Software Engineering

Software Architecture

Software & Systems Engineering | Prof. Dr. Andreas Vogelsang | 25.10.2023

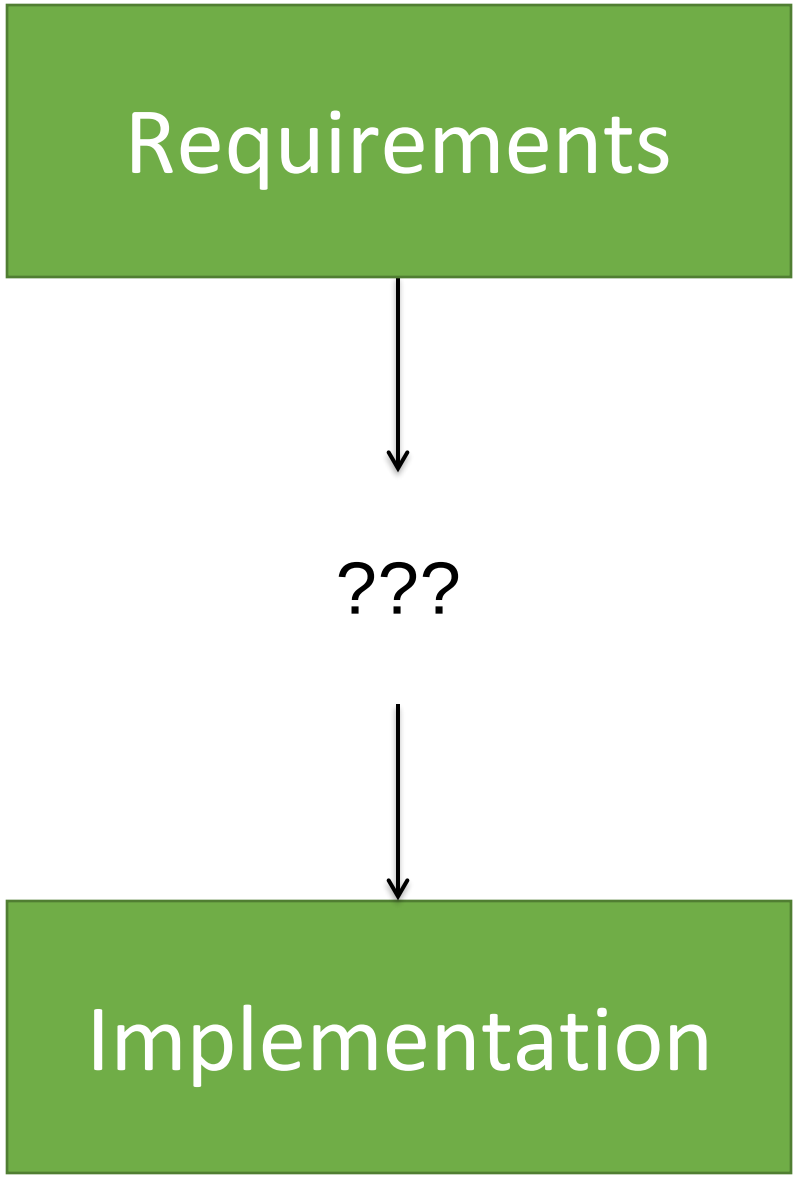
@andivogelsang
vogelsang@cs.uni-koeln.de

Learning Goals for Today

- Understand what SW architecture is and why it is important
- Understand what component diagrams are and how they can be used to describe SW architectures
- Know and understand common architectural patterns

Software Architecture

How do I get from requirements to code?



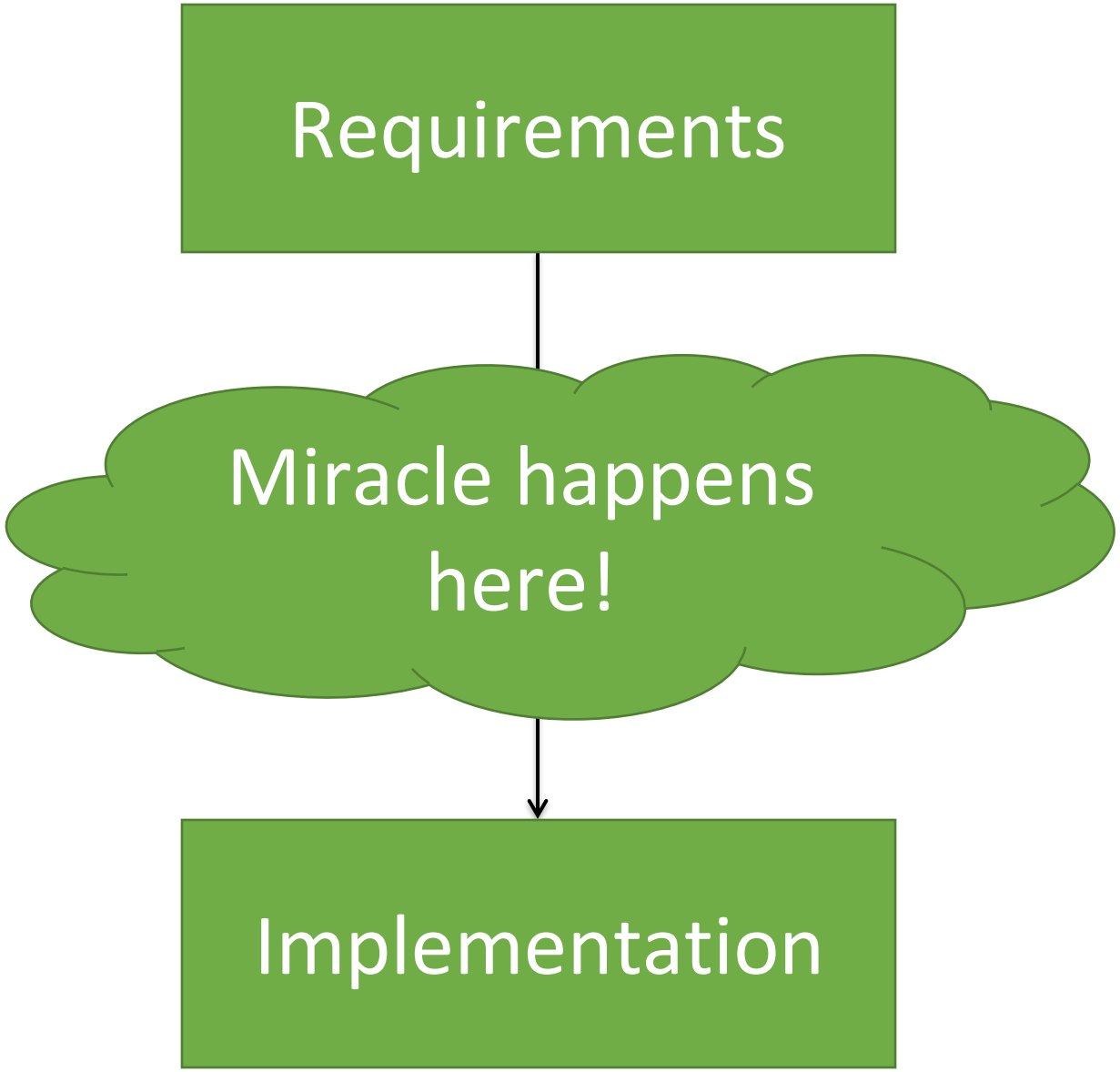
Large SW systems…

- have numerous requirements
- require many developers
- need separation of concerns (Trennung von Belangen)

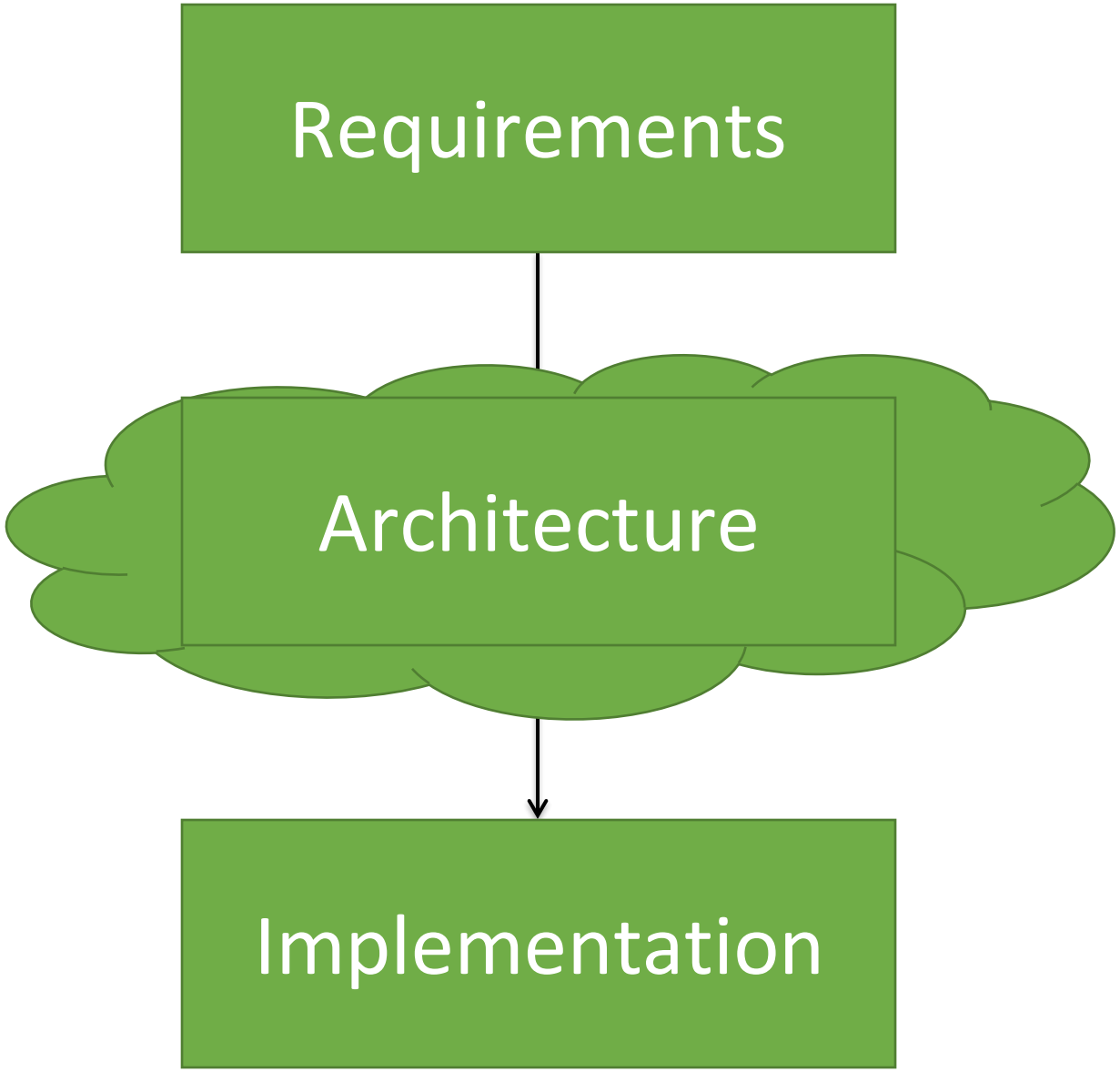
"Weeks of coding can save you hours of planning"

[anon.]

- Ad hoc
- Experience
- Unpredictable
- Costly…



- Composition of components
- Abstraction
- Reuse



Software Architecture

Software Architecture

The **software architecture** of a program or computing system is the **structure or structures** of the system, which comprise **software elements**, the **externally visible properties** of those elements, and the **relationships** among them.

[Bass et al. 2003]

Software Architecture

A **software architecture** is a description of how a software system is organized. Properties of a system such as performance, security, and availability are influenced by the architecture used.

[Sommerville]

Note

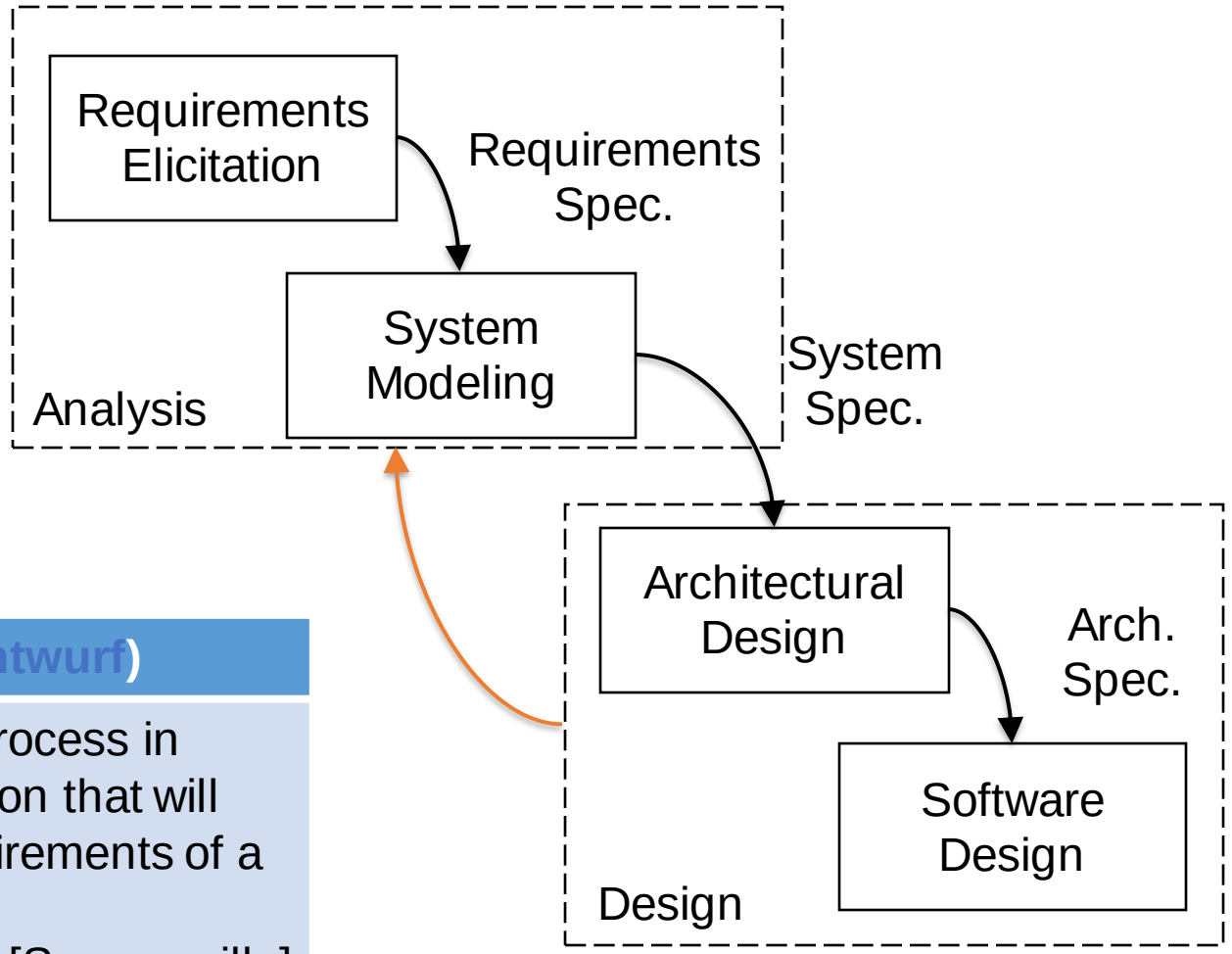
This definition is ambivalent to whether the architecture is known, or whether it's any good!

In practice

You might propose an abstract system architecture where you associate groups of system functions or features with large-scale components or subsystems. You then use this decomposition to discuss the requirements and more detailed features of the system with stakeholders.

[Sommerville]

Architectural Design

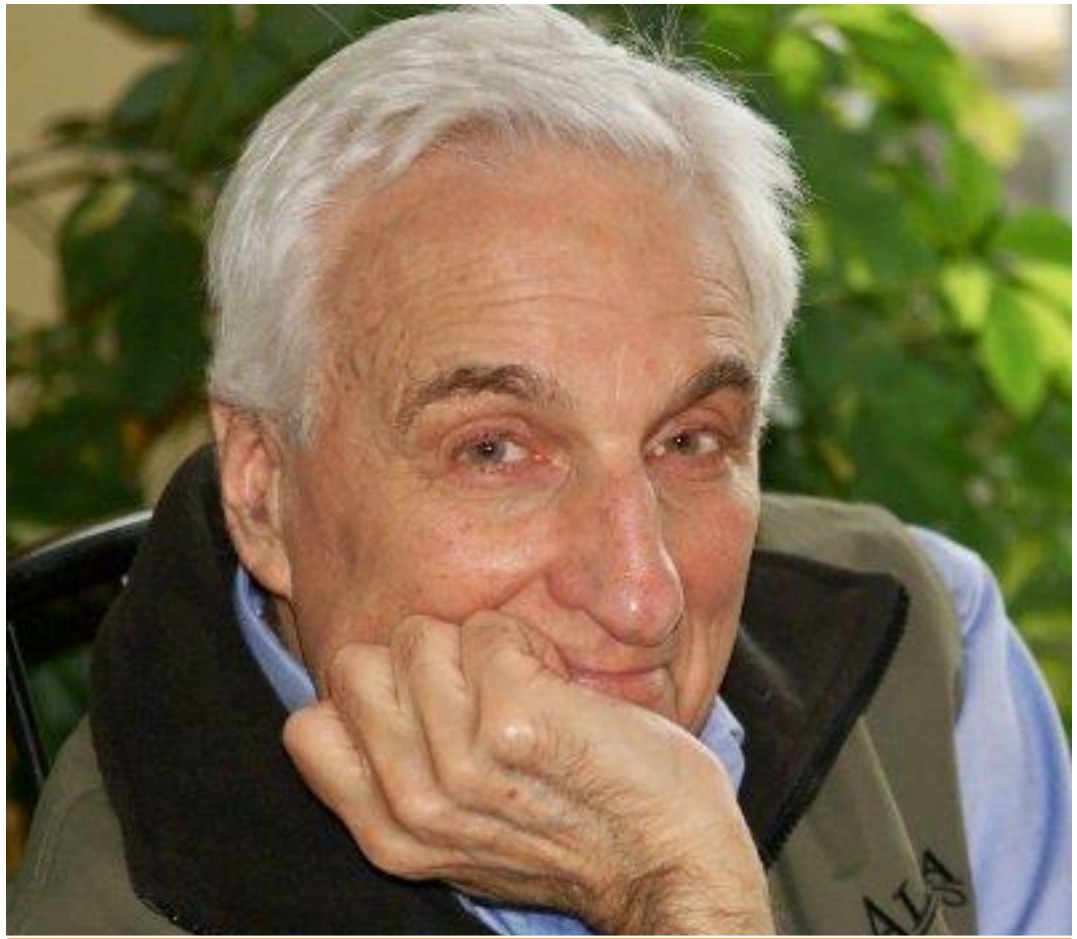


Architectural Design (Architekturentwurf)

"**Architectural design** is a creative process in which you design a system organization that will satisfy the functional and quality requirements of a system."

[Sommerville]

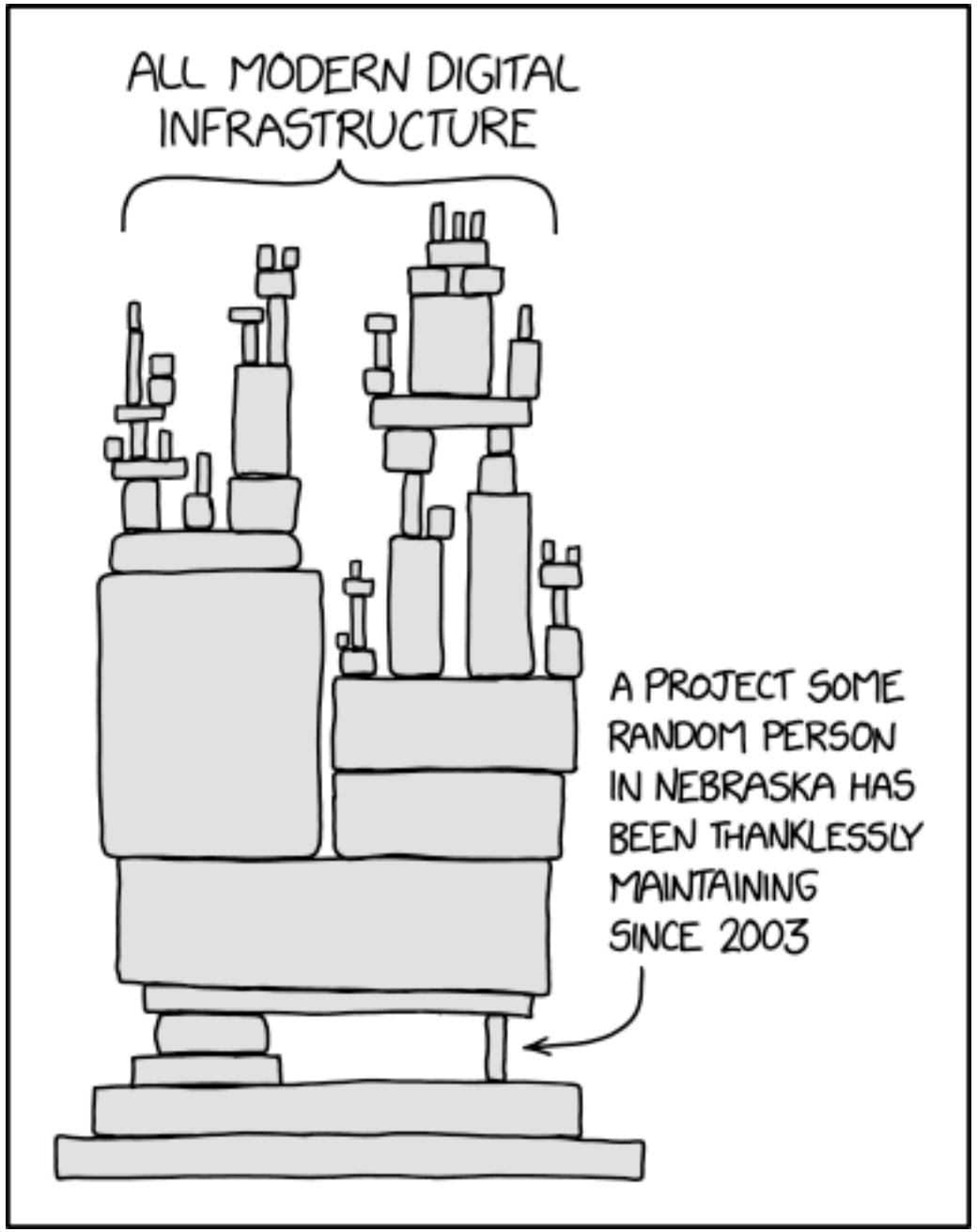
Conway‘s Law



Conway’s Law

“Any organization that designs a system [...] will produce a design whose structure is a copy of the organization’s communication structure.”

- What does that mean?
- Is that good or bad?
- Why is this law important?


ALL MODERN DIGITAL INFRASTRUCTURE
A PROJECT SOME RANDOM PERSON IN NEBRASKA HAS BEEN THANKLESSLY MAINTAINING SINCE 2003

Goals of SW Architecture

- Communication with stakeholders
  - Shows solution on a level they can understand
- Meet critical requirements
  - Especially quality requirements
- Support reuse
- Support project organizations
  - Predicting cost, quality, and schedule
  - Teams work on different components

Component Diagrams

Component Diagrams

Component Diagram (Komponentendiagramm)

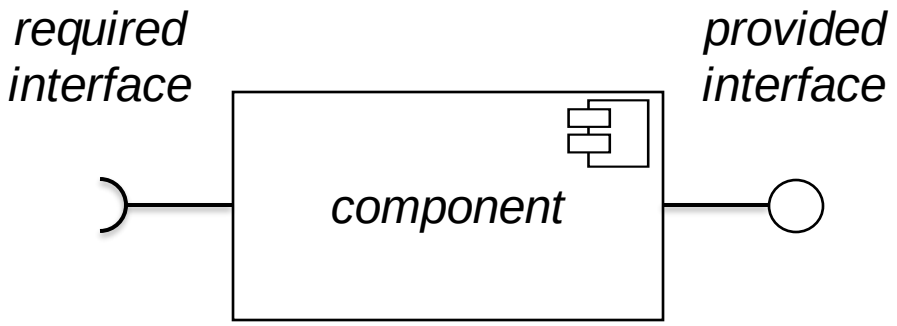
A **component** is a replaceable part of a system that conforms to and provides the realization of a set of interfaces.

An **interface** is a collection of operations that specify a service that is provided by or requested from a class or component.

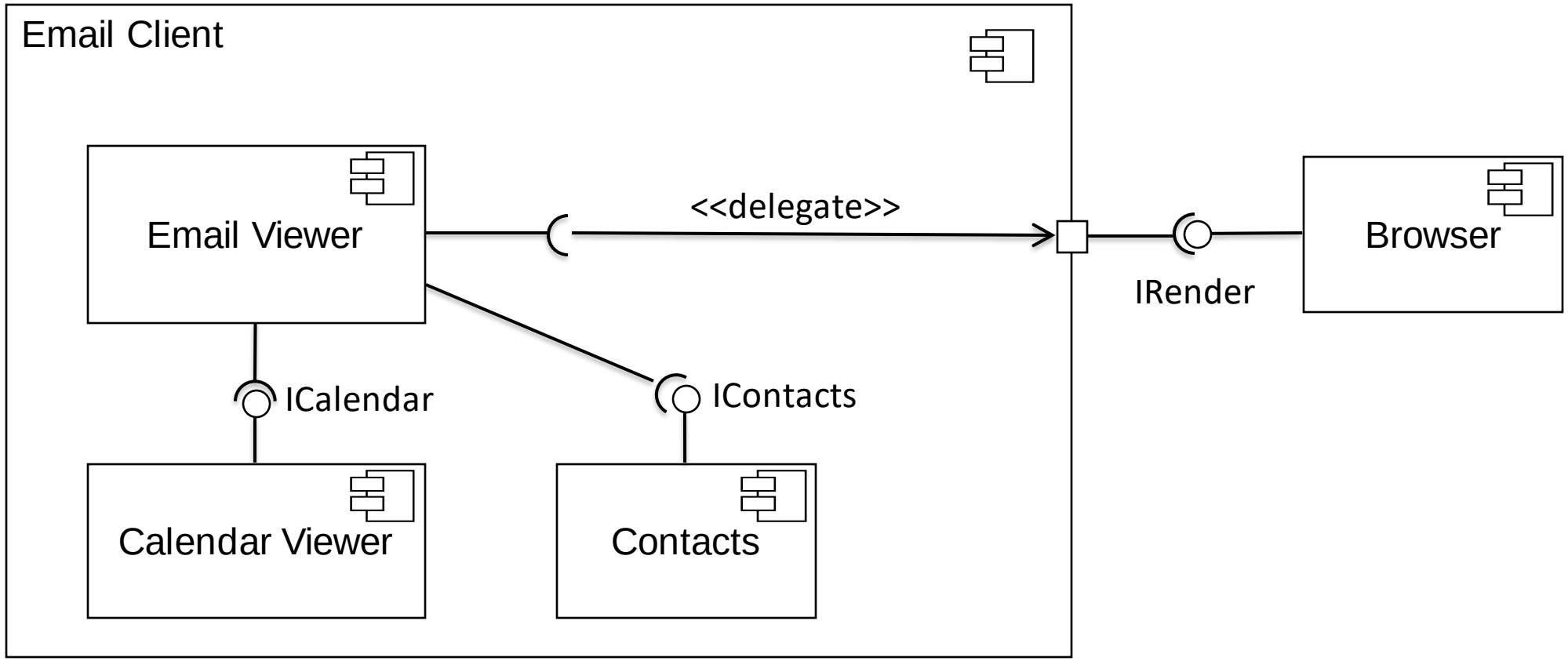
An interface that a component realizes is called a **provided interface**, meaning an interface that the component provides as a service to other components.

The interface that a component uses is called a **required interface**, meaning an interface that the component conforms to when requesting services from other components.
(Komponente, angebotene/benötigte Schnittstelle)

[adapted from UML User Guide]



Example of a Component Diagram



Hierarchical Component Diagrams

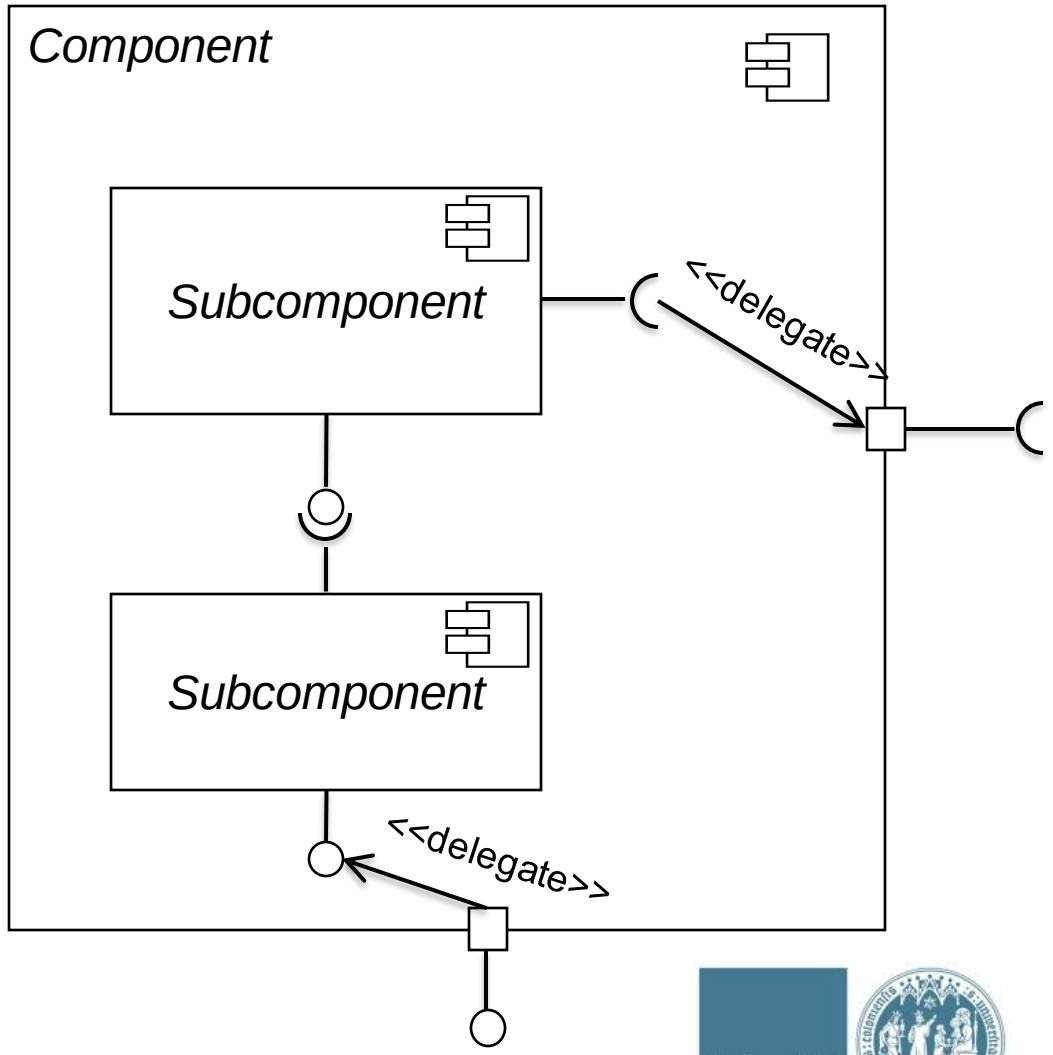
Nesting of Components (Verschachtelung)

Motivation: decompose/structure large systems

Nesting: A component may contain any number of **subcomponents**. (Teilkomponenten)

Ports and Delegates: A **port** is an explicit window into an encapsulated component. A **delegate** connects provided or required interfaces with ports.

[adapted from UML User Guide]



Rules for Component Diagrams

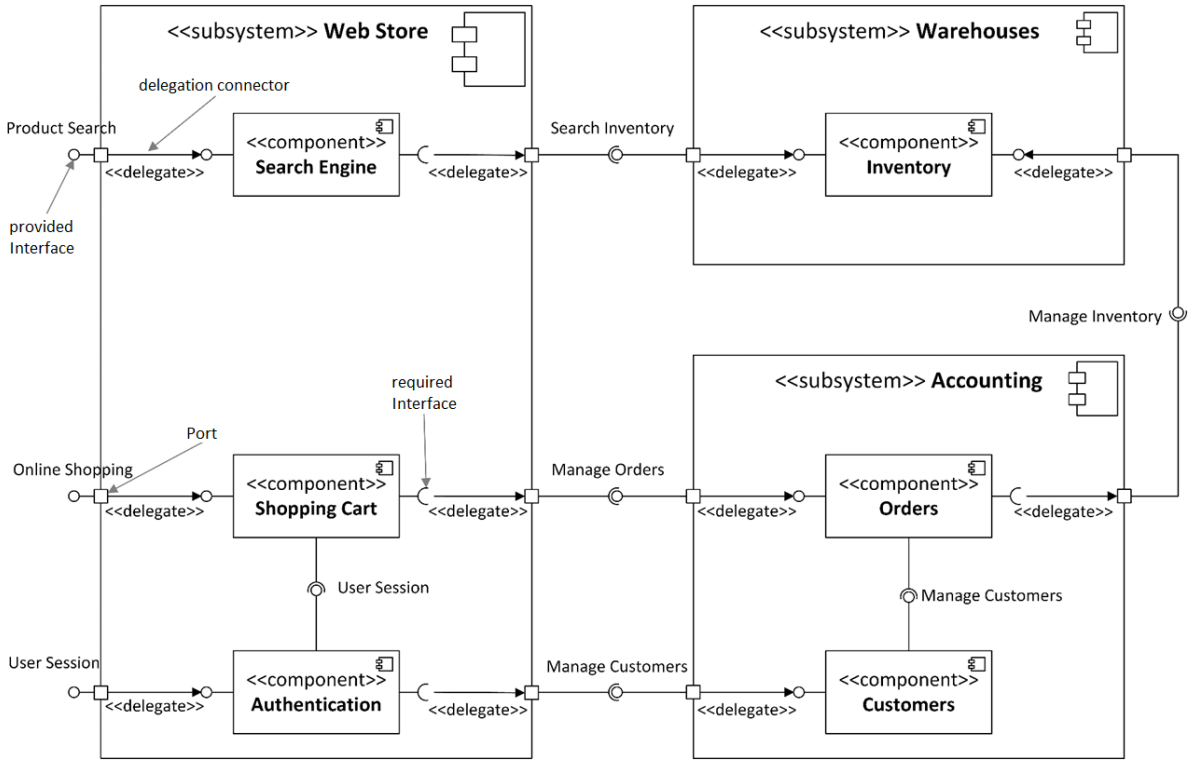
Rules for Component Diagrams

- component names are unique
- a component may have any number of required or provided interfaces
- every required interface is connected to a provided interface
- every component is directly or indirectly connected to every other component
- subcomponents may be nested to any level
- when subcomponents communicate to a higher-level component, they need to communicate via ports

Use of Component Diagrams

Use of Component Diagrams

Component Diagrams are mainly used to describe the *development view* but may also be used to describe the *physical view*



On Interfaces in Java

- Interfaces in Java define a set of methods
- A class that *implements* an interface must provide code for the methods defined in the interface.
- How does that relate to *required* and *provided interfaces* in UML component diagrams?

```java
// Interface
interface Animal {
  public void animalSound(); // interface method (does not have a body)
  public void sleep(); // interface method (does not have a body)
}

// Pig "implements" the Animal interface
class Pig implements Animal {
  public void animalSound() {
    // The body of animalSound() is provided here
    System.out.println("The pig says: wee wee");
  }
  public void sleep() {
    // The body of sleep() is provided here
    System.out.println("Zzz");
  }
}

class Main {
  public static void main(String[] args) {
    Pig myPig = new Pig();  // Create a Pig object
    myPig.animalSound();
    myPig.sleep();
  }
}
```

Common Architectural Patterns

Architectural Patterns

Architectural Pattern (Architekturmuster)

"Architectural patterns capture the essence of an architecture that has been used in different software systems. [...] Architectural patterns are a means of reusing knowledge about generic system architectures."

[Sommerville]

Goals

- Preserve knowledge of SW architects
- Reuse of established architectures
- Enable efficient communication

Layered Architecture

Layered Architecture (Schichtenarchitektur)

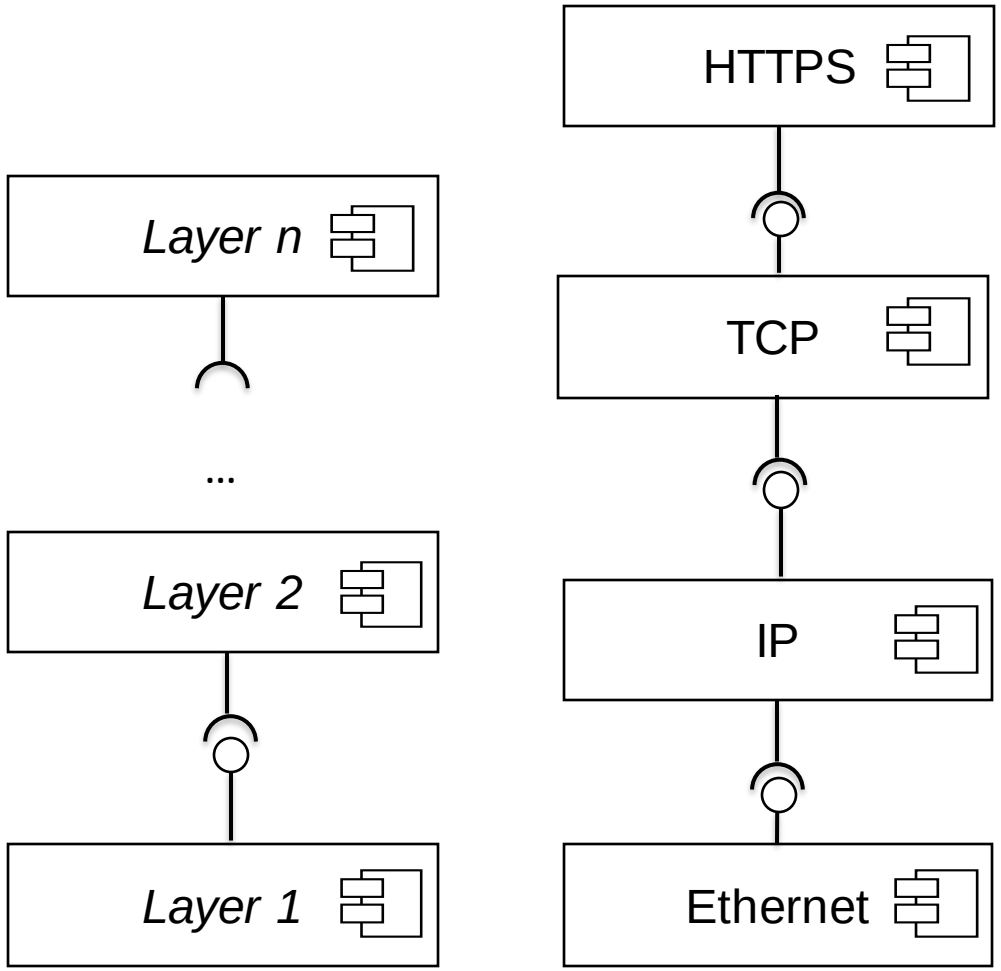
Problem: subsystems are hard to adapt and replace

Idea: decomposition into layers (Schichten)

- A layer provides services to layers above
- A layer delegates subtasks to layers below
- strict layers: every layer can only access the next layer
- relaxed layers: every layer can access all layers below

information hiding: layers hide implementation details behind interface

[Sommerville]



Client-Server Architecture (2-tier architecture)

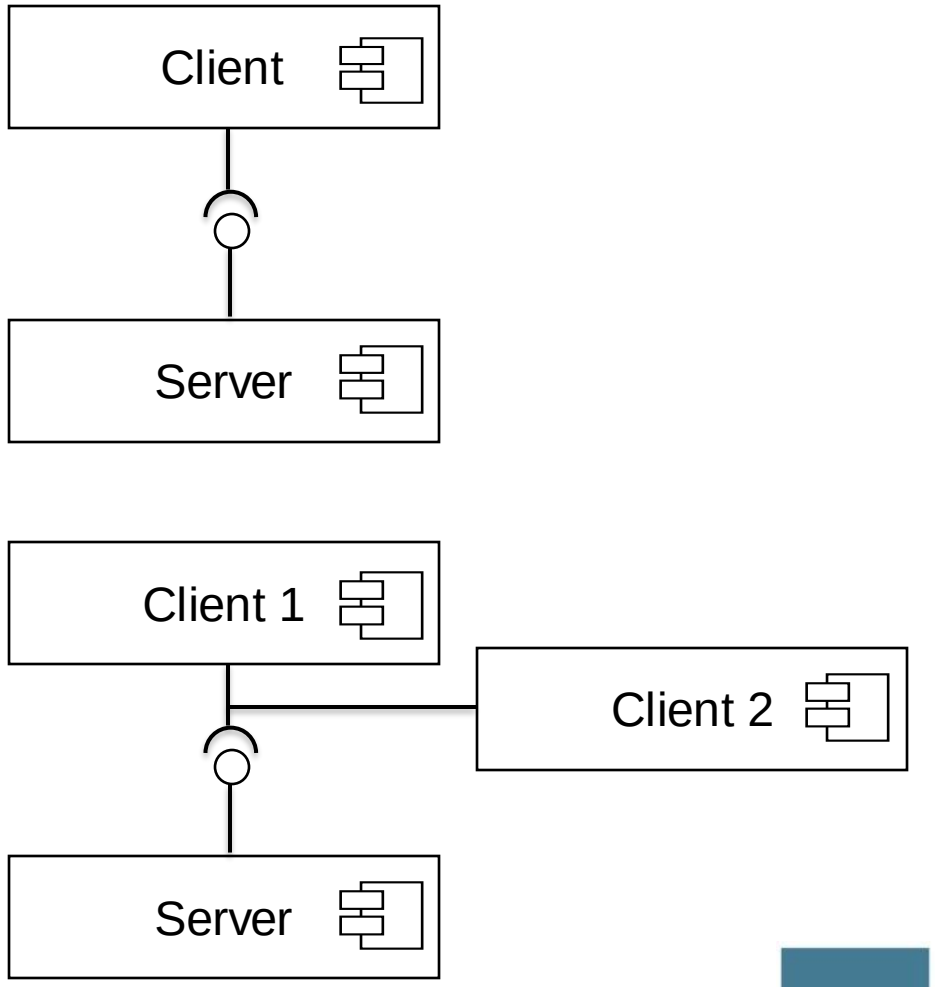
Client-Server Architecture (aka. 2-Tier)

Problem: several clients need to access the same data

Idea: separation of application (client) and data management (server)

- clients initiate the communication with a server
- typical: multiple clients of the same kind
- optional: multiple clients of different kinds

[Sommerville]



Example

a browser uses a URL to connect to a server in the world wide web and receives an HTML page

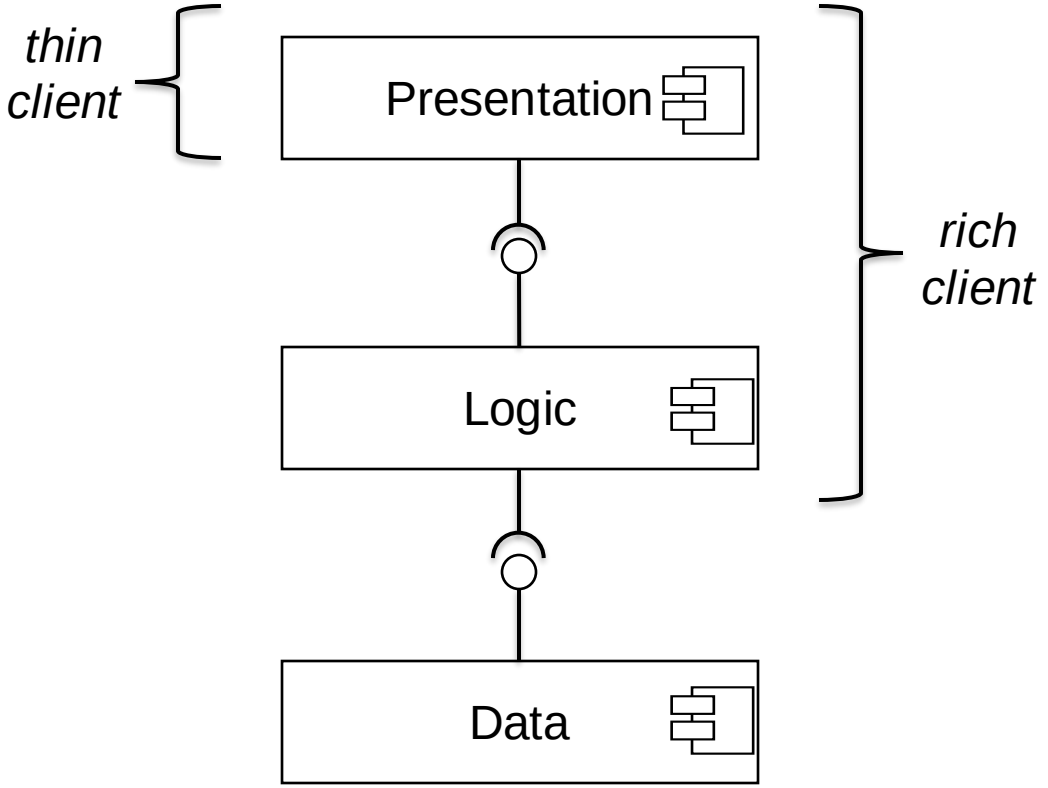
3-Tier Architecture

3-Tier Architecture (3-Schichten Architektur)

Problem: clients with same functionality but different presentation needed

Idea: separation of data presentation, application logic, and data management

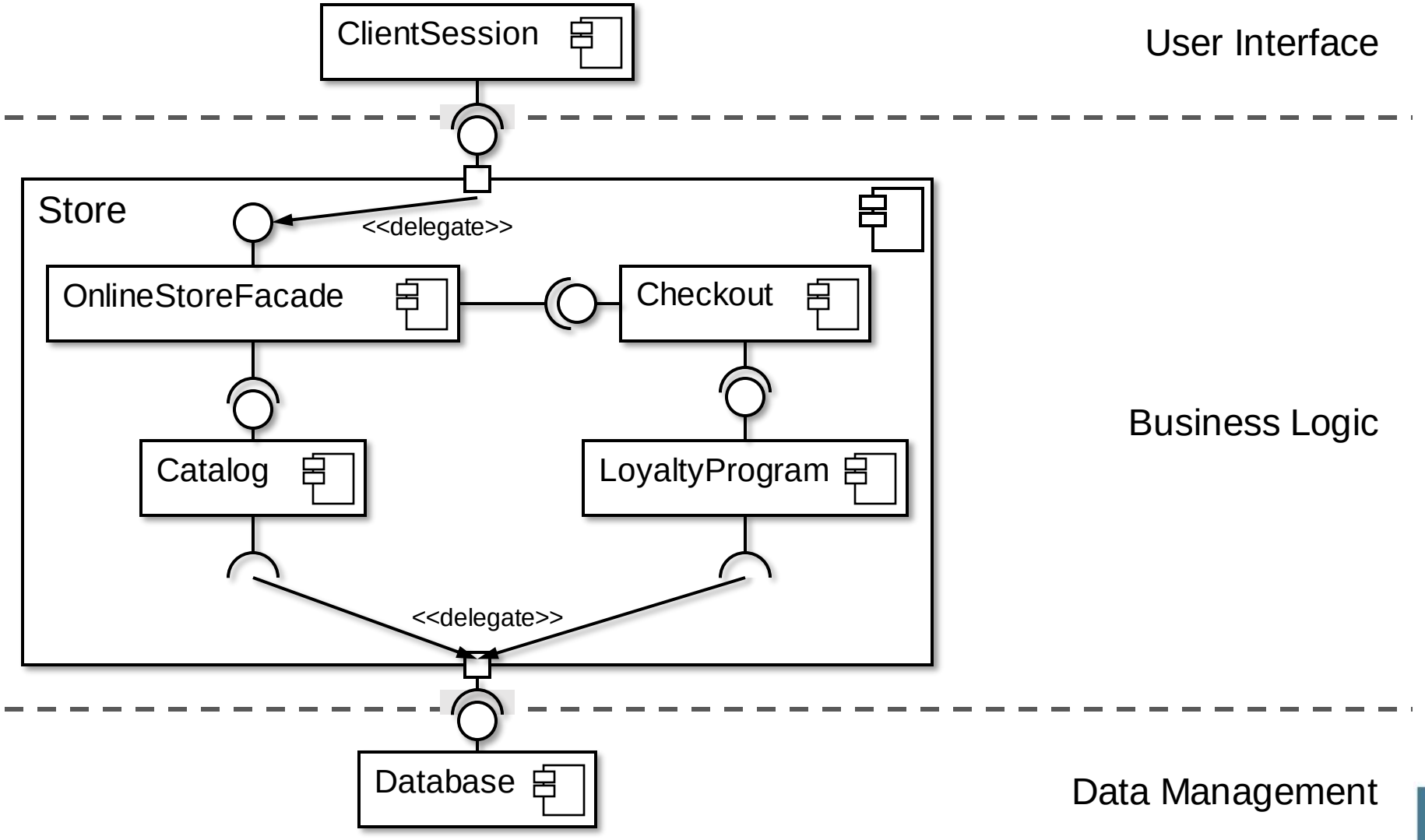
- thin-client application: application logic on the server
- rich-client application: application logic in the client



Rule of Thumb

If you can use the application offline, then it is most likely a rich-client application.

3-Tier Architecture – Example



Model-View-Controller Architecture

Model-View-Controller (MVC) Architecture

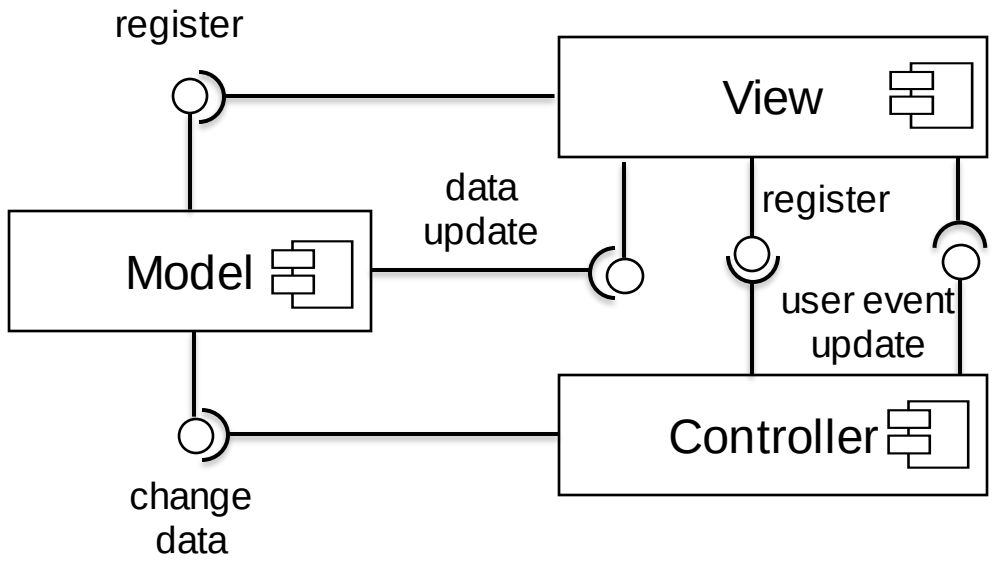
Context: data is presented and manipulated over several views

Problem: data inconsistent and new views hard to add

Idea: separation into three components

- *Model*: stores the relevant data independent of their presentation
- *View*: shows (a part of) the data independent of manipulations
- *Controller*: user interface for the manipulation of data

Independence is realized by the *observer pattern*



Example

In a spreadsheet, data is presented in tables and diagrams. Changing values in a table leads to an update of affected diagrams and tables.

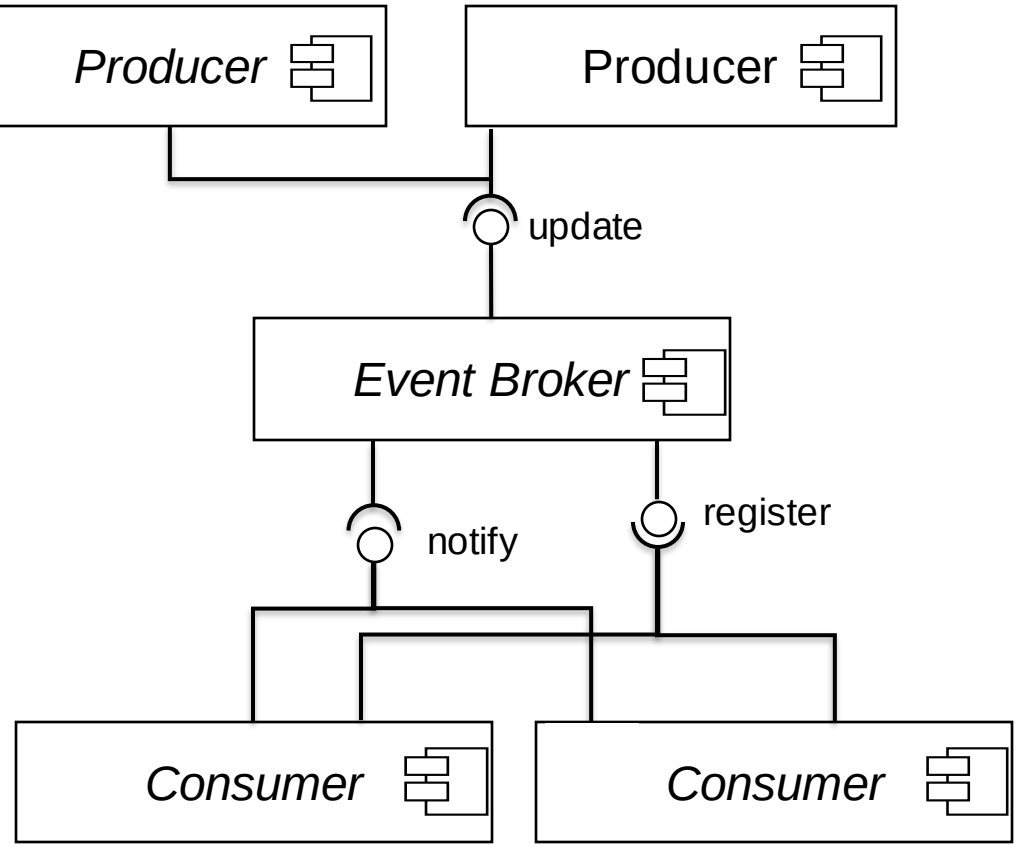
Publish-Subscribe Architecture

Publish-Subscribe Architecture

Problem: Message passing in distributed and loosely coupled systems

Idea: Introduce a broker that collects data or messages from producers and passes them to consumers

- Producers and consumers are independent
- Consumers need to register at the broker



Example

A (changing) set of sensors is producing measurement values that may be analyzed by a (changing) set of services.

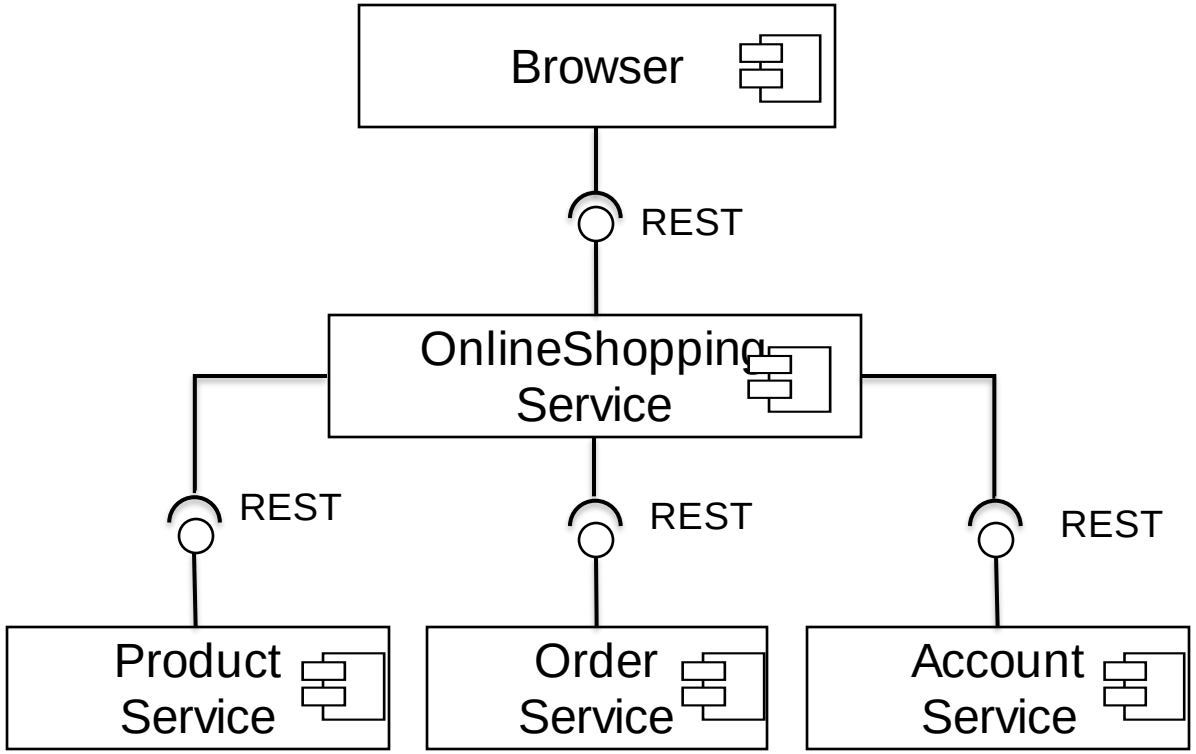
Microservice Architecture

Microservice Architecture

Problem: in large applications, components tend to become big (hard to maintain)

Idea: break down the application into smaller services that can be developed independently

- Each service is self-contained (contains data, logic, presentation)
- Services communicate via standardized interfaces (e.g., REST)

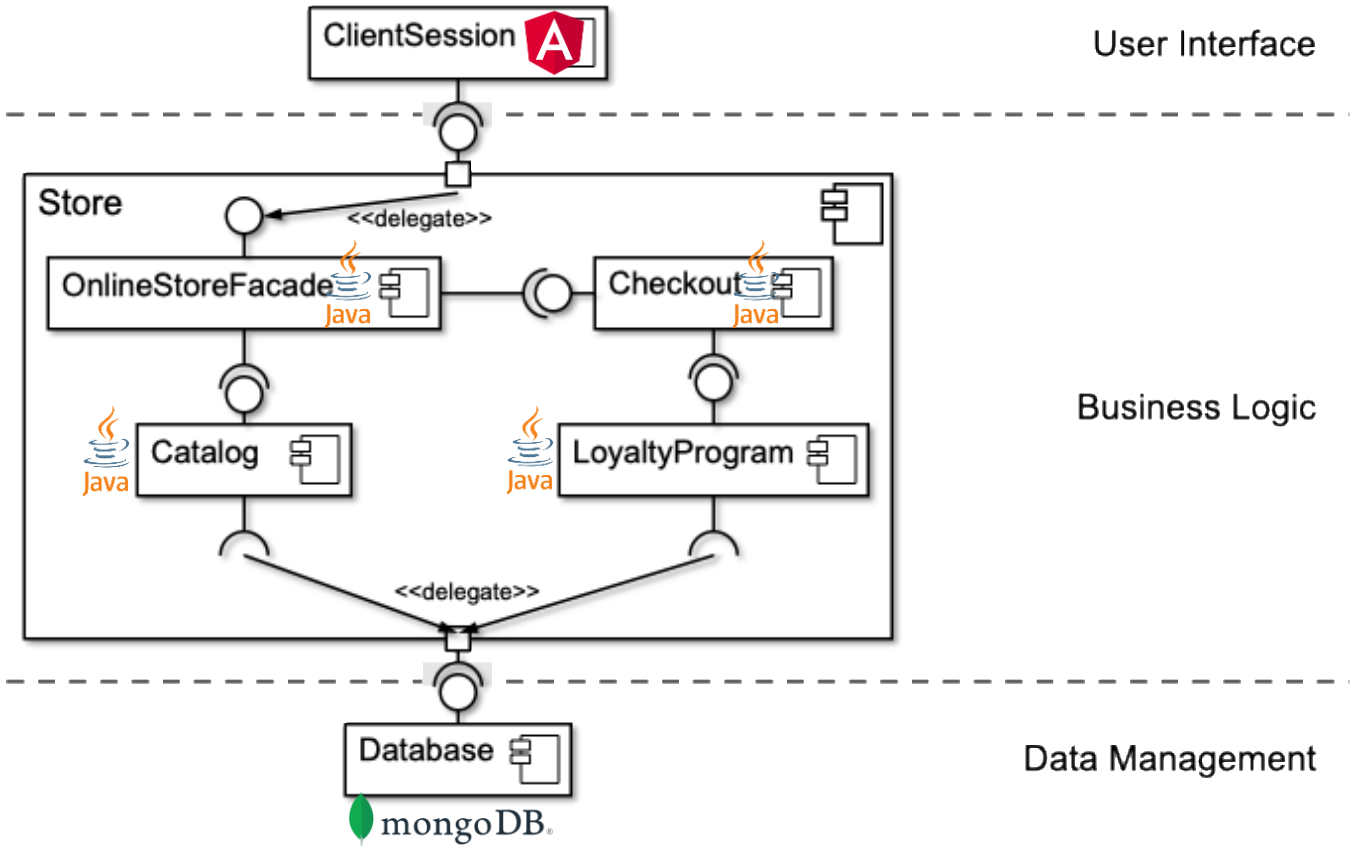


Microservice Hype

A lot of the MS hype is also related to the way how the architecture influences the organization (agile and independent teams) and how it can be scaled (*DevOps* and *Cloud*).

Application Programming Interface (API)

From Program Architecture to System Architecture



Interfaces within a program

If components work together as part of one program (i.e., within one project in one programming language), you can use the interface mechanisms of that programming environment (e.g., `Interface` in Java)

Interfaces within systems

But what if the components are written independently from each other, in completely different programming languages and environments?

Application Programming Interface (API)

Application Programming Interface (API)

An **application programming interface (API)** is an interface for machine-to-machine (or system-to-system or component-to-component) communication

It is a type of **software interface**, offering a service to other pieces of software, independent from its implementation.

A document or standard that describes how to build or use such a connection or interface is called an **API specification**.

A computer system that meets this standard is said to implement or expose an API.

APIs everywhere

APIs for…

- software libraries and frameworks (e.g., `java.io`)
- operating systems (e.g., POSIX)
- remote execution (e.g., JDBC)
- web applications (e.g., REST)

APIs in practice

APIs define the main contact point of a component (like a counter in a fast-food restaurant). Thus, they should…

- be stable (and versioned)
- be secure
- be well documented
- define and guarantee a certain level of quality

Universität zu Köln

Example: RESTful APIs

Motivation

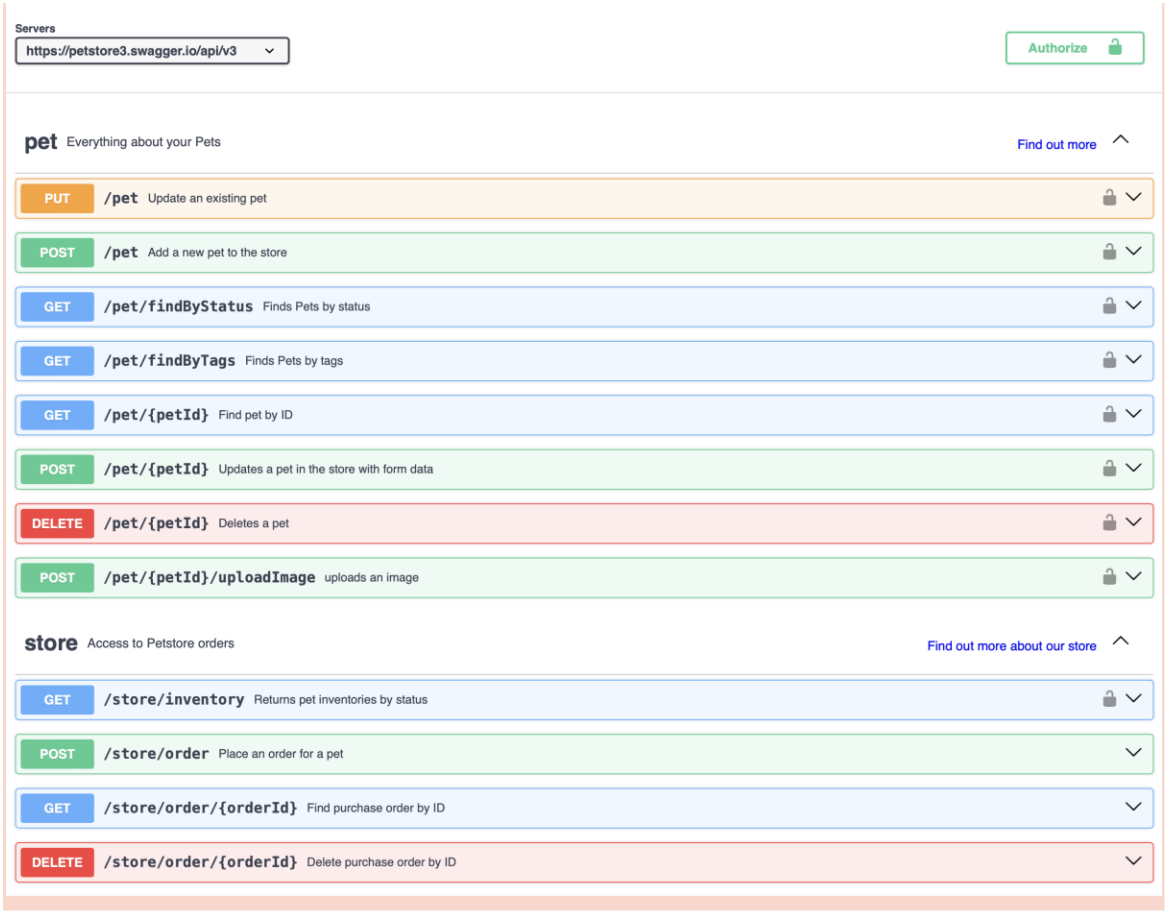
- Components shall be as independent as possible
- Components shall be scaled as easy as possible
- Components shall have uniform interfaces

RESTful APIs

- Everything a server has to offer is a **resource**
- Resources are identified by global IDs (URIs)
- A client can request or change a resource
- The HTTP standard methods (GET, POST,…) are used as the uniform interface for all API calls.

More on this later in the lectures on web applications

Pet store example



Summary

Software Architecture

- Software architecture is the link between requirements and code
- Is especially important for ensuring quality requirements
- Component diagrams can be used to describe the development view
- Architectural patterns help finding good architectural solutions for recurring problems.