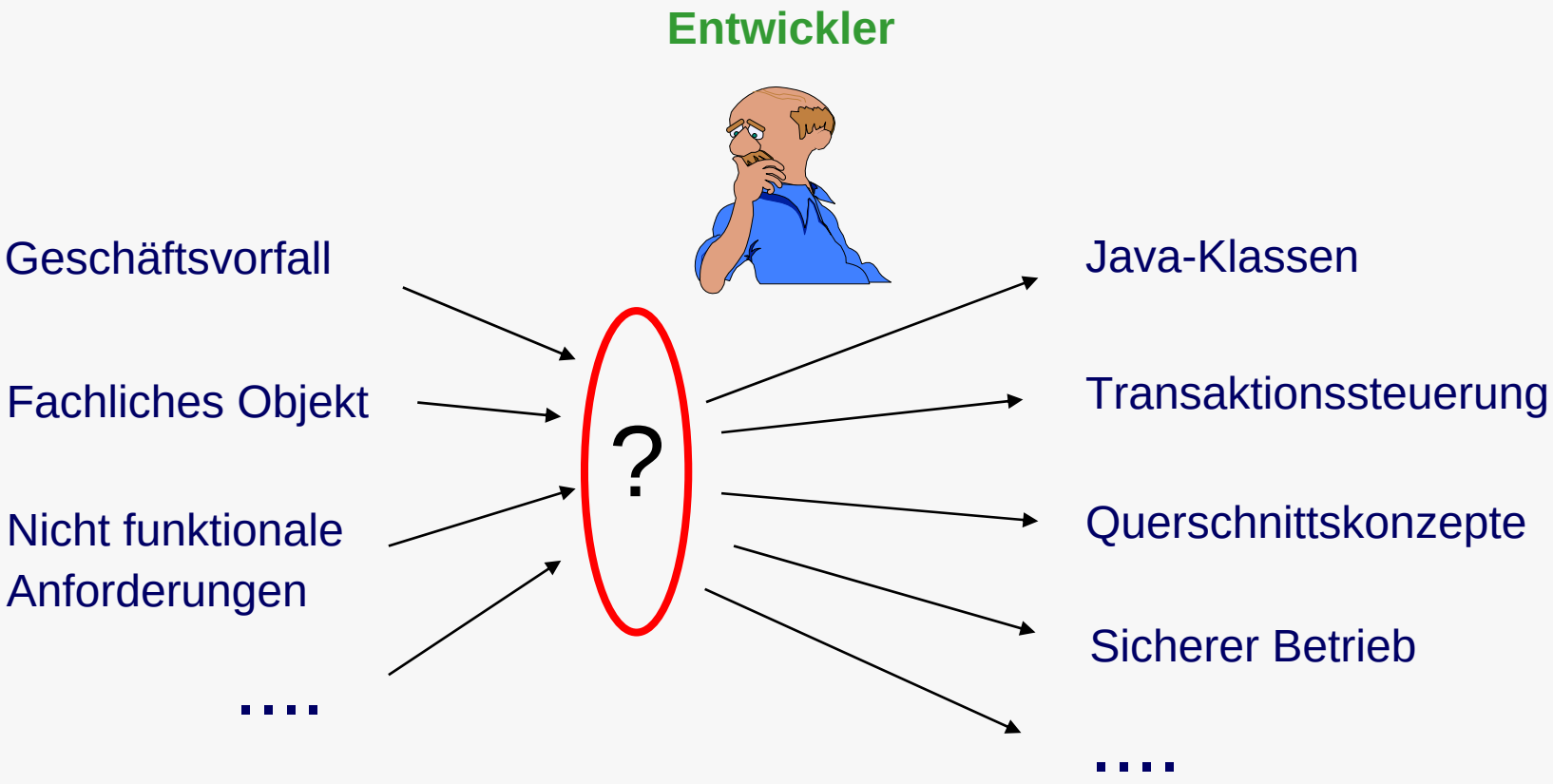
Systemkonstruktion

Übergang vom Fachkonzept zum DV-Konzept

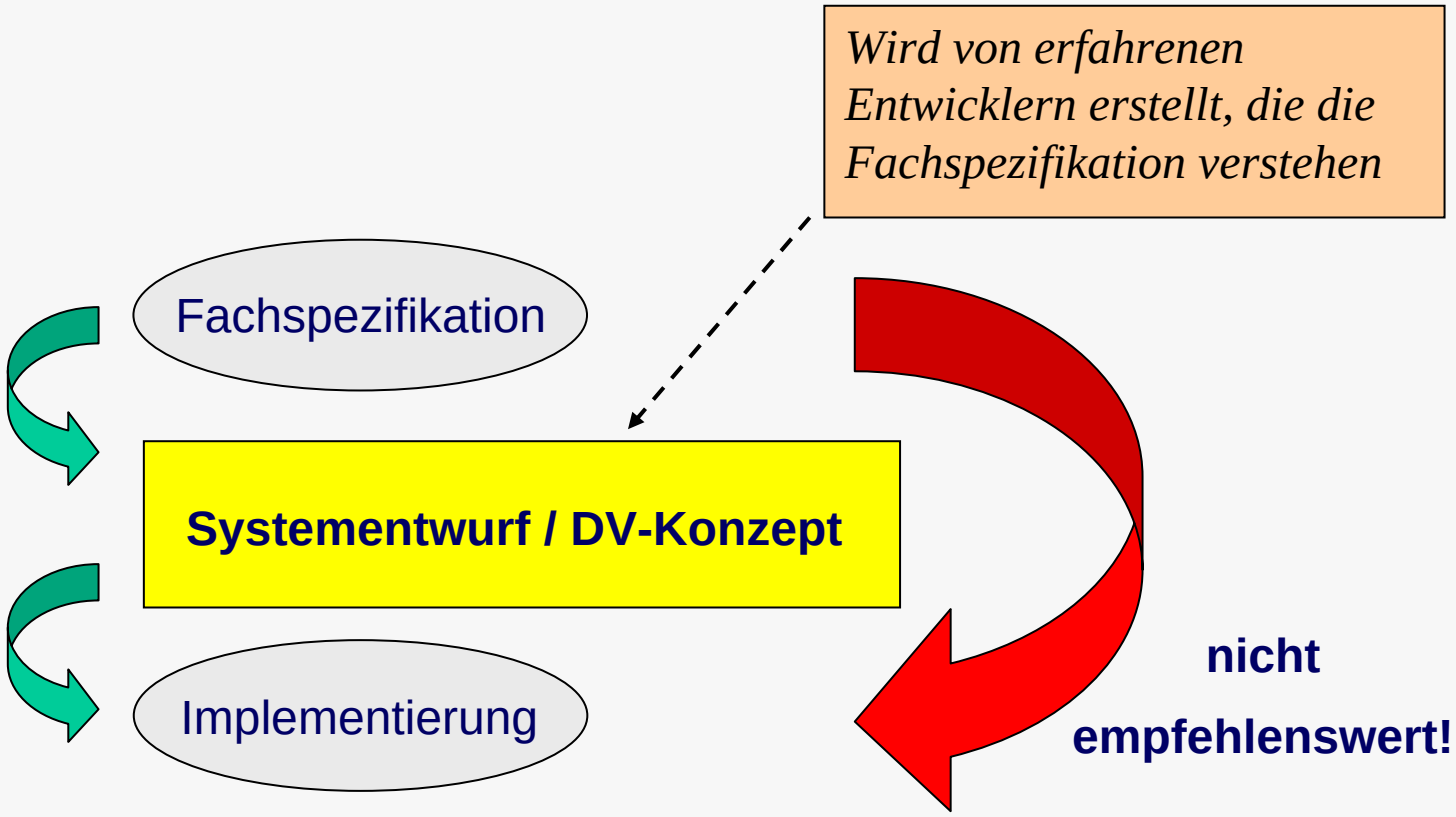
Aufgabenstellung



Bedeutung des DV-Konzepts

Das DV-Konzept dient als Beschreibung des Vorgehens, wie die Fachspezifikation später in der Realisierung umgesetzt werden soll.

Das DV-Konzept ist auch als „Bauplan“ des zukünftigen Systems zu verstehen.

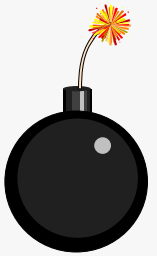


Folgen einer fehlenden Systemkonstruktion

Beispiel

(tatsächlich passiert, Multi-Millionenprojekt im Mobilfunkbereich im Juli 1995):

- Grobe Probleme bei der Installation
- System stürzt oft ab
- Datenbank wird inkonsistent
- Multi-User-Betrieb kaum möglich
- Performance miserabel
- In der ersten Woche mehr als 200 gefundene Fehler



Typische Inhalte eines DV-Konzepts (1)

Architektur

- Darstellung der Systemstruktur (Komponenten, Zusammenspiel)
- Schichtenmodell
- Beschreibung einzelner Komponenten
- Fremdmodule
- Schnittstellenbeschreibungen, -techniken und –kontrakte
- Aspekte der Verteilung
- Prozessmodell

Typische Inhalte eines DV-Konzepts (2)

Programmierkonzepte

- Umsetzung von Anwendungsfällen und geforderten Funktionen
- Umsetzung nicht funktionaler Anforderungen
- Physisches Datenmodell (Datenbankdesign)
- Beschreibung betriebsrelevanter Techniken
- Batches (mit Steuerung)

Typische Inhalte eines DV-Konzepts (3)

Technische Fragestellungen

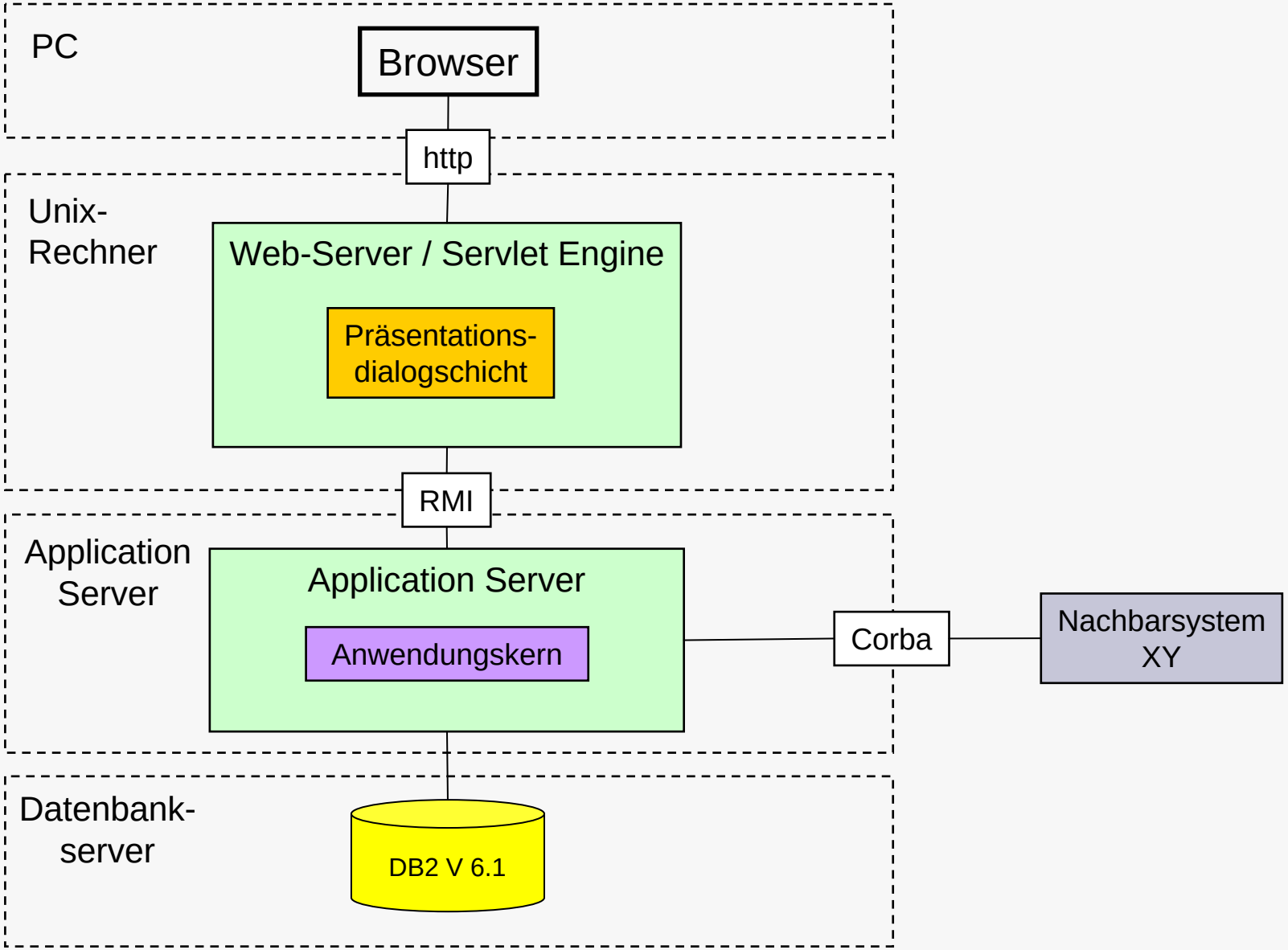
- Dialogkonzept / GUI-Programmierung
- Parallelverarbeitung
- Druckthematik
- Archivierung
- Historisierung
- Datensynchonisation
- Sicherheit (Echtheit, Verschlüsselung, …)
- Datenhaltung / Anbindung der Datenbank an den Anwendungskern
- Multi-User-Betrieb
- Austausch von Daten über Rechnergrenzen hinweg (Kommunikation)
- Monitoring des Systems (u.a. Performance)
- Fachliches Accounting
- Technisches Logging
- Workflow
- Fehlerbehandlung
- Transaktionskonzept
- Berechtigungskonzept

Typische Inhalte eines DV-Konzepts (4)

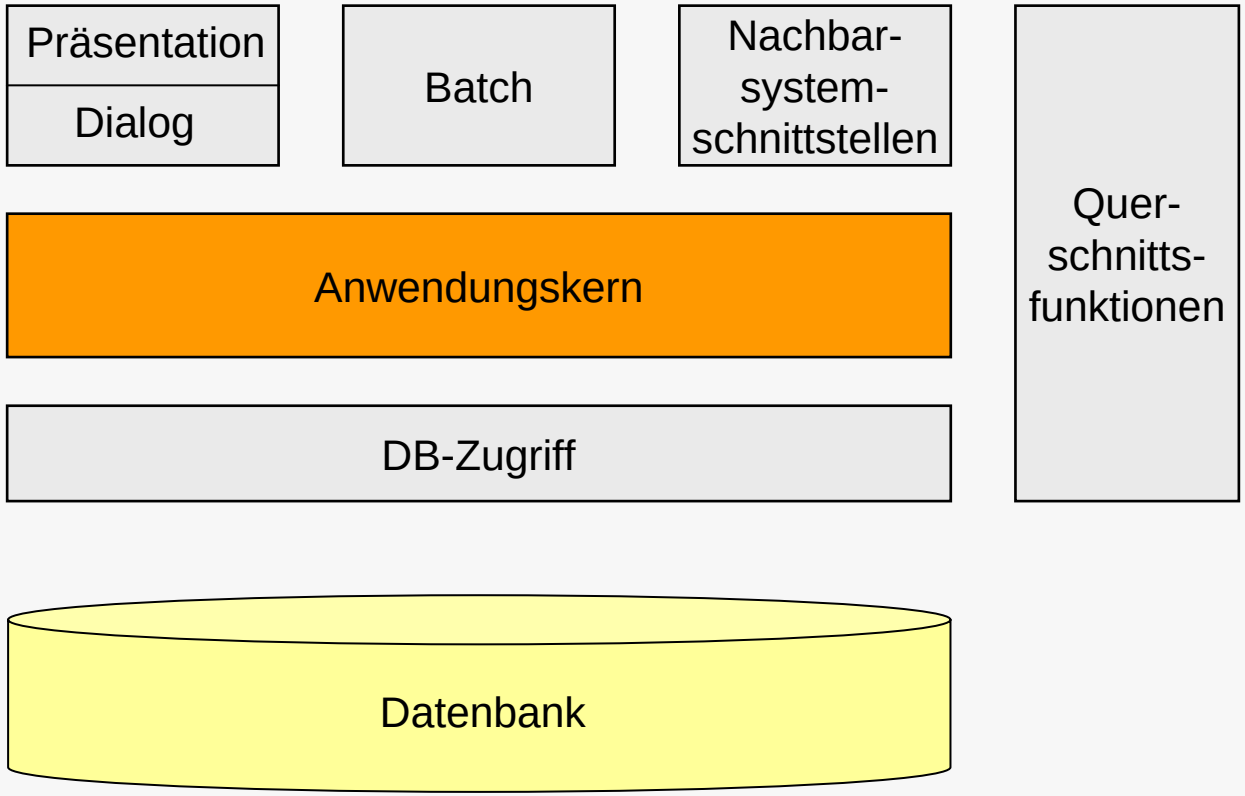
Weitere Themen (unsortiert):

- Erfahrungen aus Prototypen?
- Einstellmöglichkeiten des Systems?
- Platzanforderungen / Mengengerüste
- Datenmigration
- Notbetrieb
- Berücksichtigung von Richtlinien und Standards
- Vorgaben der Systeminfrastruktur?
- Infrastruktur von Testumgebung, Produktivumgebung, ...
- Einsatz von Entwurfsmustern
- „Release-Fähigkeit“
- Integrationsstrategie
- Anlaufplan zur Produktivsetzung
- Werkzeuge (Testdatengeneratoren, Testwerkzeuge, ...)
- Testbarkeit von Einzelkomponenten
- Wiederverwendbarkeit

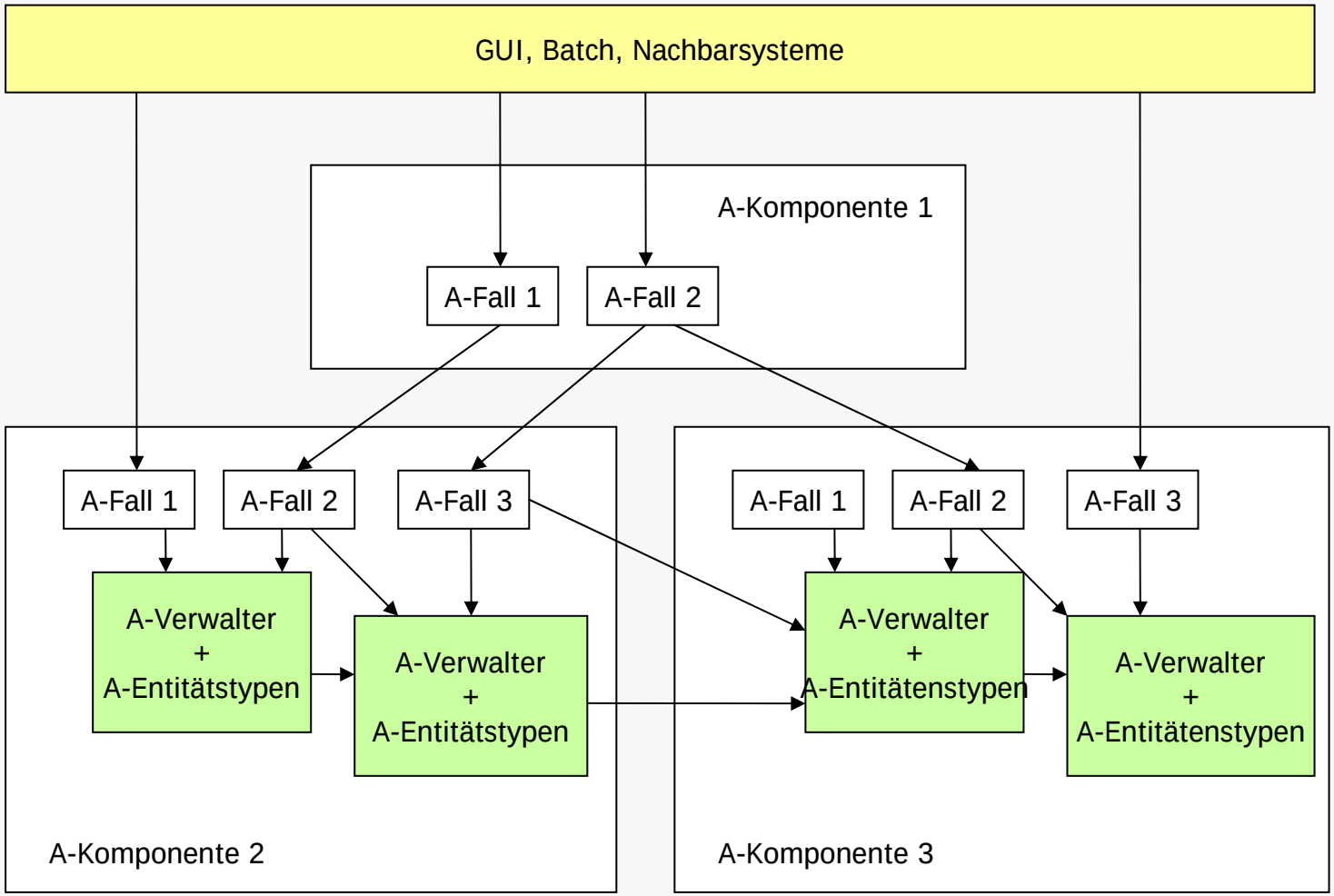
Architektur der technischen Infrastruktur



Übliche Dreischichtenarchitektur



Eine mögliche Anwendungskernarchitektur



Architektur des Anwendungskerns nach Quasar (Domäne „Betriebliche Informationssyteme“)

Anwendungsobjekte

Beispiel

```
public class Skilehrer extends Person implements ISkilehrer,
   Serializable
{
    private Id id;
    private Verfuegbarkeit verfuegbarkeit;
    //...
    public Skilehrer(Adresse adresse,
                     Datum geburtsdatum,
                     String name,
                     Sprache sprache,
                     Kurstyp kurstyp) { ... }

    public Id getId() { return id; }
    public Verfuegbarkeit getVerfuegbarkeit(){
       return verfuegbarkeit;}

    public boolean equals(Object x) { ... }
    …
}
```

Trennung der Zuständigkeiten

Auch bekannt unter „Separation of concerns“

- **Jede Softwarekomponente bzw. jedes Softwaremodul sollte sich möglichst nur mit einer (technischen oder fachlichen) Aufgabe befassen**

→ Klarer Code

→ Verständliche Architektur

→ Bessere Wartbarkeit

→ Möglichkeit, wiederverwendbare Komponenten zu identifizieren

→ Kapselung von herstellerabhängigen APIs, um Austauschbarkeit von Produkten zu erreichen

Software-Kategorien nach Siedersleben

Auch bekannt unter: „Software-Blutgruppen“

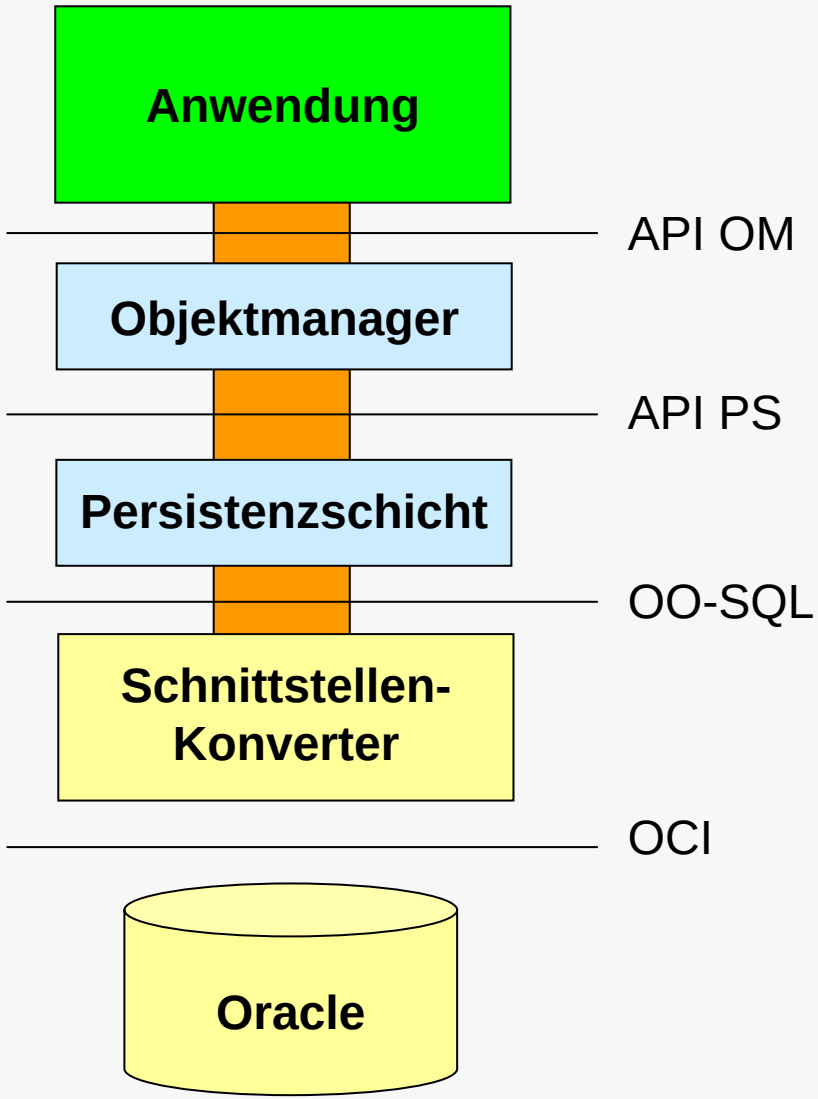
Software kann sein ...

0	bestimmt von gar nichts (Behälter, Strings) → *ideal wiederverwendbar, für sich alleine nutzlos*
A	bestimmt von der Anwendung (Kunde, Auftrag, Bestellung) → *das eigentliche Projektziel*
T	bestimmt von mindestens einem technischen API (z.B. Datenverwaltung) → *muss sein*
AT	bestimmt von der Anwendung und mindestens einem technischen API → *vermeiden; im Notfall sorgfältig abgrenzen*
R	Repräsentationssoftware (Transformation zwischen A und T; milde Art von AT)

Kombinationen	A + 0 = A T + 0 = T A + T = AT

Schnittstellen und Schichten

Beispiel



Schnittstellen aus Entwicklersicht

Warum sind Schnittstellen wichtig?

- Schnittstelle = Vertrag zwischen Nutzer und Anbieter

  → Der Anbieter ist austauschbar (ohne dass dies der Nutzer merkt)

Schnittstellen ...

- helfen, Abhängigkeiten zu reduzieren
- „verstecken“ Komplexität
- unterstützen die Entwicklung von Software im Team
- können problematisch werden, wenn sie häufigen Änderungen ausgesetzt sind

Fehlerbehandlung (1)

Was kann alles passieren?

- Fachliche Probleme
  - Konto nicht gedeckt
  - Es ist nicht die notwendige Berechtigung vorhanden
- Verletzte Vorbedingungen
  - Es wurde ein falscher Parameter übergeben
  - Die Buchung ist bereits storniert
- Technische Probleme
  - Netz temporär nicht verfügbar
  - Datenbank meldet unbekannte Fehlercodes zurück
  - Nachbarsysteme verhalten sich unerwartet
- „Hausgemachte“ Probleme
  - Programmierfehler
  - NullPointerException, ClassCastException, ….

Fehlerbehandlung (2)

Probleme mit Exceptions und Fehlern

- Welche Kategorien von Ausnahmen gibt es?
- Wie wird ein „Wildwuchs“ von Ausnahmen verhindert?
- Sind Ausnahmen von herkömmlichen Fehlern zu unterscheiden?
- Wer hat das Recht, Ausnahmen zu setzen?
- Wie werden „normale Return-Codes“ von Ausnahmen unterschieden?
- Wer hat das Recht bzw. die Pflicht, Ausnahmen zu fangen und zu behandeln?
- Was macht man in einer Ausnahmebehandlung mit einer unbekannten Ausnahme?
- Ab wann macht eine Fortführung des Programms keinen Sinn mehr?
- Wo liegen die Fehlermeldungstexte?

Fehlerbehandlung (3)

Anwendungsfehler ….

- haben nichts mit „Notfällen“ zu tun
- müssen vollständig spezifiziert werden
- werden üblicherweise über Return-Codes gemeldet
- werden unmittelbar vom Aufrufer (in der Anwendung) behandelt
- Der Rufende entscheidet letztlich, was ein echtes Problem ist und was nicht!