interaktionen (nachrichtenbasiert)
- Unterschiede in der Modellierung vorhanden zwischen den Versionen UML 1.x und 2.x. Fokus hier: UML 2.5 (Rupp, 2012)



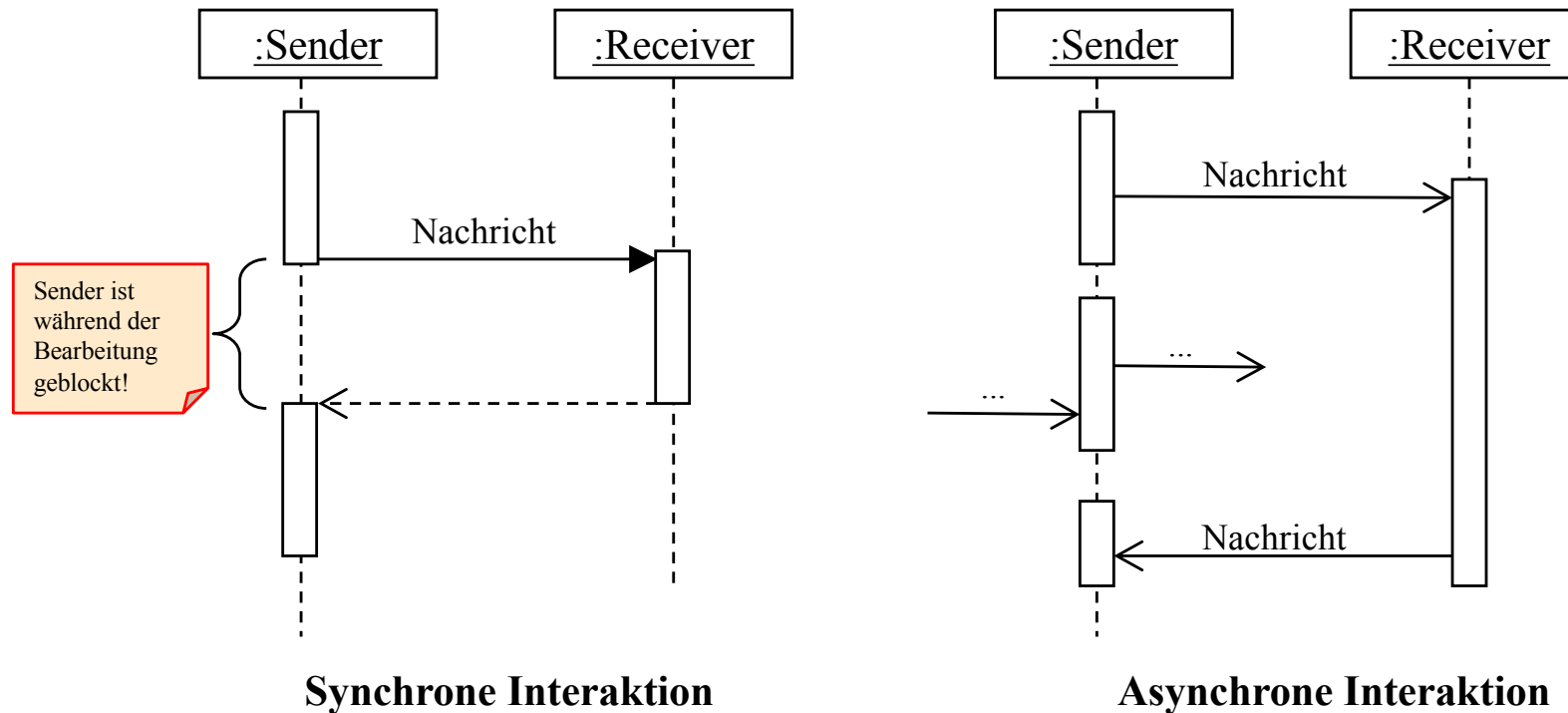
# UML Sequenzdiagramme – ein Überblick



- **Kommunikationspartner** repräsentieren Teile, die einem Ablauf involviert sind (*Objekte, Akteure, Klassen*)
  - Syntax Bezeichnung Objekte:  
`[ variable ] : Typ`
- **Lebenslinien** repräsentieren die zeitliche Existenz eines teilnehmenden Objekts
- **Eine Interaktion** ist das Zusammenspiel zwischen Kommunikationspartnern
- **Nachrichten** stellen Informationen dar, die während der Interaktion zwischen zwei teilnehmenden Objekten ausgetauscht wird. Nachrichten können **Argumente** enthalten
- **Aktivierungen** präsentieren Zeiten, in denen die teilnehmenden Objekte aktiv sind (vertikale breite Kasten). *Optional* zu verwenden (seit UML 2.x), nur wenn explizit gefordert



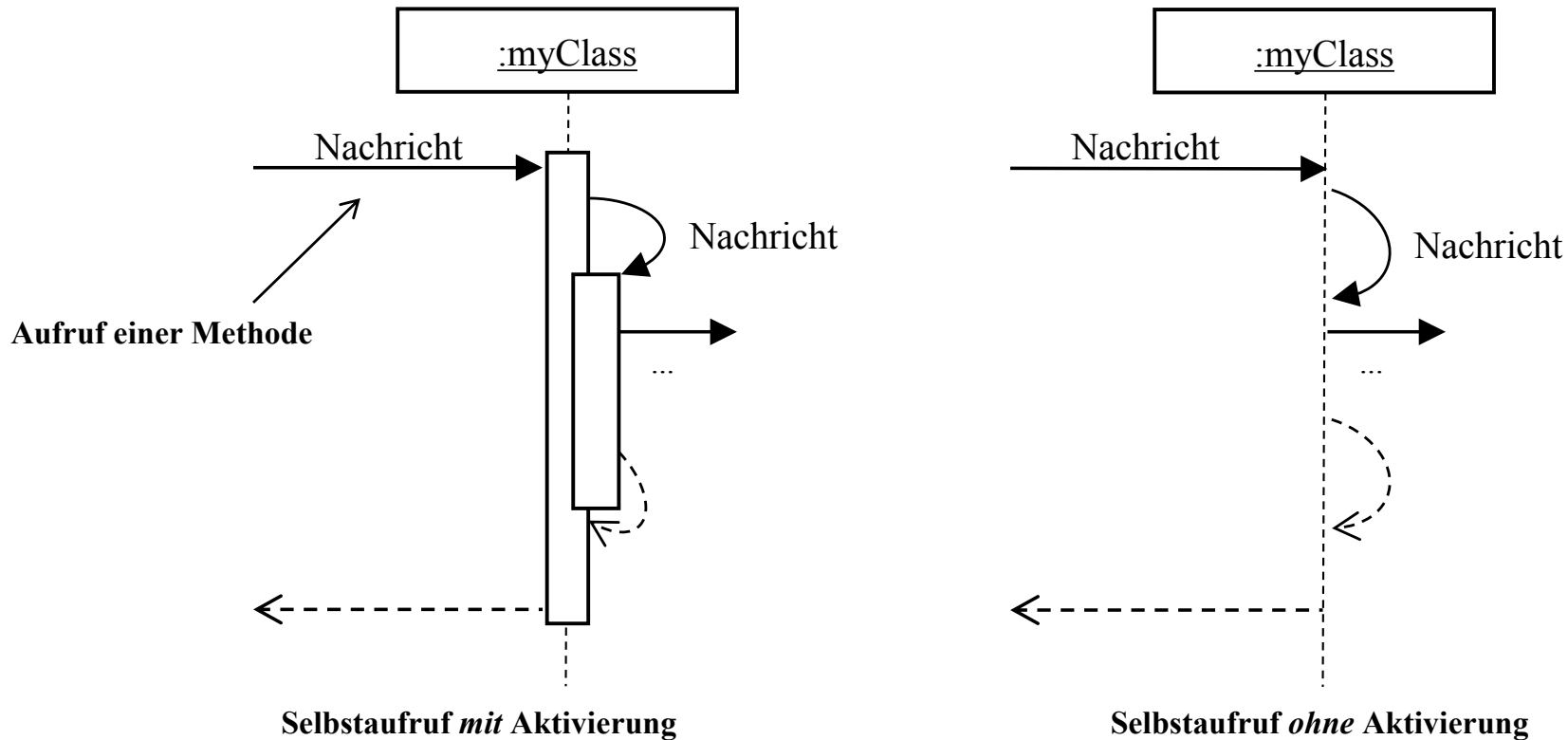
# Synchrone und asynchrone Interaktionen



- Bei der **synchrone Interaktion** geht die Kontrolle an den Kommunikationspartner, der aufgerufen wird (Receiver)
  - Der Sender ist während der Aktivierung geblockt und wartet auf Antwortnachricht
- Bei der **asynchronen Interaktion** wird der Sender nicht geblockt
  - Optionale Rückgabe durch separate Nachricht (*call back*)
  - Erleichtert die Modellierung gerade wenn Antworten *nicht* relevant sind



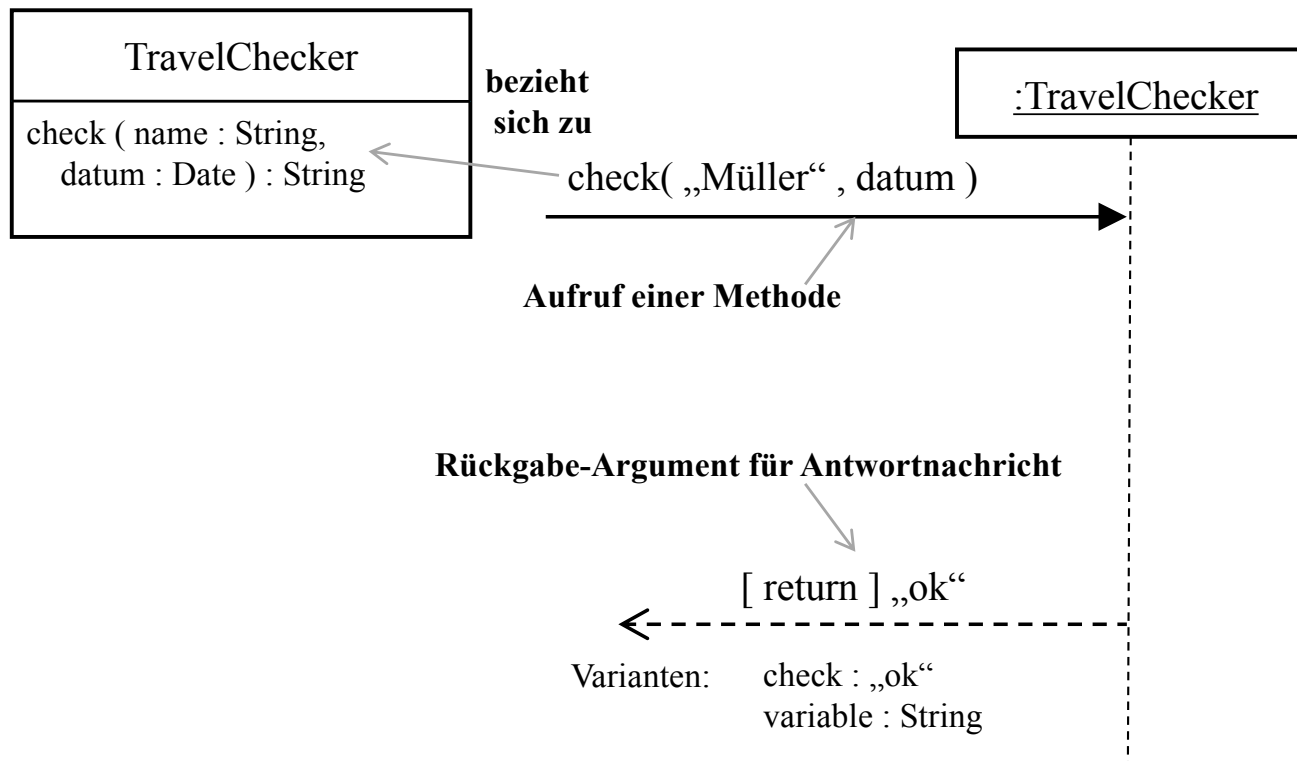
# Sequenzdiagramme: Methodenaufruf als Nachricht



- Eine nachrichtenbasierte Interaktion zwischen Objekten entspricht einem **Methodenaufruf** (Bezeichnung Nachricht = Methodename)
- Eine Methode kann interne Methoden auf dem gleichen Objekt aufrufen
- Eine Methode kann sich auch selber rekursiv aufrufen



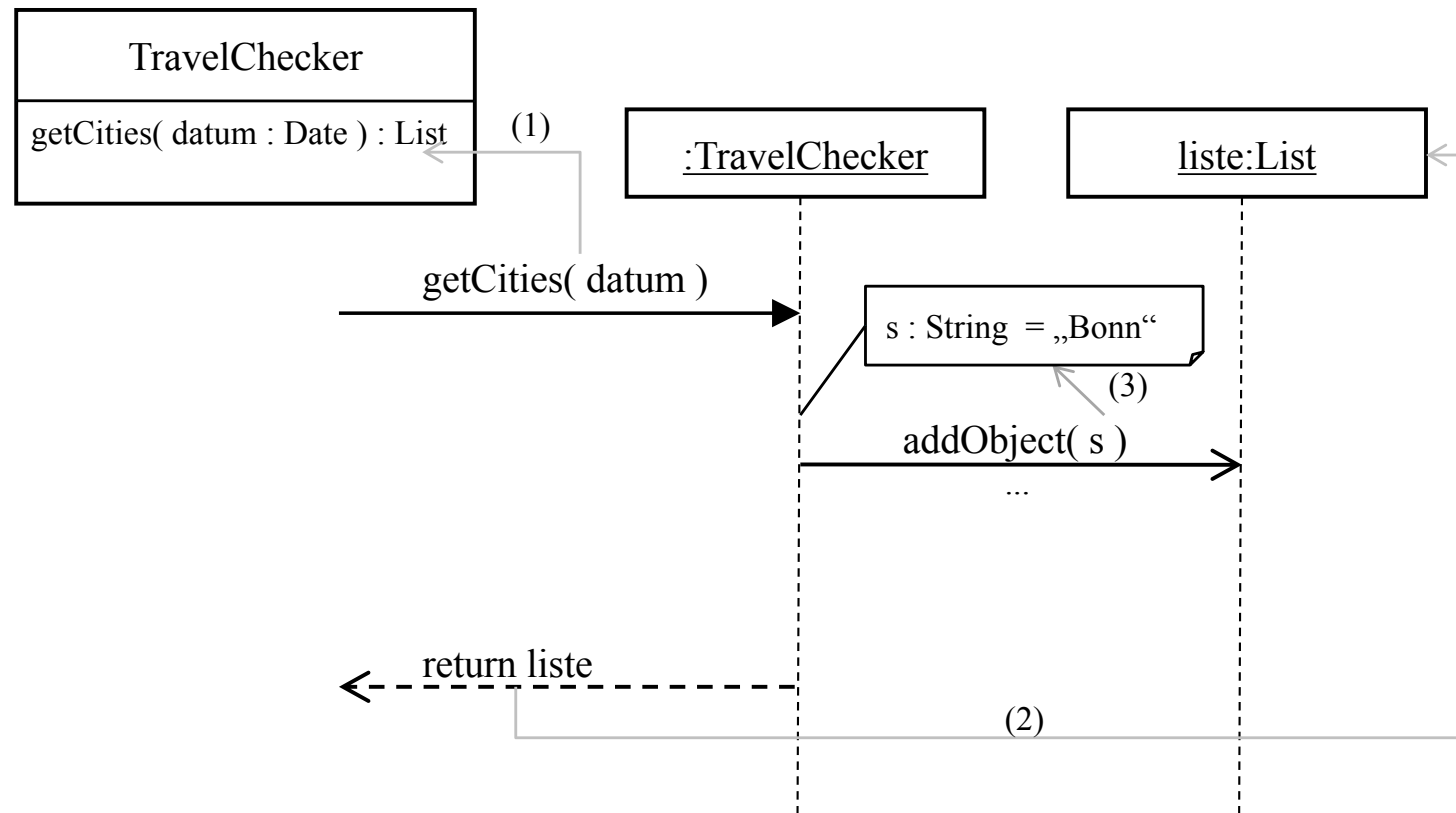
# Sequenzdiagramme: Nachrichten mit Argumenten



- Nachrichten *können* Argumente beinhalten (Bezug meist zu einer Klasse).
- Folgende Ausprägungen sind üblich (synchron wie asynchron)
  - Konstanten (z.B. „Tokio“, „Müller“, „122“)
  - Variable des *sendenden* Objekts (z.B. `ziel`, `datum`), ggf. mit Typ-Angabe
  - Zuordnung zu den Variablen möglich (z.B. `ziel = „Bonn“`, `datum = „29.4.2013“`)
- Argumente bei Antwortnachricht möglich (Varianten siehe [Rupp, 2012])
  - Argument entfällt, wenn kein Rückgabewert in Klasse spezifiziert („void“)

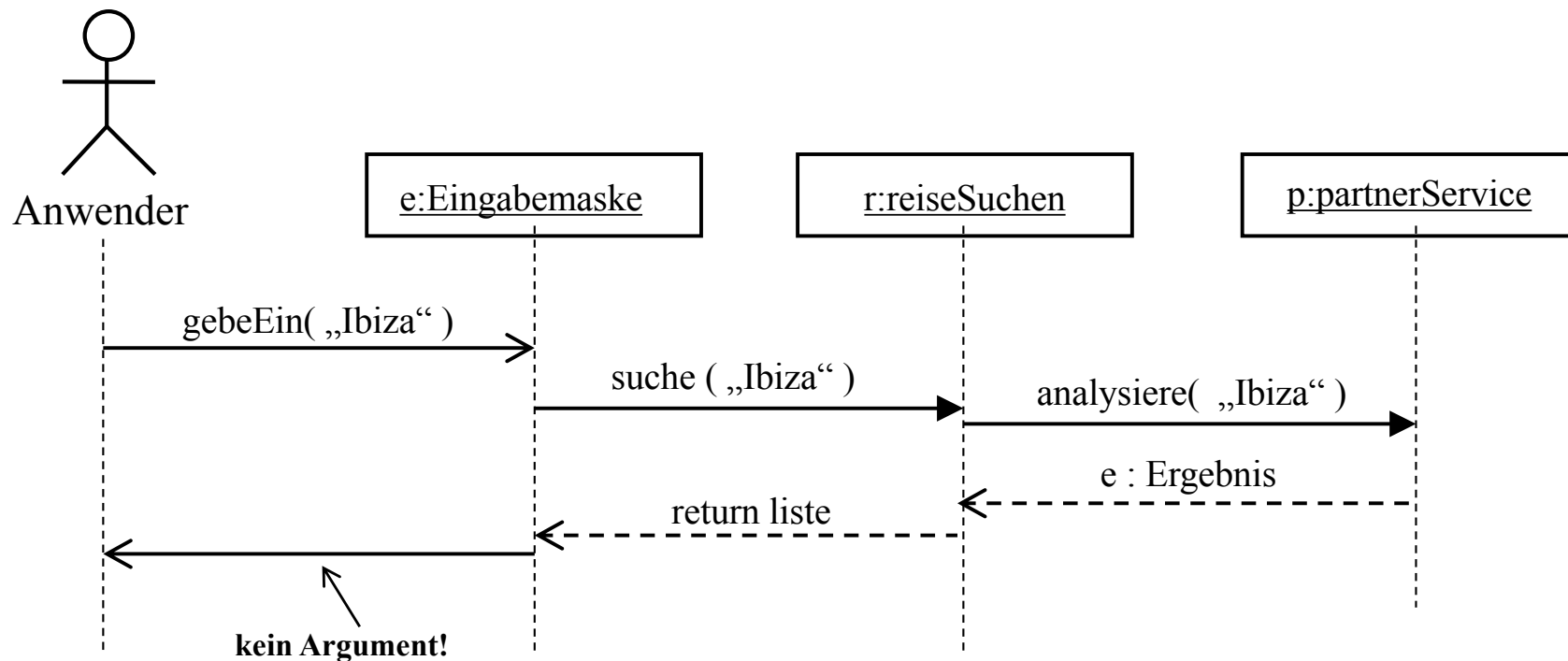


# Sequenzdiagramme: Nachrichten mit Argumenten



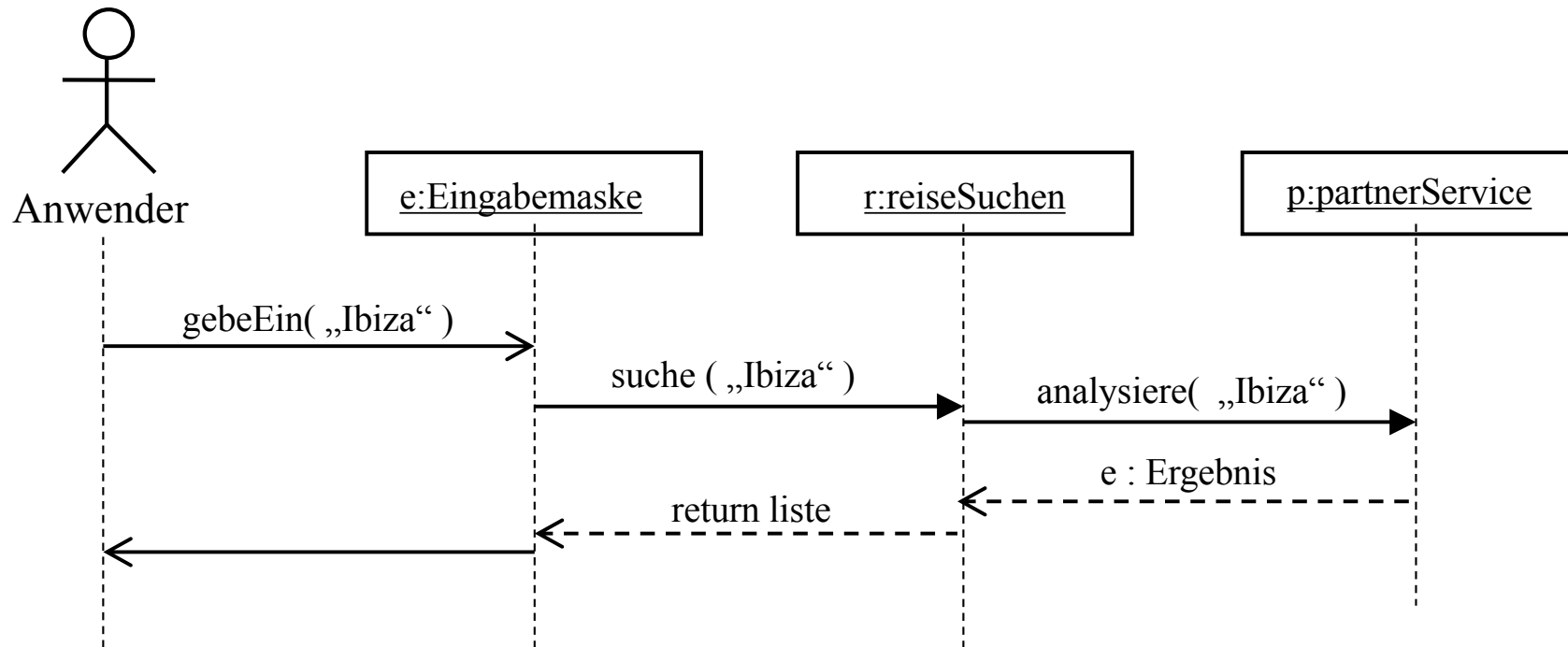
- Bei der Verwendung von Variablen können sich diese beziehen auf
  - (1) auf den Eingabe- und Rückgabeparametern der Methoden einer assoziierten Klasse
  - (2) auf die Variable eines teilnehmenden Objekts
  - (3) auf eine lokale Variable (durch Kommentarfeld auf Lebenslinie hinzugefügt, meist primitive Datentypen)

# Sequenzdiagramm: Interaktion mit Akteuren



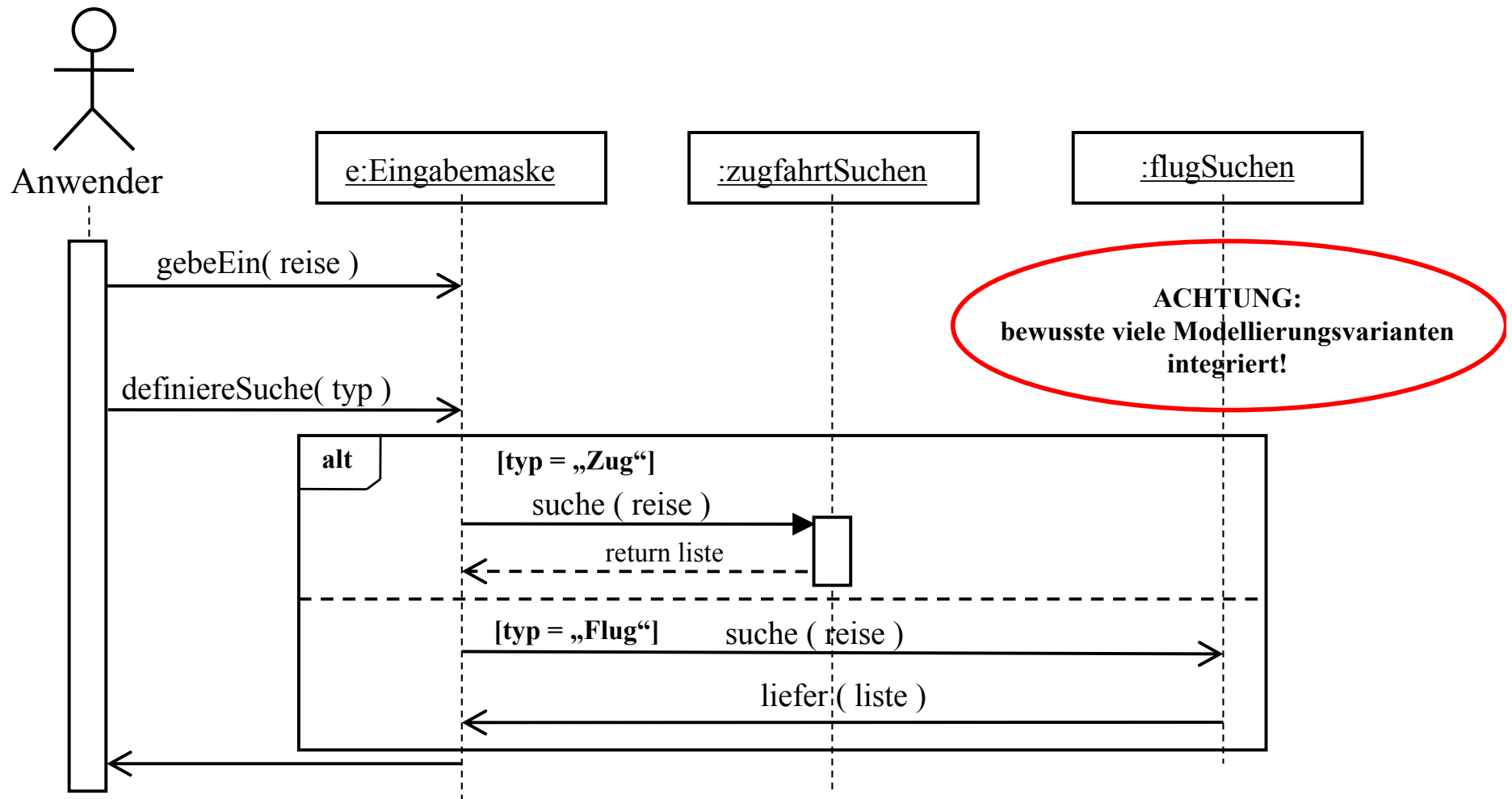
- Neben konkreten Objekten können auch **Akteure** in ein Sequenzdiagramm eingebunden werden (auch hier: Aktivierung optional!)
- Interaktion zwischen Akteur und Objekt hat eher imaginären Charakter (kein direkter Methodenaufruf, sondern Signal)
  - Interaktion wird manuell vom Akteur ausgeführt (z.B. Eingabe Daten in GUI)
  - Verwendung von offenen Pfeilen üblich (asynchrone Interaktion)
  - Call-back Nachricht leer: Signal zurück an Akteur: weitere Interaktionen ausführen!





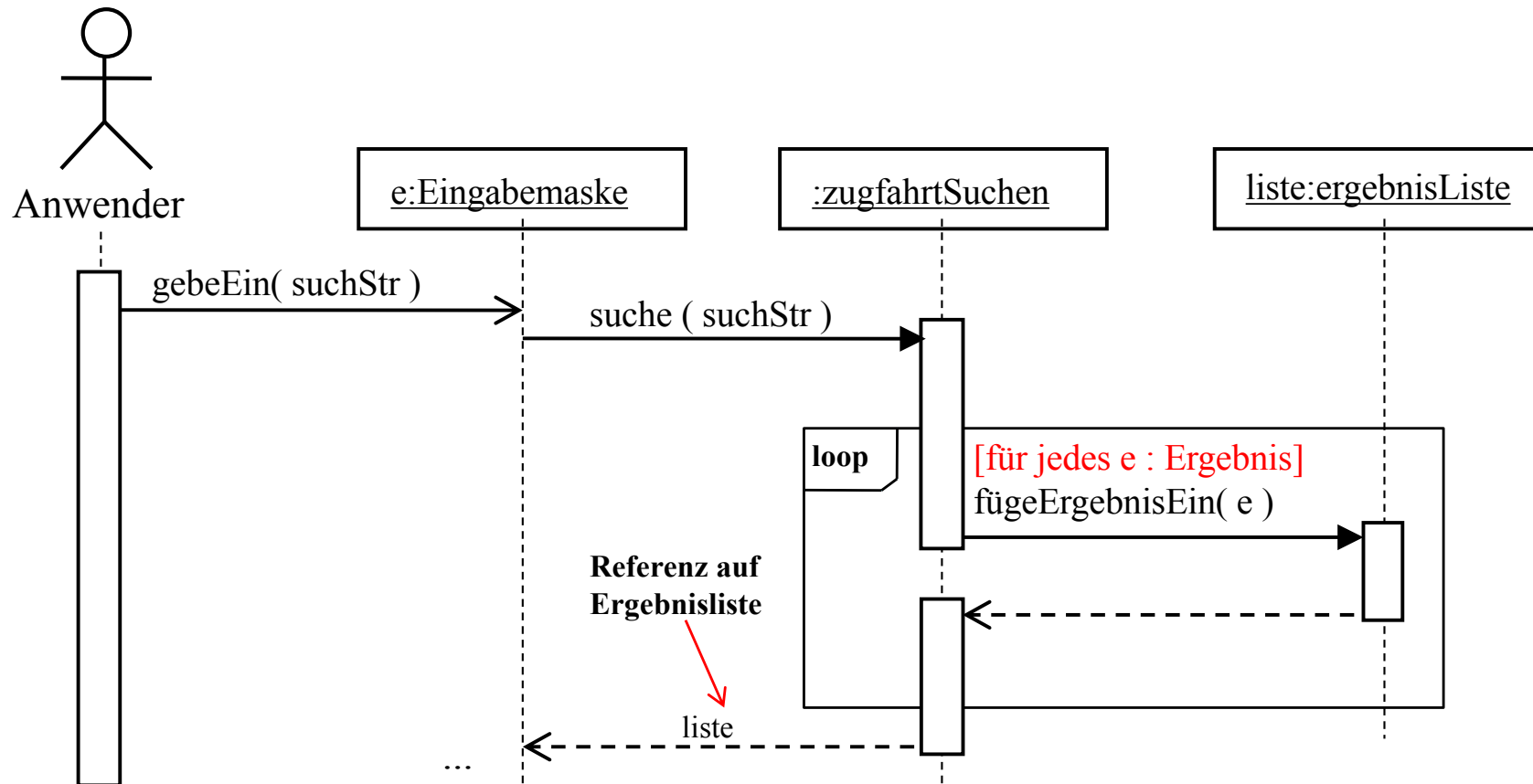
- **Konkrete Sequenzen** modellieren eine mögliche Interaktion (ein spezielles Szenario) zwischen den teilnehmenden Objekten
- Ausnahmefälle oder abweichende Abläufe werden nicht modelliert
- Primäre Anwendung bei Sequenzdiagrammen!

# Sequenzdiagramm: Modellierung von abstrakten Sequenzen



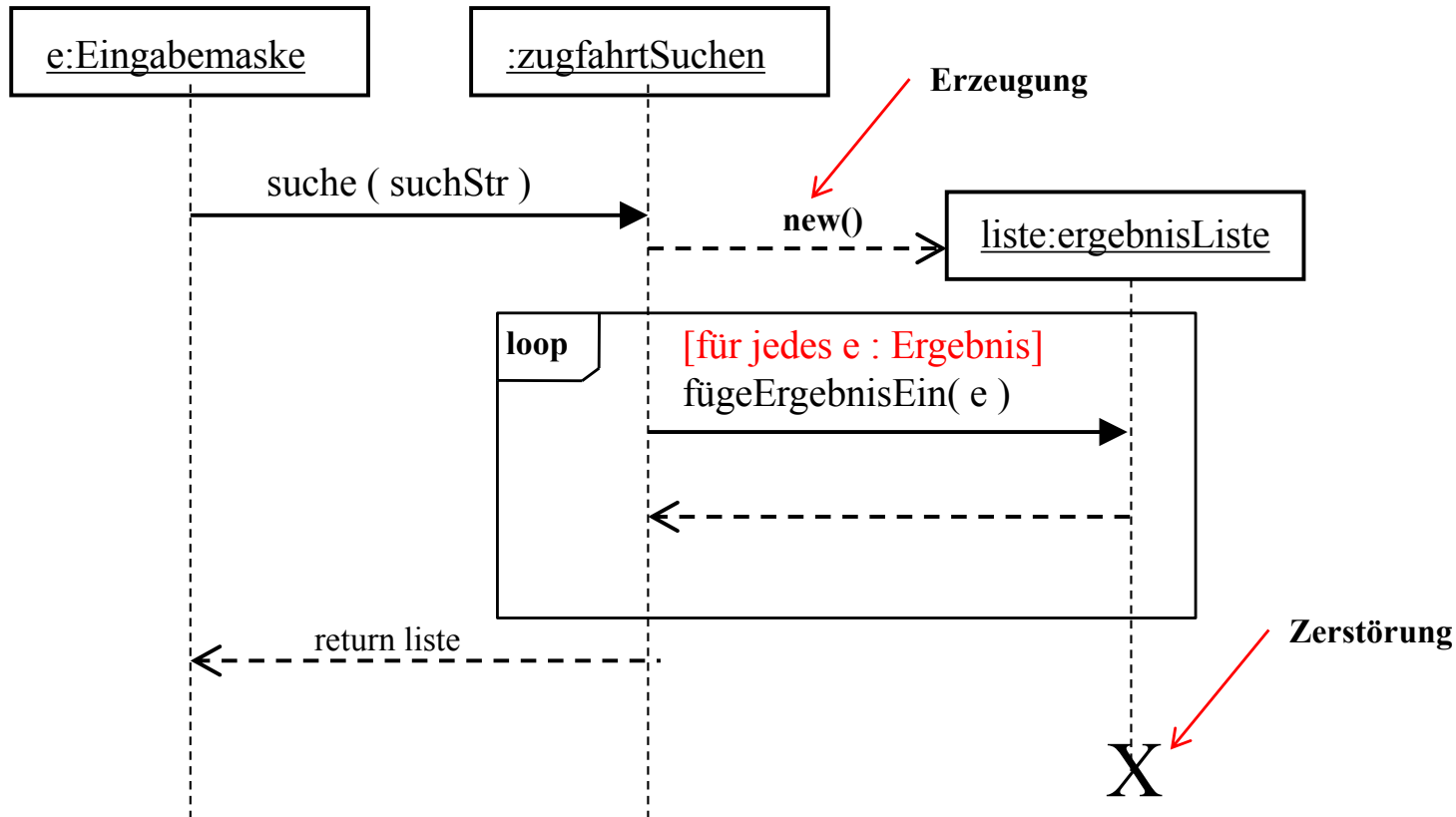
- **Abstrakte Sequenzen** modellieren alle möglichen Interaktionen zwischen den teilnehmenden Objekten
- Verwendung **Bedingungen** (alt-Operator) und Iterationen (loop-Operator) (weitere Infos: UML2 Glasklar)

# Sequenzdiagramm: Modellierung von abstrakten Sequenzen



- **Abstrakte Sequenzen** modellieren alle möglichen Interaktionen zwischen den teilnehmenden Objekten
- Verwendung Bedingungen (alt-Operator) und **Iterationen** (loop-Operator) (weitere Infos: UML2 Glasklar)

# Sequenzdiagramm: Erzeugung und Löschung von Objekten



- Erzeugung wird modelliert mit einem gerichteten Pfeil, der zu dem zu erzeugenden Objekt zeigt (inkl. Nachricht wie `new()` oder `create()`)
- Zerstörung von Objekten wird mit einem X am Ende der Lebenslinie modelliert



- Die Modellierung der Interaktionen zwischen den verschiedenen Objekt-typen, die sich aus der Analyse von Use Case ergeben, kann mittels eines Sequenz-Diagramms erfolgen.
- Grundlegende Regeln:
  - Die (linke) erste Spalte entspricht dem Akteur, der den Use Cases initiiert
  - Der Akteur greift nie auf Entity oder Control Objects direkt zu
  - Die zweite Spalte sollte ein (initiales) Boundary Object sein
  - Boundary Objects interagieren nur mit Control-Objekten (ggf. mit weiteren Sub-Boundaries)
  - Control Objects verwalten die restlichen Interaktionen, die im Use Case modelliert sind
  - Control Objects können weitere Boundary Objekte erzeugen
  - Entity Objects werden nur von Control-Objects erzeugt
  - Entity Objects können auf keine Boundary oder Control Objects zugreifen



## Kapitel 5: System Design (Grundlagen von Software-Architekturen)

1	Wiederholung und Motivation	✓
2	Definition und Eigenschaften einer Software-Architektur	✓
3	Modellierung von Software-Architekturen mit UML	✓
4	<b>Grundlegende Architekturmuster</b>	
5	Exkurs: Abbildung von Klassen auf eine Architektur	
6	Zusammenfassung und Ausblick	



# Was ist ein Muster?

---

- Ein Muster (engl.: *pattern*) zielt auf ein in speziellen Entwurfsituationen häufig auftretendes Entwurfsproblem und beschreibt eine Lösung für dieses Problem
- Muster dokumentieren bekannte, **erprobte** Lösungen von Entwurfsproblemen.
- Muster kann man **nicht erfinden**, sondern werden aus Entwurfswissen **abgeleitet**
- Identifizieren und Spezifizieren von Abstraktionen auf einer Ebene von Klassen, Subsysteme, Methoden
- Etablierung eines gemeinsamen Vokabulars



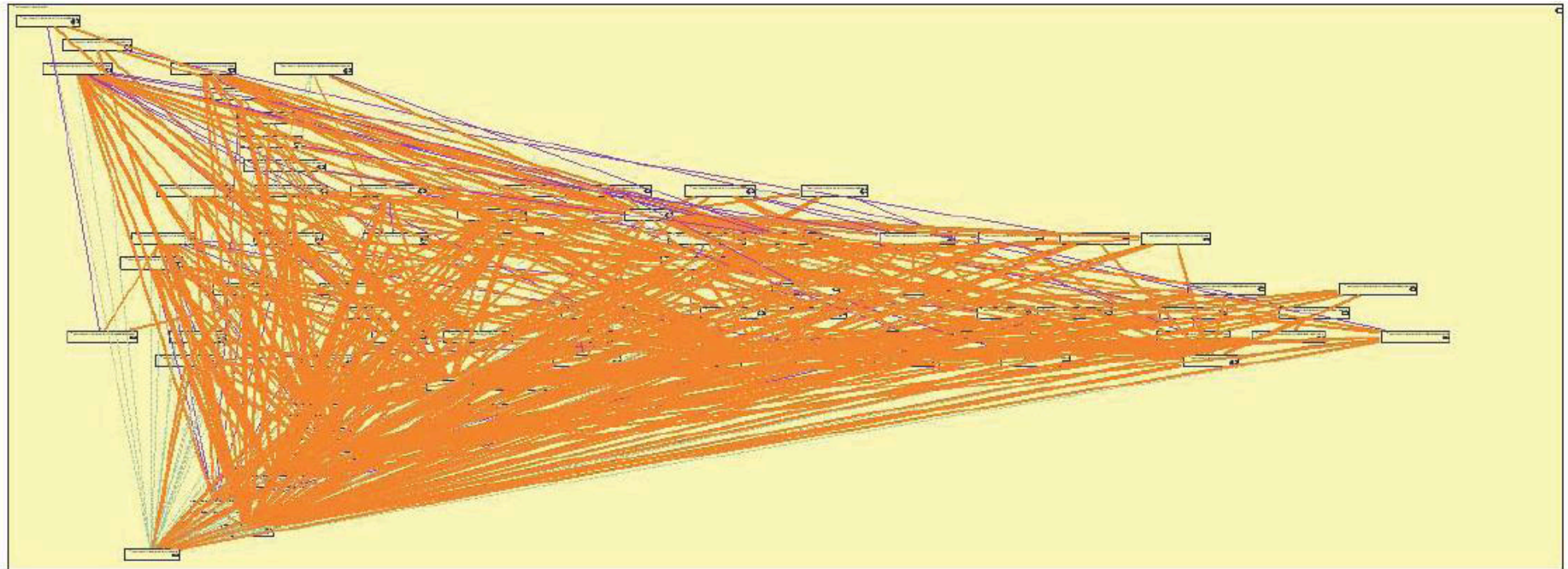
# Kategorien von Muster - Architekturmuster



- Ein **Architekturmuster** (*architectural pattern*) spiegelt ein grundsätzliches Strukturierungsprinzip von Softwaresystemen wider.
- Es beschreibt eine Menge vordefinierter Subsysteme, spezifiziert deren jeweiligen Zuständigkeitsbereich sowie deren Beziehungen
- Schablonen für konkrete Software-Architekturen
- Basis für eine Grundsatzentscheidung im Entwurf eines Software-Systems
- Fokus in dieser Vorlesung die Muster **MVC** und **Schichtenbildung**



# Problem: Unkontrolliert gewachsene Software-Architekturen



„Just ONE (out of 20) subsystems of a software system“

Quelle: Vortrag Dr. Jens Knodel, Fraunhofer IESE  
beim Software-Foren Leipzig, Febr. 2011



# Grobe Einteilung des Systems – Schichten (Layer)

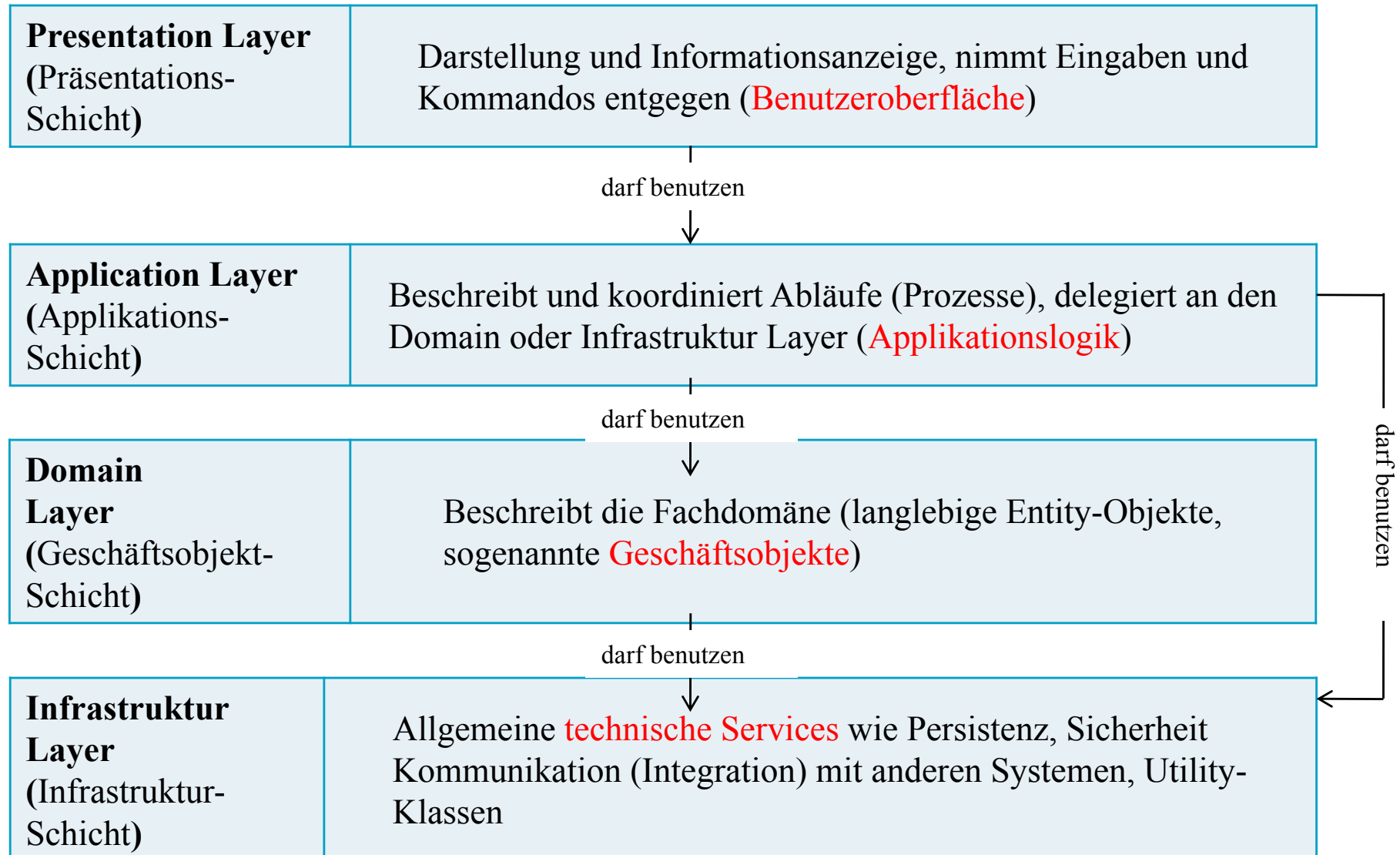
---

- Problem:
  - Teil-Systeme müssen oft ersetzt oder angepasst werden, Restfunktionalität soll gleich sein  
→ Schwierig bei unkontrolliert gewachsenen Software-Architekturen!
  - Änderungen von System-Funktionalitäten können sich durch das ganze System hindurch ziehen
  - Komplexe Software-Systeme müssen durch mehrere Entwickler unabhängig entwickelt werden
- Lösung: „Schichten“ Muster (engl.: *layer pattern*)
  - Unterteilung der System-Funktionalität in einzelne, aufeinander aufbauende Schichten
  - Jede Schicht implementiert eine zusammenhängende Funktionalität auf einem bestimmten **Abstraktionsniveau**
  - Eine Schicht verwendet die *darunter liegende* Schicht über **Schnittstellen**
  - Jede Gruppe kann *unabhängig* entwickelt und angepasst werden kann.



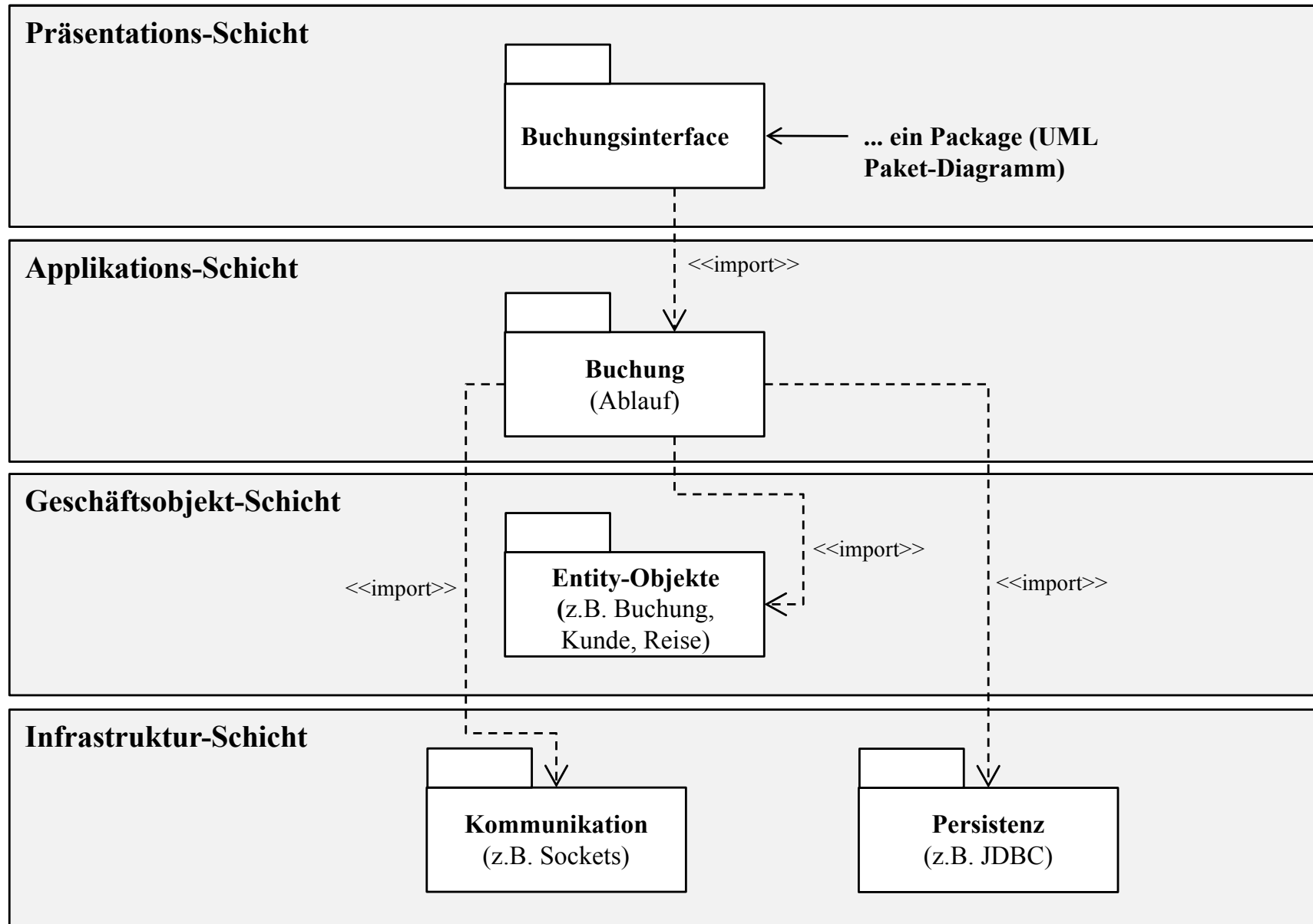
# Schichten-Architektur nach Evans (Evans, 2004)

- Aufteilung einer Software-Architektur in vier verschiedene Schichten (*layer*)



# Typischer Aufbau einer Schicht-Architektur

## Modellierung mit UML Paket-Diagrammen





# Flexible Darstellung von Benutzeroberflächen

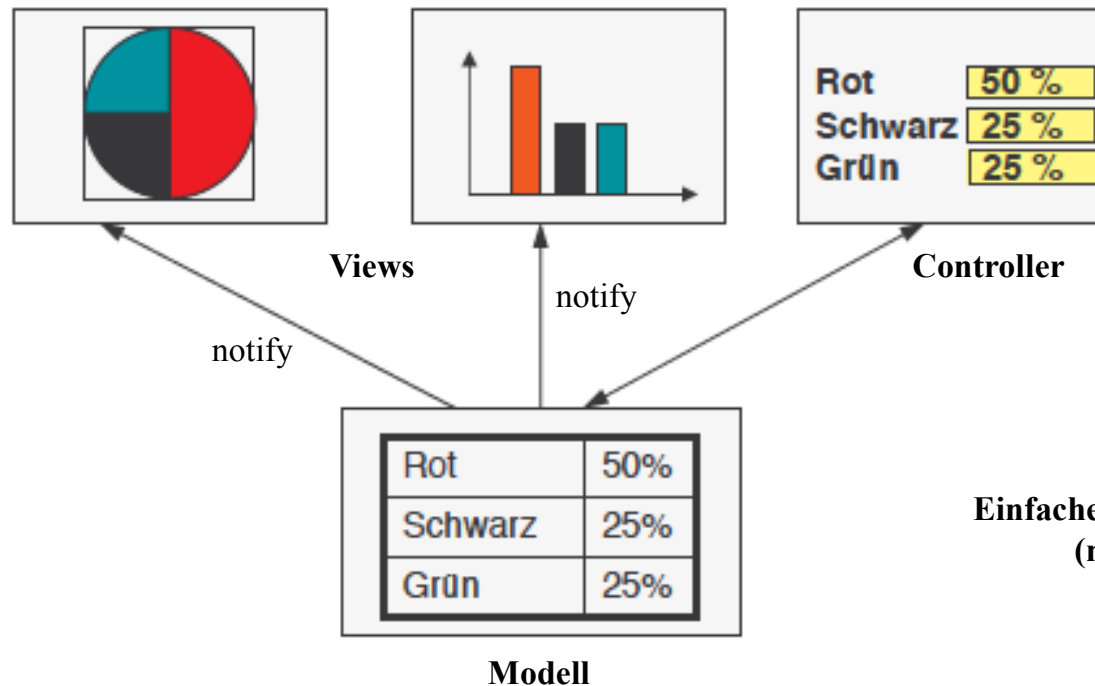
---

- Problem:
  - Bei der Implementierung von Benutzeroberflächen werden häufig Ablaufsteuerung und die Datenhaltung komplett in Benutzeroberflächen vereint
- Folgen (Starke, 2011):
  - Sollen mehrere Masken den gleichen Datenbestand in unterschiedlichen Sichten darstellen, müssen Daten *redundant* gehalten oder synchronisiert werden
    - **Konsistenzprobleme, Kapselung** nicht garantiert
  - Die Vermengung von Präsentation, Datenhaltung und Ablaufsteuerung macht den Programmcode schwer verständlich
    - Seiteneffekte bei **häufigen Erweiterungen**
  - Eine **Portierung** der Anwendung z.B. auf eine andere DB kann sehr aufwändig sein
    - Jede Klasse (Maske) muss u.U. angepasst werden
  - Neue Masken müssen stets von Grund auf neu programmiert werden.
    - **Wiederverwendung** bestehender Klassen ist kaum möglich



# Das Model-View-Controller Muster (MVC)

- **Lösung:** Aufteilung der Anwendung in drei Subsysteme (Komponenten), je eine für
  - Verwaltung der fachlichen Daten (**Model**)
  - Darstellung der Ergebnisse durch Fachmasken (**View**)
  - Eingabe und Ablaufsteuerung (**Controller**)



Einfache schematische Darstellung  
(nach Gamma, 1995)



# Erweiterung des Model-View-Controller Musters

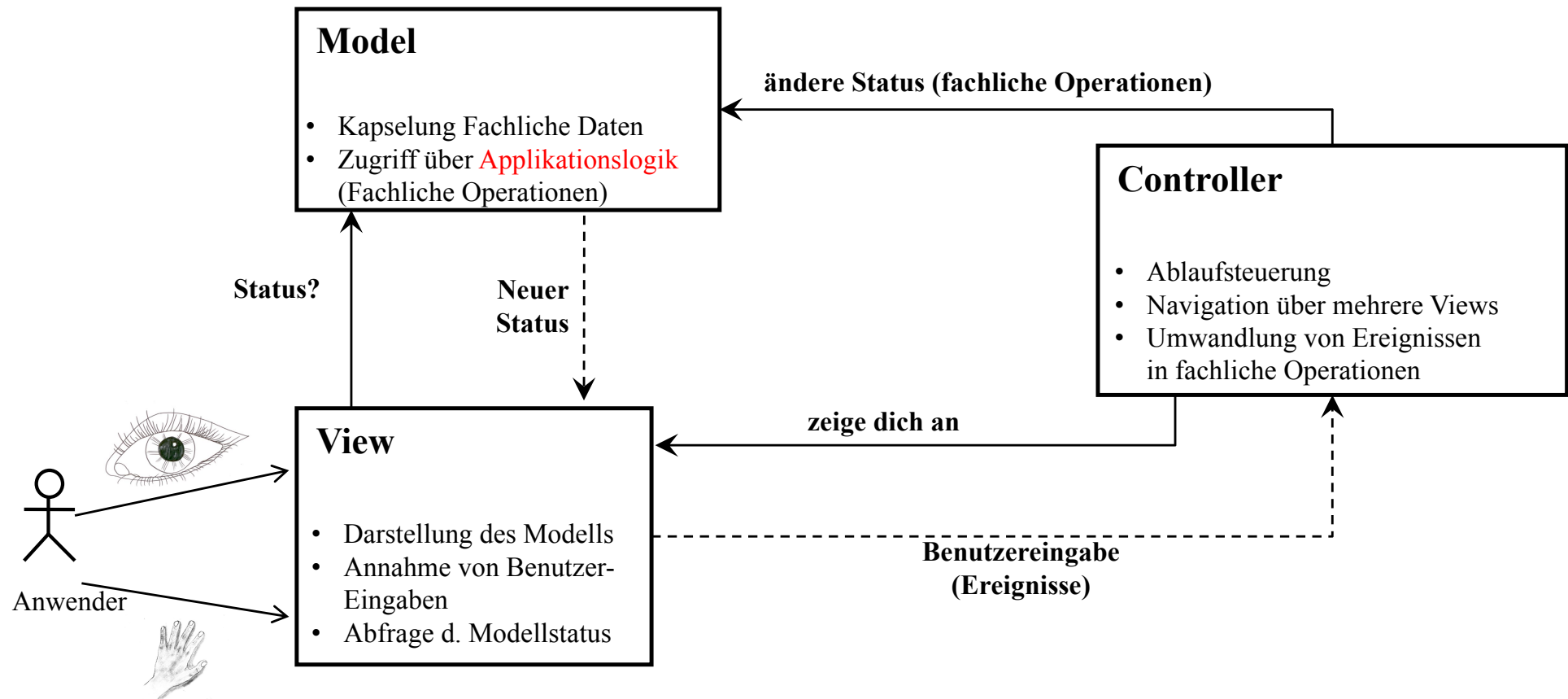
---

- Durch die Weiterentwicklung von Möglichkeiten zur Implementierung von Benutzeroberflächen (insbesondere im Web-Bereich) hat sich das ursprüngliche MVC-Muster (Gamma, 1995) auch weiterentwickelt
- Eine einheitliche Darstellung des MVC Pattern ist heutzutage kaum mehr möglich, bedingt durch **unterschiedliche Auslegungen** einer Vielzahl von Web Frameworks
  - Weiterführende Quelle: (Syromiatnikov, 2014)
  - Diverse Varianten werden hier beschrieben, z.B.: MVP (Model View Presenter) oder MVVM (Model View ViewModel)
- Übersicht bekannter Web Frameworks:
  - <https://hotframeworks.com/>
- Die folgende Darstellung basiert auf das Buch von Starke (Starke, 2011)
- Typische Verbesserungen (bzw. Erweiterung)
  - Views dienen auch zur Eingabe von Daten, Controller dient rein der Ablaufsteuerung



# Struktur des MVC Pattern

- View-Komponenten die Benutzer-Eingaben entgegen und geben diese an den Controller weiter.



Quelle: in Anlehnung an (Starke, 2011, S. 237)





## Kapitel 5: System Design (Grundlagen von Software-Architekturen)

1	Wiederholung und Motivation	✓
2	Definition und Eigenschaften einer Software-Architektur	✓
3	Modellierung von Software-Architekturen mit UML	✓
4	Grundlegende Architekturmuster	✓
<b>5</b>	<b>Exkurs: Abbildung von Klassen auf eine Architektur</b>	
6	Zusammenfassung und Ausblick	



# Objekttypen zur Strukturierung der Analyse

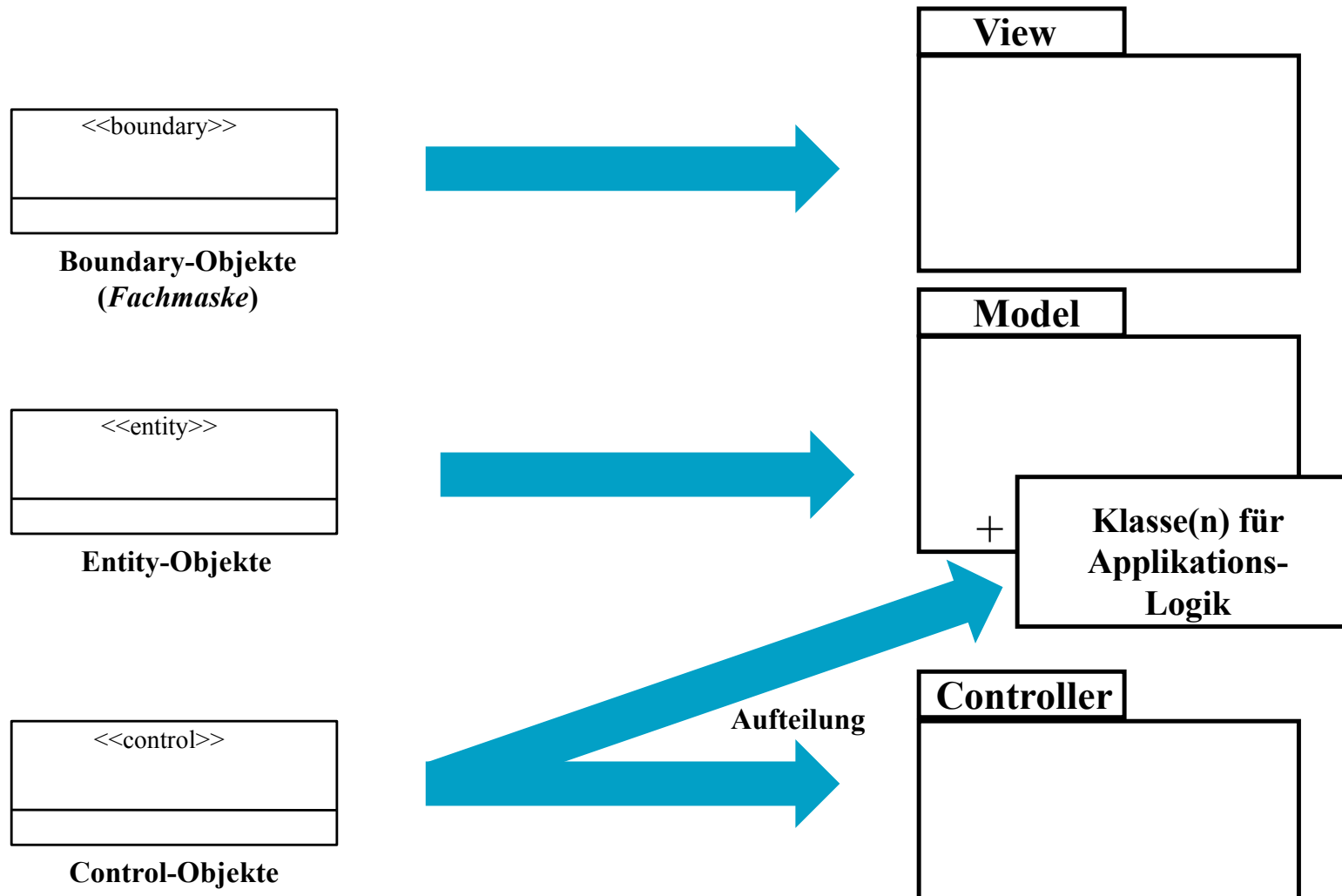
---

- Die Unterteilung der Klassen aus dem Analyse-Modell in Objekttypen ermöglichen eine erste Einteilung und Abbildung der Konzepte auf eine mögliche Software-Architektur.
- Das objektorientierte Analyse-Modell muss nach den Vorgaben konkreter Architekturmuster abgebildet werden
- Ziel: Bildung von größeren Subsystemen mit konkreten Schnittstellen
- Fokus hier: Abbildung der Analyse-Objekte auf eine Schichtenarchitektur *sowie* gemäß des MVC-Musters



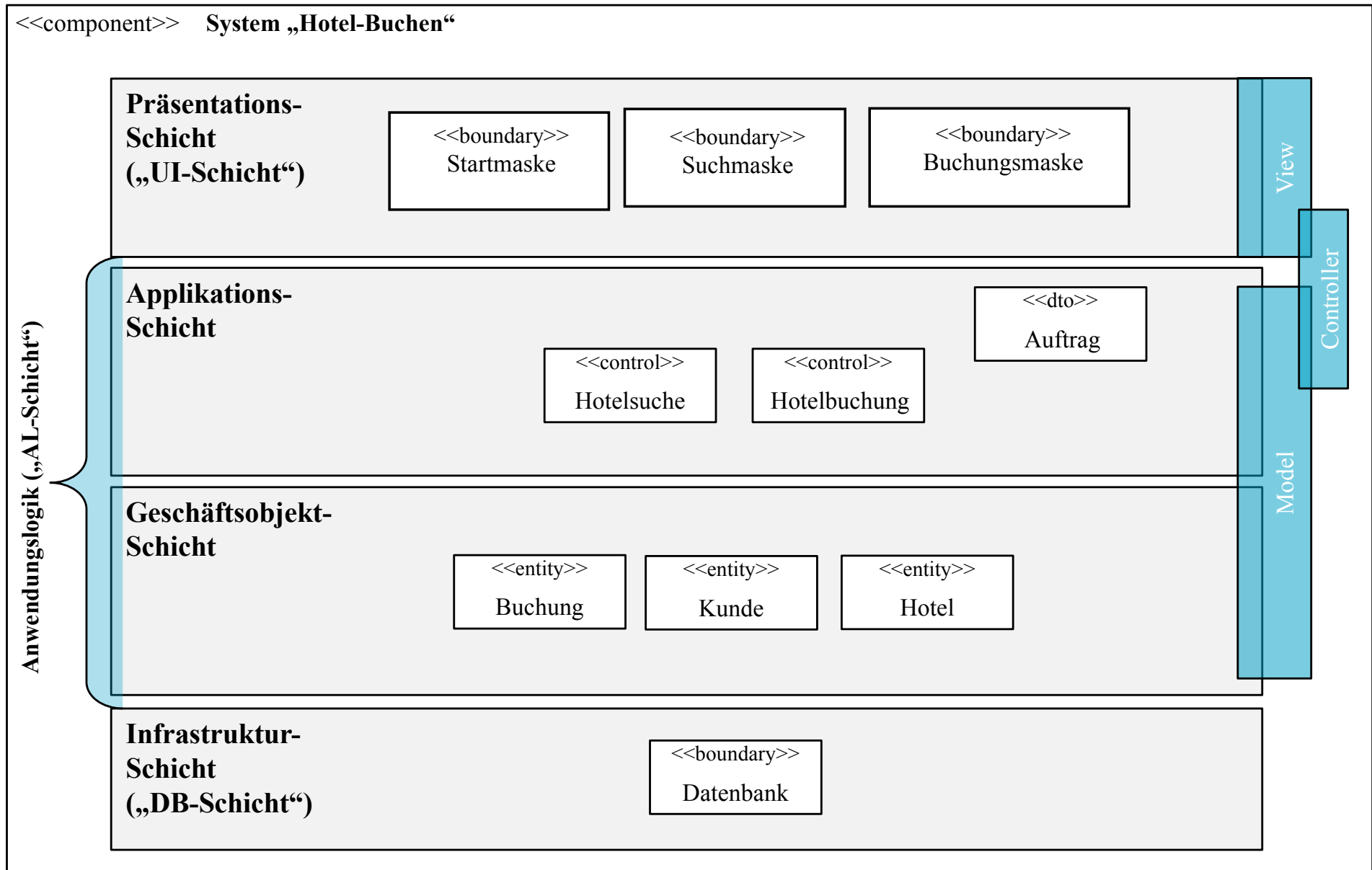
# Abbildung der Objekttypen auf das MVC Muster

- Die eingeführten Objekttypen lassen sich auf die Komponenten des Model-View-Controller Musters abbilden. Wichtigste Abbildungen:





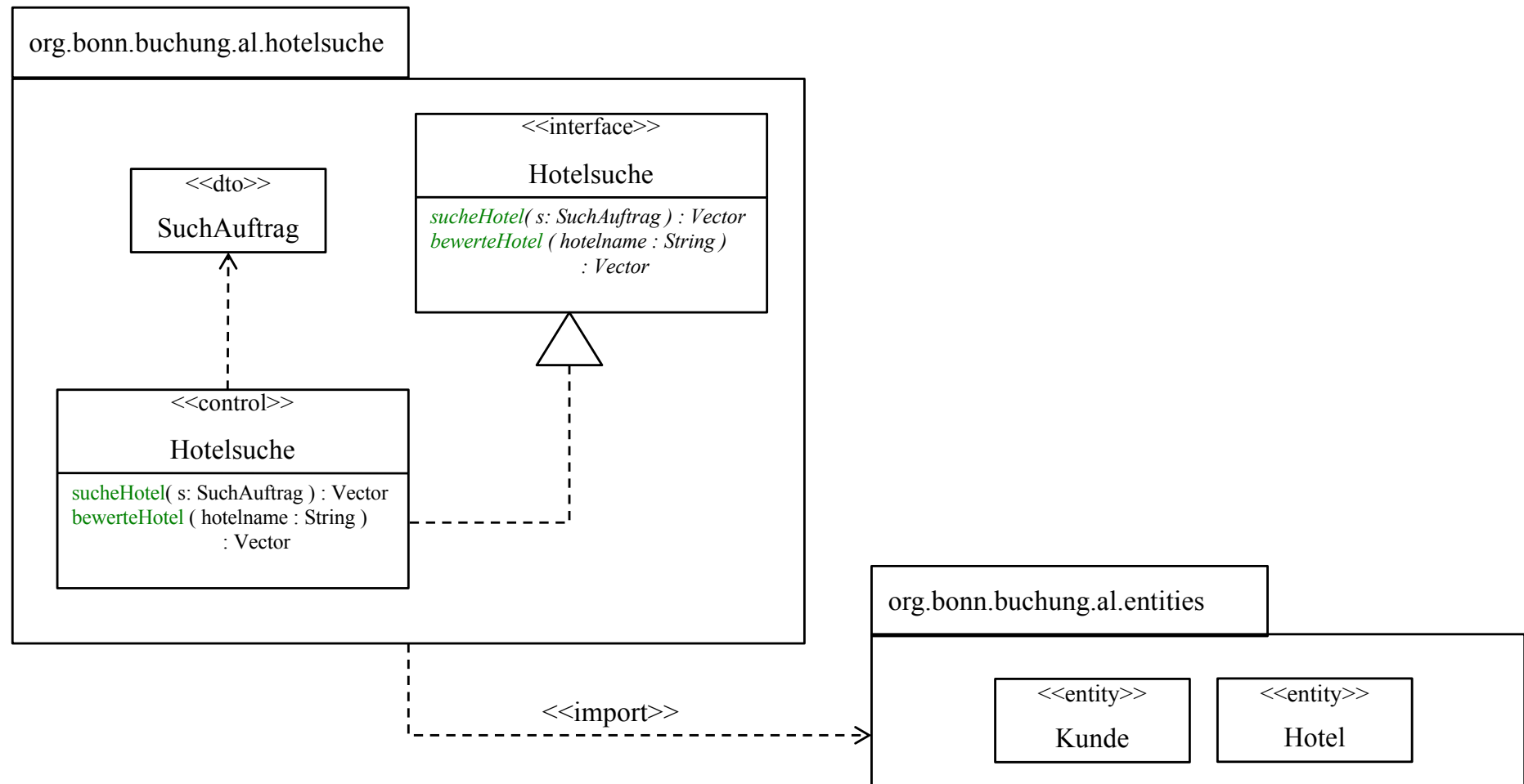
# Abbildung der Objekt-Typen das Layer Muster & MVC





# Abbildung Objektmodell auf eine Architektur

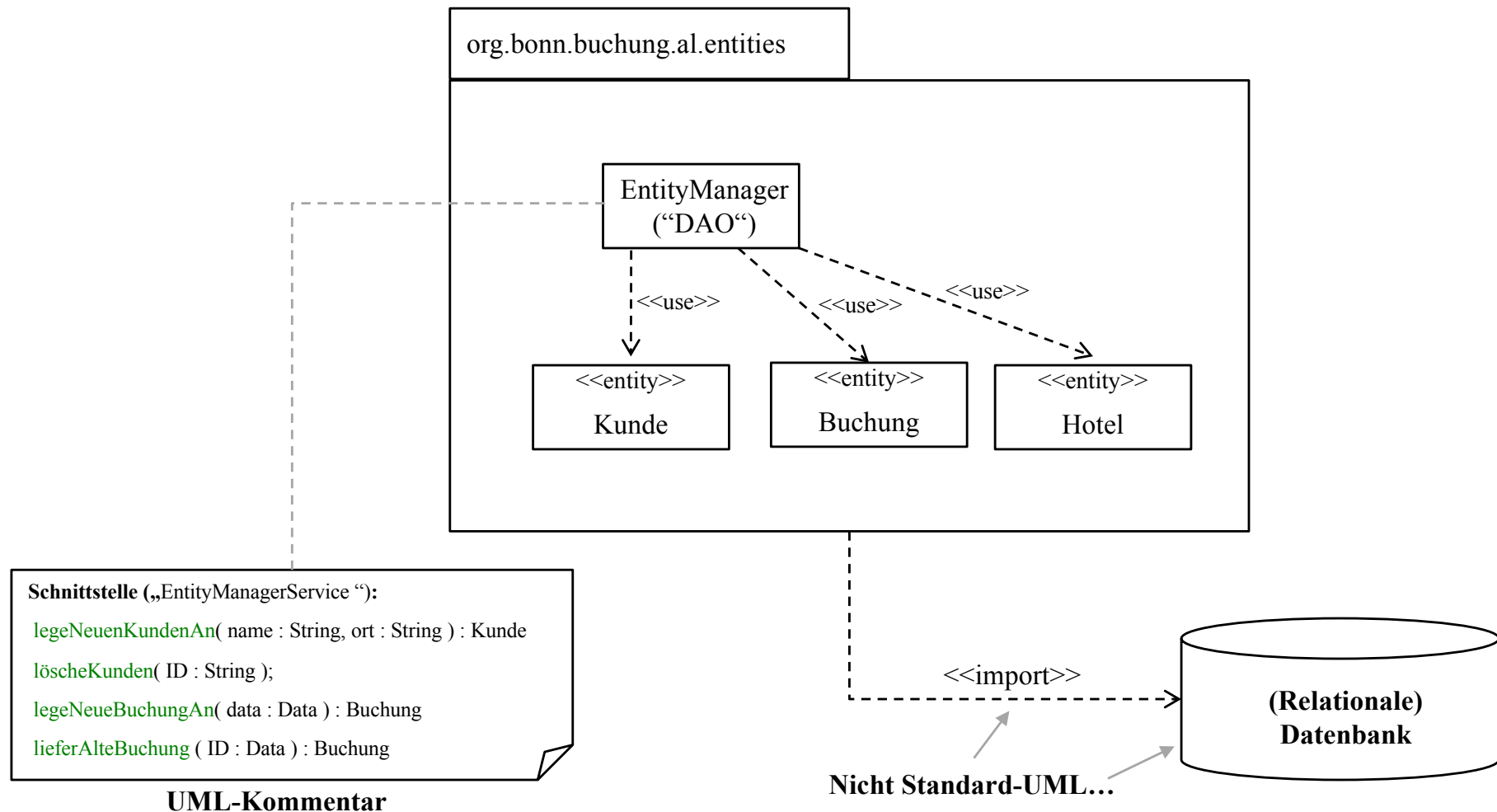
- Innerhalb einer Schicht können die Klassen zu Subsystemen zusammengefasst werden sowie **angebotene** und **benötigte** Schnittstellen definiert werden. Beispiel AL-Schicht:





# Abbildung Objektmodell auf eine Architektur

- Die Verwaltung (Erzeugen, Lesen, Update, Löschen) auf Entities kann über eine zentrale Klasse („EntityManager“) erfolgen
- Vertiefung ebenfalls in SE-2: Data Access Object (DAO) Pattern



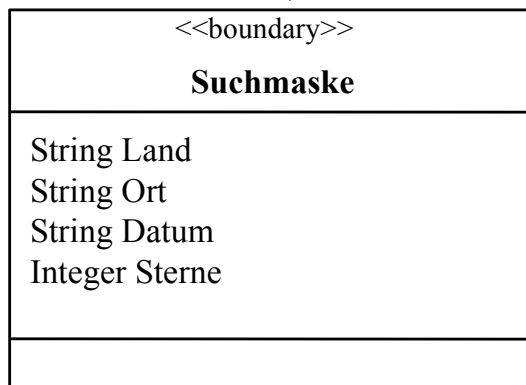


# Abbildung von Benutzerschnittstellen

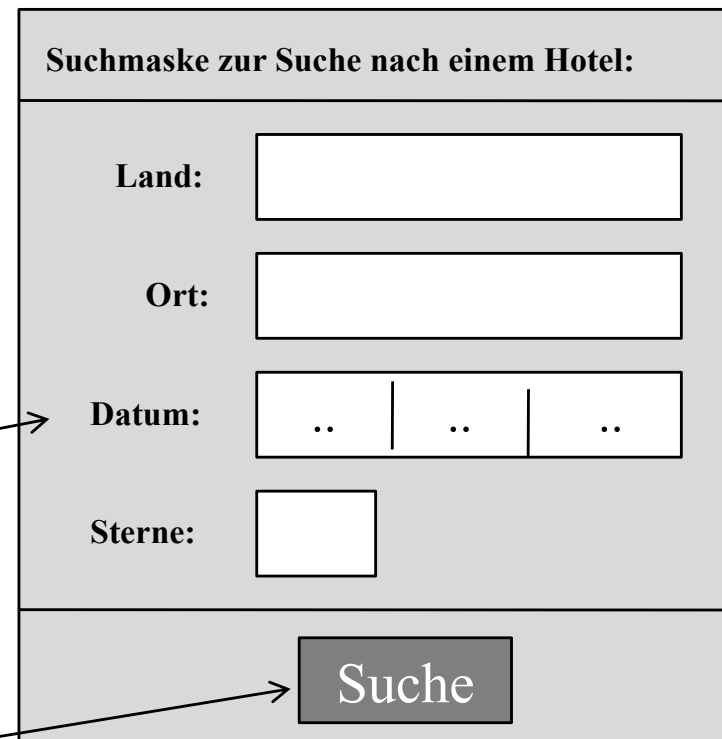
- Für Boundary-Objekte, die Benutzerschnittstellen repräsentieren, werden in der Regel keine Methoden definiert
- Attribute (z.B. was muss ein Benutzer eingeben für eine Suche) können über den Aufbau der Benutzerschnittstelle Auskunft geben (Input für Mock-Up und / oder Implementierung, Page Flow)

Der Anwender soll nach Hotels in der **Suchmaske** suchen können. Als Suchkriterien gibt er das **Land** und den **Ort** an sowie das **Datum** und die **Sterne**.

Auszug aus Use Case „Hotel Suchen“



Button, ruft Operation im Control-Objekt auf



Mock-Up (Paper Prototype)



## Kapitel 5: System Design (Grundlagen von Software-Architekturen)

1	Wiederholung und Motivation	✓
2	Definition und Eigenschaften einer Software-Architektur	✓
3	Modellierung von Software-Architekturen mit UML	✓
4	Grundlegende Architekturmuster	✓
5	Abbildung von Anforderungen auf Architektur-Konzepte	✓
<b>6</b>	<b>Zusammenfassung und Ausblick</b>	





# Zusammenfassung und Ausblick

---

## Zusammenfassung

- In der Phase des System Designs wird die Software-Architektur des Systems entwickelt
- Herleitung von grundlegenden Entwurfsentscheidungen für das gesamte System:
  - Dekomposition des Systems in elementare Bestandteile (Subsysteme) und ihren zentralen Abhängigkeiten
  - Entwicklung von weiteren Architektur-Direktiven (z.B. Hardware-Mapping, Bestimmung der Persistenz-Strategie)

## Ausblick

- In der folgenden Phase des Object Designs werden die einzelnen Subsysteme weiter in Richtung einer konkreten Umsetzung entworfen



- Euler, E.E., Jolly, S.D., and Curtis, H.H. "The Failures of the Mars Climate Orbiter and Mars Polar Lander: A Perspective from the People Involved." Proceedings of Guidance and Control 2001, American Astronautical Society, paper AAS 01-074, 2001.
- Melvin E. Conway: *How Do Committees Invent?*. In: F. D. Thompson Publications, Inc. (Hrsg.): *Datamation*. 14, Nr. 5, April 1968, S. 28–31
- Starke, G.: *Effektive Software-Architekturen*, 5. Auflage, Hanser, 2011
- Starke, G.: *Effektive Software-Architekturen*, 7. Auflage, Hanser, 2015