

Software Engineering 1

Hochschule für angewandte Wissenschaften Hamburg

Fachbereich Informatik

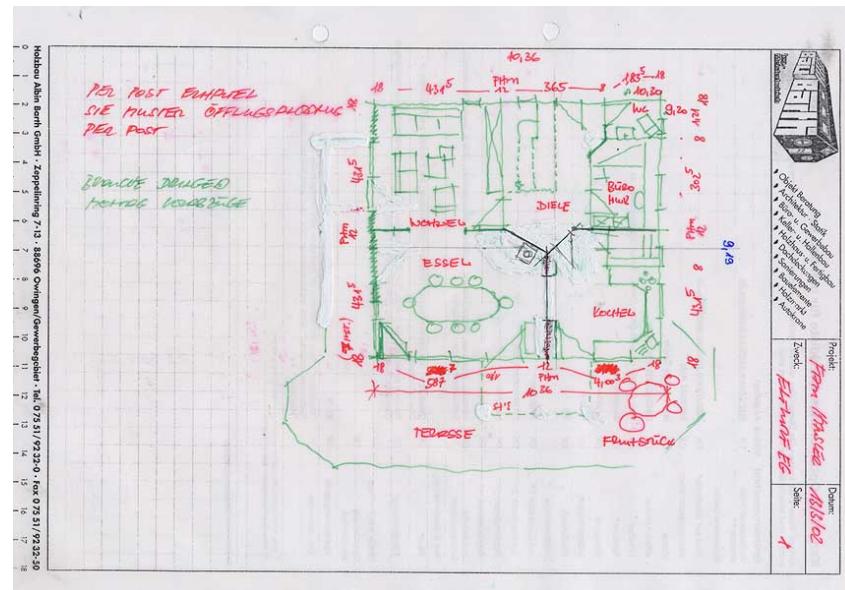
Prof. Dr. Stefan Sarstedt

(stefan.sarstedt@haw-hamburg.de)

Raum: 10.85

Entwurf/Design/Architektur

Iteration 1





Lernziele: Software-Engineering 1

- Was ist Software-Engineering?
- Wie läuft ein Softwareprojekt ab und wer ist beteiligt?
- Wie kann ich Kundenanforderungen analysieren und spezifizieren?
 Was macht ein gutes Anwendungsdesign aus und wie komme ich dorthin?
- Was ist der Unterschied zwischen „schmutzigem“ und „sauberem“ Code? Was ist bei der Implementierung zu beachten?
- Wie, wann und was sollte ich testen?



Spezielle Literaturempfehlungen für den Entwurf

[Gamma et.al. 94]

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns – Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

[Kecher 2011]

Kecher: UML 2: Das umfassende Handbuch, Galileo Computing, 2011.

[Larman 2004]

Craig Larman: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, Prentice Hall, 2004.

[Ludewig-Lichter 2007]

Ludewig, Lichter: Software Engineering, dpunkt.verlag, 2007.

[Martin 2002]

Robert C. Martin: Agile Software Development. Principles, Patterns, and Practices, Prentice Hall, 2002.

[Siedersleben 2003]

J. Siedersleben (Hrsg.): Softwaretechnik. Praxiswissen für Softwareingenieure. 2. Aufl., Hanser, München, Wien.

[Siedersleben 2004]

J. Siedersleben: Moderne Software-Architektur: Umsichtig planen, robust bauen mit Quasar, Dpunkt.



Literatur: Architektur und Komponenten



Siedersleben 2004

Quasar Teil 1

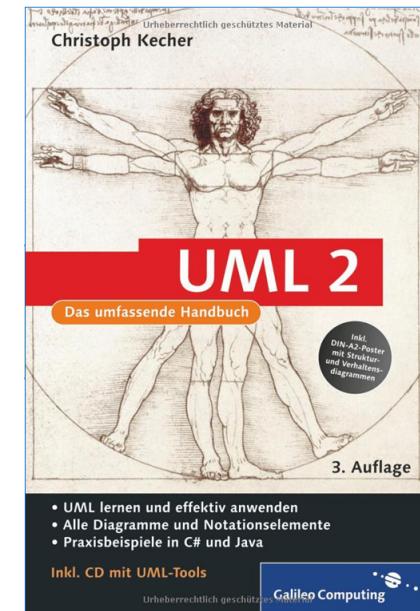
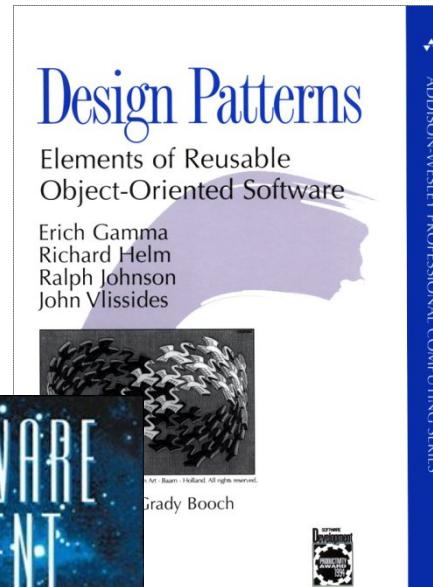
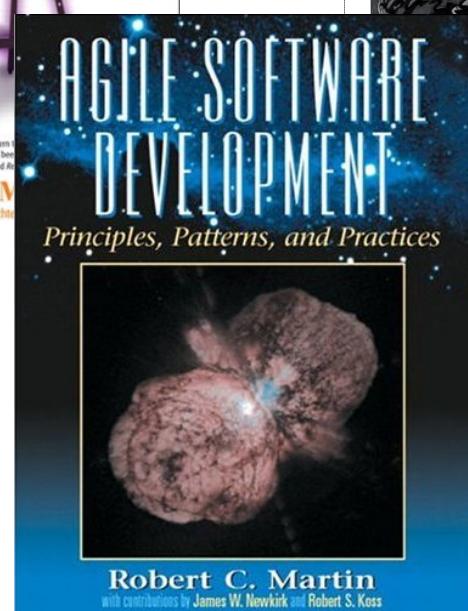
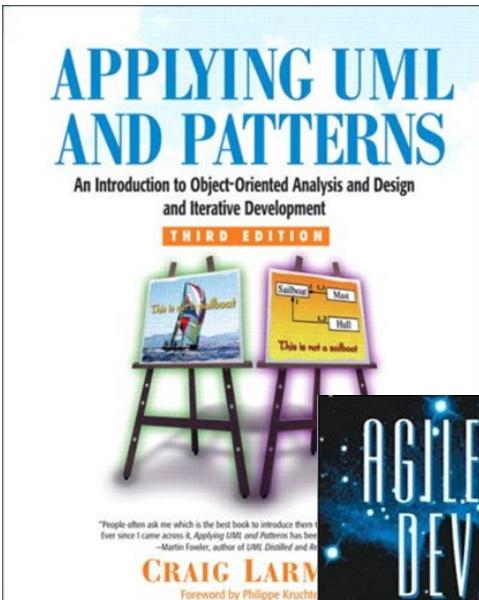
[https://www.fbi.h-da.de/fileadmin/personal/b.humm/
Publikationen/Siedersleben_-
Quasar_1_sd_m_Brosch_re_.pdf](https://www.fbi.h-da.de/fileadmin/personal/b.humm/Publikationen/Siedersleben_-Quasar_1_sd_m_Brosch_re_.pdf)

Quasar Teil 2

[https://www.fbi.h-da.de/fileadmin/personal/b.humm/
Publikationen/Siedersleben_-
Quasar_2_sd_m_Brosch_re_.pdf](https://www.fbi.h-da.de/fileadmin/personal/b.humm/Publikationen/Siedersleben_-Quasar_2_sd_m_Brosch_re_.pdf)



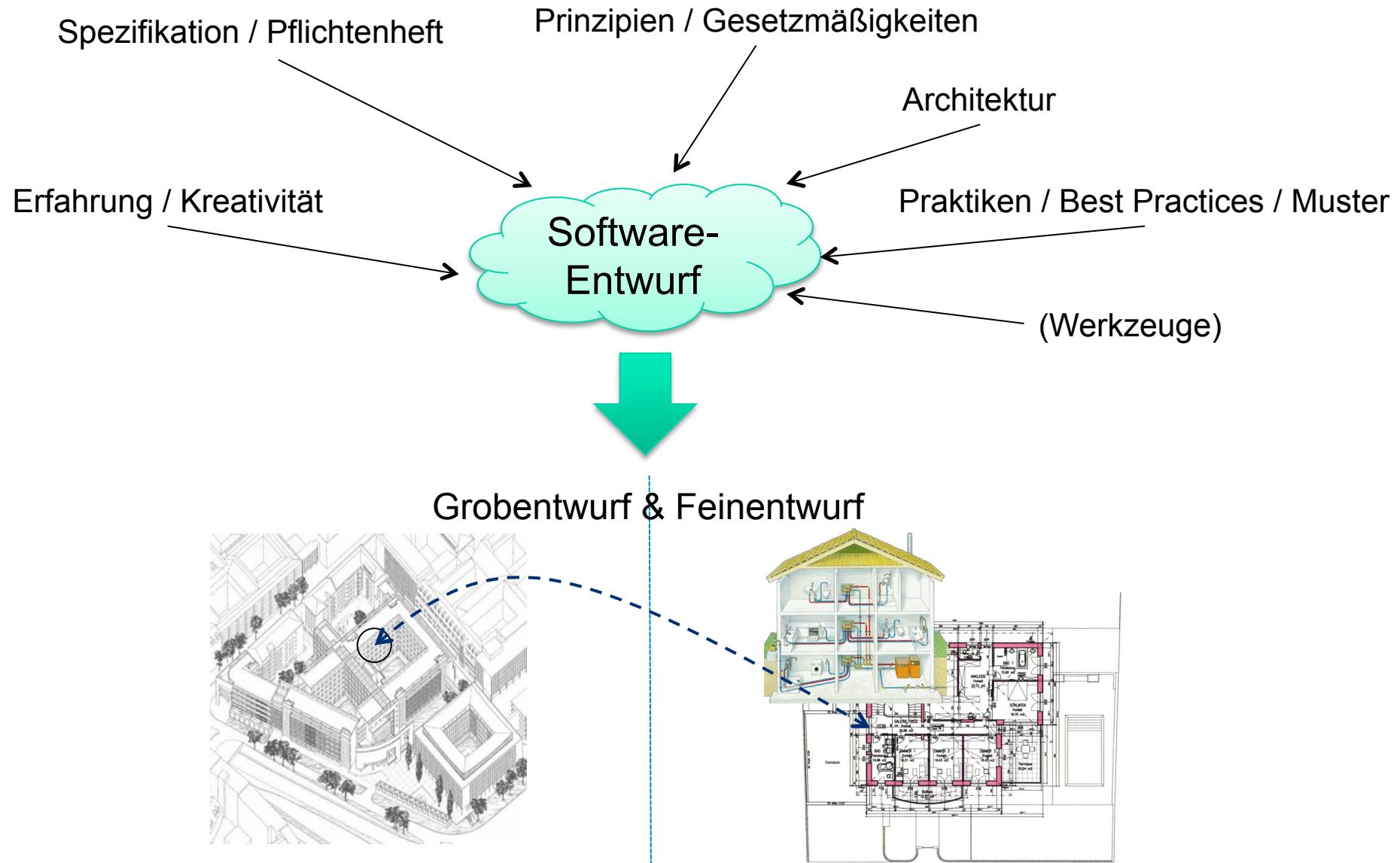
Weitere Literaturempfehlungen für den Entwurf

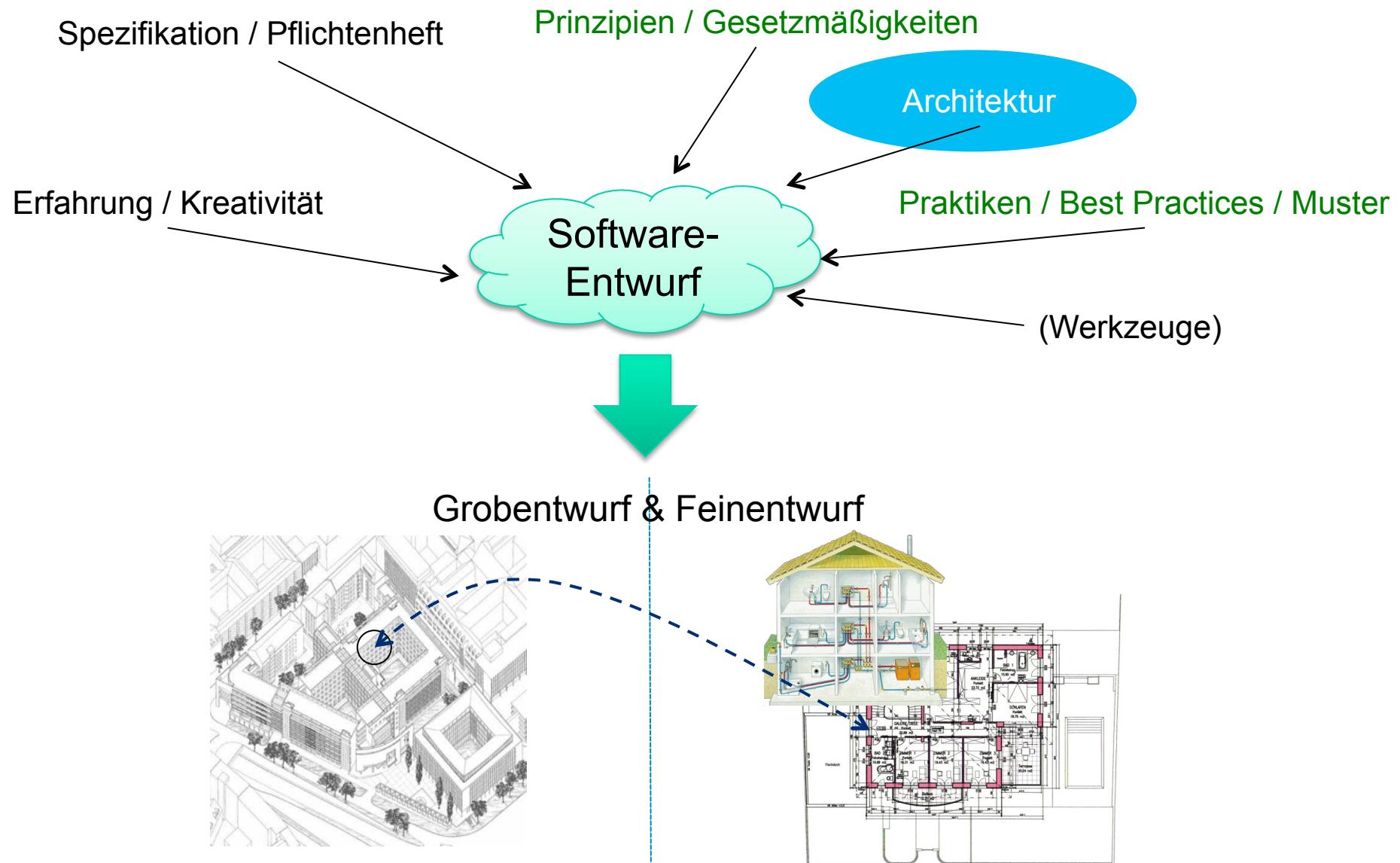




Aspekte eines Entwurfs?





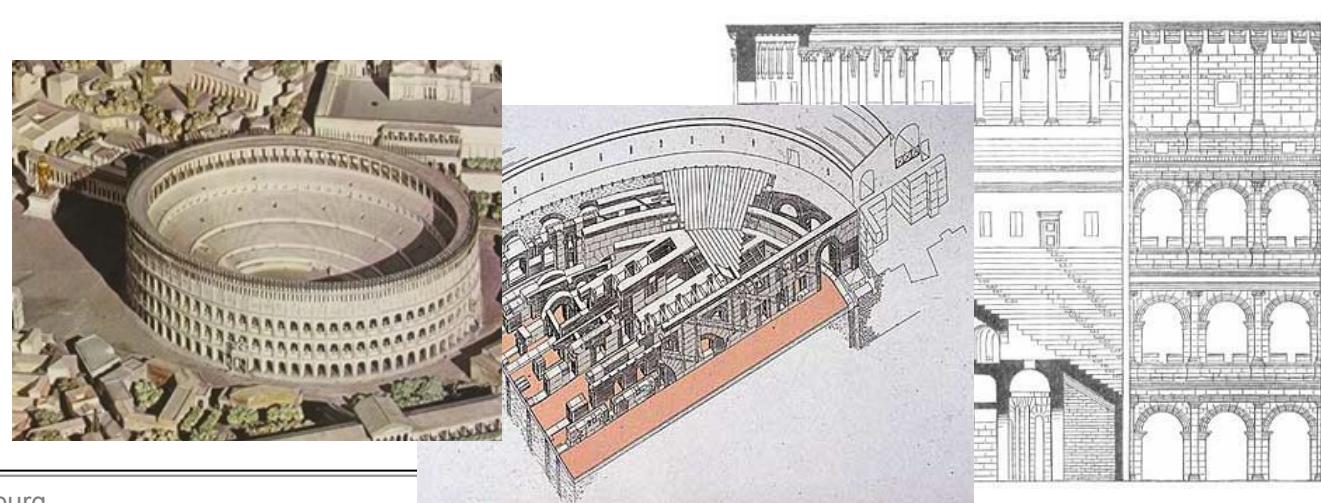




Architektur

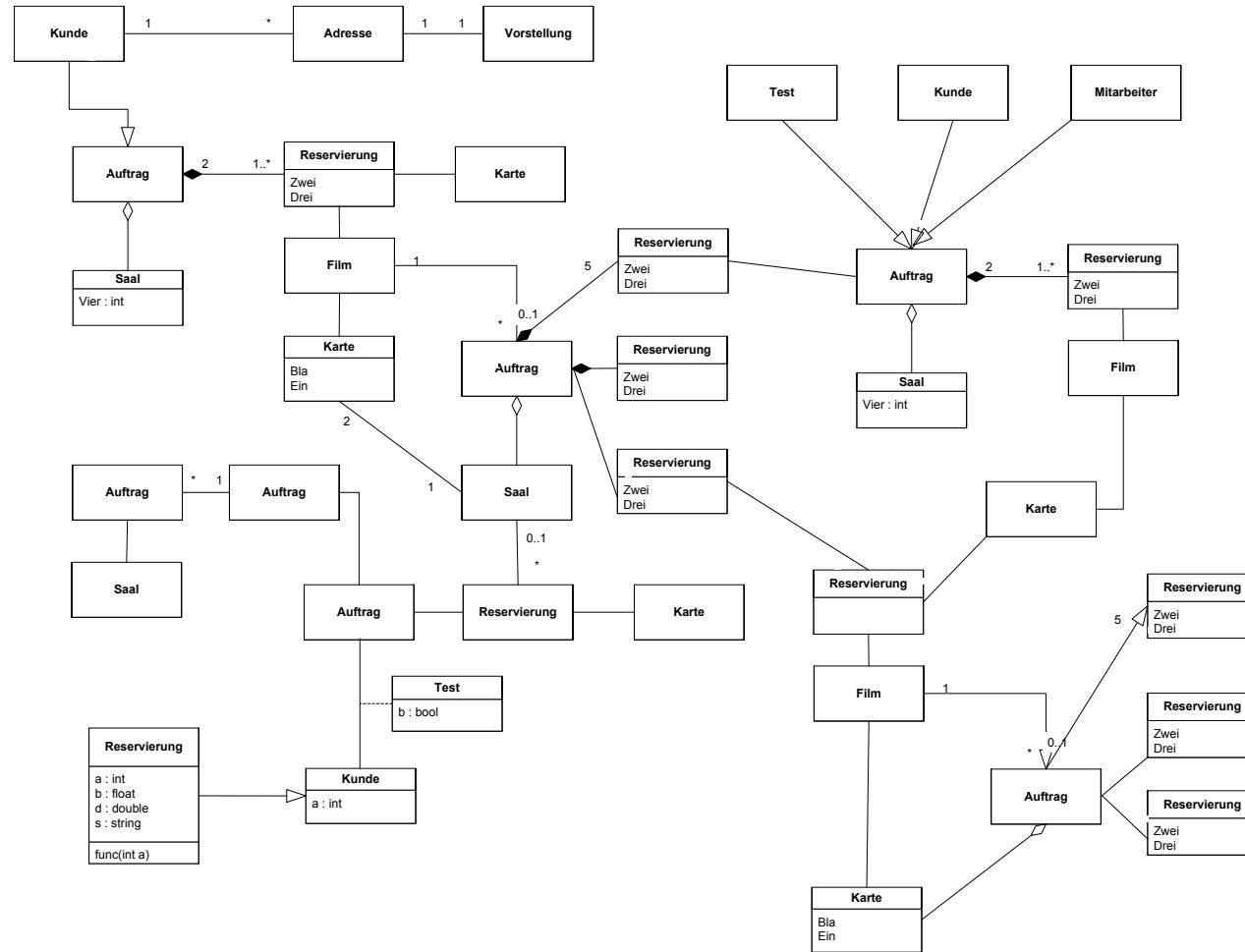
„Software-Architektur: Das ist die Königsdisziplin des Software-Engineering.“ (aus: Siedersleben, Software Architektur)

- Aber was ist eine „Software-Architektur“?
- Wie gelangt man zu einer guten Architektur?
- Und was bedeutet überhaupt „gut“?





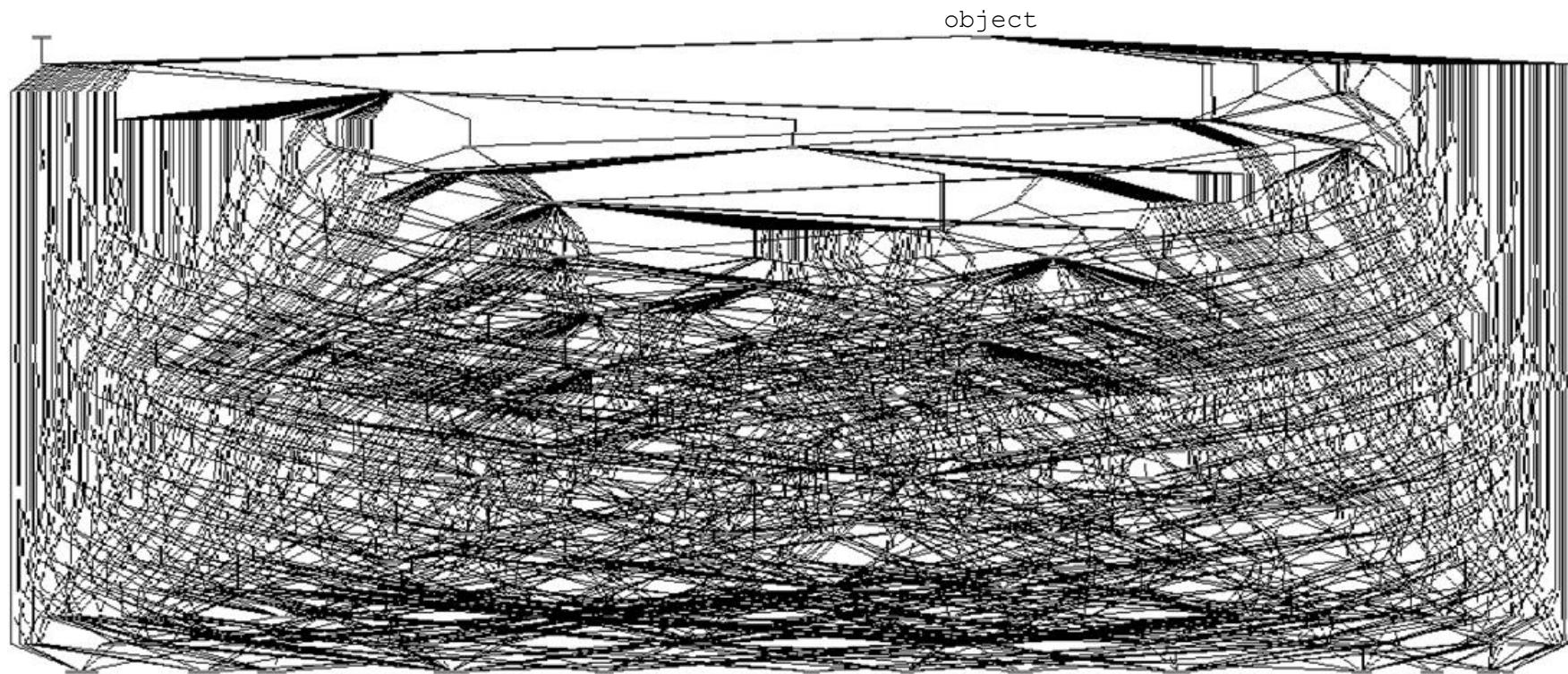
Ist dies eine gute Architektur?





Ist dies eine gute Architektur?

(Abhängigkeitsgraph im OO-Betriebssystem Taligent)



Erich Gamma, "100 OO Frameworks, Pitfalls and Lessons Learned", 1997



Entwurf

- Es gibt nicht „die“ Herangehensweise an den Entwurf
- Analogie zum Hausbau
 - ein Haus kann aus Holz, Stein, Beton, ... gebaut sein
 - es kann flach, hoch, rund, eckig, ... sein
 - ...
- Je nach Art der Software und Entwurfsstrategie spielen verschiedene Überlegungen eine Rolle
 - z. B. Informationssystem vs. Embedded System vs. Multimedia-Streaming-Anwendung vs. ...
 - Weitere Einflussfaktoren: Größe, nichtfunktionale Anforderungen, ...
- Wir konzentrieren uns auf Informationssysteme



Bedeutung des Entwurfs

- Ein komplexes System muss **in überschaubare Einheiten gegliedert** werden
 - Facetten und Wechselwirkungen sonst nicht offensichtlich
 - Oft hilft erst der Entwurf zum notwendigen Verständnis einiger Anforderungen
 - Ein Mensch kann nur +/- 7 Gegenstände ohne weiteres erfassen

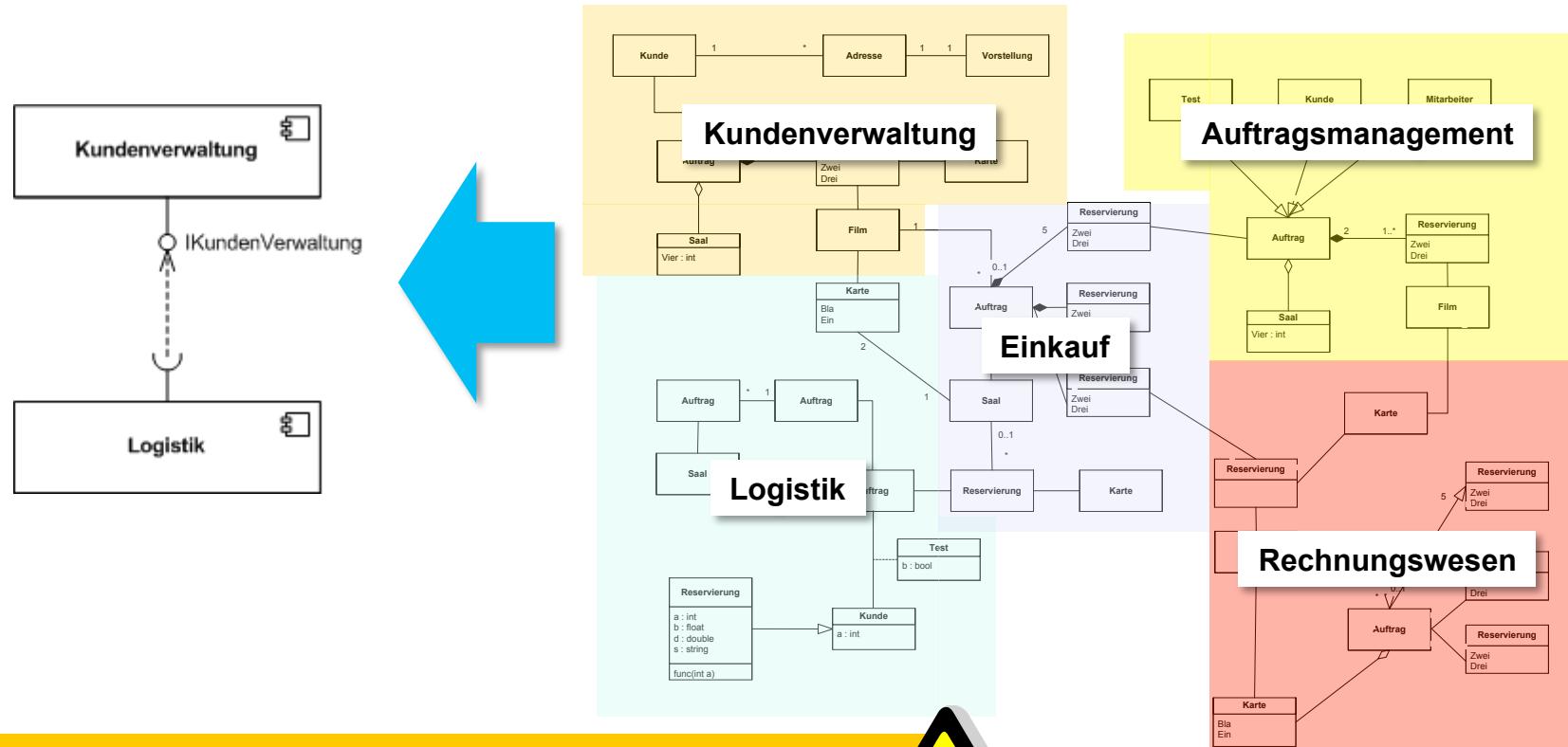
Die Software-Struktur hat großen
Einfluss auf die Wartbarkeit!





Entwurf

- Hier ist die Struktur und Wartung!



deshalb: Klassen sind als Einheit des
Software-Entwurfs viel zu klein!





Begriffe

*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, Clements, Kazman (2003)



Komponenten (1/7)

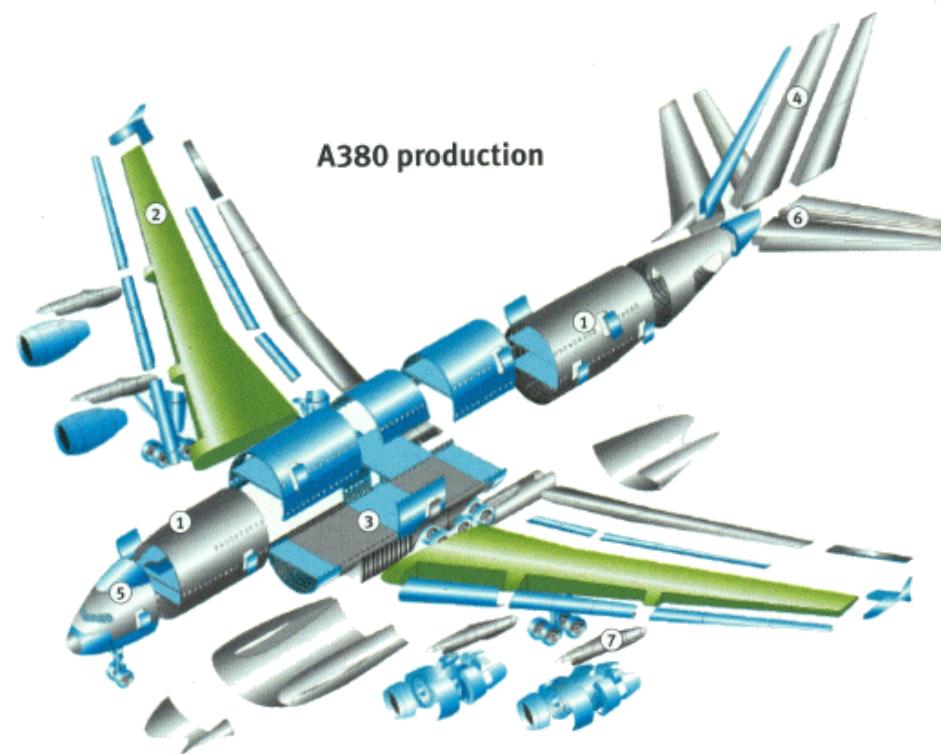
- Technische Systeme bestehen aus **Komponenten**





Komponenten (2/7)

- Technische Systeme bestehen aus **Komponenten**



① Hambourg (All.)
② Broughton (UK)
③ St. Nazaire/Nantes (France)

④ Stade (All.)
⑤ St. Nazaire/Méaulte (France)
⑥ Getafe/Puerto Real (Esp.)
⑦ Toulouse (France)

AIC manufacturing units
Risk-Sharing Partners
Risk-Sharing Partners sub-assemblies





Komponenten (3/7)

- Ein Flugzeug besteht aus vielen Bestandteilen:
 - diese Komponenten wirken sinnvoll **zusammen**
 - sind jedoch **weitgehend unabhängig**
- Ein Fahrwerk ist leicht **austauschbar**, denn es hat nur **einfache Schnittstellen** zu benachbarten Komponenten
- „Passende“ Schnittstellen reichen jedoch nicht aus
 - Es kommt auch auf deren „Dimensionierung“ (nichtfunktionale Eigenschaften) an!
- Heutige Programmiersprachen kennen das Konzept der Komponenten nicht oder nur in einer versteckten Form
 - DLLs / shared libraries (C und C++)
 - Pakete / jar-files (Java)
 - Assemblies (C#)



Komponenten (4/7)

- Es gibt nicht „die“ Definition des Begriffs „Komponente“ ...

A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

Szypersky (2002)

A software component is a coherent package of software implementations that

- has explicit and well-specified interfaces for services it provides;*
- has explicit and well-specified interfaces for services it expects; and*
- can be composed with other components, perhaps customizing some of their properties, without modifying the components themselves.*

D'Souza & Wills (1999)



Komponenten (4/7) – UML-Definition

A component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment.

A component defines its behavior in terms of provided and required interfaces. As such, a component serves as a type whose conformance is defined by these provided and required interfaces (encompassing both their static as well as dynamic semantics). One component may therefore be substituted by another only if the two are type conformant. [...]

UML 2.1.2 (2008)



Komponenten (5/7)

- **Merkmale einer Komponente nach Siedersleben**

1. Eine Komponente **exportiert** eine oder mehrere Schnittstellen, sie im Sinne eines Vertrages garantiert sind.
2. Sie **importiert** andere Schnittstellen. Sie ist erst lauffähig, wenn die importierten Schnittstellen zur Verfügung stehen.
3. Sie **versteckt** ihre Implementierung.
4. Sie eignet sich als Einheit der **Wiederverwendung**.
5. Komponenten können andere (Sub-)Komponenten enthalten. Man kann Komponenten somit über beliebig viele Stufen **komponieren**.
6. Die Komponente ist neben der Schnittstelle die **wesentliche Einheit** des Entwurfs, der Implementierung und damit der Planung.



Komponenten (6/7)

Die erste Frage des Entwurfs sollte lauten:
Aus welchen Komponenten besteht das System?

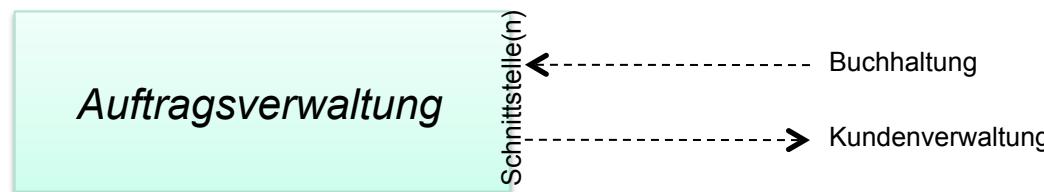


- Wie unterscheidet man sinnvolle und weniger sinnvolle Komponenten?
 - Geschick und Erfahrung des Architekten sind gefragt
 - Gestaltungsprinzipien durch Klassifikation (→ später)
- Die „Extreme“ vermeiden:
 - Klassen sind keine Komponenten!
 - Layer/Schichten (GUI, Applikation, Persistenz, ...) reichen nicht als „Komponenten“!
- Ein großes Softwaresystem ohne **überzeugende** Komponentenstruktur **kann nicht gut sein!**

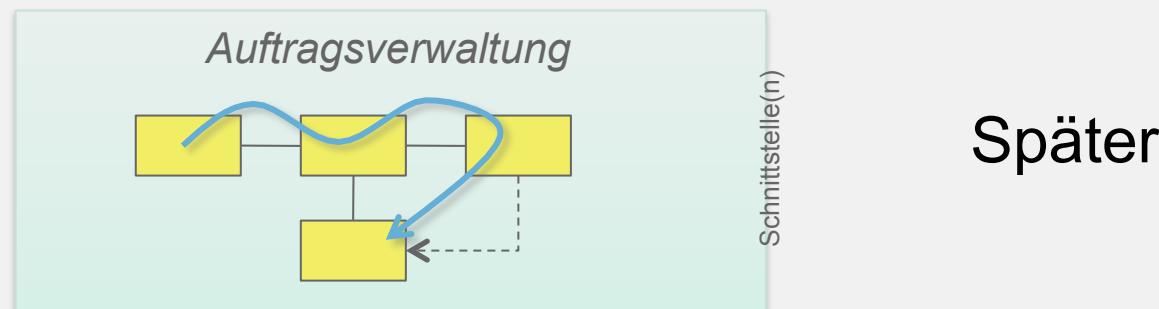


Komponenten (7/7)

- Komponenten haben eine **Außensicht** und eine **Innensicht**
- **Außensicht** 
 - Welche Schnittstellen definiert und verwendet die Komponente?



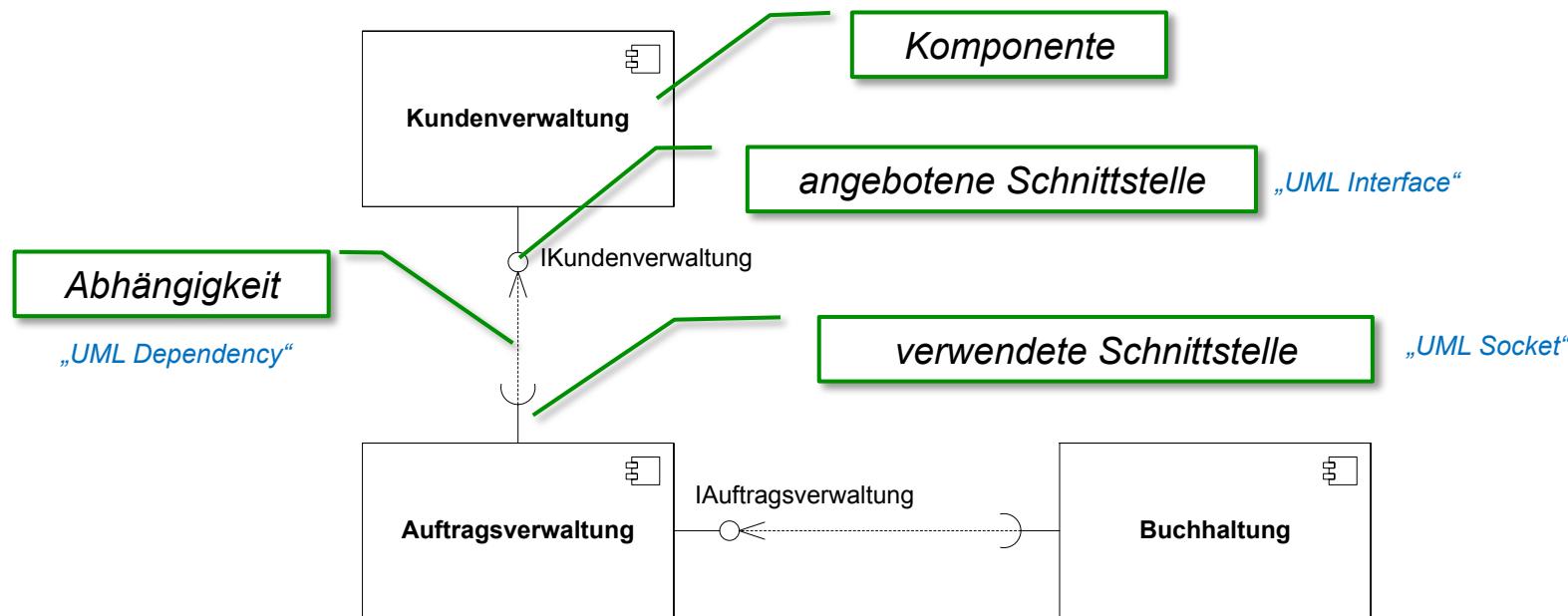
- **Innensicht** 
 - Wie ist der interne Aufbau der Komponente? Welche Anwendungsfälle und Entitäten enthält sie? Wie ist die (interne) Funktionalität strukturiert?





UML 2.x Komponentendiagramme

- Wie modelliert man die **Architektur** und **Außensicht** von Komponenten und Schnittstellen?
→ UML 2.x Komponentendiagramme [3]





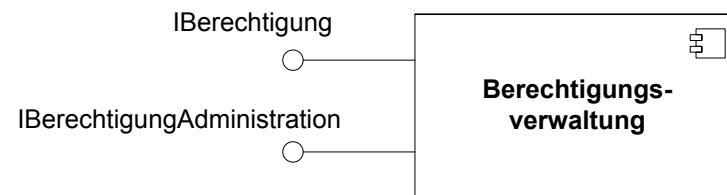
Beispiel 1: Berechtigungskomponente (1/5)

- Features
 - verwaltet Benutzer und Berechtigungsobjekte
 - Benutzern Zugriff auf Berechtigungsobjekte gestatten oder verbieten
- Die Komponente ist völlig **generisch**: ihre Funktionen lassen sich perfekt von den Funktionen der sie nutzenden Anwendung trennen!
- Welche Schnittstellen werden benötigt?
 1. Die **operative Schnittstelle** klärt die Frage
„Hat Benutzer x Zugriff auf Berechtigungsobjekt y?“
 - muss einfach zu benutzen sein
 - unzählige Aufrufer verwenden sie
 - muss performant und stabil sein
 2. Über die **Administrations-Schnittstelle** kann man: Benutzer eintragen/entfernen, Rechte erteilen/entziehen, ...
 - u. U. steckt dahinter ein komplexes Datenmodell mit Benutzern, Gruppen, Hierarchien, ...
 - Die operative Schnittstelle ist davon nicht berührt!



Beispiel 1: Berechtigungskomponente (2/5)

- Möglicher Entwurf:

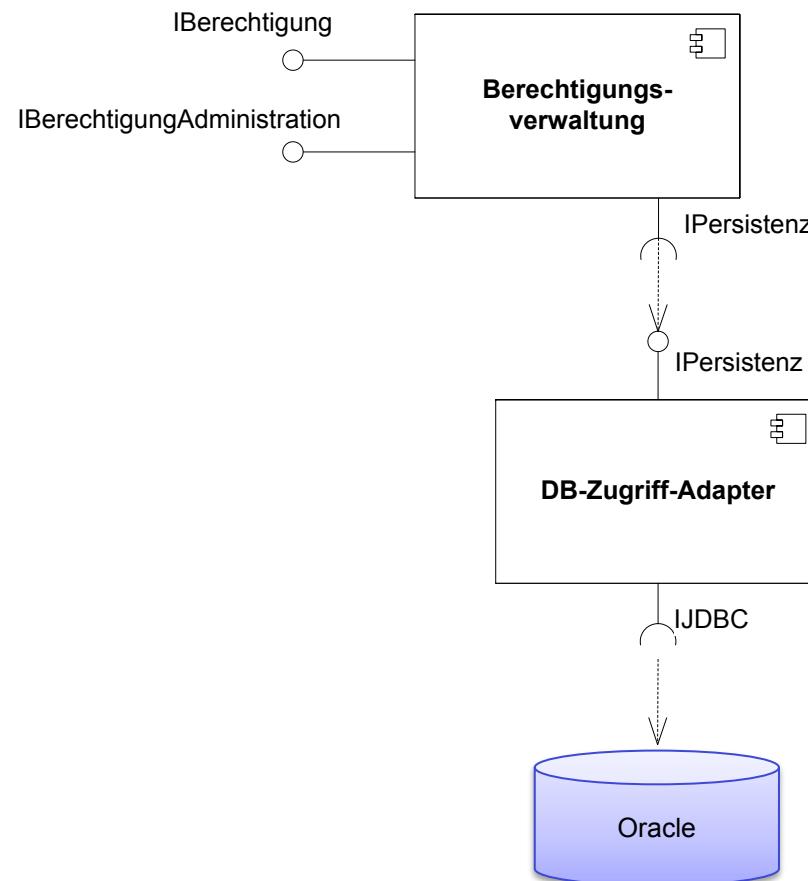


- Wie speichert die Komponente die Daten? (Zuordnungen, Gruppen, ...)



Beispiel 1: Berechtigungskomponente (3/5)

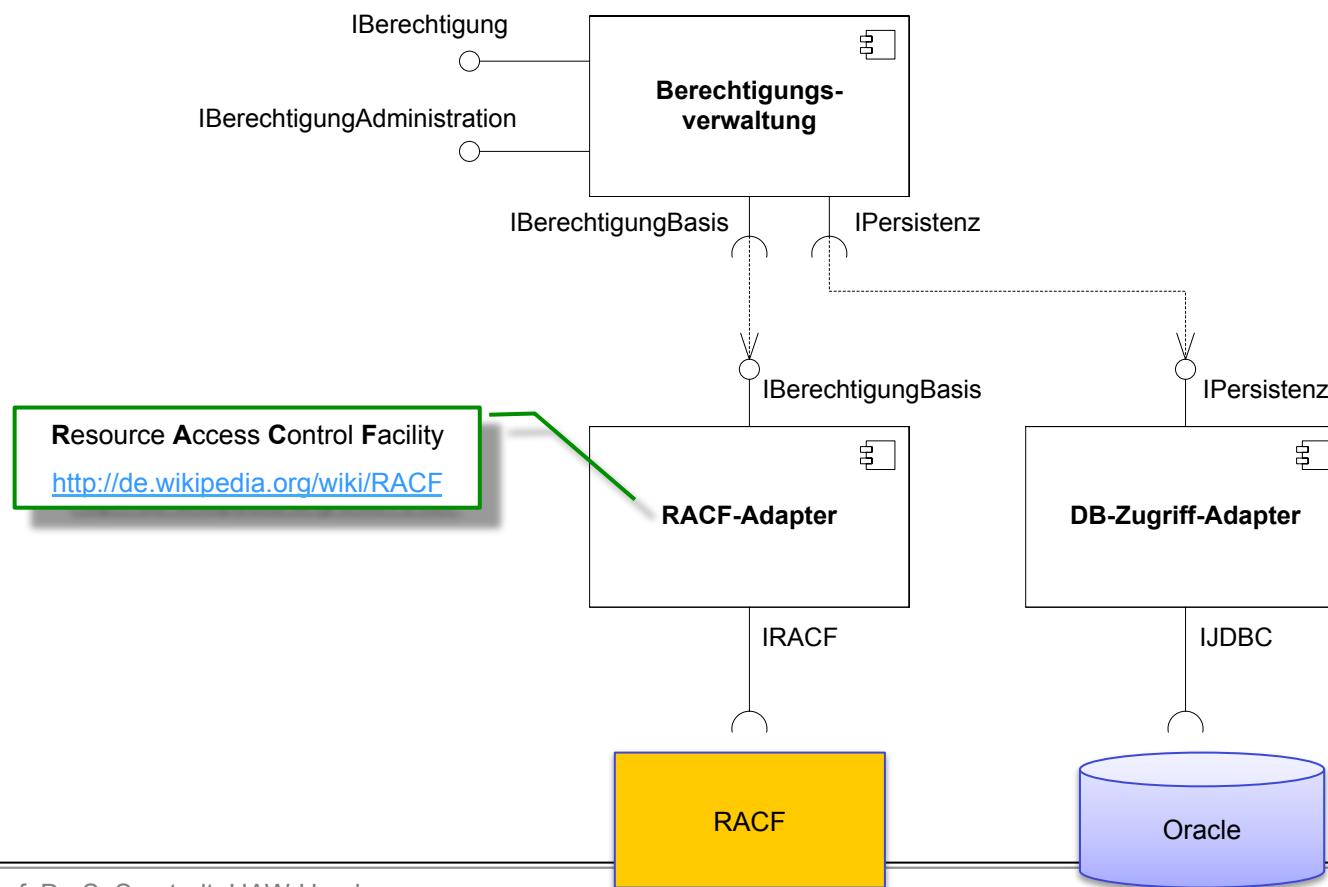
- Möglicher Entwurf mit Datenbank-Anbindung:





Beispiel 1: Berechtigungskomponente (4/5)

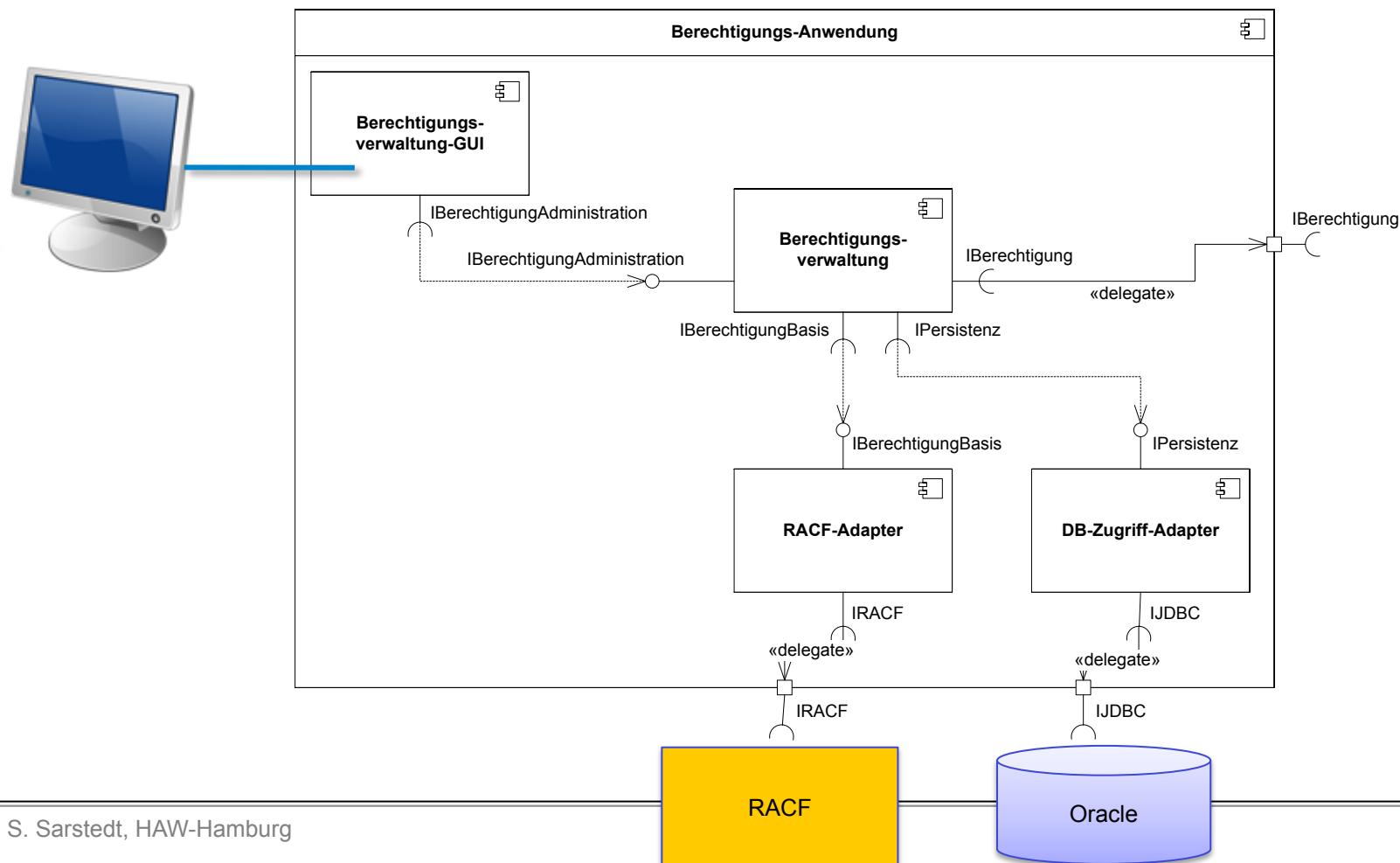
- Möglicher Entwurf mit Datenbank-Anbindung und zusätzlichem externem Berechtigungssystem:





Beispiel 1: Berechtigungskomponente (5/5)

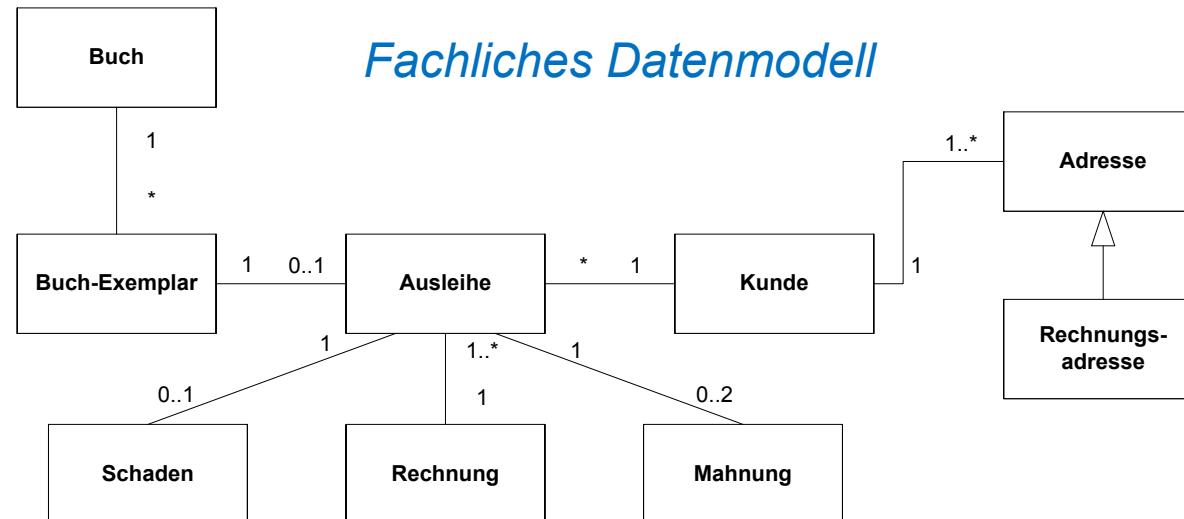
- Möglicher Entwurf mit Datenbank-Anbindung, RACF und Administrations-GUI:





Beispiel 2: Bibliotheksverwaltung (1/5)

- Features
 - Bücher verleihen
 - Titelkatalog
 - Bestandsführung (u. U. mehrere Exemplare pro Buch)
 - Kundenverwaltung
 - Gebühren / Mahngebühren



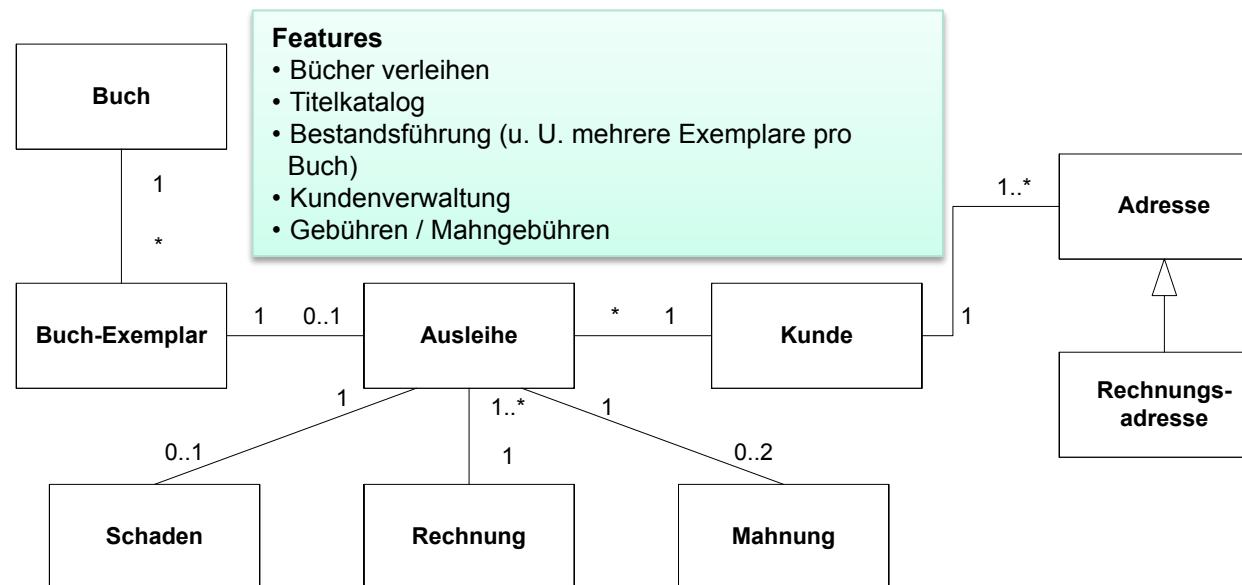


Gruppenübung

Schnittstellenspezifikation

Erstellen Sie eine fachliche Architektur für das Bibliothekssystem.

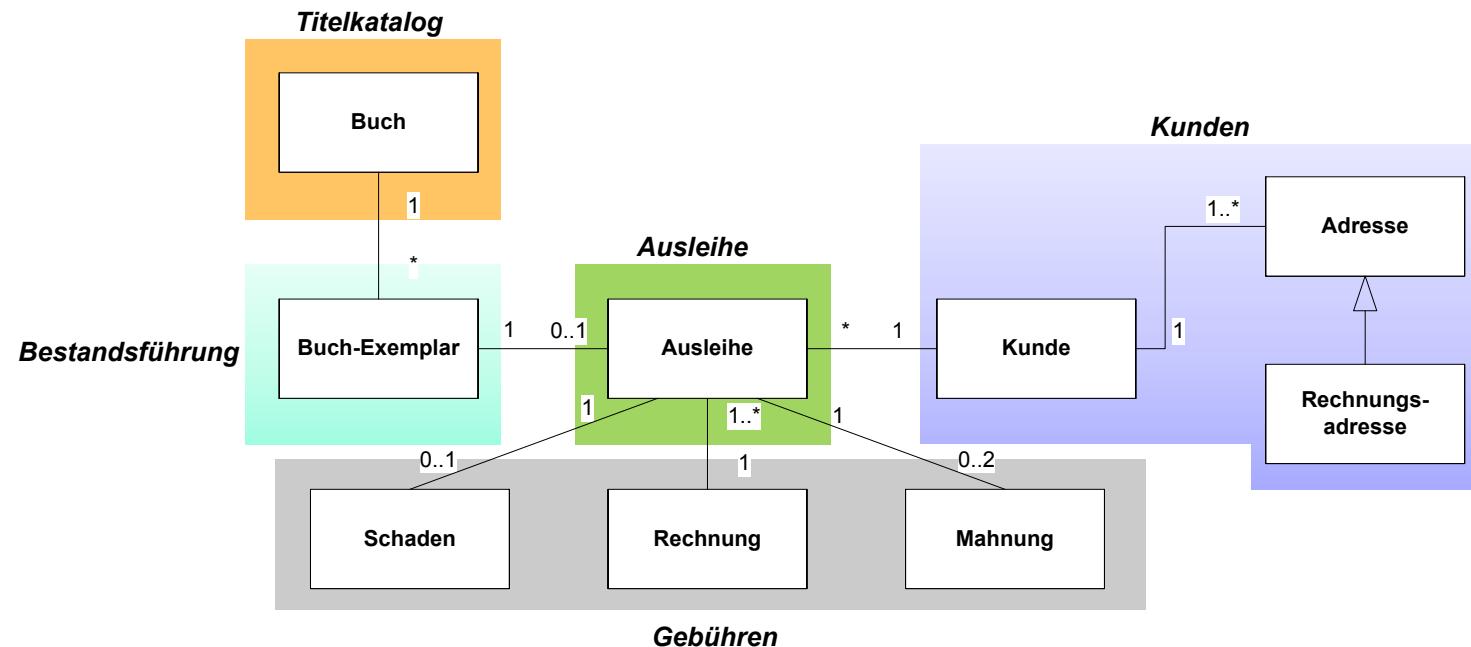
- Definieren Sie sinnvolle **Komponenten** und **Schnittstellen**
- Welche **Aufgaben/Verantwortlichkeiten** haben ihre Komponenten?
- Welche Operationen können Sie sich für Ihre Schnittstellen vorstellen? Geben Sie **Beispiele!**
- Zeichnen Sie ein **UML Komponentendiagramm**, das die Komponenten, Schnittstellen und Abhängigkeiten darstellt



25 Minuten

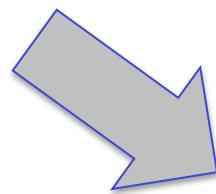
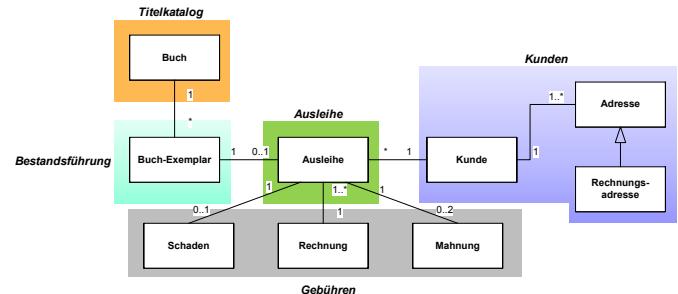


Beispiel 2: Bibliotheksverwaltung (2/5)



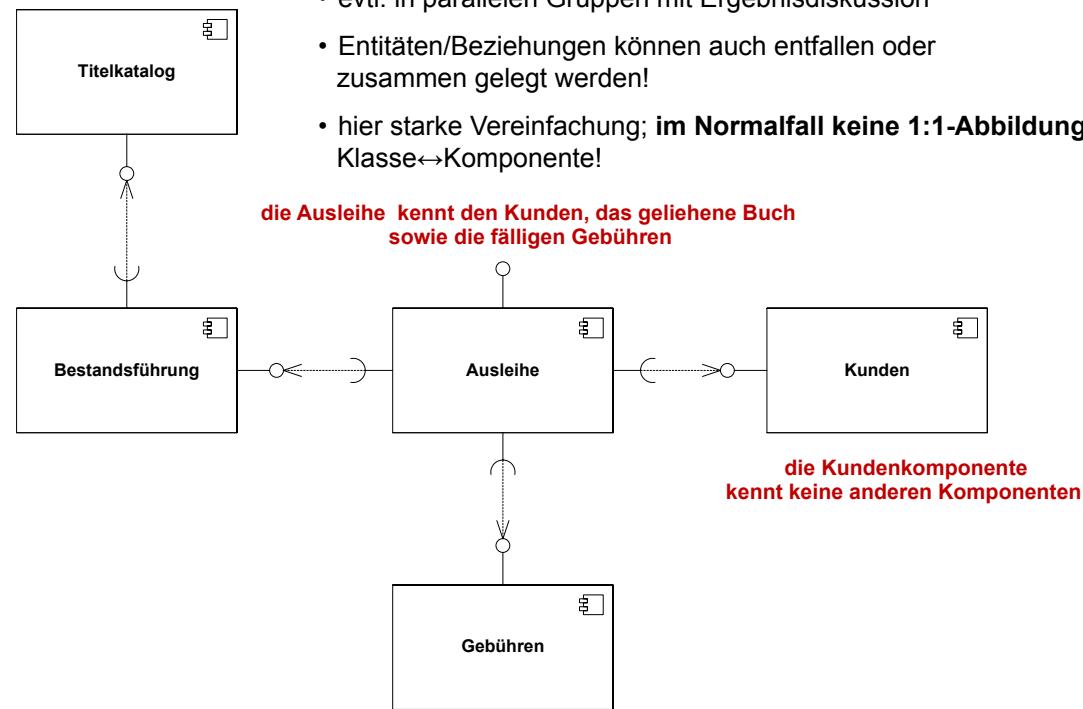


Beispiel 2: Bibliotheksverwaltung (3/5)



Wie kommen wir zu den Komponenten?

- Verschiedene fachliche Themen im Lastenheft/Pflichtenheft
- „gesunden Menschenverstand“
- Komponenten sollten in anderen Kontexten wiederverwendbar sein
- evtl. in parallelen Gruppen mit Ergebnisdiskussion
- Entitäten/Beziehungen können auch entfallen oder zusammen gelegt werden!
- hier starke Vereinfachung; **im Normalfall keine 1:1-Abbildung Klasse↔Komponente!**





Beispiel 2: Bibliotheksverwaltung (4/5)

- **Außensicht** der Komponente „Gebühren“

The diagram illustrates the external view of the 'Gebühren' component. It shows the **IGebuehrenMgmt** interface and its implementation **Gebühren**.

IGebuehrenMgmt Interface:

```
interface IGeuehrenMgmt
{
    /**
     * Liefert die am Tag 'tag' gültige Leigebuehr
     * @param tag: Tag, für den die Leihgebuehr
     *            geliefert werden soll
     * @return: Leihgebühr für den angegebenen Tag
     */
    Geld GetLeihgebuehr(const Datum tag);

    /**
     * ...
     * @pre: von < bis
     */
    List<Geld> GetLeihgebuehren(const Datum von,
                                 const Datum bis);

    /**
     * ...
     * @pre: tag > HEUTE
     */
    void SetLeihgebuehr(const Datum tag,
                        const Geld neueGebuehr);
}
```

Implementation Class: Gebühren

Annotations:

- Machen Sie im Namen kenntlich, dass es sich um eine Schnittstelle handelt!
→ **IGebuehrenMgmt**
- Abfrage (query)
- Kommando (command)
- Wie kommt man zu diesen Schnittstellen-Operationen?
→ später



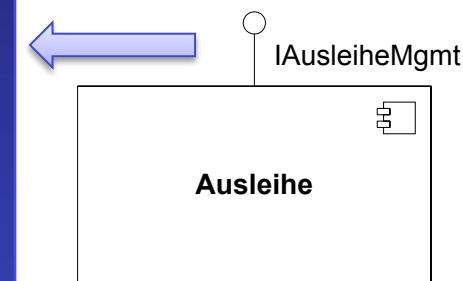
Beispiel 2: Bibliotheksverwaltung (5/5)

- **Außensicht** der Komponente „Ausleihe“

```
interface IAusleiheMgmt
{
    /**
     * Legt für jedes übergebene Exemplar eine neue Ausleihe
     * für den Kunden an
     * @param entleiher: Kunde, der die Exemplare ausleiht
     * @param exemplare: Liste der zu leihenden Buch-Exemplare
     * @pre  exemplare != null && exemplare.size > 0
     * @return: die erzeugten Ausleihen; fehlt für ein Exemplar
     *          die entsprechende Ausleihe, wurde das Exemplar
     *          in der Zwischenzeit bereits anderweitig
     *          verliehen
    */
    List<IAusleihe> SucheAusleihen(const IBuch buch);

    List<IAusleihe> LeiheBuecher(const IKunde entleiher,
                                const List<IExemplar>
                                exemplare);
}

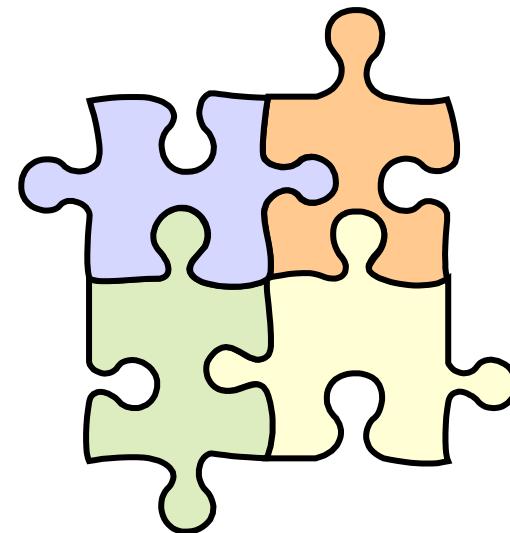
..."
```



**Wieso IAusleihe, IBuch, IKunde, ... in den Schnittstellen?
→ später**



Komponenten verbinden





Konfiguration von Komponenten

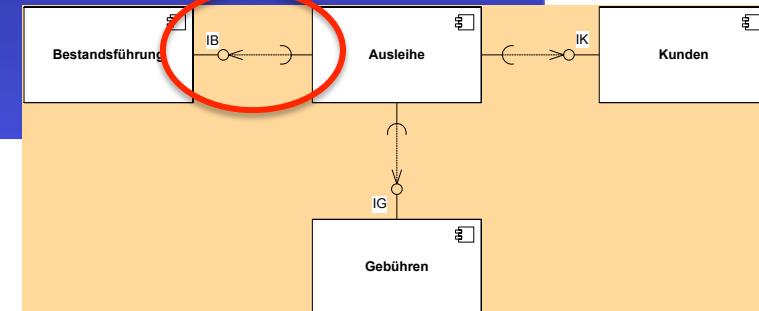
- Die Separierung der Komponenten-Konfiguration bietet zahlreiche Vorteile
 - Komponenten sind in anderen Kontexten **wiederverwendbar**
 - Komponenten-Implementierungen sind **austauschbar**, ohne in den Quellcode eingreifen zu müssen
 - Komponenten sind leicht **testbar** durch sog. Mock-Objekte; diese „tun so“, als ob sie eine richtige Komponentenimplementierung wären
- Nicht nur fachliche Komponenten „stöpselt“ man zusammen, sondern hier erfolgt auch die Verbindung von fachlichen mit den technischen Bestandteilen des Systems
- Dependency Injection kann in vielen Systemteilen angewendet werden, so z.B. auch zur Konfiguration des inneren Aufbaus einer Komponente („Innensicht“, siehe später)
- Beim Start einer Applikation wird zunächst die Anwendung „konstruiert“, erst dann fängt sie an zu arbeiten (BuildComponents()->Run())



Konfiguration – Komponenten verbinden (1/2)

- Wer verbindet die Komponenten zu einem lauffähigen Ganzen?

```
class AusleiheKomponente {  
    private:  
        IB refIB;  
        ...  
  
    public AusleiheKomponente() {  
        this.refIB = new BestandsführungKomponente();  
        ...  
    }  
}
```



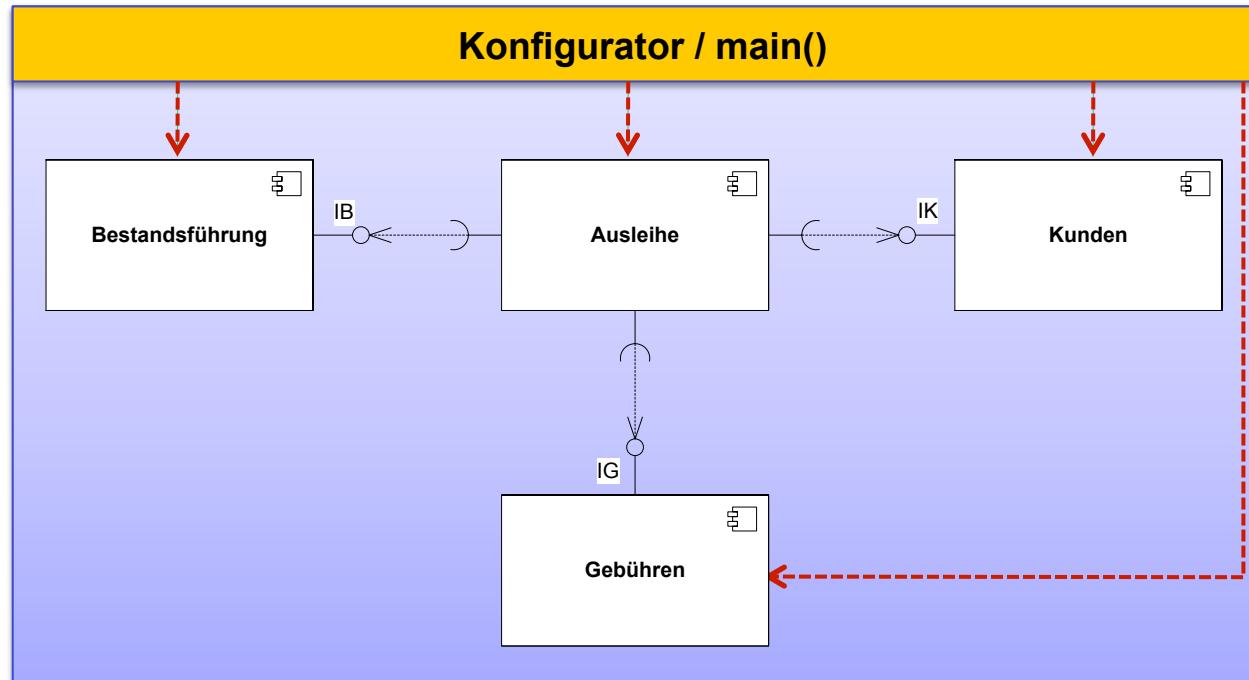
(im Folgenden ignorieren wir aus Übersichtgründen den Titelkatalog)

Was halten Sie davon?



Konfiguration – Komponenten verbinden (2/2)

- Bessere Lösung: Konfiguration
 - dies erledigt z.B. die `main()`- oder eine `BuildComponents()`-Methode





Konfiguration – Dependency Injection (1/4)

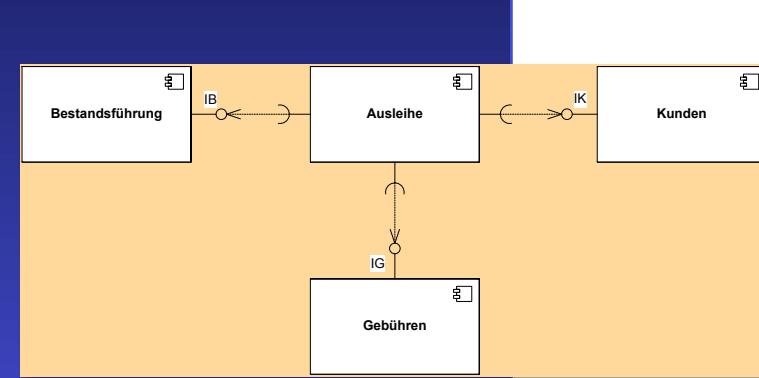
- Für die Konfiguration von Verbindungen gibt es ein Entwurfsmuster:
Dependency Injection
 - <http://www.martinfowler.com/articles/injection.html>
- Bei der **Dependency Injection** wird die Abhängigkeit von „außen“ in die Komponente eingebracht
 - dies verhindert die direkte Kopplung an eine konkrete Implementierung durch die Komponente selbst
 - ist eine Ausprägung des IoC-Prinzips („Inversion of Control“ – auch Hollywood-Prinzip genannt: „*Don't call us, we call you*“)
- Es gibt mehrere Varianten:
 1. Constructor Injection
 2. Setter Injection
 3. Interface Injection (hier nicht betrachtet, da nicht gängig)



Konfiguration – Dependency Injection (2/4)

Variante 1: Konfiguration über den Konstruktor

```
class KundenK implements IK {...}  
class GebührenK implements IG {...}  
class BestandsführungK implements IB {...}  
class AusleiheK {  
    private:  
        IB refIB;  
        IH refIG;  
        IK refIK;  
  
        public AusleiheK(IB ib, IG ig, IK k) { ... }  
}
```



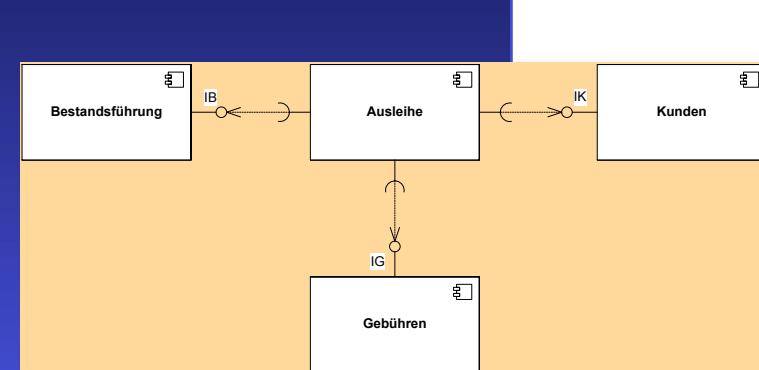
```
IK k = new KundenK();  
IG g = new GebührenK();  
IB b = new BestandsführungK();  
Ausleihe a = new AusleiheK(b, g, k);
```



Konfiguration – Dependency Injection (3/4)

Variante 2: Konfiguration über Setter

```
class AusleiheK {  
    private:  
        IB refIB;  
        ...  
  
    public void bindIB(IB ib) {  
        this.refIB = ib;  
    }  
}
```



```
IK k = new Kunden();  
IG g = new Gebuehren();  
IB b = new Bestandsfuehrung();  
  
AusleiheK a = new Ausleihe();  
a.bindIB(b); ←
```



Konfiguration – Dependency Injection (4/4)

- Konfiguration mittels XML-Datei (Java-Spring-IoC-Framework)

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//Spring//DTD BEAN 2.0//EN"
 "http://www.springframework.org/dtd/spring-beans-2.0.dtd">

<beans>

    <bean name="datasource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
        <property name="driverClassName" value="org.hsqldb.jdbcDriver" />
        <property name="url" value="jdbc:hsqldb:file:springbuchhsqldb" />
        <property name="username" value="sa" />
        <property name="password" value="" />
    </bean>

    <bean name="kundeDAO" class="springjdbcdao.kundeDAO">
        <property name="datasource" ref="datasource" />
    </bean>

    <bean name="bestellungDAO" class="springjdbcdao.BestellungDAO">
        <property name="datasource" ref="datasource" />
        <property name="kundeDAO" ref="kundeDAO" />
        <property name="wareDAO" ref="wareDAO" />
    </bean>

    <bean name="bestellung" class="businessprocess.BestellungBusinessProcess">
        <property name="bestellungDAO" ref="bestellungDAO" />
        <property name="kundeDAO" ref="kundeDAO" />
        <property name="wareDAO" ref="wareDAO" />
    </bean>

</beans>
```



Konfiguration – Service Locator

- **Service Locator** ist ein anderes Entwurfsmuster zur Reduzierung der Kopplung
- Idee: Eine Art **Verzeichnis** für alle benutzbaren Objekte
- Die Komponente sucht sich ihre Verbindungen mit Hilfe des Service Locator

```
void main() {  
    ServiceLocator.RegisterComponent(new Bestandsführung(), "IB");  
}  
  
class Ausleihek {  
    ...  
    public Ausleihek() {  
        this.refIB = ServiceLocator.Lookup("IB");  
    }  
    ...  
}
```

- Nachteil: Abhängigkeit vom Service Locator, schwerer konfigurierbar