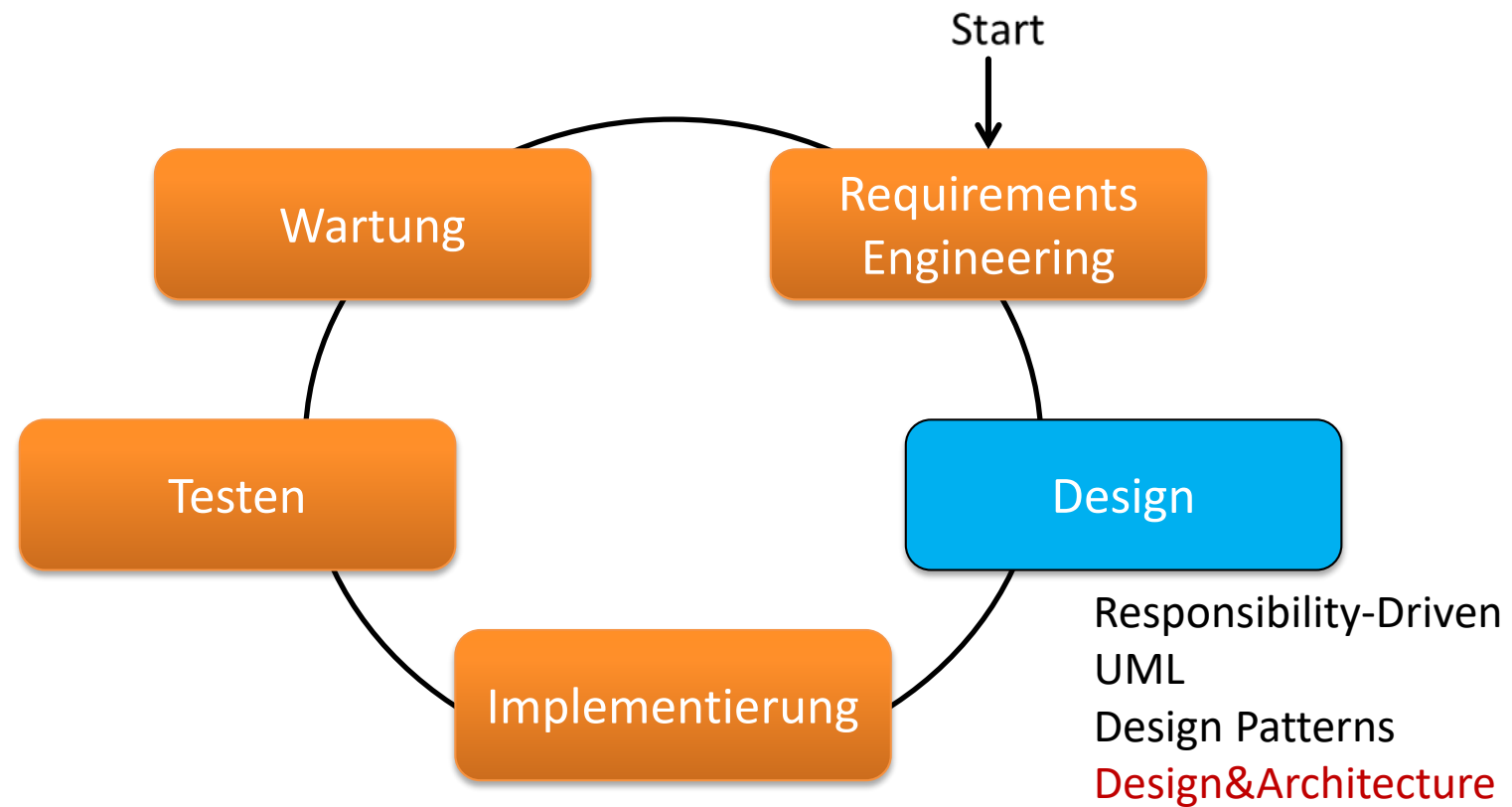


# Software Engineering

## Design & Architektur

Authors of slides:  
Norbert Siegmund  
Janet Siegmund  
Oscar Nierstrasz  
Sven Apel

# Einordnung



# Lernziele

- Wichtige Kriterien für das Design von Software kennenlernen
- Qualität von Software-Design bewerten können
- Arten von Software-Architekturen kennen
- Besonderheiten des Model-Driven Engineerings wissen



## Warum Design von Software?

- Für jedes Verhalten gibt es unendlich viele Programme
  - Wie unterscheiden sich diese Varianten?
  - Welche Variante ist die beste?
- Wie soll Variante designed werden, damit sie gewünschte Eigenschaften hat?
- Kosten für Fehler werden größer, je später Fehler bzw. Schwächen entdeckt werden; darum gute Modellierung!

# Qualität von Software-Design



# Was ist Qualität?

- Interne Qualität
  - Erweiterbarkeit, Wartbarkeit, Verständlichkeit, Lesbarkeit
  - Robust gegenüber Änderungen
  - Coupling und Cohesion
  - Wiederverwendbarkeit
  - Typischerweise beschrieben als Modularität
- Externe Qualität
  - Korrektheit: Erfüllung der Anforderungen
  - Einfachheit in der Benutzung
  - Ressourcenverbrauch
  - Legale und politische Beschränkungen

## Design

- Nach der Modellierung werden Methoden definiert und zu Klassen zugeordnet sowie die Kommunikation zwischen den Objekten festgelegt, um die spezifizierten Anforderungen zu erfüllen.
- Wie genau?
  - Welche Methode kommt wohin?
  - Wie sollen die Objekte interagieren?
  - Wichtige, kritische, nicht-triviale Fragestellung!

# Fünf Kriterien für gutes Design





# Modularität

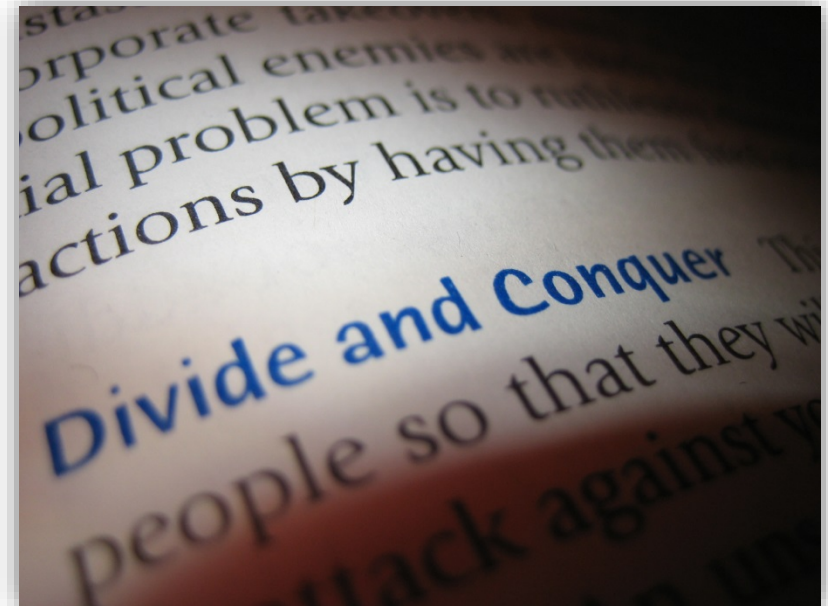
- Beschreibt, in wie weit man ein Softwaresystem aus autonomen Elementen bauen kann, die mit einer kohärenten, einfachen Struktur miteinander verbunden sind
- Dabei helfen 5 Kriterien und 5 Regeln

## Fünf Kriterien

- Modular Decomposability
- Modular Composability
- Modular Understandability
- Modular Continuity
- Modular Protection

## Fünf Kriterien: Modular Decomposability

- Problem kann in wenige kleinere, weniger komplexe Sub-Probleme zerlegt werden
- Sub-Probleme sind durch einfache Struktur verbunden
- Sub-Probleme sind unabhängig genug, dass sie einzeln bearbeitet werden können



## Fünf Kriterien: Modular Decomposability

- Modular decomposability setzt voraus: Teilung von Arbeit möglich
- Beispiel: Top-Down Design
- Gegenbeispiel: Ein globales Initialisierungsmodul, eine große Main-Methode

## Fünf Kriterien: Modular Composability

- Gegenstück zu modular decomposability
- Softwarekomponenten können beliebig kombiniert werden
- Möglicherweise auch in anderer Domäne
- Beispiel: Tischreservierung aus NoMoreWaiting kann auch für das Vormerken von Büchern benutzt werden (gutes Design!)

## Fünf Kriterien: Modular Understandability

- Entwickler kann jedes einzelne Modul verstehen, ohne die anderen zu kennen bzw. möglichst wenige andere kennen zu müssen
- Wichtig für Wartung
- Gilt für alle Softwareartefakte, nicht nur Quelltext
- Gegenbeispiel: Sequentielle Abhängigkeit zwischen Modulen

## Fünf Kriterien: Modular Continuity

- Kleine Änderung der Problemspezifikation führt zu Änderung in nur einem Modul bzw. möglichst wenigen Modulen
- Beispiel 1: Symbolische Konstanten (im Gegensatz zu Magic Numbers)
- Beispiel 2: Datendarstellung hinter Interface kapseln
- Gegenbeispiel: Magic Numbers, (zu viele) globale Variablen

## Fünf Kriterien: Modular Protection

- Abnormales Programmverhalten in einem Modul bleibt in diesem Modul bzw. wird zu möglichst wenigen Modulen propagiert
- Motivation: Große Software wird immer Fehler enthalten
- Beispiel: Defensives Programmieren
- Gegenbeispiel: Nullpointer in einem Modul führt zu Fehler in anderem Modul



## Aufgabe: Fünf Kriterien

- Finden Sie weitere Beispiele und Gegenbeispiele:
  - Modular Decomposability
    - Problem kann in wenige kleinere, weniger komplexe Sub-Probleme zerlegt werden
  - Modular Composability
    - Softwarekomponenten können beliebig kombiniert werden
  - Modular Understandability
    - Entwickler kann jedes einzelne Modul verstehen, ohne die anderen zu kennen
  - Modular Continuity
    - Kleine Änderung der Problemspezifikation führt zu Änderung in nur einem Modul
  - Modular Protection
    - Abnormales Programmverhalten in einem Modul bleibt in diesem Modul



# Fünf Regeln für gutes Design



# Fünf Regeln

---

- Fünf Regeln für qualitativ hochwertiges Software-Design:
  - Direct Mapping
  - Few Interfaces
  - Small Interfaces
  - Explicit Interfaces
  - Information Hiding

# Fünf Regeln: Direct Mapping

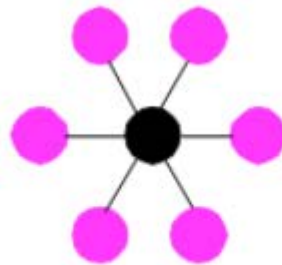
---

- Modulare Struktur des Softwaresystems sollte modularer Struktur des Modells der Problemdomäne entsprechen
- Folgt aus continuity und decomposability
- A.k.a. “low representational gap”[C. Larman]

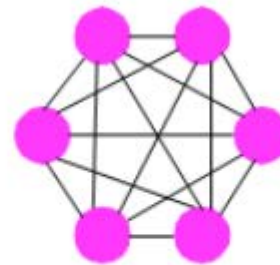
# Fünf Regeln: Few Interfaces

- Jedes Modul sollte mit möglichst wenigen anderen Modulen kommunizieren
- Folgt aus continuity and protection
- Struktur mit möglichst wenigen Verbindungen

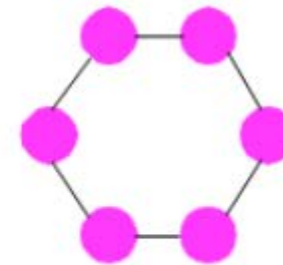
*Types of module  
interconnection  
structures*



(A)



(B)



(C)

# Fünf Regeln: Small Interfaces

- Wenn zwei Module kommunizieren, sollten sie so wenig Informationen wie möglich austauschen
- Folgt aus continuity und protection, notwendig für composability
- Gegenbeispiel: Big Interfaces 😊
  - Wenn ich jede Methode ins Interface aufnehme, muss jede Klasse, die das Interface implementiert, alle Methoden realisieren -> Vorteil geht verloren
  - Man weiß nicht, welche Methoden entscheidend sind
  - Zu viele Einfallstore für Fehler

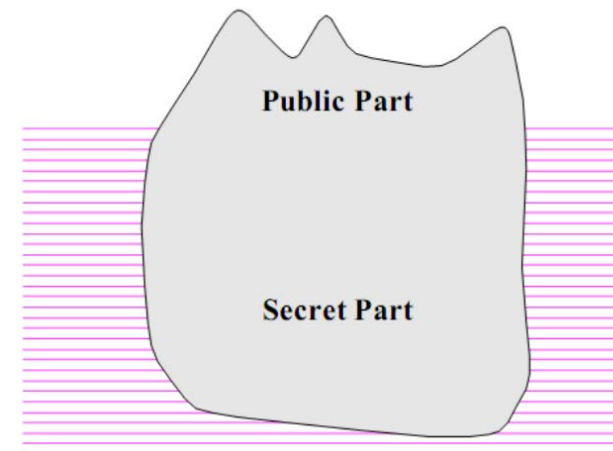
# Fünf Regeln: Explicit Interfaces

---

- Wenn zwei Module miteinander kommunizieren, muss das aus dem Interface von mindestens einem hervorgehen
- Gegenbeispiel: Globale Variablen
  - Wer benutzt sie?
  - Welche Auswirkungen (und wo) werden durch Änderungen an der globalen Variable verursacht?
  - Wer ändert sie und wann?
  - Wer war verantwortlich für einen inkonsistenten oder fehlerhaften Zustand?

# Fünf Regeln: Information Hiding

- Jedes Modul muss eine Teilmenge seiner Eigenschaften definieren, die nach außen gezeigt werden
- Alles andere wird “versteckt”
- Nicht nur Inhalt, auch Implementierung wird versteckt
- Impliziert durch continuity





# Aufgabe: Fünf Regeln

- Finden Sie weitere Beispiele und Gegenbeispiele:
  - Direct Mapping
    - Modulare Struktur des Software Systems sollte modularer Struktur des Modells der Problem-Domäne entsprechen
  - Few Interfaces
    - Jedes Modul sollte mit möglichst wenigen anderen Modulen kommunizieren
  - Small Interfaces
    - Wenn zwei Module kommunizieren, sollten sie so wenig Informationen wie möglich austauschen
  - Explicit Interfaces
    - Wenn zwei Module miteinander kommunizieren, muss das aus dem Interface von mindestens einem hervorgehen
  - Information Hiding
    - Jedes Modul muss eine Teilmenge seiner Eigenschaften definieren, die nach außen gezeigt werden

# GRASP – Pattern

General Responsibility Assignment  
Software Patterns

# GRASP Pattern

---

- Allgemeine Lernhilfe, um
  - grundlegendes objekt-orientiertes Designen zu verstehen
  - Design-Entscheidungen methodisch, rational und erklärbar zu treffen
- Basiert auf Responsibilities (Responsibility-Driven Design)

# Responsibilities

---

- Bezogen darauf, wie sich Objekt verhalten soll
- Zwei Arten
  - Wissen
    - Wissen über private Daten
    - Wissen über Objekte, die sich auf einander beziehen
    - Wissen, was es ableiten oder berechnen kann
  - Tätigkeit
    - Etwas selbst tun, z.B. Objekt erstellen oder berechnen
    - Aktionen in anderen Objekten initiieren
    - Kontrolle und Koordination von Aktivitäten anderer Objekte

# GRASP-Pattern

---

- Information Expert
- Creator
- Low Coupling
- High Cohesion

# GRASP-Prinzip: Information Expert

---

- Wer soll die Responsibilities bekommen?
- Das Objekt, das die meisten Informationen hat, diese Responsibility zu erfüllen
- Beispiel NoMoreWaiting: Wer hat die Information, ob ein Tisch frei ist oder nicht?

# GRASP-Prinzip: Creator

- Wer erstellt Instanzen eines Objekts?
- Creator braucht erstelltes Objekt häufig in seinem Lebenszyklus
- Objekt B bekommt Verantwortung, Objekt A zu erstellen, wenn:
  - B A-Objekte aggregiert
  - B A-Objekte enthält
  - B Instanzen von A-Objekten loggt
  - B Initialisierungsdaten von A hat

# GRASP-Prinzip: High Cohesion

Cohesion ist ein Maß, *wie gut Teile einer Komponente “zusammen gehören”*.

- Cohesion ist schwach, wenn Elemente nur wegen ihrer Ähnlichkeit ihrer Funktionen zusammengefasst sind (z.B., `java.lang.Math`).
- Cohesion ist stark, wenn alle Teile für eine Funktionalität tatsächlich benötigt werden (z.B. `java.lang.String`).
  - Starke cohesion *verbessert Wartbarkeit* and Adaptivität durch *die Einschränkung des Ausmaßes von Änderungen* auf eine kleine Anzahl von Komponenten.

Es gibt viele Definitionen und Interpretationen von cohesion.  
*Die meisten Versuche, dies formal zu definieren, sind inadäquat!*



# GRASP-Prinzip: High Cohesion

---

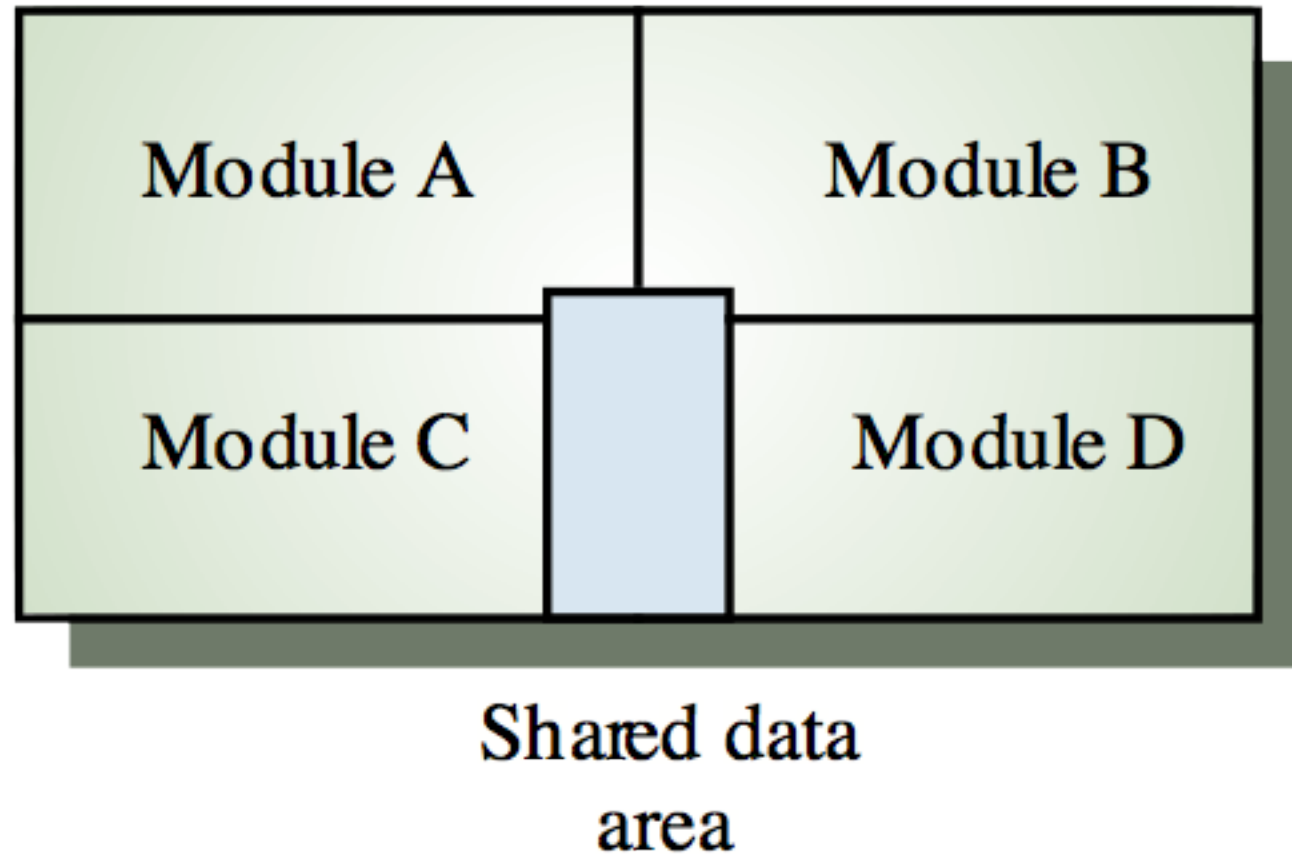
- Responsibilities so zuordnen, dass Kohäsion hoch ist
- Faustregel: Klasse mit starker Kohäsion hat meistens wenige Methoden, die verwandte Funktionalität haben, und macht nicht zu viel (d.h., ist keine Gottklasse)

# GRASP-Prinzip: Loose Coupling

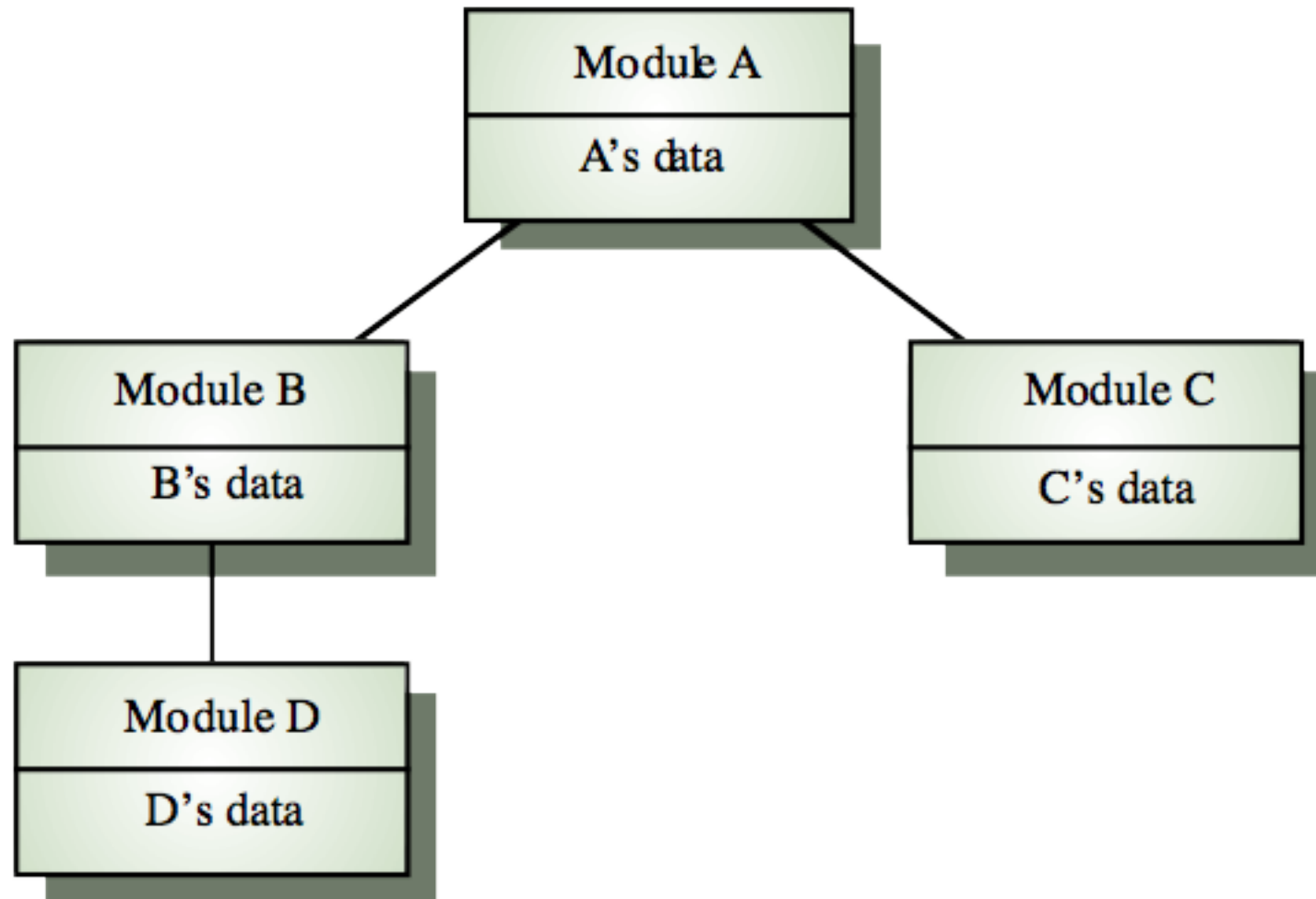
Coupling ist ein Maß der *Stärke der Verbindungen* zwischen Systemkomponenten.

- Coupling ist eng (tight) zwischen Komponenten, wenn sie stark voneinander abhängig sind (z.B., wenn sehr viel Kommunikationen zw. beiden statt findet).
- Coupling ist lose (loose), wenn es nur wenige Abhängigkeiten zwischen Komponenten gibt.
  - Loose coupling *verbessert Wartbarkeit* und Adaptibilität, da *Änderungen in einer Komponente sich weniger wahrscheinlich auf anderen Komponenten auswirkt.*
  - Responsibility so zuteilen, dass coupling schwach ist

# Tight Coupling



# Loose Coupling



# Was ist Software Architektur?



# Was ist Software Architektur?

---

*A neat-looking drawing of some boxes, circles, and lines, laid out nicely in Powerpoint or Word, does not constitute an architecture.*

— D'Souza & Wills

# Was ist (wirklich) Software Architektur?

Die Architektur eines Systems besteht aus:

- der *Struktur(en) ihrer Teile*
  - einschließlich Design-, Test und Laufzeit Hardware und Software Teile
- den *extern sichtbaren Eigenschaften* dieser Teile
  - Module mit Interfaces, Hardware-Einheiten und Objekte
- den *Beziehungen und Bedingungen* zwischen ihnen

*In anderen Worten:*

The set of *design decisions* about any system (or subsystem) that keeps its implementors and maintainers from exercising “*needless creativity*”.

# Design vs. Architektur

---

- Design beschreibt Aufbau von Subsystemen und Komponenten (fein granular)
  - Welche Klassen gibt es (in Modul X) und wie interagieren sie?
- Architektur beschreibt den groben Aufbau eines Systems (welche Komponenten gibt es?)
  - Welche Komponenten / Module gibt es und wie interagieren sie?



# Sub-systeme, Module und Komponenten

- Ein Sub-system ist selbst ein System, dessen Operation *unabhängig* von den Leistungen und Funktionen anderer Sub-systeme ist.
- Ein Modul ist eine Systemkomponente, die *Dienstleistungen / Funktionen anbietet*, welche andere Komponenten benötigen, aber welche nicht als komplett separates System angesehen werden.
- Eine Komponente ist eine *unabhängig auslieferbare Einheit* von Software, die ihr Design und Implementierung eingeschlossen hat (hiding) und ihr Interface zur Außenwelt anbietet, so dass sie mit anderen Komponenten zusammengefasst werden kann, um ein größeres System zu bilden.

# Arten von SW Architektur



# Parallelen zur (echten) Architektur

- Architekten sind die *Schnittstelle* zwischen den Kunden und den Auftragnehmern, die das System/Gebäude bauen
- Eine schlechte Architektur für ein Gebäude *kann nicht mehr durch gute Konstruktion gerettet werden*— gleiches gilt für Software
- Es gibt *spezielle Typen* von Gebäuden und Software-Architekten.
- Es gibt *Schulen oder Styles* des Bauens und der Software –Architektur.

# Architectural Styles

---

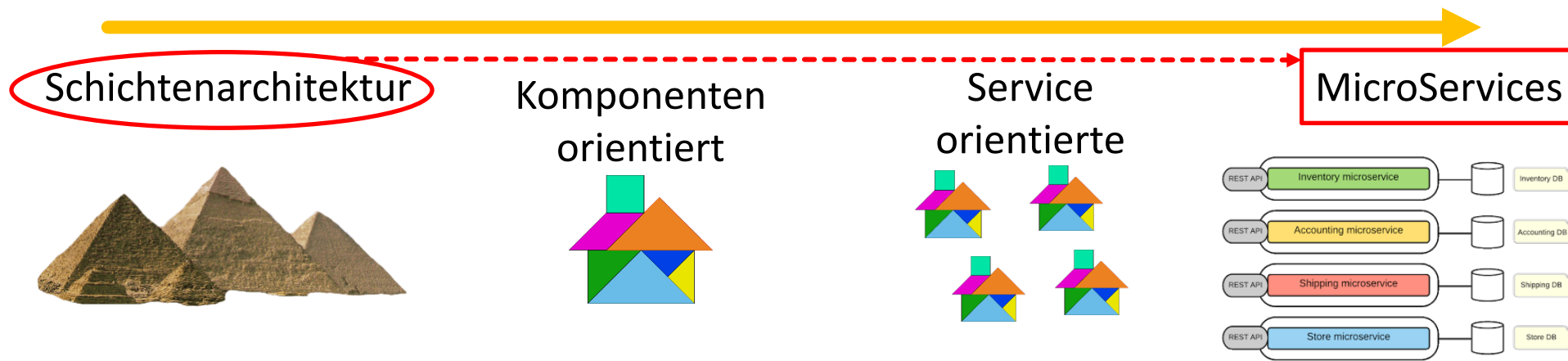
*An architectural style defines a **family of systems** in terms of a pattern of structural organization. More specifically, an architectural style defines a vocabulary of **components** and **connector** types, and a set of **constraints** on how they can be combined.*

— Shaw and Garlan

# SW-Architekturen für große Softwaresysteme

Monolith

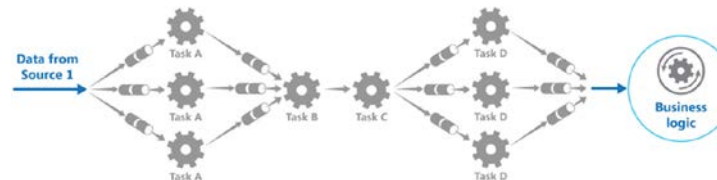
System of Systems



Spezialarchitekturen:

Pipes and Filters

Peer-to-Peer



# Welche Architektur?

amazon.de  
Prime

Koffer, Rucksäcke & Taschen

Q

Prime music Über 2 Millionen Songs. Ohne Werbung.

DE

Hallo, Jonas  
Mein Konto

Mein Prime

Meine Listen

Einkaufswagen

Amazon Fashion

DAMEN

HERREN

KINDER & BABY


GEPÄCK

MARKEN

SALE

KOSTENLOSER RÜCKVERSAND  
Innerhalb von 30 Tagen > Mehr Informationen

Koffer, Rucksäcke & Taschen > Taschen > Umhängetaschen



Für größere Ansicht Maus über das Bild ziehen

AmazonBasics Tasche für Laptop / Tablet mit Bildschirmdiagonale 15,6 Zoll / 39,6 cm

von AmazonBasics

★★★★★ 616 Kundenrezensionen | 64 beantwortete Fragen

Bestseller Nr. 1 in Notebook-Aktentaschen

Preis: EUR 19,49 Prime  
Alle Preisangaben inkl. USt

Auf Lager.

Verkauf und Versand durch Amazon. Geschenkverpackung verfügbar.

Größe: 15,6 Zoll / 39,6 cm

11,6 Zoll / 29,5 cm  
EUR 13,99 Prime

14,1 Zoll / 35,8 cm  
EUR 14,99 Prime

15,6 Zoll / 39,6 cm  
EUR 19,49 Prime

17,3 Zoll / 44 cm  
EUR 19,99 Prime

- Schlanke und kompakte Tasche, perfekt für Laptops bis 29,5 cm (15,6 Zoll), ohne unnötigen Ballast
- Zubehörtaschen für Maus, iPod, Handy und Stifte
- Einschließlich gepolstertem Schulterriemen

[Weitere Produktdetails](#)

[Falsche Produktinformationen melden](#)

Möchten Sie Ihr Elektro- und Elektronik-Gerät kostenlos recyceln? [Erfahren Sie mehr.](#)

Teilen

Menge: 1

In den Einkaufswagen

oder

Jetzt mit 1-Click® kaufen

Innerhalb von 3h 9min bestellen und Lieferung erfolgt am:  
**Samstag, 21 Jan**  
(GRATIS Premiumversand)

Lieferort:  
Jonas Hecht- Weimar - 99423

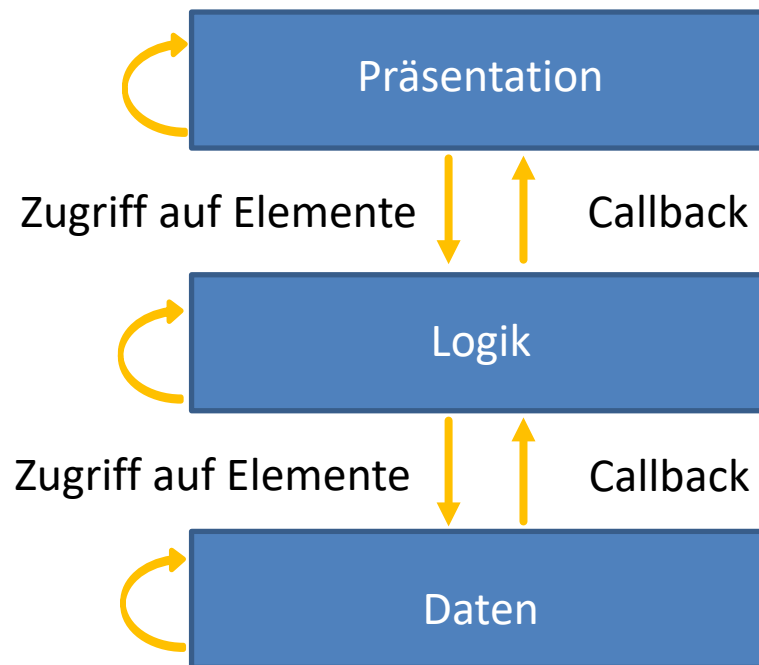
☐ Dies ist ein Geschenk

Auf die Liste

Möchten Sie verkaufen?  
Bei Amazon verkaufen

# Schichtenarchitekturen

- Eine Schichtenarchitektur organisiert ein System in eine Menge von Schichten, wobei jede Schicht eine Menge von Leistungen / Funktionen für die Schicht “darüber” anbietet.



## Vorteile:

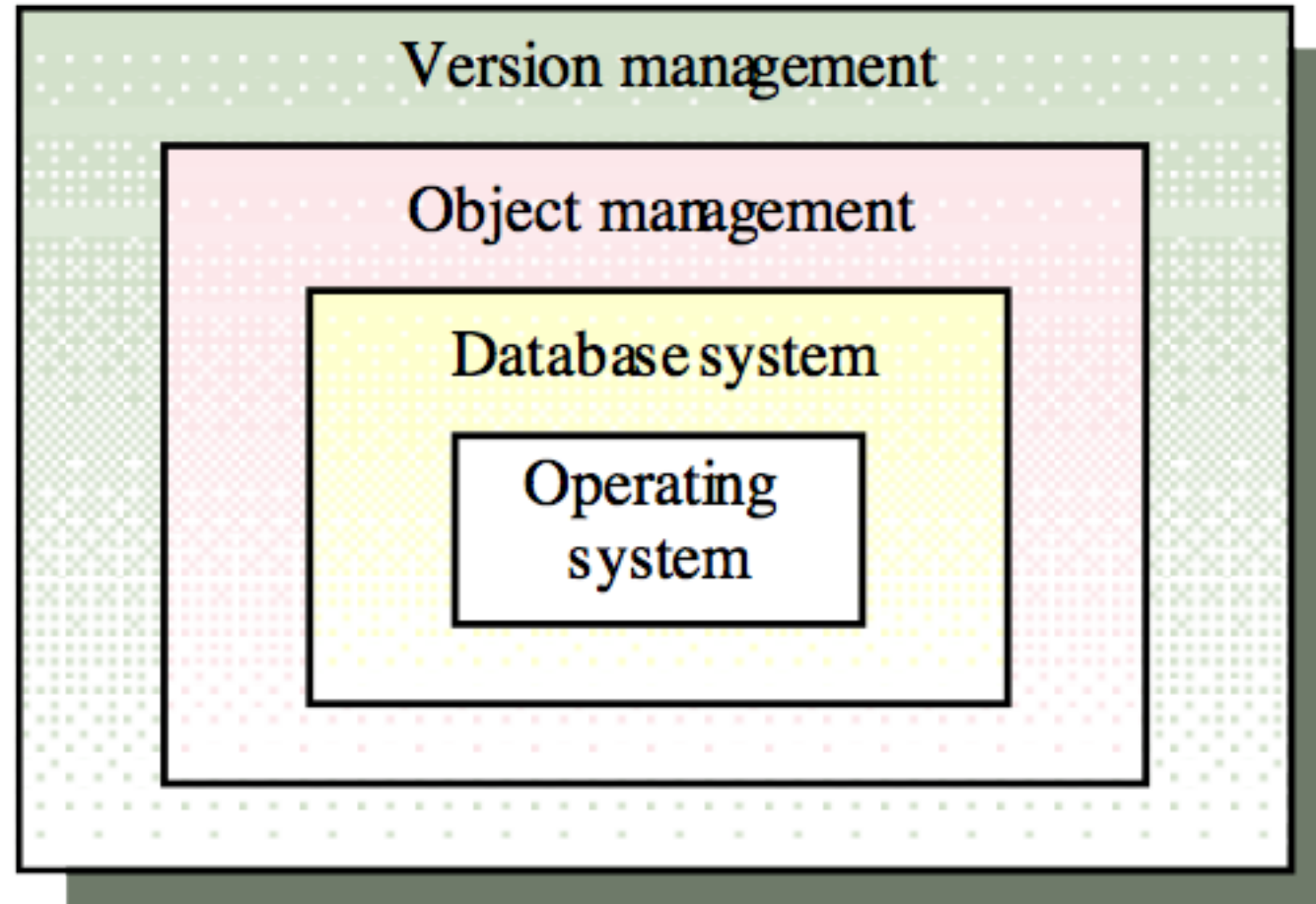
- Inkrementelle Entwicklung von Subsystemen
- Wenn ein Interface einer Schicht sich ändert, *sind nur benachbarte Schichten betroffen*
- Modularität, Kohäsion

# Layered Architectures

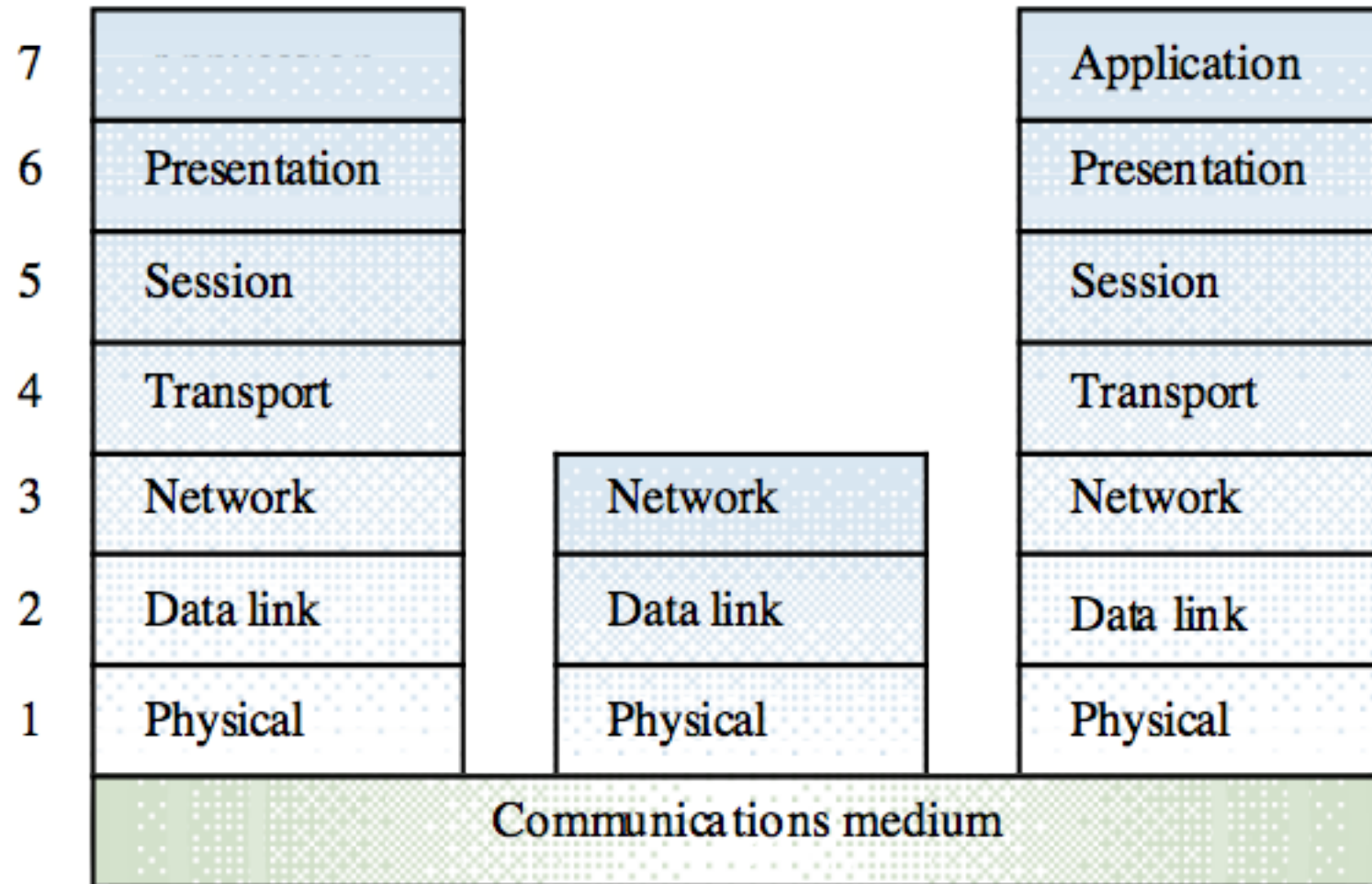
- Eine Schichtenarchitektur organisiert ein System in eine Menge von Schichten, wobei jede Schicht eine Menge von Leistungen / Funktionen für die Schicht “darüber” anbietet.
- Schichten sind normalerweise *beschränkt*, so dass Elemente nur
    - Andere Element in der gleichen Schicht oder
    - Elemente von der Schicht darunter sehen können
  - *Callbacks* können verwendet werden, um mit höheren Schichten zu kommunizieren
  - Unterstützt die *inkrementelle Entwicklung* von Sub-systemen in unterschiedlichen Schichten
    - Wenn ein Interface einer Schicht sich ändert, *sind nur benachbarte Schichten betroffen*.



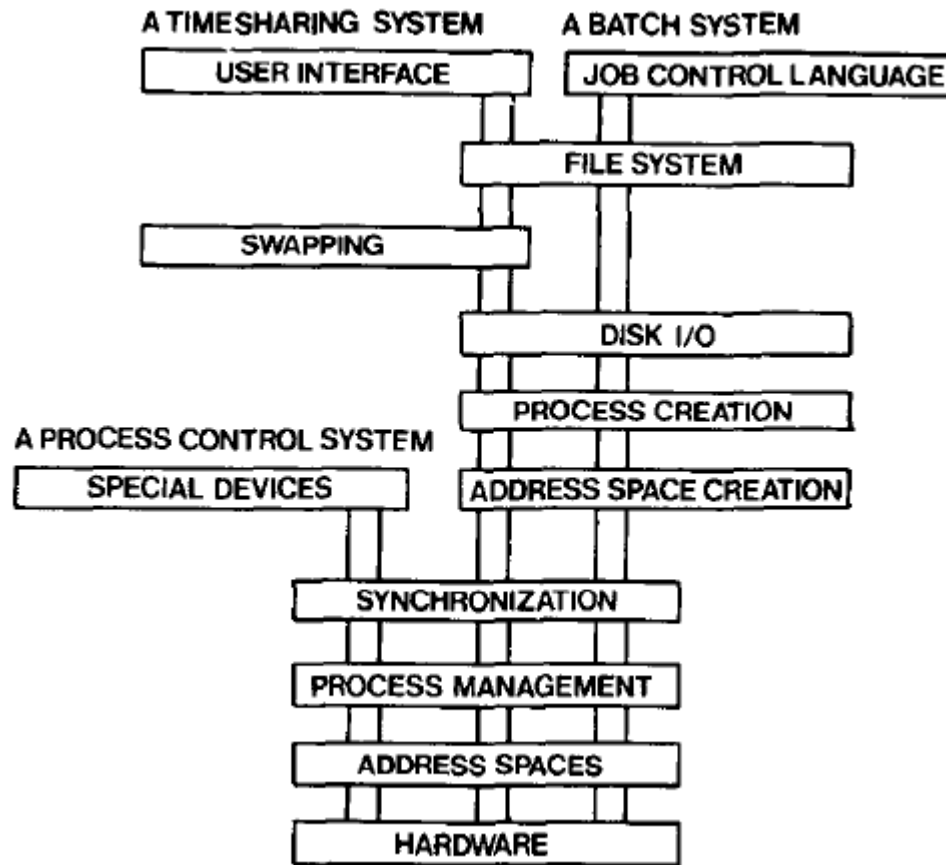
# Version Management System



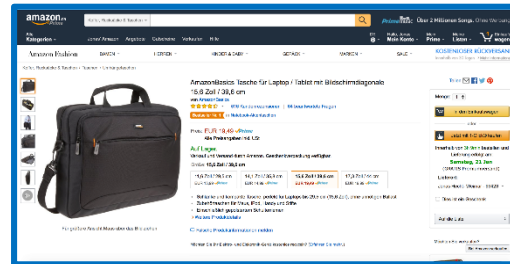
# OSI Reference Model



# Operating System Family



# Beispiel: 3-Layer Architektur



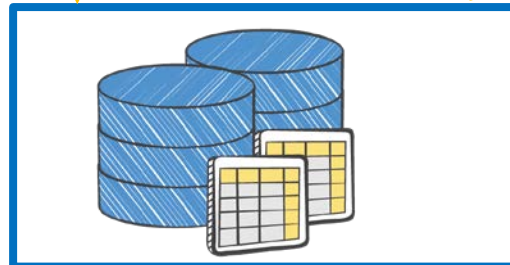
JavaScript Implementierung

↓ Apache, Tomcat ↑



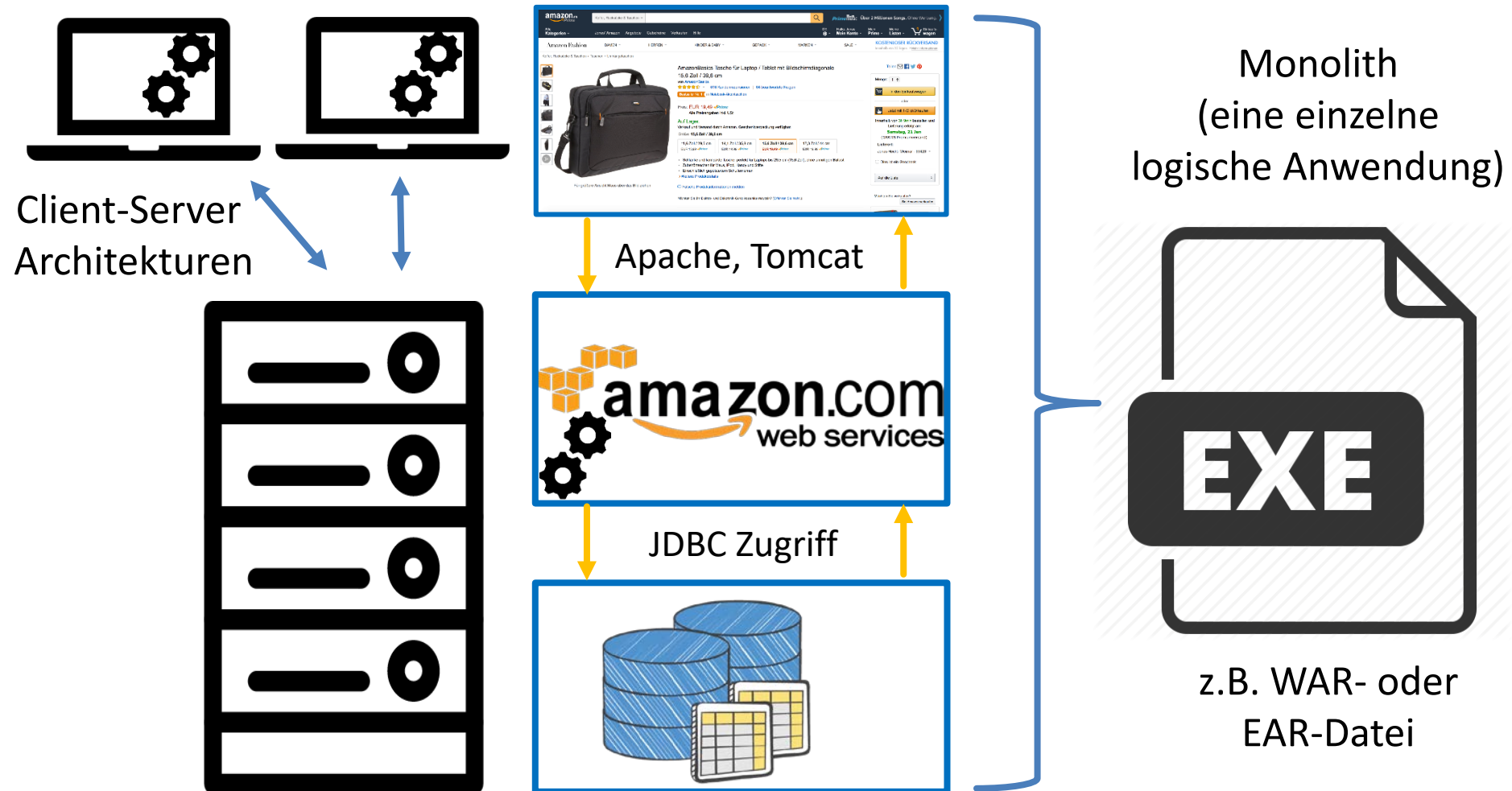
Java Implementierung

↓ JDBC Zugriff ↑



Oracle, etc.

# Client-Server & Monolith



# Client-Server Architektur

Eine Client-Server Architektur *verteilt Applikationslogik und Funktionalität* zu einer Anzahl von Klienten (clients) und Server-Subsystemen, wobei jede potentiell auf einer unterschiedlichen Maschine läuft und über das Netzwerk kommuniziert.

## Vorteile:

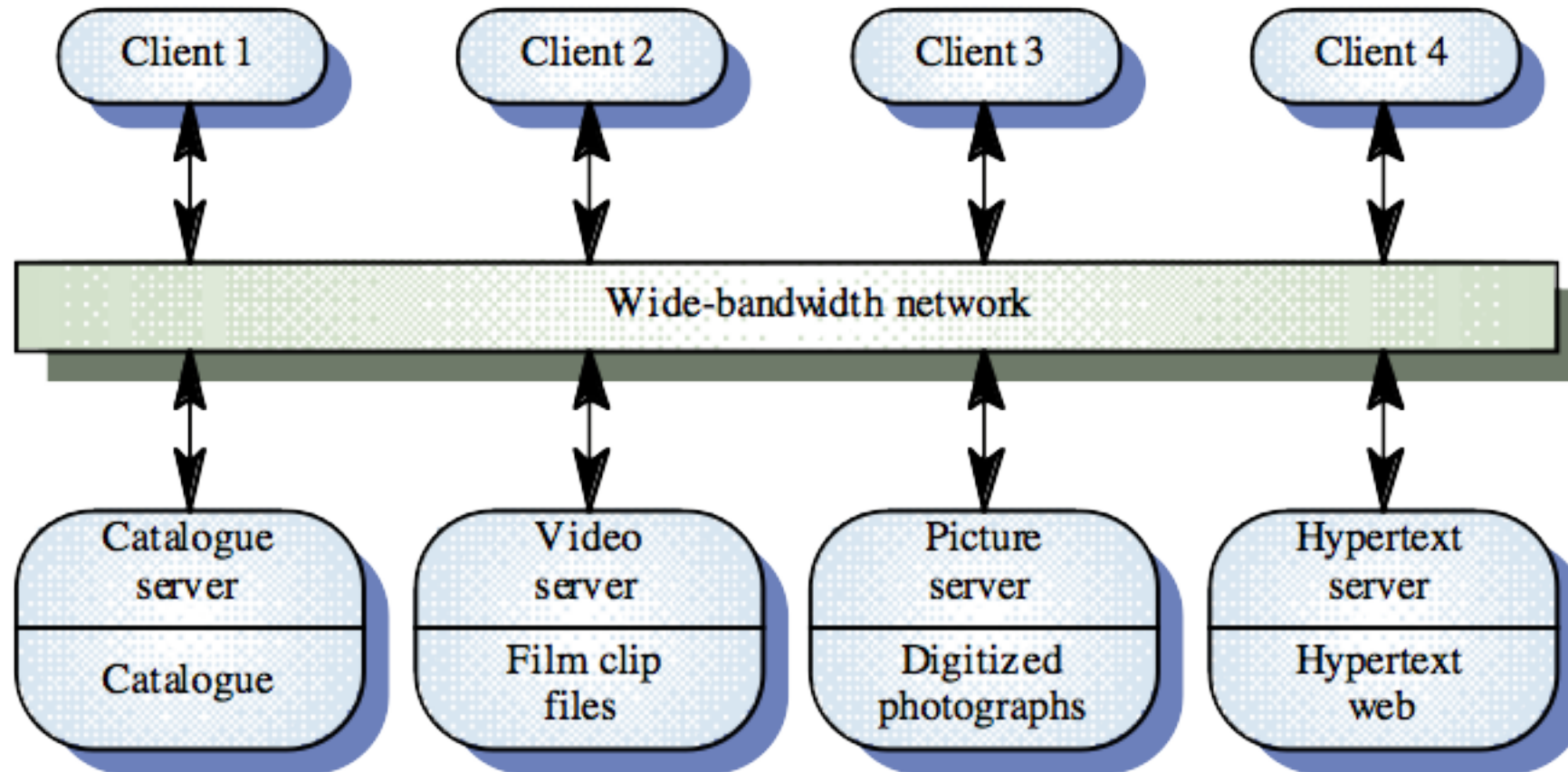
- Einfache *Datenverteilung*
- Effektive *Hardwareauslastung*
- Einfaches *Hinzufügen* neuer Server

## Nachteile:

- Kein *geteiltes Datenmodell*
- *Redundante* Verwaltung
- Evtl. *zentrale Registrierung* erforderlich  
(welcher Server stellt welche Dienstleistung zur Verfügung?)



# Film and Picture Library



# Und ohne Web?

Wie würde Sie die Architektur entwerfen, wenn wir eine Desktop-Anwendung schreiben würden?

Kommunikation / Controller

Apache, Tomcat, JavaScript

Sicht / View

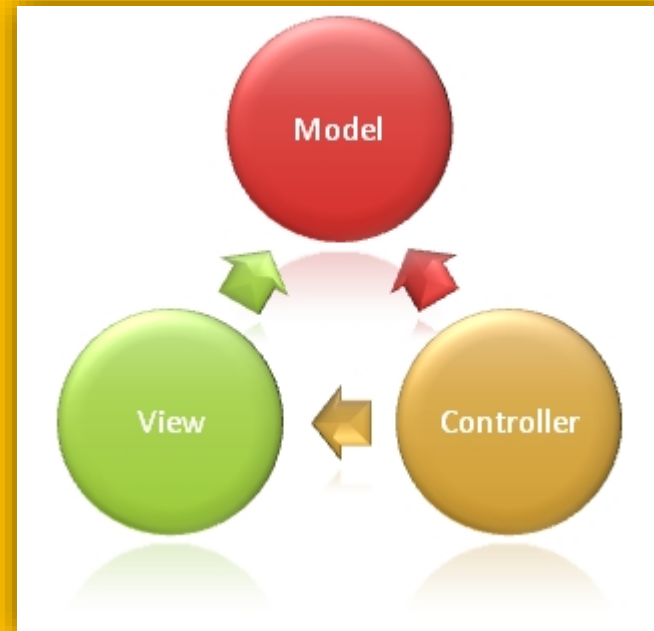
JavaScript Frontend

Java Implementierung, DB, ...

Model / Business-Logik / Daten



# Model-View-Controller

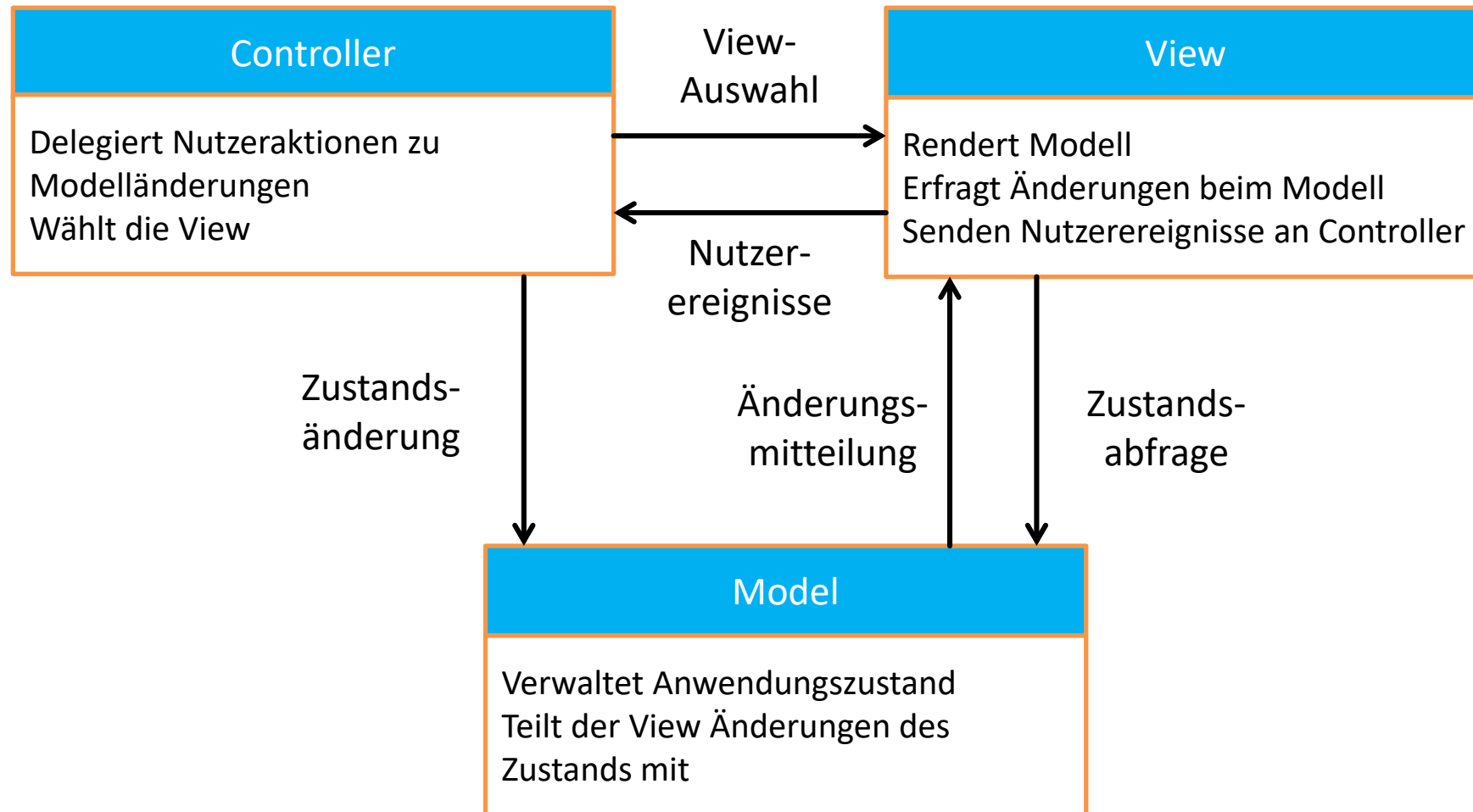


# Model-View-Controller (MVC) Architektur

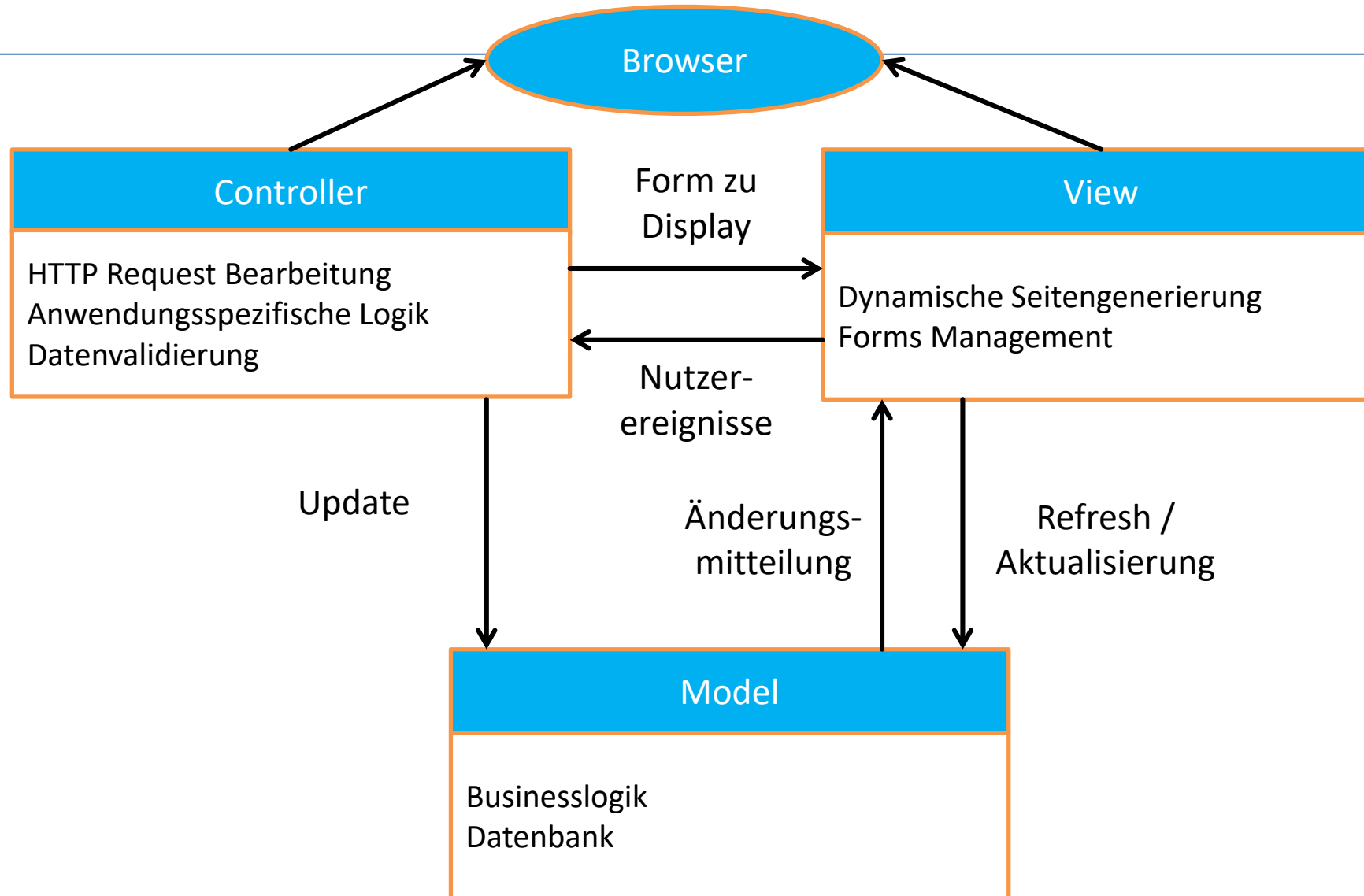
**Idee**: Separiere *Präsentation* und *Interaktion* von den *Daten* des Systems

- Das System ist strukturiert in drei Komponenten:
  - *Model*: verwaltet Systemdaten und Operationen auf den Daten
  - *View*: Präsentiert die Daten zum Nutzer
  - *Controller*: händelt Nutzerinteraktion; schickt Informationen zur View und zum Model
- Nützlich, wenn es mehrere Wege gibt auf die Daten zuzugreifen
- Ermöglicht das Ändern der Daten unabhängig von deren Repräsentation
- Unterstützt unterschiedliche Präsentationen der gleichen Daten

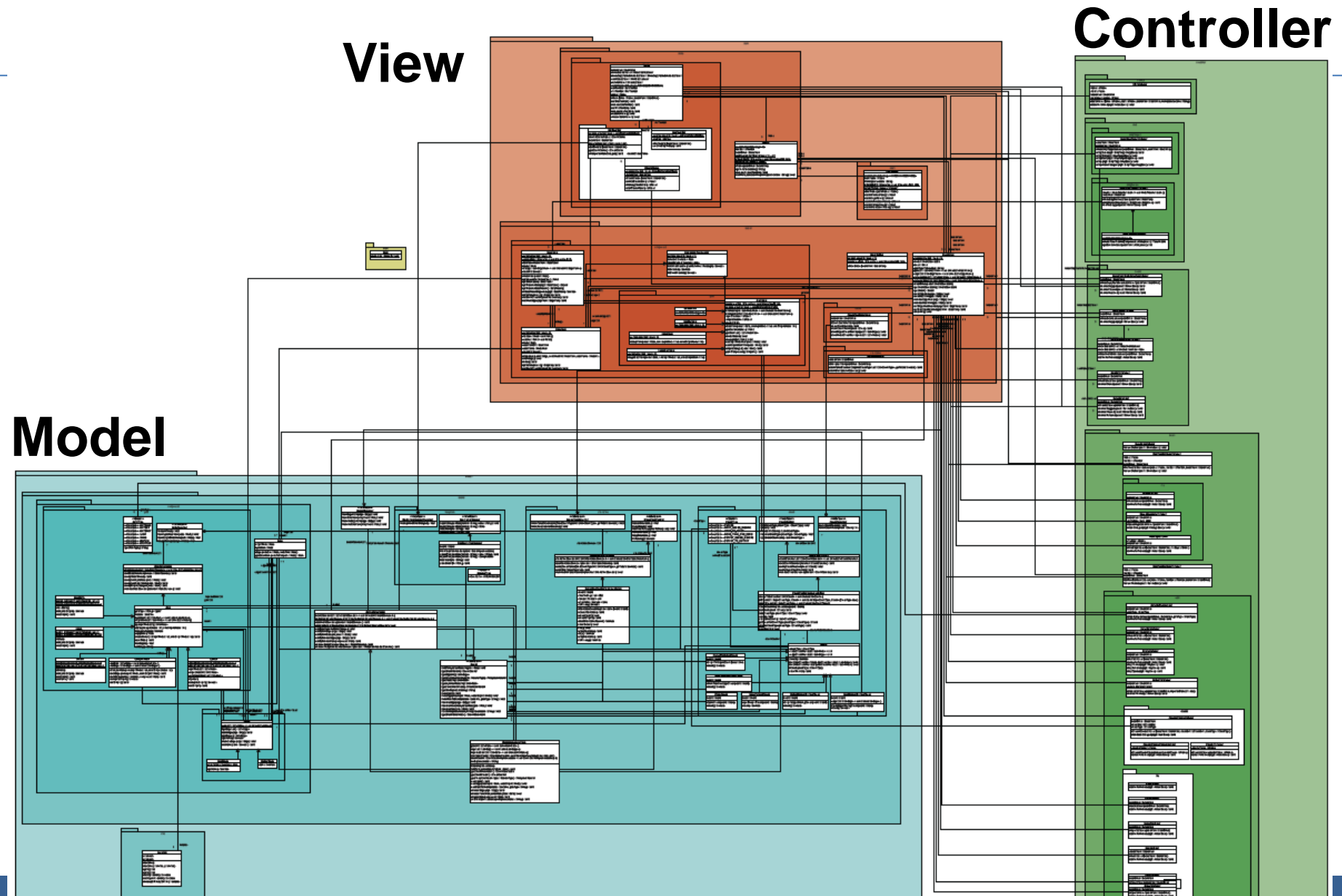
# MVC Übersicht



# MVC Beispiel

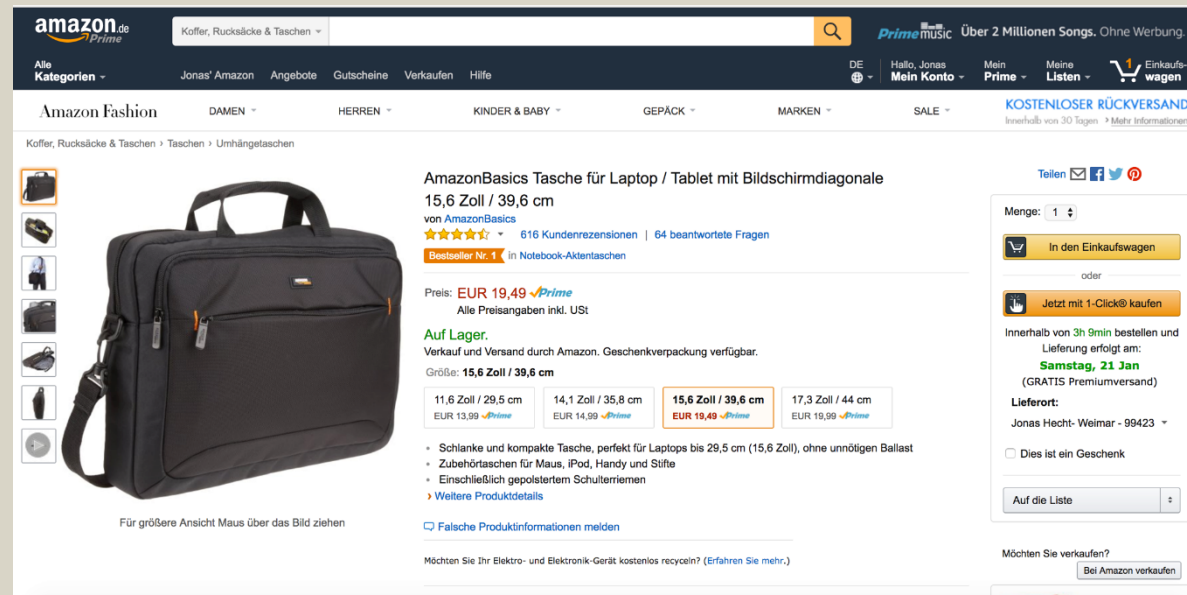


# MVC in Action (Circuit Simulation, SEP'11)



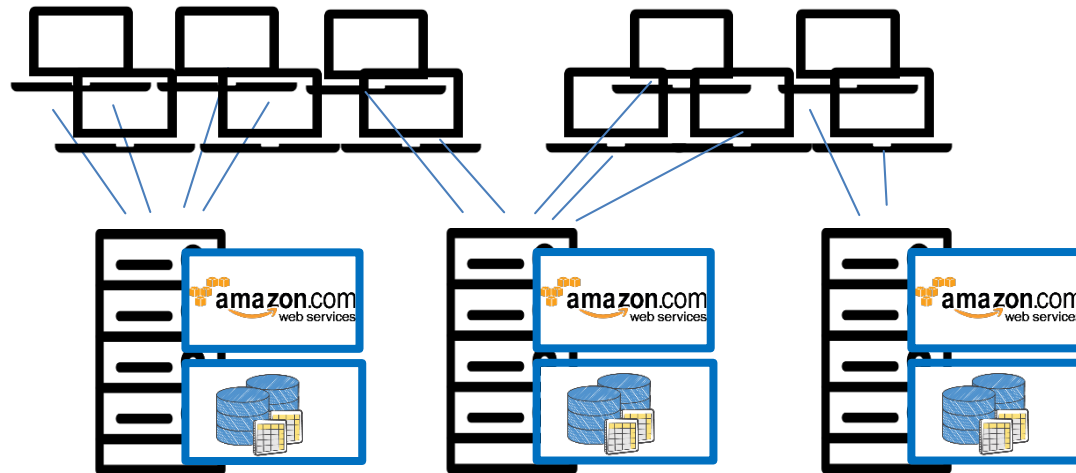
# Probleme der Architekturen

- Welche Probleme erwarten Sie, wenn wir diese Architekturen in der Praxis für unser Beispiel einsetzen?
- Hinweise:
  - Großes Softwaresystem
  - Hohes Nutzeraufkommen



# Probleme traditioneller Architekturmuster für große Softwaresysteme

- Häufige Änderungen
  - Monolithisches System muss komplett neu gebaut werden
  - Abhängigkeiten zwischen Subsystemen erschweren und verzögern Änderungen
- Skalierbarkeit der Hardware
  - Alle Architekturen sind schwer skalierbar, selbst Client-Server



Neue Probleme:

- Verteilte, *replizierte* Daten
- *Konsistenz* und Synchronität?
- *Gesamtes* System repliziert, obwohl nur Subsysteme ausgelastet sind

# Probleme traditioneller Architekturmuster für große Softwaresysteme

- Was passiert bei Ausfällen und Fehlern von Subsystemen?
  - Oft zu starke Kopplung der Subsysteme und kaum Möglichkeit des „Fail-overs“
  - Alternative Views bei MVC möglich, aber alternative Modelle?
- Wie wird das laufende System geupdatet?
  - Herunterfahren der Server ist keine Option
- Wie kann die Entwicklung von Subsystemen parallelisiert werden?
  - Schichtenarchitektur und MVC oft zu grob-granular
  - Komponenten und Services nicht gut bei querschneidenden Belangen und erfordern zu viel Glue-Code / Kommunikation
- Wie vereinfache ich Testen? Usw.



# Gewünschte Eigenschaften

---

- Schneller Austausch von Subsystemen ohne, dass das gesamte System betroffen ist (lose Kopplung + hohe Modularität)
- Skalierbarkeit bei großen Lasten von einzelnen Subsystemen (Beispiel: Black Friday)
- Weniger Abstimmungsaufwand innerhalb der Unternehmensorganisation (bei Entwicklern und Sachverständigen)
- Parallelisierung in der Entwicklung sowie Anwendbarkeit von agilen Methoden der Softwareentwicklung
- Einfache Testbarkeit von Subsystemen

# Welche Architektur?

The screenshot shows the Amazon.de product page for an AmazonBasics laptop bag. The page is annotated with colored boxes highlighting different architectural components:

- Green Box:** The product image of the laptop bag, with a zoom-in icon and the text "Für größere Ansicht Maus über das Bild ziehen".
- Blue Box:** The product title and description, including the price, availability, and size options.
- Red Box:** The purchase options, including the quantity selector, "In den Einkaufswagen" button, "Jetzt mit 1-Click® kaufen" button, delivery date, and location.
- Orange Box:** The search bar and navigation links at the top of the page.

**Product Details:**

- AmazonBasics Tasche für Laptop / Tablet mit Bildschirmdiagonale 15,6 Zoll / 39,6 cm**
- von AmazonBasics**
- 5 Sterne** (616 Kundenrezensionen | 64 beantwortete Fragen)
- Bestseller Nr. 1** in Notebook-Aktentaschen
- Preis: EUR 19,49** (Prime)
- Auf Lager.**
- Verkauf und Versand durch Amazon.** Geschenkverpackung verfügbar.
- Größe: 15,6 Zoll / 39,6 cm**
- 11,6 Zoll / 29,5 cm** (EUR 13,99)
- 14,1 Zoll / 35,8 cm** (EUR 14,99)
- 15,6 Zoll / 39,6 cm** (EUR 19,49)
- 17,3 Zoll / 44 cm** (EUR 19,99)

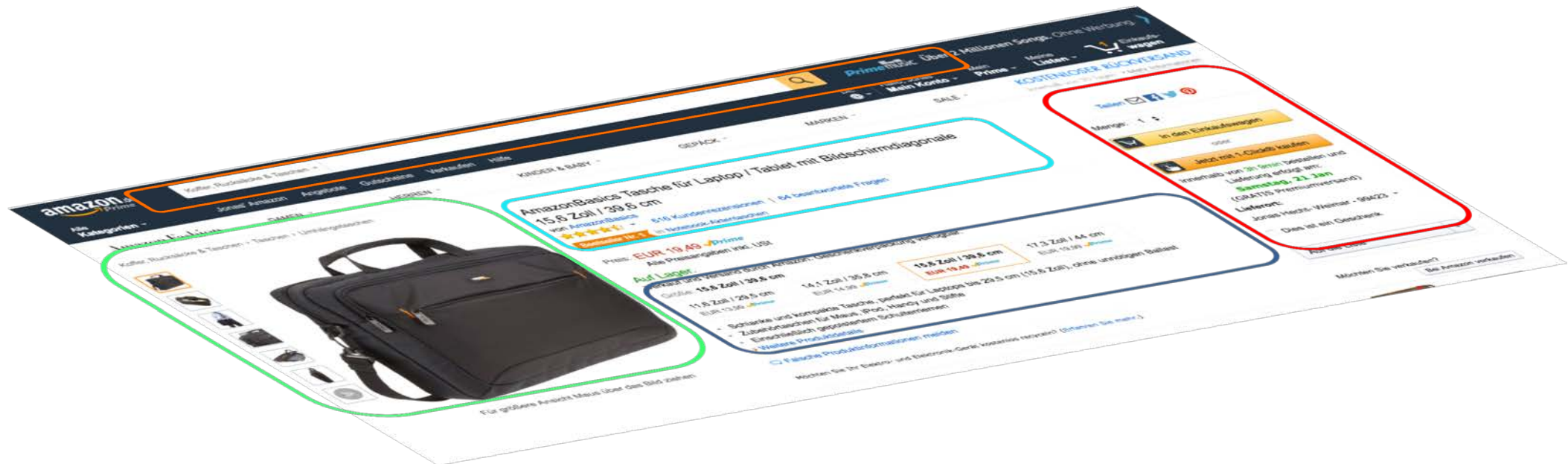
**Features:**

- Schlanke und kompakte Tasche, perfekt für Laptops bis 29,5 cm (15,6 Zoll), ohne unnötigen Ballast
- Zubehörtaschen für Maus, iPod, Handy und Stifte
- Einschließlich gepolstertem Schulterriemen

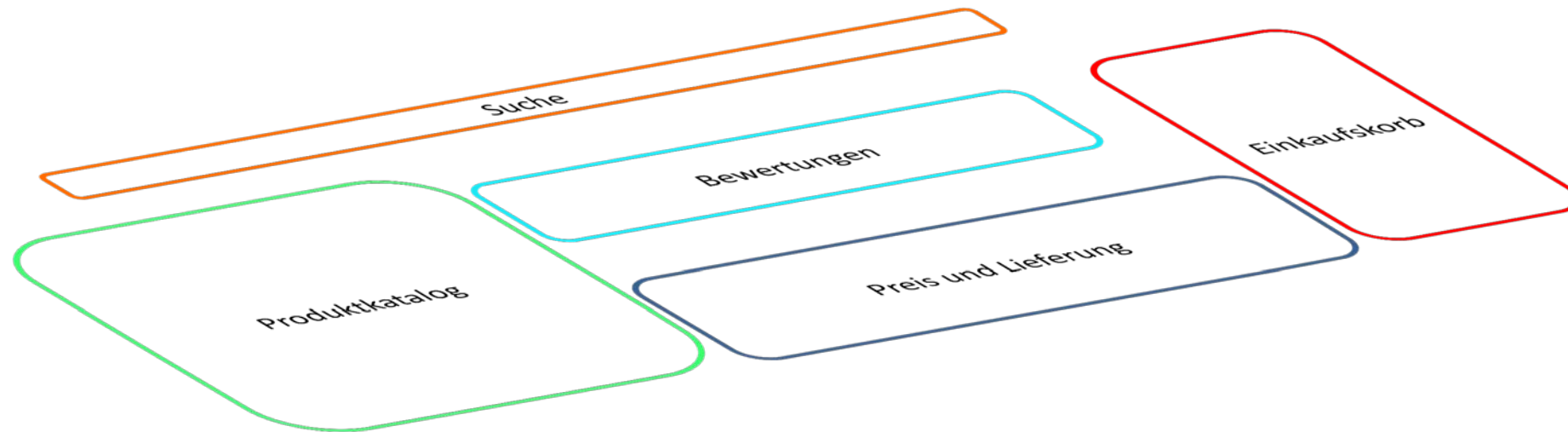
**Buttons:**

- Teilen (Email, Facebook, Twitter, Pinterest)
- Menge: 1
- In den Einkaufswagen
- oder
- Jetzt mit 1-Click® kaufen
- Innerhalb von 3h 9min bestellen und Lieferung erfolgt am: **Samstag, 21 Jan** (GRATIS Premiumversand)
- Lieferort: Jonas Hecht- Weimar - 99423
- ☐ Dies ist ein Geschenk
- Auf die Liste
- Möchten Sie verkaufen? Bei Amazon verkaufen

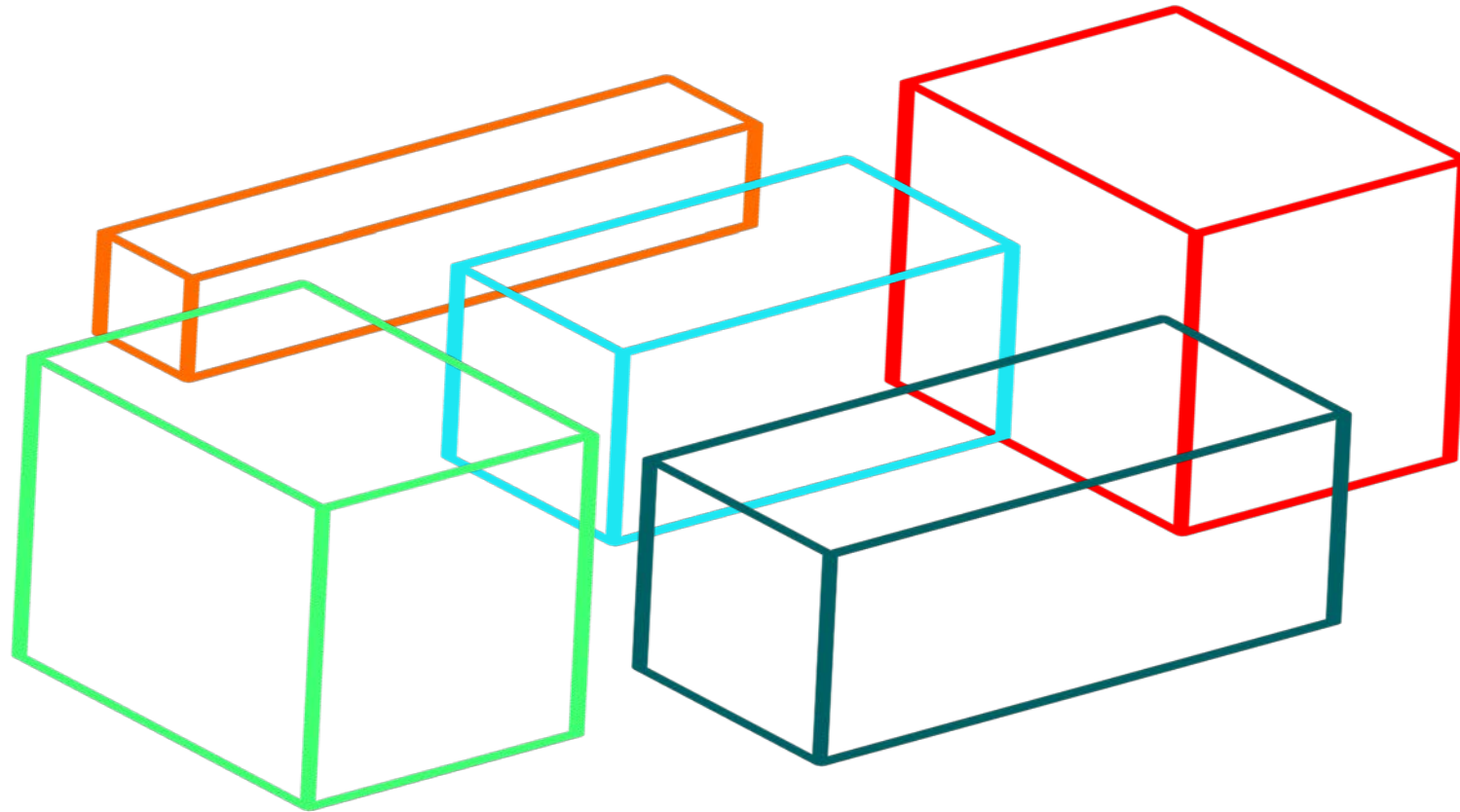
# Zerlegung des Systems...



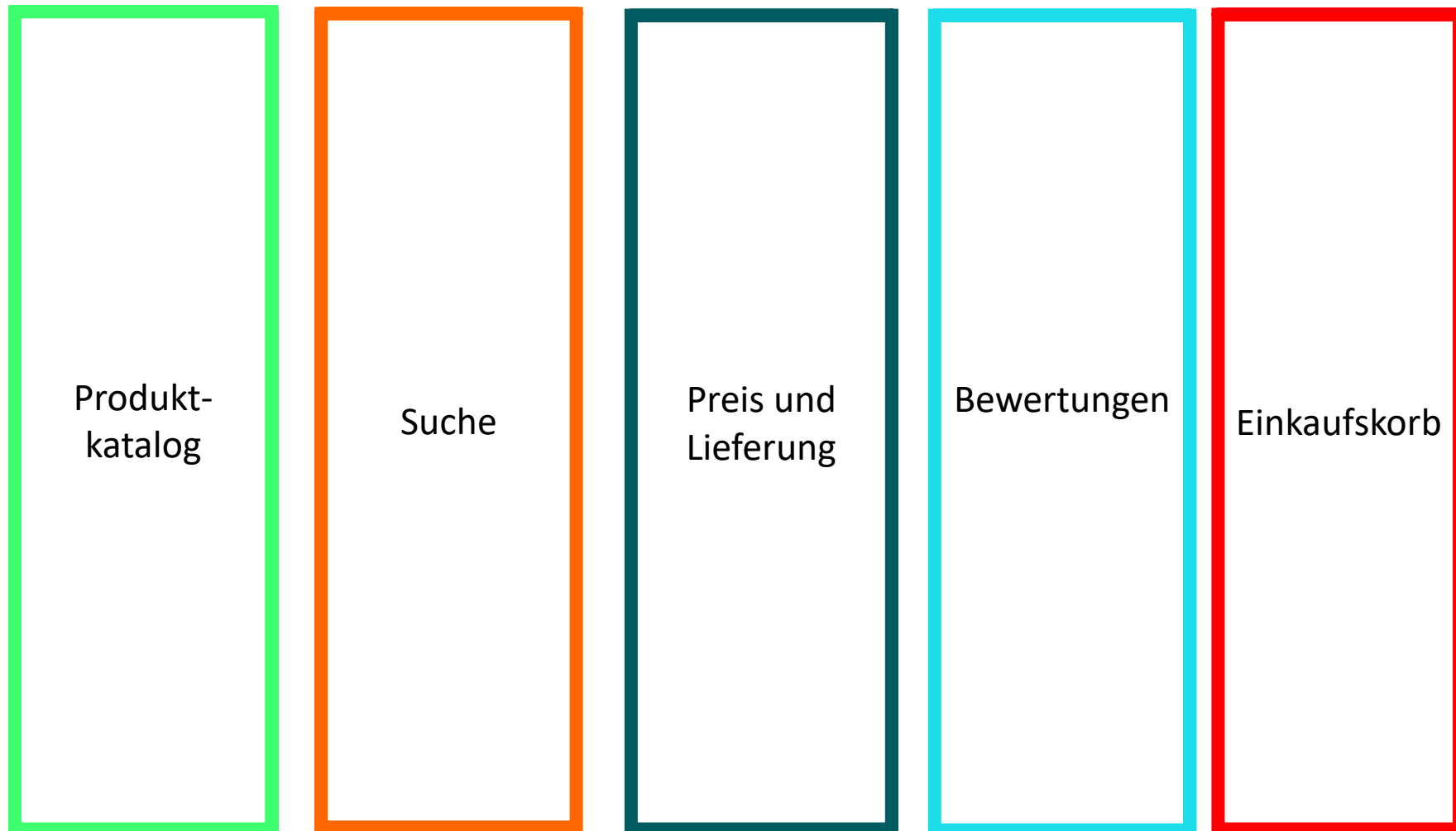
# Zerlegung des Systems...



# Zerlegung des Systems...



## ... nach fachlichen Funktionen



# nach fachlichen Funktionen



# MicroServices: Idee

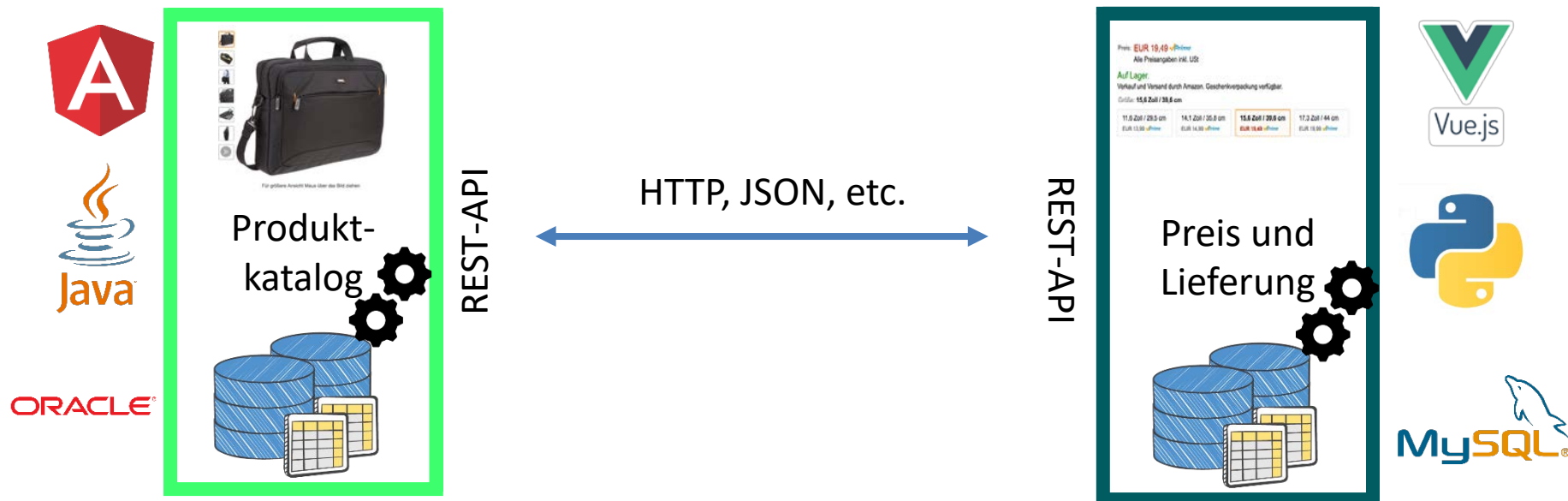
- Jede *fachliche* Funktion in einen *autonomen, unabhängigen Subsystem* modularisieren
- Jeder Service bietet die *vollständige* Funktionalität für die jeweilige fachliche Aufgabe an
  - Alle *Daten*, die hierfür notwendig sind
  - Gesamte *Business-Logik* für die Aufgabe
  - Alle *Sichten* und *Interaktionsmöglichkeiten*
- Teamzusammensetzung ideal für agile Entwicklung
  - Fachexperte, Entwickler, Tester, DevOps

*Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure. -- Melvyn Conway, 1967*

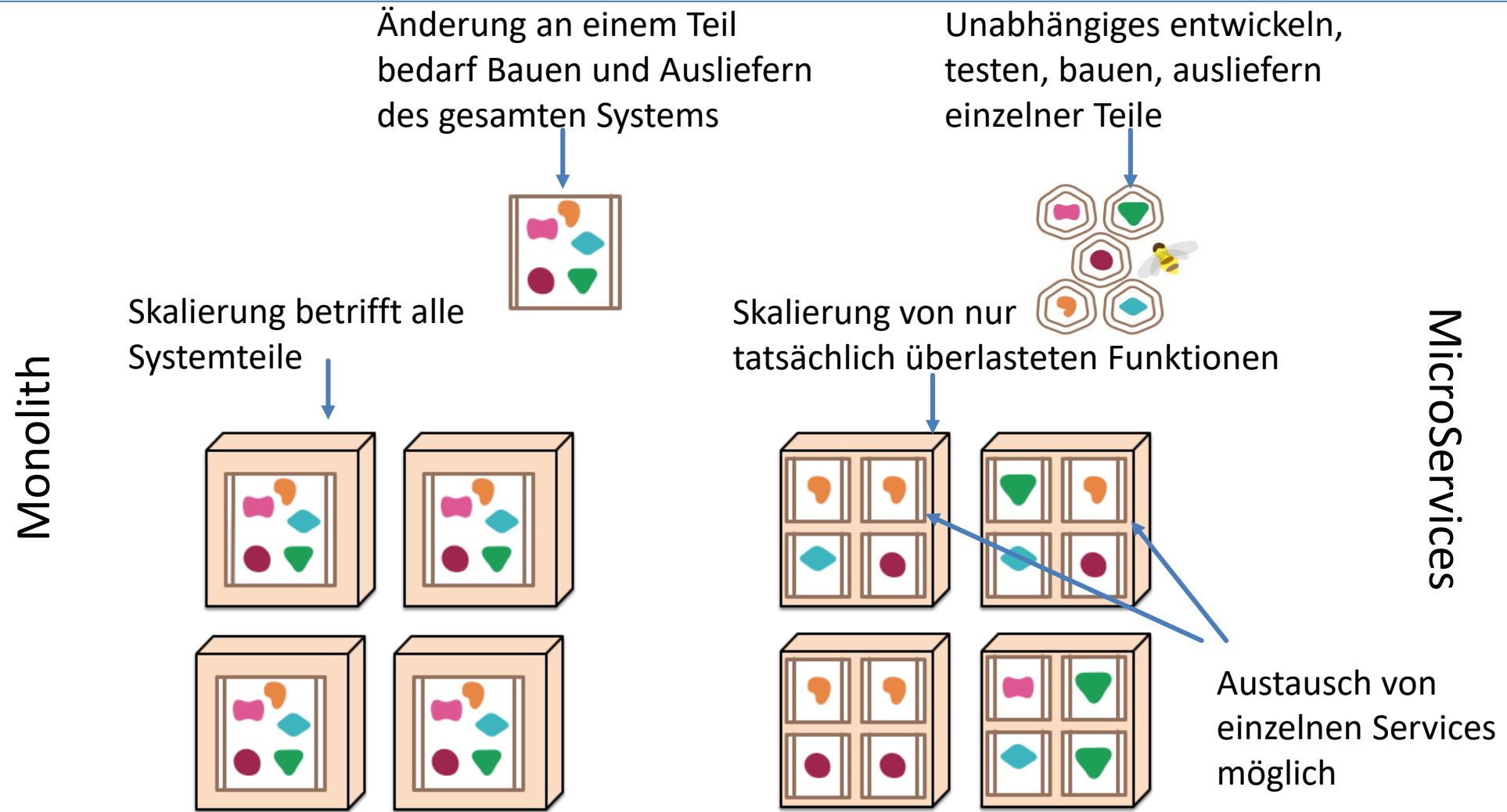


# Von MicroService zum Softwaresystem

- Anwendung besteht aus einer Menge an MicroServices
  - Jeder hat eigene Prozesse und Daten
  - Leichtgewichtige Kommunikation (meist REST-API)
  - Unabhängig voneinander auslieferbar, testbar, entwickelbar
- Maximale Unabhängigkeit der MicroServices

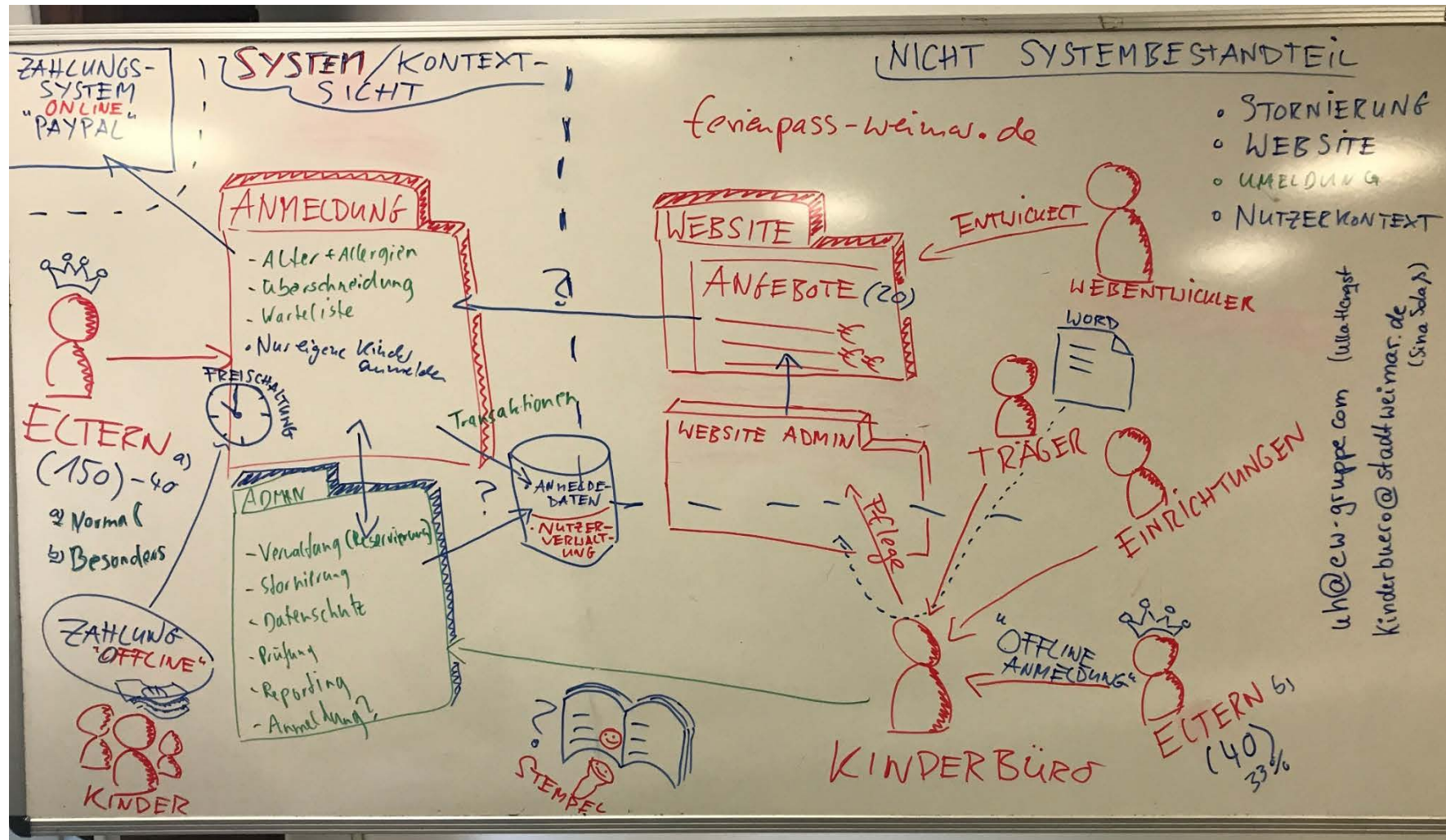


# Monolith vs. MicroServices



by James Lewis & Martin Fowler

# Erfahrungen Ihrer VorgängerInnen



# Und in der Praxis?

amazon NETFLIX  zalando

UBER

 TheGuardian

  
REWE digital

ebay

 28. SEPTEMBER 2018  
JUG SAXONYDAY

# Kritische Diskussion

---

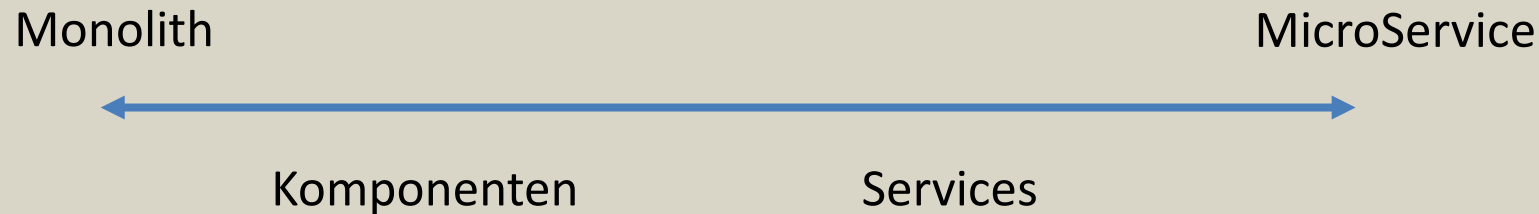
- Was sind mögliche Nachteile einer MicroService Architektur?
- Welche Besonderheiten und Voraussetzungen sollten beim Einsatz dieser Architektur betrachtet werden?
  - MicroServices für Word oder Virens Scanner?

# Nachteile von MicroServices

- Benötigt richtigen „Mindset“ der Entwickler und klare Definition von einem MicroServices
- Immer noch hoher Kommunikationsaufwand zwischen Teams
- Moderne DevOps Pipeline erforderlich, Stichwort: Continuous Integration and Delivery
- Technologien entwickeln sich schnell weiter
- Architektur resultiert in einem verteilten System mit all seinen Vor- und Nachteilen

# Services vs. MicroServices

- Bisher:



- Die extreme bzgl. Kopplung von Subsystemen sind nun bekannt. Jetzt:
  - Welche Abstufungen sind dazwischen möglich?
  - Welche Kommunikationsformen und Granularitäten?
  - Welche Einsatzszenarien für welche Architektur?

# Kriterien zur Auswahl von Architekturmustern

---

- Welche Struktur des Softwaresystems?
  - Komponenten-orientiert, monolithisch, Schichten, Pipes and Filters
- Wie kommunizieren die Sub-Systeme?
  - Ereignis-basiert, Publish-Subscribe , asynchrone Nachrichten
- Wie sind die Sub-Systeme verteilt?
  - Client-Server, shared nothing, Peer2Peer, service-orientiert, Cloud

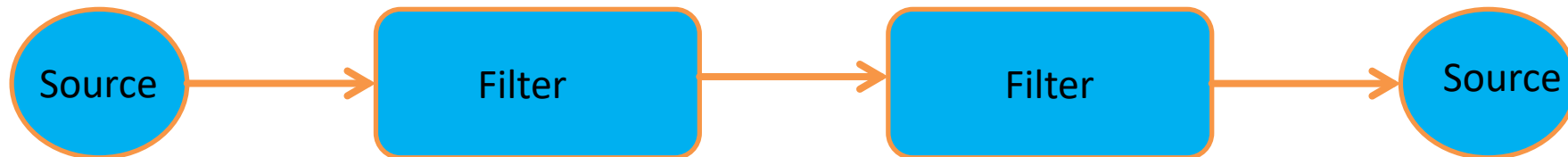


# Pipes and Filters



# Aufbau

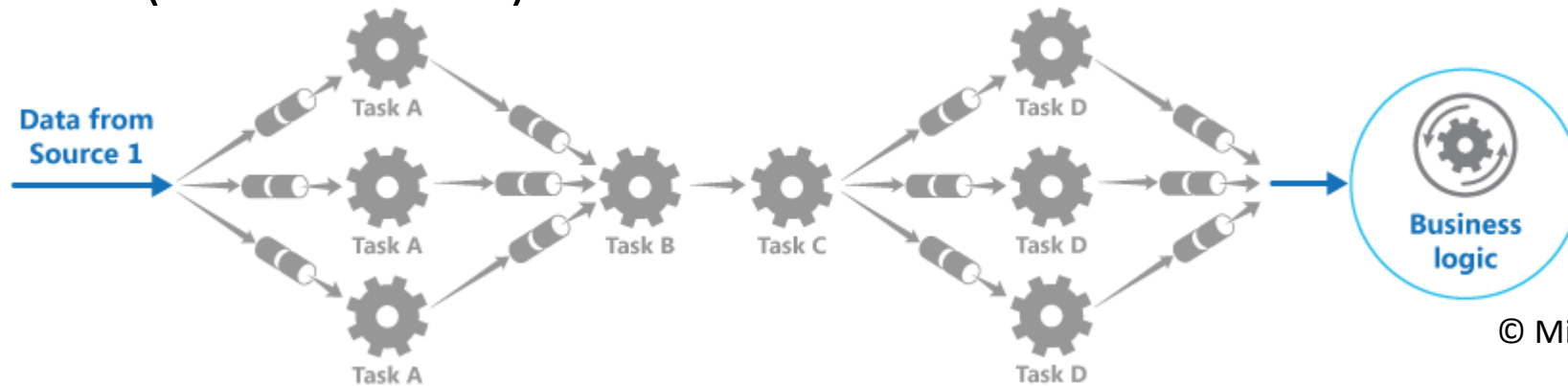
- Pipe: Verbindungsglied, welches Daten von einem Filter zu einem anderen weiterleitet
- Filter: Transformiert Daten, die es durch eine Pipe bekommen hat



- Vorteile:
  - Filter können einfach hinzugefügt und herausgenommen werden
  - Robust, performant, skalierbar, gute Wartbarkeit!

# Vor- und Nachteile

- Parallelisierbar (Lastbalanciert)

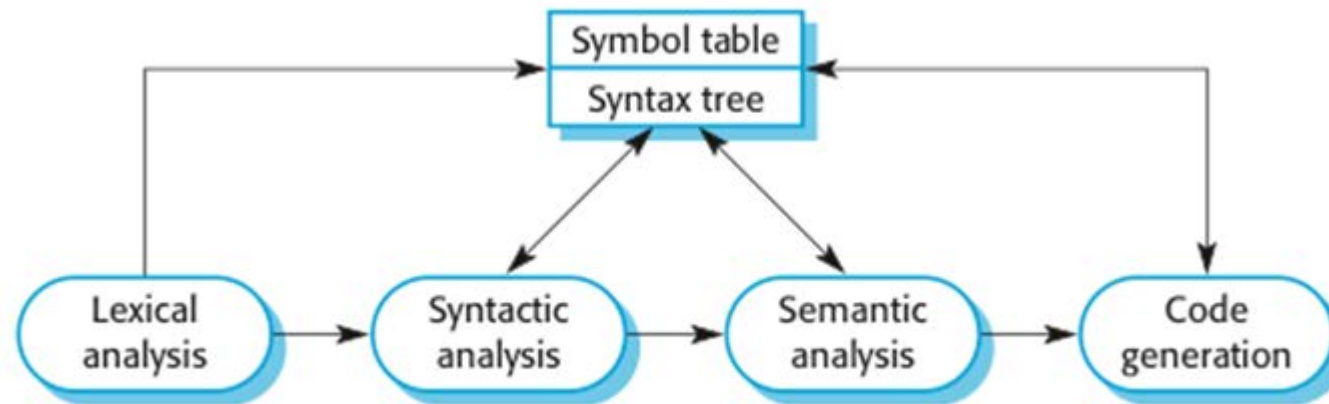


© Microsoft

- Probleme:
  - Komplexität steigt
  - Bufferoverflow möglich
- Wann anwendbar?
  - Anwendung kann in mehrere unabhängige Teile zerlegt werden
  - Wenn viele Transformationen auf Daten nötig sind
  - Wenn Flexibilität notwendig ist (z.B. in der Cloud)

# Beispiele

<i>Domain</i>	<i>Data source</i>	<i>Filter</i>	<i>Data sink</i>
<i>Unix</i>	<code>tar cf - .</code>	<code>gzip -9</code>	<code>rsh picasso dd</code>
<i>CGI</i>	HTML Form	CGI Script	generated HTML page



# Was Sie mitgenommen haben sollten:

---

- Anwendung:
  - [Modell/Quelltext]
  - Bewerten Sie anhand der 5 Kriterien
  - Bewerten Sie anhand der 5 Regeln
  - Bewerten Sie anhand der vorgestellten Elemente des GRASP-Patterns
- Wissen:
  - Was ist modular decomposability? Erklären Sie an einem Beispiel
  - [analog die restlichen Kriterien, Regeln, und Prinzipien]

# Was Sie mitgenommen haben sollten:

- Inwiefern wirkt sich die Architektur auf das Softwaresystem aus?
- Wie kann die Auswahl einer geeigneten Architektur den Entwurf / Implementierung vereinfachen?
- Was bedeutet Kopplung und Kohäsion auf Architekturebene?
- Was ist ein Architektur-Stil?
- Für welche Szenarien sind MVC oder Pipes and Filters sinnvoll?
- Was sind Limitierungen von monolithischen Systemen?
- Was sollten Schichten nicht auf Elemente in Schichten darüber Zugriff haben?
- Wann macht der Einsatz von MicroServices Sinn und wann nicht?
- Welche Kriterien für den Einsatz bestimmter Architekturmuster gilt es zu beachten?

# Was Sie mitgenommen haben sollten:

---

- How does software architecture constrain a system?
- How does choosing an architecture simplify design?
- What are coupling and cohesion?
- What is an architectural style?
- For which application scenarios is MVC beneficial? For which is pipes and filters?
- Why shouldn't elements in a software layer "see" the layer above?
- What is the role of programming in model-driven architecture?

# Literatur

---

- Bertrand Meyer, Object-Oriented Software Construction, Prentice Hall, 1997 [Chapter 3, 4]
- *Software Engineering*, I. Sommerville, 7th Edn., 2004.
- *Objects, Components and Frameworks with UML*, D. D'Souza, A. Wills, Addison-Wesley, 1999
- *Pattern-Oriented Software Architecture — A System of Patterns*, F. Buschmann, et al., John Wiley, 1996
- *Software Architecture: Perspectives on an Emerging Discipline*, M. Shaw, D. Garlan, Prentice-Hall, 1996



# Literatur

- Service-Oriented Architecture (SOA) vs. Component Based Architecture. Helmut Petritsch ([http://petritsch.co.at/download/SOA\\_vs\\_component\\_based.pdf](http://petritsch.co.at/download/SOA_vs_component_based.pdf))
- Microservices. James Lewis und Martin Fowler.  
<https://martinfowler.com/articles/microservices.html>

