Softwarearchitektur

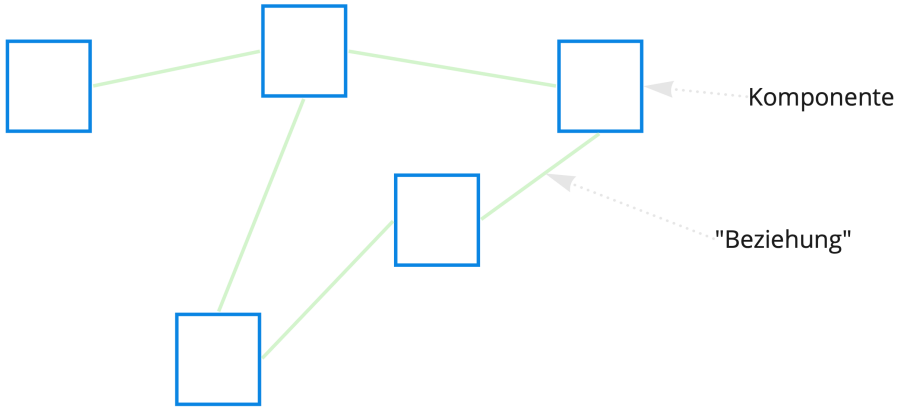
Prof. Dr. Roland Schätzle

Inhalt

- Einführung und Definition
- Komponenten und Schnittstellen
- Architekturen beschreiben und dokumentieren
- Schnittstellen in Programmiersprachen

Einführung und Definition

Ein „Architekturdiagramm”



- Was ist die Bedeutung der Beziehungen?
  - … *„benutzt”*, *„ist abhängig von”*, *„interagiert mit”*?
  - dies ist oft nicht klar definiert

Softwarearchitektur – Definitionen

> The software architecture of a program or computing system is the **structure** or structures of the system, which comprise software **components**, the **external visible properties** of those components and the **relationships** among them.
>
> L. Bass, P. Clements, R. Kazman: *Software Architecture in Practice*. Addison-Wesley, 2013 1

...

> Eine Softwarearchitektur beschreibt die **Strukturen** eines Softwaresystems durch **Architekturbausteine** und ihre **Beziehungen** und Interaktionen untereinander sowie ihre physikalische Verteilung. Die extern sichtbaren Eigenschaften eines Architekturbausteins werden durch **Schnittstellen** spezifiziert.
>
> 2

… noch mehr Definitionen

Softwarearchitektur ist …

> … the **shared understanding** that the expert developers have of the system design … the **decisions** you wish you could **get right early** in a project

→ Nachträgliche Änderungen an der Architektur sind teuer und aufwändig!

...

> Architecture is about **important stuff**. Whatever that is.

Martin Fowler ,
Software Architecture Guide

...

> Architecture is the **stuff you can't Google**
>
> 3

Entwurf einer guten Softwarearchitektur

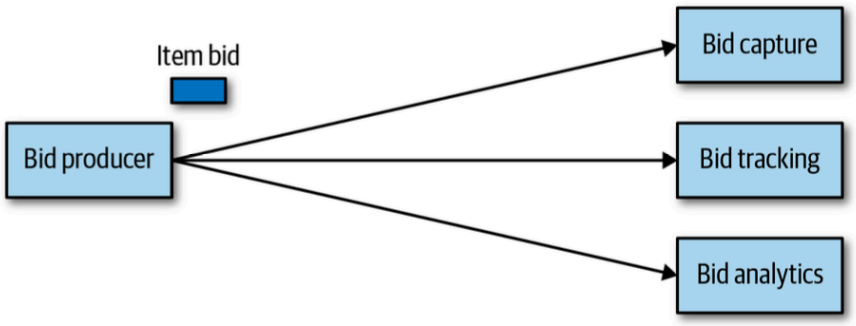
- Für die meisten Problemstellungen gibt es **nicht "die beste Lösung"**
- Vor- und Nachteile von **Lösungsalternativen** müssen gegeneinander abgewogen werden
- Die gewählte Lösung ist meist ein **Kompromiss**

...

> ***"Laws of Software Architecture"*** 3
>
> - Everything in software architecture is a **trade-off**
> - If an architect thinks they have discovered something that *isn't* a trade-off, more likely they just haven't **identified** the trade-off yet
> - **Why** ist more important than **how**

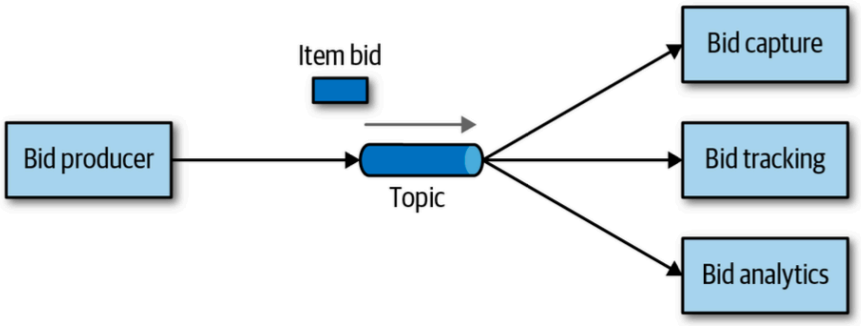
Beispiel für Architekturentscheidungen

Beispiel: Ein System zur Abwicklung von Auktionen



- Komponente `Bid producer` sendet die Gebote der verschiedenen Bieter
- Komponente `Bid capture` dient der Gebotsabgabe/-erfassung
- Komponente `Bid tracking` überwacht die gesamte Auktion
- Komponente `Bid analytics` führt verschiedene Analysen durch (z.B. um Betrugsversuche zu entdecken)

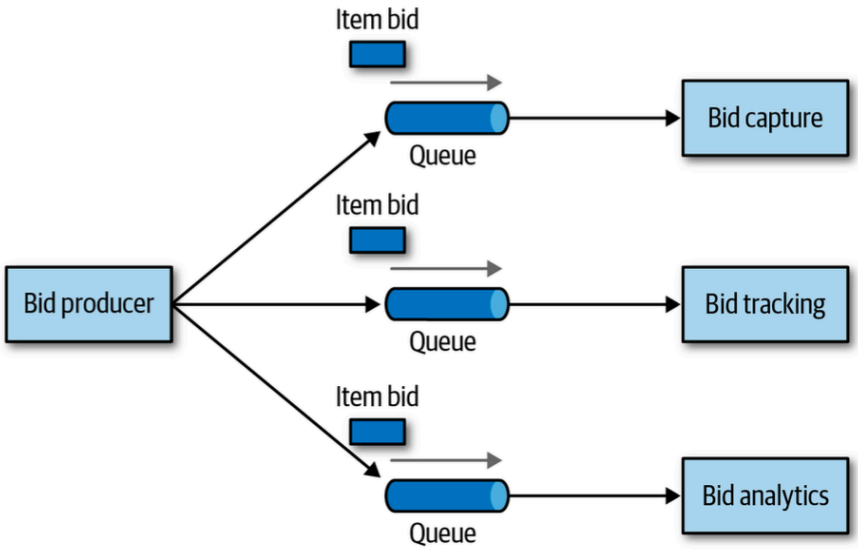
Beispiel: Lösungsalternative 1



- Verwendung einer zentralen **Event-Broker**-Komponente
- interessierte Komponenten interagieren über "**publish & subscribe**"-Mechanismus

---

Beispiel: Lösungsalternative 2



- **Punkt-zu-Punkt**-Kommunikation
- Nutzung von **Warteschlagen**-Komponenten zur Ereignisübertragung

---

Beispiel: Bewertung (1)

Anforderung: System muss um eine Komponente `Bid History` ergänzt werden, die für jeden Bieter über alle Auktionen sämtliche Gebote protokolliert.

- In **Lösung 1** ist diese Erweiterung **einfach**:
  - Die neue Komponente ist ein weiterer "Subscriber"
  - `Bid producer` muss dafür in keinster Weise angepasst werden

...

- In **Lösung 2** muss
  - eine **neue Warteschlangen-Komponente** hinzugefügt werden
  - der `Bid producer` **erweitert** werden, damit er diese auch mit Nachrichten versorgt

---

Beispiel: Bewertung (2)

Aber so einfach ist die Lage nicht …

- **Sicherheit**:
  - **beliebige** neue Komponenten können sich in Lösung 1 als "Subscriber" anmelden – auch **bösartige**
  - in Lösung 2 hat man sehr **genaue Kontrolle** über die Weitergabe der Ereignisse

…

- **Nachrichtenvarianten**:
  - in Lösung 1 bekommt jeder "Subscriber" **dieselben Informationen**
  - in Lösung 2 kann man den **Nachrichteninhalt spezifisch** entsprechend der Anforderungen des Konsumenten zuschneiden

Komponenten und Schnittstellen

Komponenten und Schnittstellen

- **Komponenten** sind die „Bausteine" einer Architektur
  - *zusammengesetzte Komponenten* bestehen aus weiteren Subkomponenten
  - *einfache Komponenten* sind nicht weiter unterteilt
- **Schnittstellen** sind deren nach außen sichtbare Eigenschaften (über die die Komponenten miteinander in Beziehung stehen)

Komponenten

> A software component is a **unit of composition**
> with
> **contractually** specified **interfaces** and
> **explicit**
> context **dependencies** only.

C.
Szypersky: Component Software, Addison-Wesley, 2009 4

...

> A software component is a **coherent package**
> of software
> implementation that
> a) has explicit and well-specified interfaces
> for **services it provides**
> b) has explicit and well-specified
> interfaces for **services it expects**; and
> c) can be
> **composed** with other components, perhaps **customizing** some of
> their **properties**, **without modifying** the components themselves.

D.F. D'Souza, A.C.
Wills: Objects, Components and Frameworks with UML:
The
Catalysis Approach, Addison-Wesley, 1999 5

Eine Komponente ...

- **exportiert** eine oder mehrere **Schnittstellen**, die im Sinne eines Vertrags garantiert sind
- **importiert** andere **Schnittstellen**
- die eine Schnittstelle S exportiert, ist eine **Implementierung** von S
- **versteckt** die **Implementierung** und kann somit durch eine andere Komponente ersetzt werden
- kann andere **Komponenten enthalten**
  - über beliebig viele Stufen hinweg → Komponentenhierarchie
  - zusammengesetzte vs. einfache Komponenten

    - in oo-Systemen: Komponenten der unterste Ebene = Klassen
- eignet sich als **Einheit der Wiederverwendung**

Komponenten – Erscheinungsformen

Je nach Kontext und Verwendung haben Komponenten unterschiedeliche Erscheinungsformen:

- **Module** (Packages, Modules, Namespaces etc.) in einer Programmiersprache, bestehend aus
    - Klassen
    - Typen/Traits
    - Funktionen
- **Kompilierte Einheiten** ((shared) Libraries)
- (eigentständig) **ablauffähige Einheiten**
    - Web-Service
    - Micro-Service
    - Subsystem

Schnittstellen

> *Eine ∗∗Schnittstelle∗∗...* beschreibt bildhaft die Eigenschaft eines Systems als Black Box, von der nur die „Oberfläche" sichtbar ist; nur über diese ist Kommunikation möglich. Zwei benachbarte Black Boxes können nur miteinander kommunizieren, wenn ihre Oberflächen „zusammenpassen".
>
> **Softwareschnittstellen** ... sind logische Berührungspunkte in einem Softwaresystem: Sie ermöglichen und regeln den Austausch von Kommandos und Daten zwischen verschiedenen Prozessen und Komponenten.
>
> **Schnittstellen für Programmkomponenten** sind eine formale Deklaration, welche Funktionen vorhanden sind und wie sie angesprochen werden können.

[Wikipedia](https://de.wikipedia.org/wiki/Schnittstelle)

Eine Softwareschnittstelle ...

definiert eine Menge von Funktionen mit folgenden Eigenschaften 6:

- **Syntax**: Funktionsname, Parameter (Name & Typ), Rückgabewert(e) (mit Typ)
- **Semantik**: Was macht die Funktion?
- **Nicht-funktionale Eigenschaften**: Komplexität, Performance, Verfügbarkeit, Ressourcenbedarf etc.

Ggf. auch ein **Protokoll**, d.h. eine Menge von Regeln, die die Kommunikation bestimmen.

Man unterscheidet folgende Arten von Funktionen:

- *Kommando (command)*: ändert den Zustand der betreffenden Komponente

- *Abfrage (query)*: lässt den Zustand unverändert

Schnittstellen auf Ebene von Programmiersprachen

Beispiel: Listen in Java (Auszug aus der Interface-Beschreibung)

```
public interface List<E> {
  int size();
  boolean isEmpty();
  boolean contains(Object o);
  Iterator<E> iterator();
  boolean add(E x);
  boolean remove(Object o);
  boolean equals(Object o);
  E get(int index);
  E set(int index, E element);
}
```

- `size` gibt die Anzahl der Elemente an, `isEmpty` prüft, ob überhaupt welche vorhanden sind
- `contains` prüft, ob ein best. Element vorhanden ist
- `iterator` liefert einen Iterator auf die Liste
- `add` fügt ein Element hinzu, `remove` löscht ein Element
- `equals` prüft, ob `o` und die Liste gleich sind
- `get` liefert das Element an einer best. Position, `set` tauscht es aus

---

Der Teufel steckt im Detail

Die scheinbar einfach zu verstehende Schnittstelle ist im Detail doch komplizierter:

- in Java gibt es **optionale Methoden**; diese müssen nicht unbedingt implementiert sein; im vorliegenden Fall sind dies `add` und `remove`
- `add` darf (je nach Implementierung) bestimmte **Werte zurückweisen** (z.B. Null-Werte)
- Es gibt eine ganze Reihe unterschiedlicher **Fehlersituationen**; bei `add` z.B. `ClassCastException`, `NullPointerException`, `IllegalArgumentException` und `OutOfMemoryError`

Diese Komplikationen lassen sich allerdings zurückführen auf

- sprachspezifische bzw. implementierungsspezfische Probleme
- Fehlerbehandlung

Schnittstelle und Implementierung

- Schnittstelle trennt **Außensicht** von der **Implementierung** (Geheimnisprinzip/Kapselung)
- Zu einer Schnittstelle kann es **mehrere Implementierungen** geben

- Schnittstellen
  - machen Software **leichter verständlich** (man muss die Impl. nicht verstehen)
  - helfen **Abhängigkeiten** zu **reduzieren**
    (nur noch
    Abhängigkeit in der Schnittstelle; keine Abhängigkeit von
    Implementierung → das muss aber auch wirklich gewährleistet sein)
  - erleichtern **Wiederverwendung**

Schnittstellen und Tests

- Tests schreibt man **gegen eine Schnittstelle**, nicht gegen die Implementierung (Black-Box-Tests!)
- **Einschränkung**: Falls es unterschiedliche Implementierungen gibt, die **unterschiedliche nicht-funktionale** Eigenschaften zusichern, kann es notwendig sein, dafür spezialisierte Tests zu schreiben
- Tests können eine Schnittstelle **präzisieren** und **verständlicher** machen

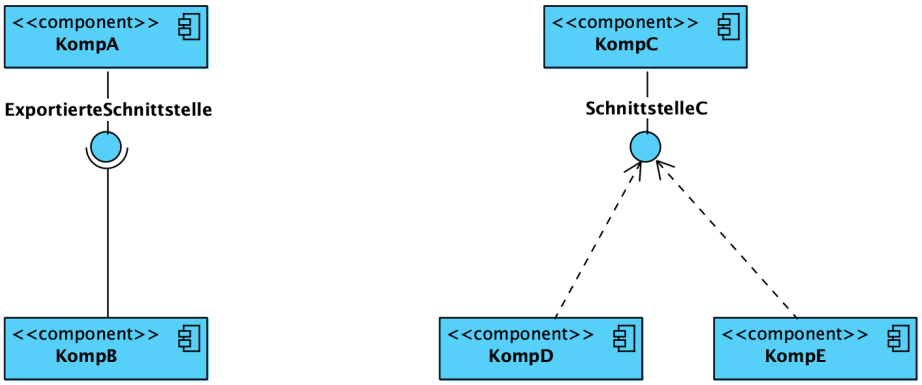
---

Schnittstellen und Softwareentwurf

- Schnittstellen sind (wie die Komponenten) eine eigene **Einheit des Entwurfs**
- Schnittstellen sind die **Träger der Softwarearchitektur**
- Beim Entwurf wird man zunächst geeignete **Schnittstellen entwerfen**

  (und sich erst im zweiten Schritt der Implementierung widmen)

UML: Komponentendiagramme



Beziehungstypen: *use* und *implement*

Erste Beispiele

Ziele:

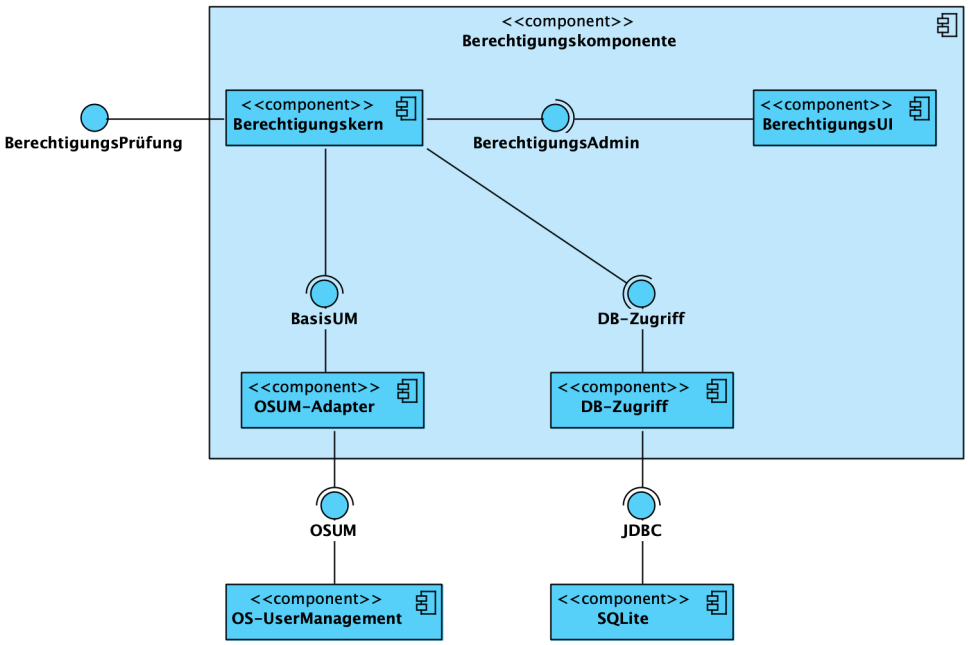
- **Veranschaulichung** der o.g. Konzepte
- Einführung **erster Konzepte** für den Architekturentwurf
- Einführung der **Notation** (UML-Komponentendiagramme)

Berechtigung: Eine Komponente zur Rechteprüfung und -verwaltung

Wesentliche Aufgaben dieser Komponente:

- Verwaltung von **Benutzern**, ihren **Zugriffsrechten** (und ggf. Nutzergruppen)
- Prüfung von **Zugriffsberechtigungen** (wenn ein Benutzer X auf ein Objekt Y zugreifen möchte)

Berechtigung: Architektur



Berechtigung: Anmerkungen zur Architektur

- Aufteilung der Funktionalität von `Berechtigungskern` in **zwei Schnittstellen**
  - beide Schnittstellen haben völlig **disjunkte Nutzer und Nutzungsprofile**
  - `BerechtigungsPrüfung` wird sehr häufig aufgerufen und muss deshalb **performant** sein
  - `BerechtigungsAdmin` spiegelt die **grundlegenden Konzepte** des Berechtigungskonzepts wider
- `OSUM-Adapter` und `DB-Zugriff` **isolieren** von externen, technischen Schnittstellen

---

Kläranlagenmonitor: Eine Komponente zur Überwachung von Kläranlagen

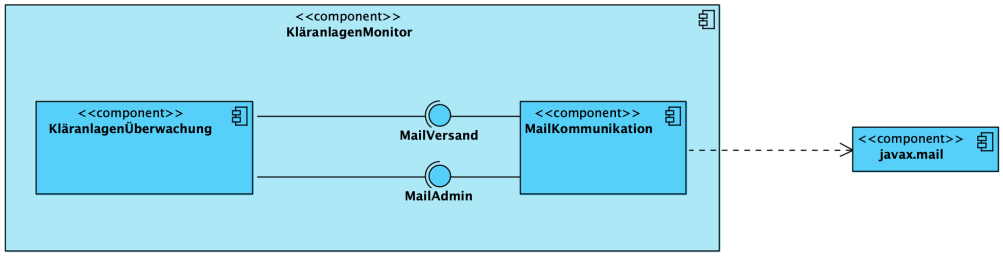
Ein ganz einfaches Beispiel!

- System **überwacht Kläranlage** (über Sensoren)
- Bei **Störungen** soll der Klärwärter **per E-Mail informiert** werden

  (es soll lediglich E-Mails an genau eine Adresse versenden)

- Implementierung mit Java unter Verwendung des `javax.mail`-Pakets
  - das
    Paket
    ist funktional sehr **reichhaltig**, aber auch sehr **komplex**
  - es umfasst 6 Interfaces und weit über 20 Klassen

Kläranlagenmonitor: Architektur



Kläranlagenmonitor: Anmerkungen zur Architektur

- Überwachung per Sensorik aus Gründen der Einfachheit ignoriert
- E-Mail-Versand:
  - **Isolation** von der komplexen und technischen `javax.mail`-Schnittstelle
    - `MailKommunikation` ermöglicht auch Einsatz einer Dummy-Komponente für verinfachten Test
  - intern Aufteilung in **zwei E-Mail-Schnittstellen**:
    - **Versenden** von E-Mails (`MailVersand`)
    - **Administration** des E-Mail-Versands (`MailAdmin`)

Arten von Schnittstellen

Abhängig davon, wer eine Schnittstelle definiert, unterscheidet man:

- **Standardschnittstellen**: Die Schnittstelle ist Allgemeingut
- **Angebotene Schnittstellen**: Der Exporteur definiert die Schnittstelle
  - Der Importeur macht sich abhängig vom Exporteur
  - Kritische Betrachtung der Notwendigkeit (und des Umfangs) der importierten Funktionalität sinnvoll
- **Angeforderte Schnittstellen**: Der Importeur definiert die Schnittstelle
  - Schnittstelle und zugeh. Implementierung muss meist speziell für diesen Fall erstellt werden
  - Für Nutzung von Standardschnittstellen ist typischerweise ein Adapter notwendig

---

Schnittstellen-Adapter

- In manchen Fällen muss zwischen angebotener und angeforderter

Schnittstelle **vermittelt** werden

- Dies erledigt ein sog. **Adapter**
- Dies sind spezielle Komponenten, die meist **genau eine Schnittstelle importieren** und auch nur **eine Schnittstelle exportieren**
- Der Adapter fungiert als "**Mittler**" zwischen diesen Schnittstellen.

  Er führt dabei

  - **Datentransformationen** und
  - **Anpassungen** (u.a. Parametrisierung, Vereinfachung) durch und
  - kümmert sich um die **Ausnahmebehandlung**

Beschreibung von Komponenten (1)

Folgende Aspekte gehören zur Beschreibung einer Komponente:

- **Übersicht**:
  - Sinn und Zweck der Komponente; grundlegende Idee für ihr Dasein
  - Wesentliche Aufgaben, Zuständigkeiten und Verantwortlichkeiten
- **Außensicht**: eine Art Benutzerhandbuch für den Importeur; dies umfasst
  - die angebotenen Schnittstellen
  - Kontextinformationen (u.a. zur Instanziierung der Komponente und zu deren Betrieb)

Beschreibung von Komponenten (2)

- **Innensicht**: (innerer) Aufbau und Implementierung; Adressat ist das Wartungsteam
- **Variabilitätsanalyse**: Vorstellung von Änderungsszenarien, welche
  - von der Architektur vorgesehen sind
  - mit gewissen Umbauten konform zur Architektur umgesetzt werden können
  - größere Um- oder Neugestaltung der Architektur erfordern

Die wichtigsten Aspekte sind dabei die Übersicht und die Außensicht

Architekturen beschreiben und dokumentieren

Sichten auf eine Architektur

- **Logische Sicht**: (Statische) Struktur der Software, Aufteilung in Komponenten (ggf. über mehrere Hierarchiestufen hinweg)
- **Laufzeitsicht/Prozesssicht**: Zeigt, welche Prozesse zur Laufzeit existieren und wie sie zusammenwirken (dynamische Sicht)
- **Verteilungsssicht/Physische Sicht**: Gibt an, welche Architekturkomponenten auf welchen Hardwarekomponenten ablaufen

---

Dokumentation einer Architektur – Was und wieviel? (1)

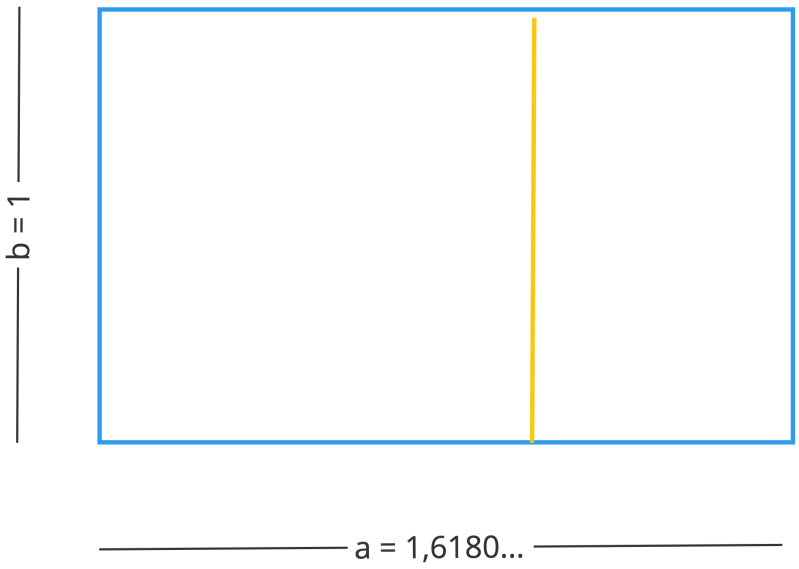
- Es müssen nicht alle Sichten in allen Details dokumentiert werden
- WICHTIG: Die **wesentlichen Komponenten der logischen Sicht** (die Top-Level-Komponenten/Abstraktionen), deren Inhalte, Zuständigkeiten und Schnittstellen untereinander müssen klar und eindeutig beschrieben werden
    - Sie bilden das „**Shared Understanding**" des Gesamtsystems
    - Dieses Wissen kann **nicht** aus der Software **reengineered** werden

---

Beispiel eines Architekturkonzepts aus dem Bauwesen

Le Corbusier – Goldener Schnitt („Modulor")

Goldener Schnitt: $\frac{a}{b} = \frac{a+b}{a}$



Was und wieviel? (2)

- Die **unterste Ebene der logischen Sicht** ist
  (in einem
  objekt-orientierten System) die **Klassenstruktur**
- Sie ist über den **Quellcode** sichtbar
- Deshalb ist es meist unnötig, sie separat vollständig zu dokumentieren
- Eine **punktuelle Dokumentation**,
  - zur Erläuterung **besonderer** Aspekte ist ausreichend
  - **besonders** im Sinne von besonders **schwierig**,
    besonders
    kritisch etc.

Was und wieviel? (3)

- Auch für die **Prozesssicht** ist es meist ausreichend
  nur
  besondere Aspekte zu dokumentieren
- Für die **Verteilungssicht** reicht oft ein einfaches

  Übersichtsdiagramm mit einigen Erläuterungen

...

- ABER: Es kommt immer auf das spezifische System an –
  wenn es z.B.
  - hohe/spezielle nicht-funktionale Anforderungen
    (z.B. für
    kritische Infrastruktur, eingebettete Systeme)
  - regulatorische Vorschriften

  gibt, muss deutlich mehr dokumentiert werden

Womit (Notation)? (1)

- In der Vorlesung verwenden wir die UML:
  - **Komponentendiagramme** für die *logische Sicht*
  - **Deploymentdiagramme** für die *Verteilungssicht*
  - **Sequenzdiagramme, Aktivitätendiagramme u.a.** für die *Prozesssicht*

Womit (Notation)? (2)

- Es ist jedoch strittig, ob die **UML**
  - nicht „**zu technisch**" ist und
  - für die Diskussion mit Nicht-IT-Experten besser informellere Notationen (z.B. Blockdiagramme) verwendet werden sollen
- Die UML hat allerdings den Vorteil, dass die Diagrammelemente (und deren Bedeutung) **eindeutig(er) definiert** sind

...

- WICHTIG:
  - Kaum ein Diagramm (egal in welcher Notation) ist selbsterklärend
  - Eine **ergänzende, textuelle Beschreibung** ist fast immer notwendig

Schnittstellen in Programmiersprachen

Exkurs: Schnittstellen in anderen Programmiersprachen

Ein Blick über den Tellerrand:

- Java ist verbreitet und and der DH die zentrale Ausbildungssprache
- Es beginnen sich jedoch neue Programmiersprachen zu etablieren,

  die **nicht mehr** oder nur noch **teilweise** auf dem Konzept
  der
  Objekt-Orientierung basieren
- Bei diesen Sprachen spielen u.a. Schnittstellen eine wichtige Rolle
  - man spricht von „**traits**" (= Eigenschaften) bzw. **Trait-Based Languages**
  - gemeint sind damit Schnittstellen
- Weiterführende Informationen: 7, 8

Schnitttellen in Julia und Rust

- Wir betrachten zwei dieser Sprachen (Julia und Rust) genauer
- Beide haben das Potenzial, bislang verbreitete Sprachen abzulösen
  - C/C++ → **Rust** (als **Systemprogrammiersprache**)
  - Python → **Julia** (im Bereich **Data Science & Machine Learning**)

Sprache	Klassen/Typen	Typisierung	Polymorphismus	Schnittstellen	Übersetzung
Java	objekt-orientiert, Klassen	statisch	klassisch oo	Klassen & Interfaces	kompiliert
Rust	Typen (user-defined)	statisch	teilweise	Traits (explizit)	kompiliert
Julia	Typen (user-defined, Typhierarchie)	dynamisch, optional deklariert, Typinferenz	multiple dispatch	Traits (informell)	JIT-kompiliert

---

Julia: informelle Schnittstellen

```
abstract type AbstractArray         # Basistyp nicht
zwingend notwendig

size(A::AbstractArray)
length(A::AbstractArray)
keys(a::AbstractArray)
getindex(A::AbstractArray, i::Int)
getindex(A, inds...)
```

...

- `size`: Return a tuple containing the dimensions of `A`. Optionally you can specify a dimension to just get the length of that dimension.
- `length`: Number of elements
- `keys`: Return an efficient array describing all valid indices for `a` arranged in the shape of `a` itself.
- `getindex`: Return a subset of array `A` as specified by `inds`, where each `ind` may be, for example, an `Int`, an `AbstractRange`, or a `Vector`. See the manual section on array indexing for details.

Rust: Schnittstellen mit `trait`

```
pub trait List {
  type E;                                 // element type
  fn new() -> Self;                       // Construct an
empty list
  fn is_empty(&self) -> bool;             // Test wether
a list is empty
  fn len(&self) -> usize;                 // Get the
length of a list
  fn head(&self) -> Option<Arc<E>>;       // Get the
first element of a list
  fn tail(&self) -> Option<Self>;         // Get teh tail
of a list
}

pub struct MyFancyDataStructure<A> {
  ...
}

impl List for MyFancyDataStructure<A> {
  type E = A;
  fn is_empty(&self) --> {
    ...    // Implementierung der Funktion is_empty
  }
  ...      // Implementierung der übrigen Funktionen
}
```

Literatur

1
L. Bass, P. Clements, and R.
Kazman, *Software architecture in practice*, 3rd ed. in SEI series in
computer software engineering. Upper Saddle River, NJ: Addison-Wesley,
2013.

2
H. Balzert and P. Liggesmeyer,
*Lehrbuch der Softwaretechnik. 2: Entwurf, Implementierung, Installation
und Betrieb / Helmut Balzert. Unter Mitw. von Peter Liggesmeyer*, 3.
Auflage. in Lehrbücher der Informatik. Heidelberg: Spektrum,
Akademischer Verlag, 2011.

3
M. Richards and N. Ford,
Fundamentals of software architecture: An engineering approach, First
edition. Sebastopol, CA: O'Reilly Media, Inc, 2020.

4
C. Szyperski, D. W. Gruntz, and S.
Murer, *Component software: Beyond object-oriented programming*, 2. ed.,
repr.. in Addison-wesley component computer software series. London
Munich: Addison-Wesley *u. a.*, 2009.

5
D. F. D'Souza and A. C. Wills,
Objects, components, and frameworks with UML: The catalysis approach.
in The addison-wesley object technology series. Reading, Mass:
Addison-Wesley, 1999.

6
J. Siedersleben, *Moderne
Softwarearchitektur: Umsichtig Planen, robust Bauen mit Quasar*, 1.
Aufl., korr. Nachdr. Heidelberg: dpunkt-Verl, 2006.

7
N. Schärli, S. Ducasse, O.
Nierstrasz, and A.P. Black, "Traits: Composable units of behaviour," in
LNCS 2743, Springer, 2003, pp. 248–274.

8
Will Crichton, "Programming
languages: traits," Vorlesung, Standford University, 2019.

#