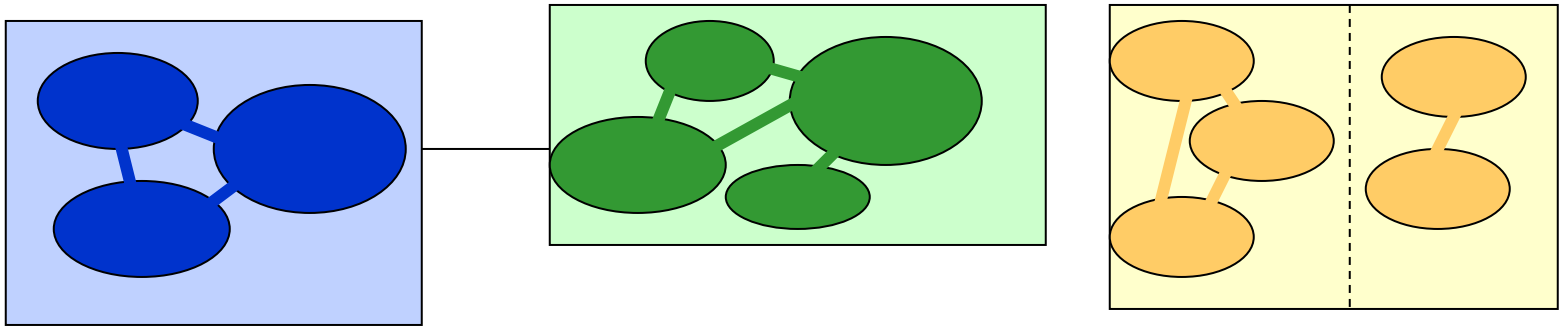
Von der Technik zum Entwurf

- Prinzipien des Entwurfs
- Kapselung der Persistenz
- Referenzarchitektur Web-Anwendungen
- Schichtentrennung



1. Prinzipien des Software-Entwurfs

Software-Entwurf

- Software-Entwurf ist **Strukturenwurf** auf verschiedenen Ebenen:

- Architekturentwurf
  - Strukturentwurf für das Gesamtsystem
- Komponentenentwurf
  - Komponente ist Gliederungseinheit des Systems
- Modul – oder Klassenentwurf
  - Modul ist Gliederungseinheit der Programmierung

- Ziel ist die Beherrschung der Komplexität, also Einfachheit:
  - "Wenn du es nicht in fünf Minuten erklären kannst, hast du es nicht verstanden oder es funktioniert nicht." *Rechtin*

- Erste Schritt: Vereinfachung durch Zerlegung

Grundformen der Zerlegung

- Horizontal:
  - "in Scheiben schneiden"
  - Schichtung
  - Abstraktionsebenen
  - jede Schicht baut auf der darunter liegenden auf
- Vertikal:
  - "in Stücke schneiden"
  - jedes Teil übernimmt eine benennbare fachliche oder technische Funktion
- Kapselung:
  - Teile (Komponenten) als Black Box betrachten
  - kommunizieren mit der Umgebung über klar definierte Schnittstellen

Entwurfsprinzipien

- Was ist ein guter Entwurf?

- Zunächst brauchen wir Ziele und Kriterien

→ Entwurfsprinzipien

- Unterschiedliche Prinzipien auf verschiedenen Ebenen:
  - Prinzipien des Architekturentwurfs
  - Prinzipien des Komponentenentwurfs
  - Prinzipien des Klassenentwurfs

Heute nur:
Hauptprinzip des Architektur-Entwurfs

Eine gute Software-Architektur verfügt über:

- Lose Kopplung
  - leichter Austausch von Komponenten
- Starke Kohäsion
  - "Unteilbarkeit" der Komponenten

... und ist verstehbar ☺

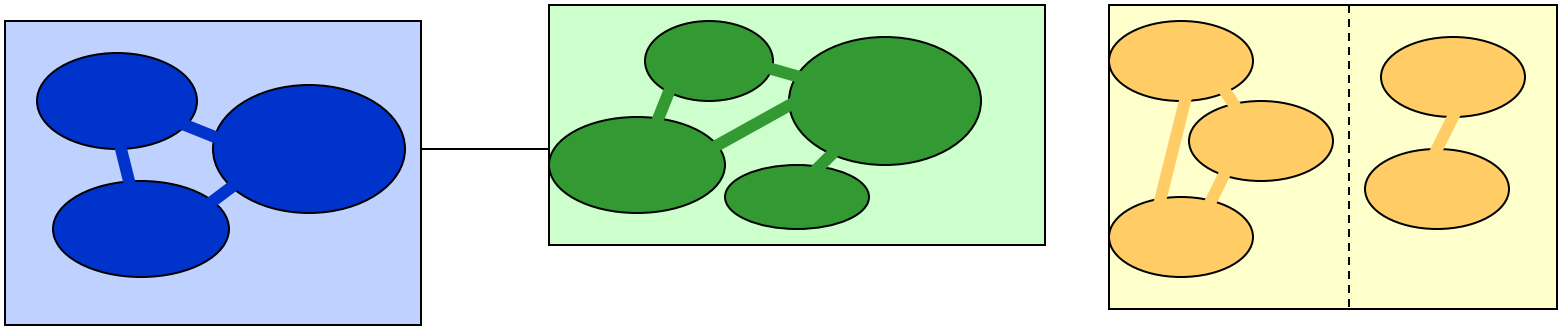
- Referenzarchitekturen
- Best Practices

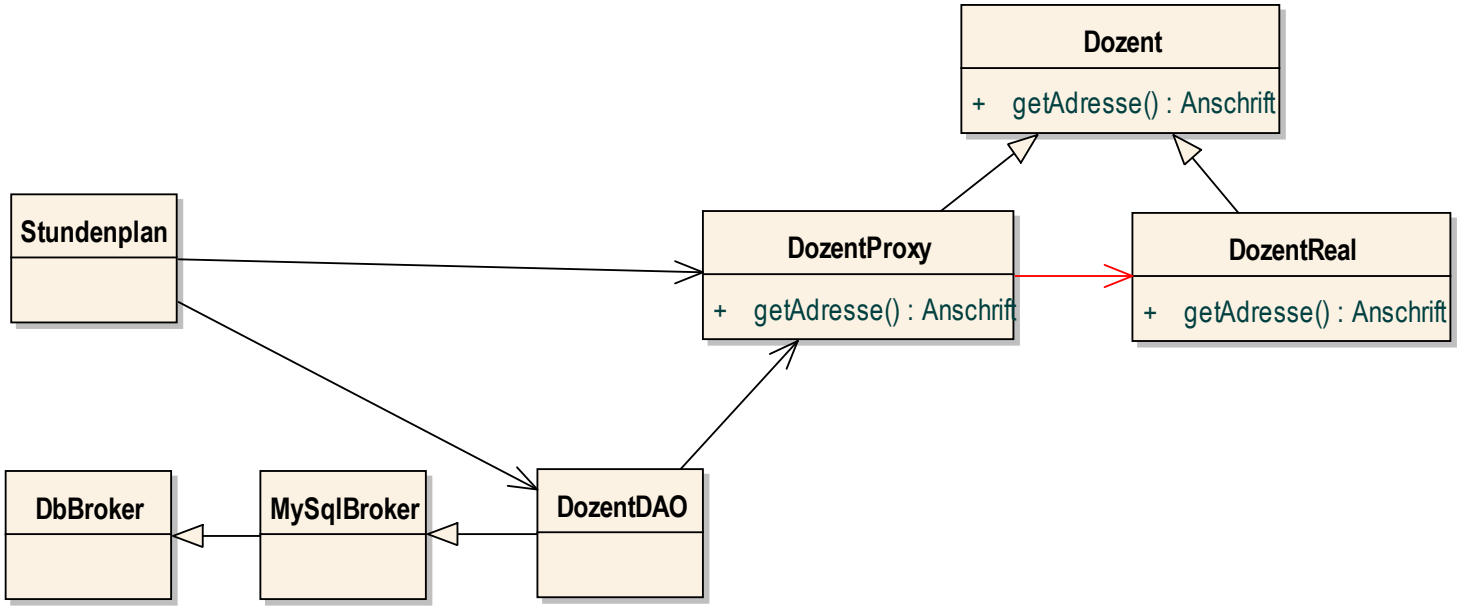
Lose Kopplung und starke Kohäsion

- Optimierung von Abhängigkeiten:
  - Flexibilität (Veränderungen können lokal vorgenommen werden)
  - Stabilität (Änderungen wirken sich nur lokal aus)

- Kopplung
  - Abhängigkeit zwischen Moduln
  - sollte möglichst gering sein: Austauschbarkeit, Veränderbarkeit

- Kohäsion
  - Abhängigkeit innerhalb eines Moduls, innerer (logischer) Zusammenhang
  - sollte stark sein → sonst Modul teilen





2 Kapselung der Persistenz

Transparente Persistenz

- Ziel ist eine transparente Persistenz
  - Die Anwendung soll mit Objekten arbeiten
  - und möglichst wenig von der Persistenz merken

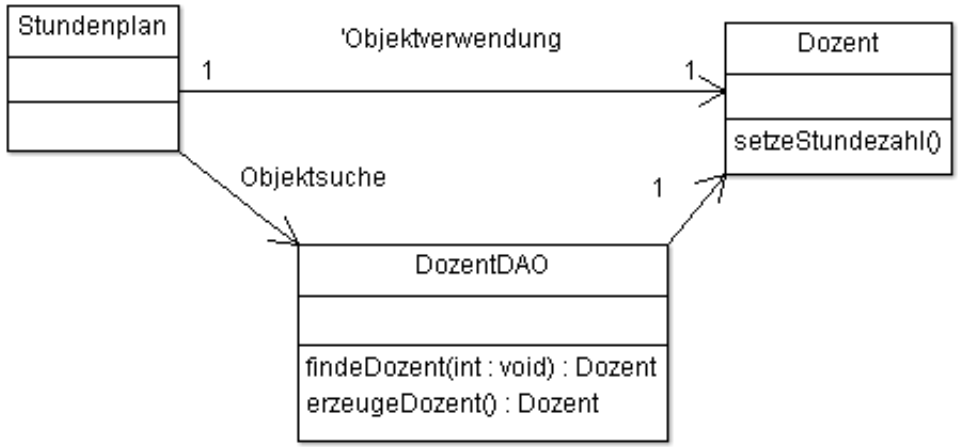
- Das bedeutet kapseln:
  - Zugriffswissen kapseln
  - Zugriffs - und Klassenwissen trennen

- 2 Ansätze:
  - Fachklassen sind "POJOS"
    - DAO- und Broker-Muster für kapseln die Persistenz
  - Fachklassen sind "Active Records"
    - Fachklassen erweitern Zugriffsklassen (Active-Record-Muster)

DAO – Data Access Objects (Persistierungsobjekte)

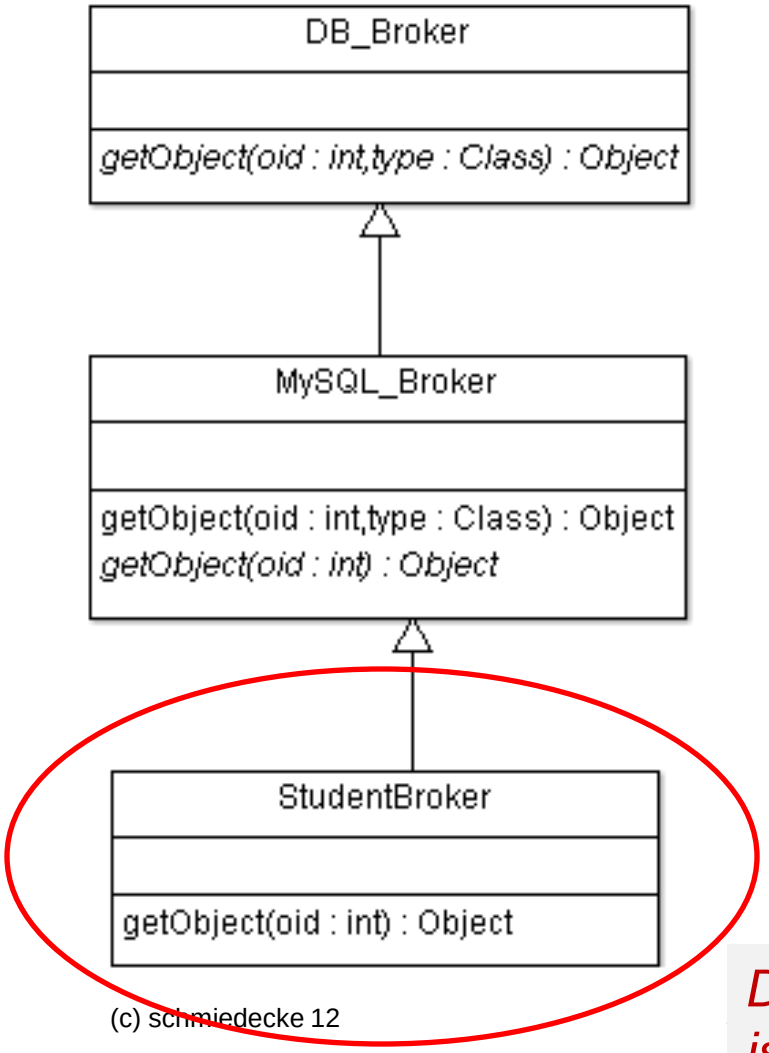
- *Annahme aus der Analysephase:*
  - *Klasse verwaltet die Menge ihrer Objekte*
  - *würde jetzt bedeuten, dass jede Klasse die DB-Details kennt*
  - *nicht transparent*
  - *DB-Wechsel sehr aufwändig*

- Data Access Object – der Objekte-Baumeister:
  - Klassenspezifisches Zugriffsobjekt
  - kapselt die DB-Details
  - liefert und persistiert die Anwendungsobjekte



- Nachteile
  - Für jede Klasse ein DAO
  - Viel redundanter Code
  - Viel Aufwand bei DB-Wechsel

Das Broker-Muster



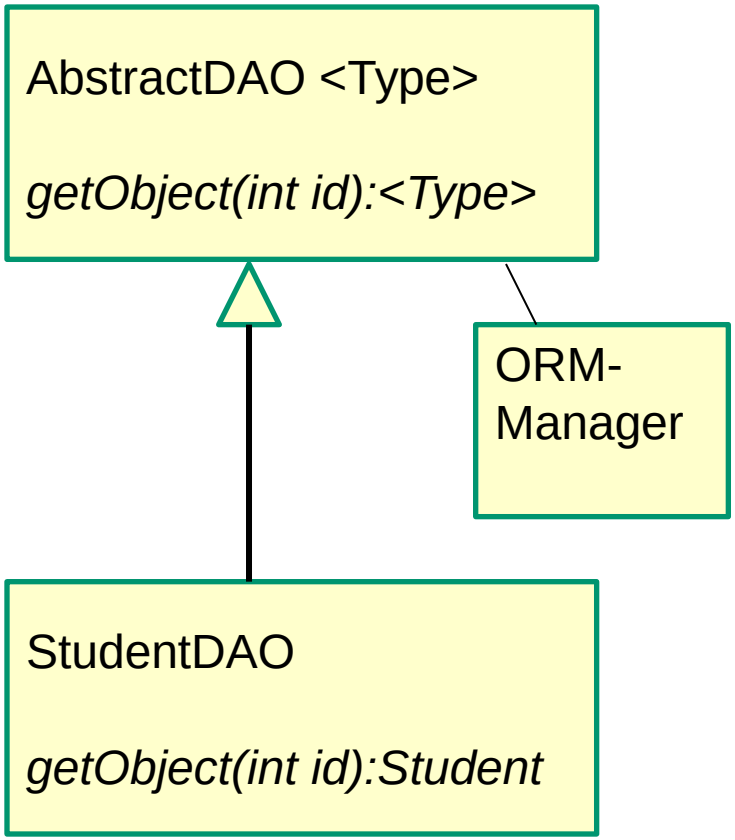
- kennt und verwaltet den Objekt-Cache
- benutzt getObject(oid, class)
- Methode getObject(oid, class) ist abstract

- kennt die DB
- kann den passenden Objektbroker ermitteln
- Methode getObject(oid, class) ist Schablone, benutzt getObject(oid)
- Methode getObject(oid) ist abstract

- kennt und verwaltet den Objekt-Cache
- Methode getObject ist implementiert

Der Klassenspezifische DAO-Anteil ist klein und DB-neutral:

Generische Lösung

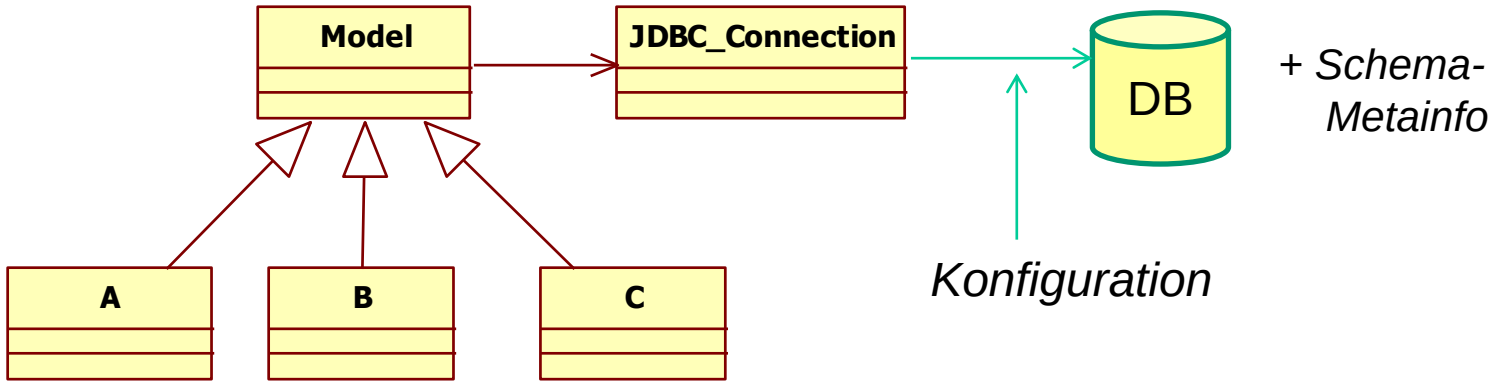


Einfachere Umsetzung mit generischen Klassen:

- Die Klasse AbstractDAO enthält getObject als generische Schablonenmethode
- Die Verbindungsinformation und der DB-Zugriff wird als separates Objekt eingebunden
- In der Konkretisierung wird der tatsächliche Typ eingesetzt

Alternative: Active Record

- ActiveRecord ist ein Entwurfsmuster von M. Fowler
- Konzeptumkehrung:
  Fachklassen aus dem DB-Schema ableiten.
- Fachklasse erweitert eine (fertige) Persistenzklasse
- Persistenzklasse enthält CRUD-Methoden, die die Metainformationen des Datenbankschemas nutzen.



Alternative: Active Record

- **Java-Implementierung: ActiveJDBC**

```
CREATE TABLE students (
  id  int(11) NOT NULL auto_increment PRIMARY KEY,
  first_name VARCHAR(56) NOT NULL,
  last_name VARCHAR(56),
  dob DATE,
  graduation_date DATE,
  created_at DATETIME,
  updated_at DATETIME
  );
```

Configuration: students→Student

```
class Student
  extends Model
  {} //fertig!
```

```
Student s = new Student();
s.set("first_name", "John");
s.set("last_name", "Doe");
s.set("dob", "1991-12-06");
s.saveIt();
```

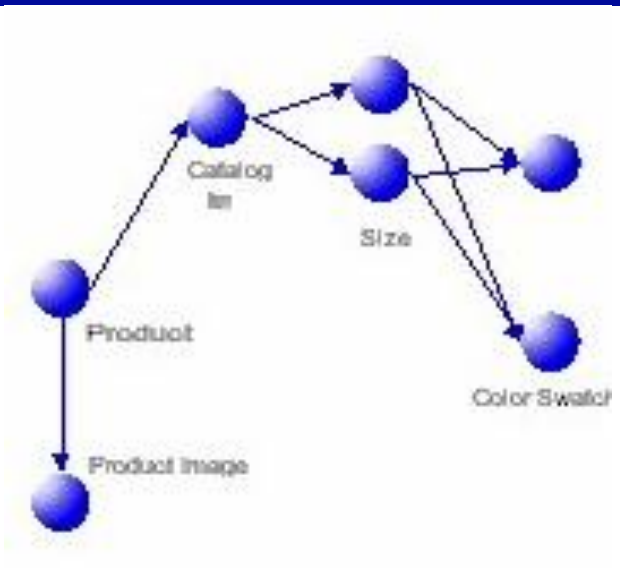
Diskussion ActiveRecord

- ✓ **Bestechend einfach!**

- OO-Sicht ist **sekundär**
  erschließt das DB-Schema im Programm
- Modellierungseinschränkung
  (keine Vererbung, Polymorphie)

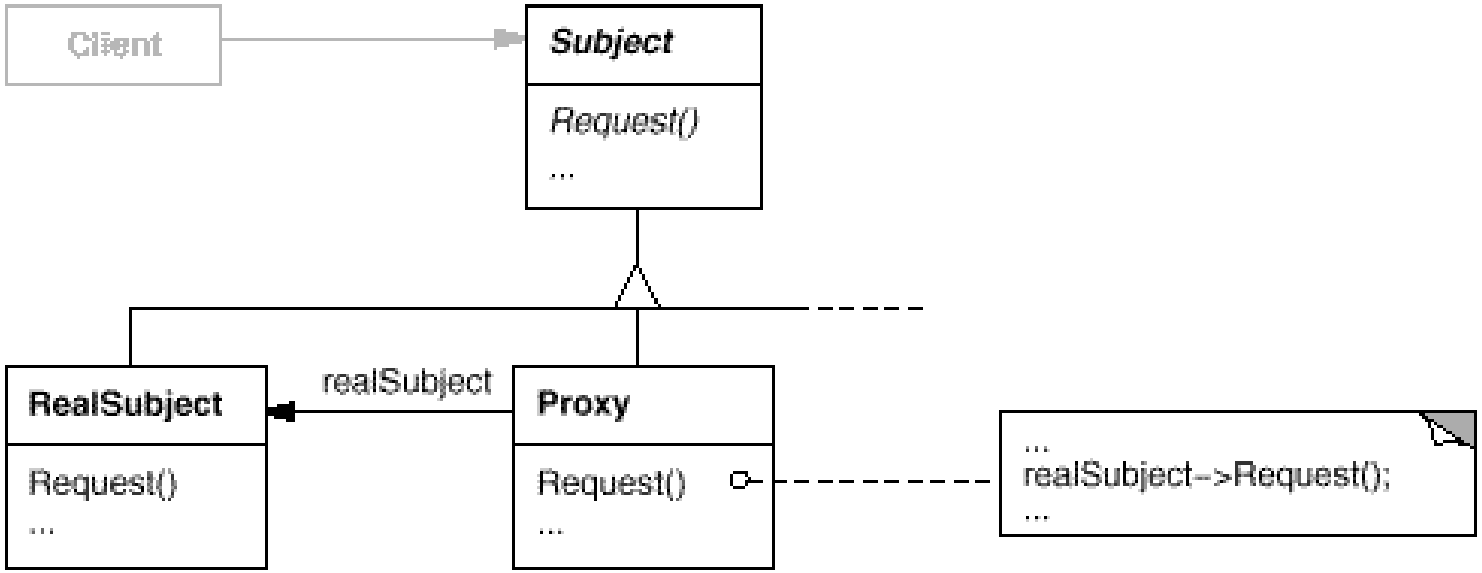
- Fachobjekte sind keine POJOS, sondern erweitern die Persistenzklasse
- Attributoperationen werden als **DB-Aktionen** verstanden

OR-Mapping – Lazy Materialization



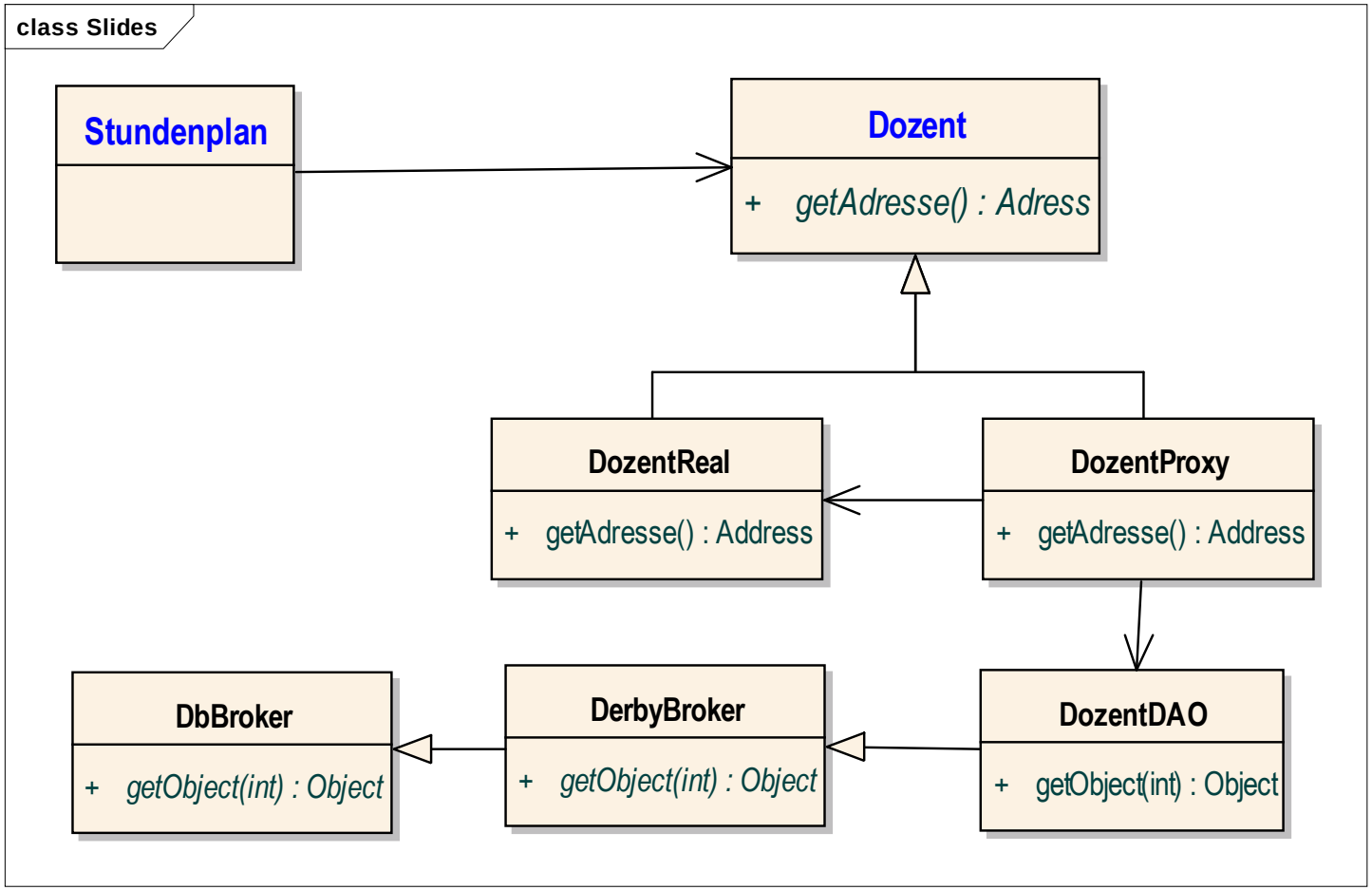
- Laden externer Objekte:
  - Was ist mit den assoziierten Objekten?
  - Objektstruktur kann stark verzweigen...
- Proxy-Entwurfsmuster:
  - Objekt erreichbar über ein Stellvertreterobjekt (Proxy)
  - Proxy liegt im Speicher
  - besorgt das Nachladen des Objekts bei Bedarf
  - enthält das OID

Proxy-Entwurfsmuster



- Objektassoziation des Client verweist auf einen Platzhalter
  - Proxy,
  - imitiert Struktur des realen Subjekts.
- Anfrage (request) erzeugt Bedarf:
  - Das reale Subjekt wird geladen
  - und die Anfrage delegiert.

DAO mit Proxy- und Broker-Muster



- erst bei Bedarf wird Dozent als DozentReal mithilfe des DAO materialisiert
- der gesamte Persistenzapparat bleibt unter der Oberfläche.

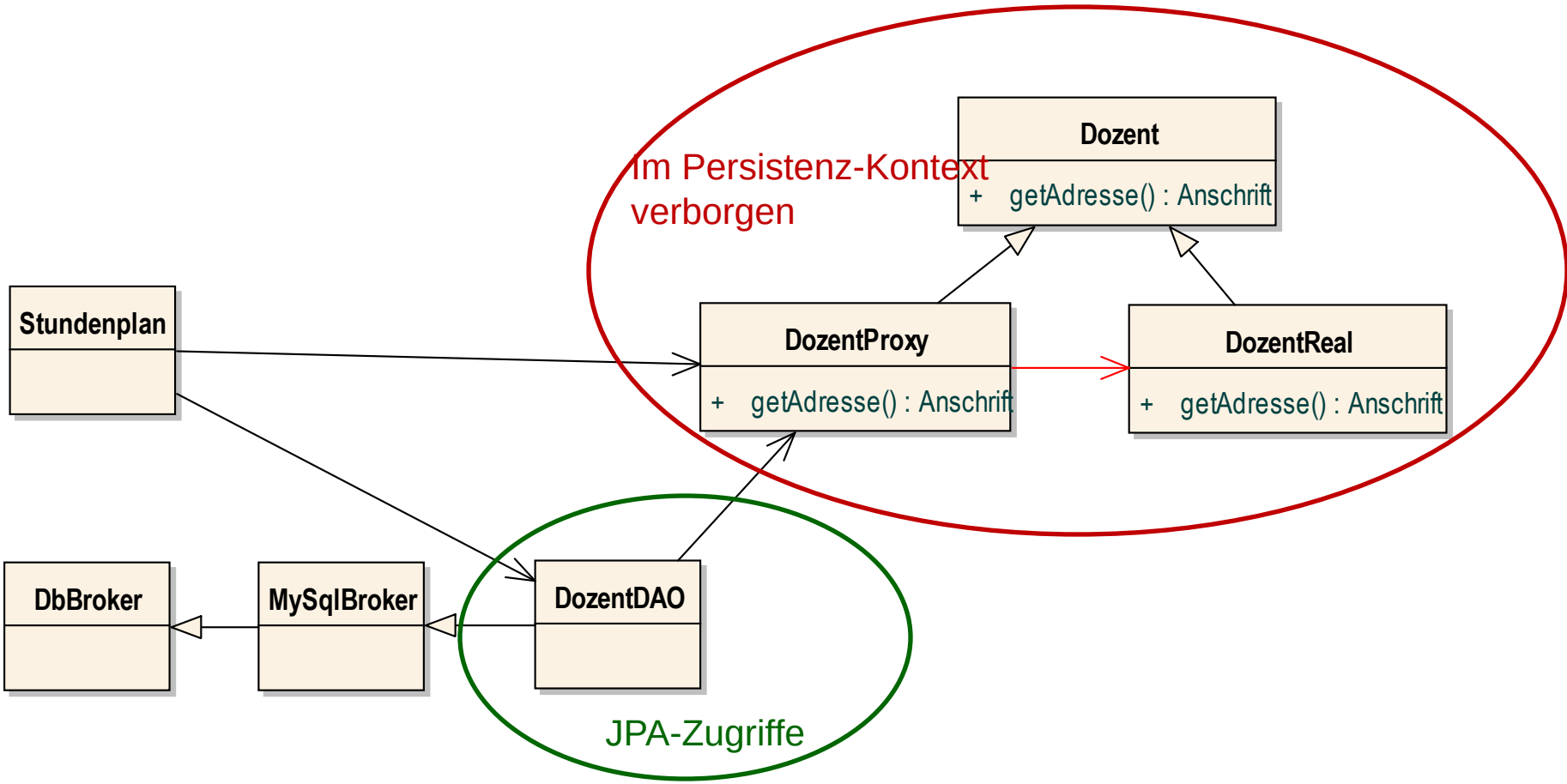
Fazit

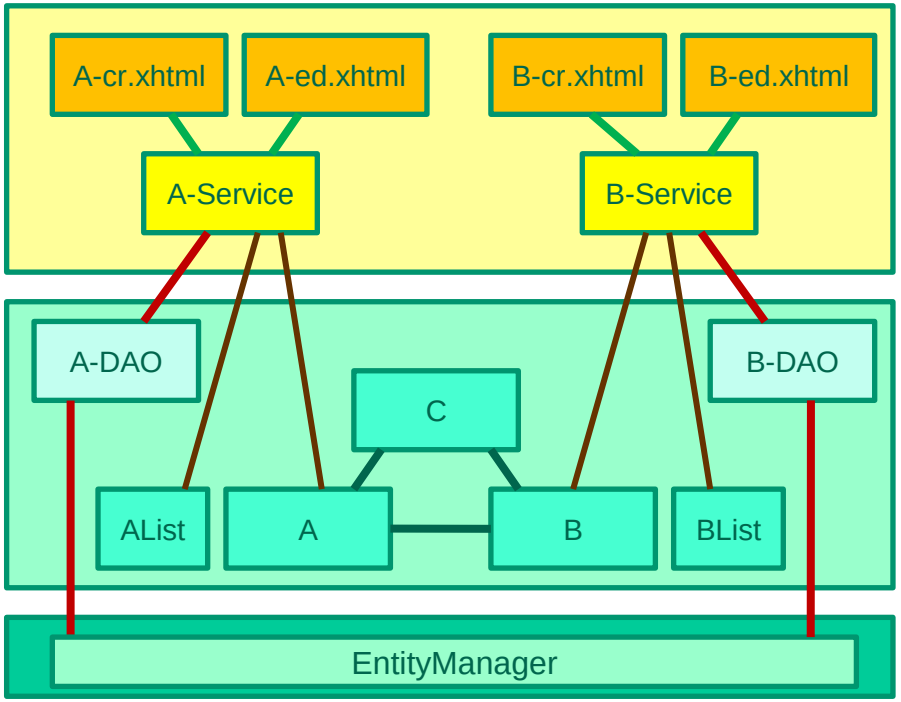
- Persistenz-Frameworks liefern den sicheren Umgang mit persistierten Objekten.

- Ziel: POJO-Architektur
  Die Schnittstelle zur Persistenz wird aus den Fachobjekten ausgelagert → DAOs.
  Fachobjekte sind POJOS

- *Alternativer Ansatz: "Active Record"*
  Jedes Fachobjekt erweitert eine Persistenzklasse, umgekehrtes Konzept: Objekt materialisiert Datensatz.

Persistenz mit JPA





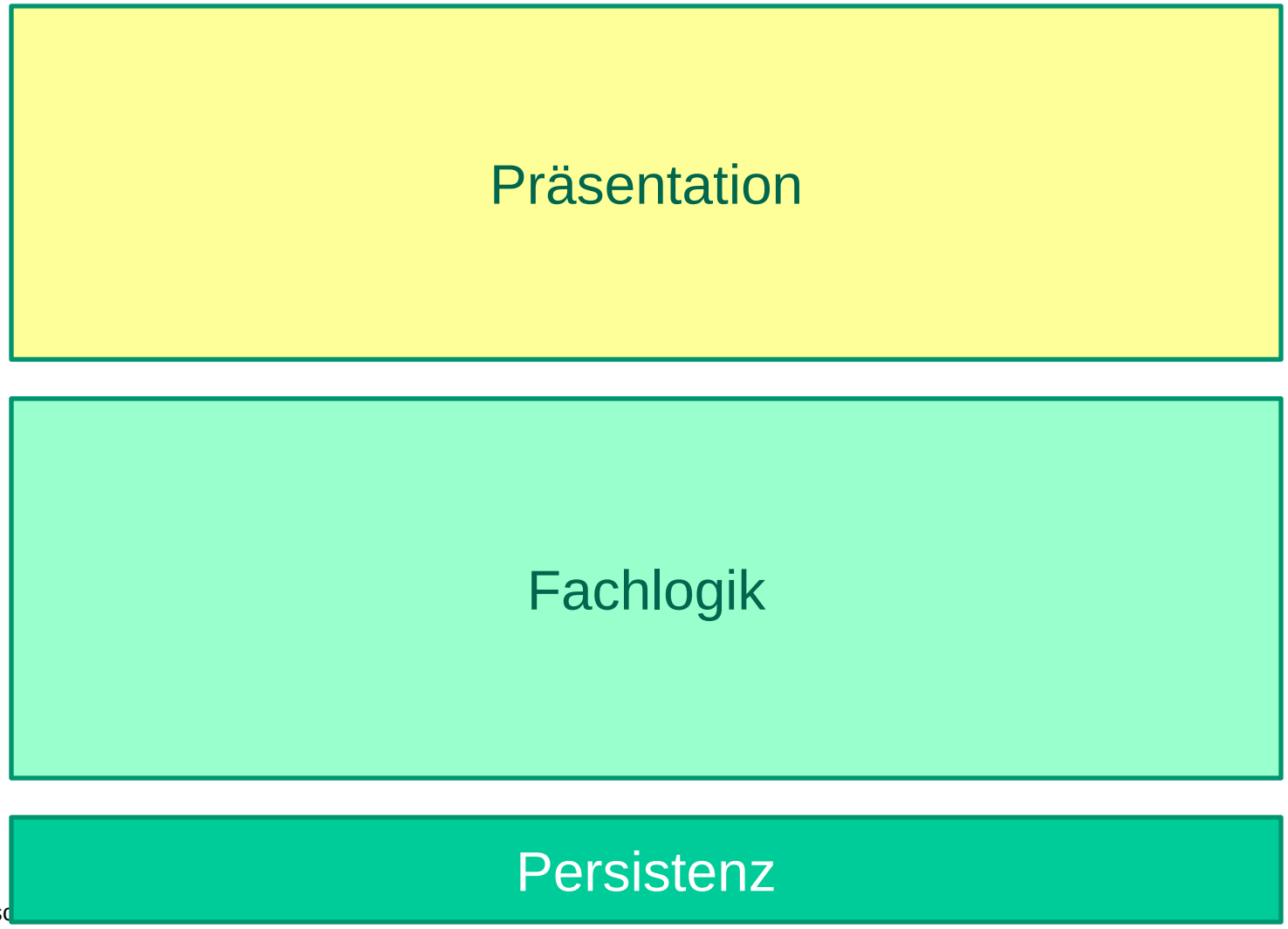
3. Referenzarchitektur für Webanwendungen

Und jetzt: Das Architektur-Puzzle

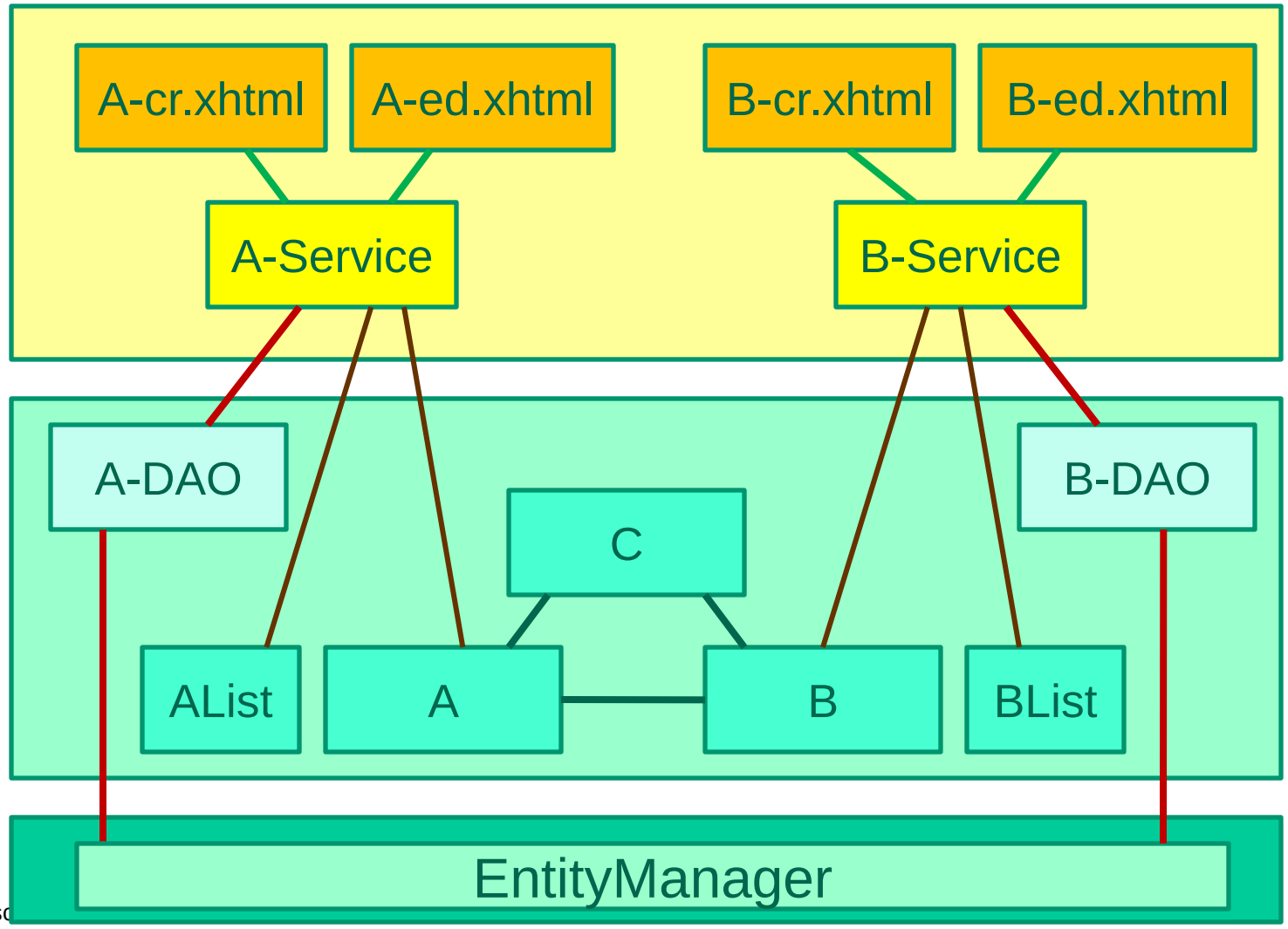
Nach der Etablierung der Techologien:

- Überarbeitung des Fachlichen Modells zu einem geeigneten Entwurfsmodell
  - Navigationen
  - Datenhaltung, Listen
  - Paketierung
  - Entwurfsmuster
- Einbindung des Entwurfsmodells in eine Gesamtarchitektur
  - mit JSF für die Präsentation
  - mit JPA-Persistenz

3 Schichten



Architektur des CRUD-Systems



4. Schichtentrennung mit IoC

Wer instanziiert und verwaltet die Klassen?

- Service- (oder Control-)Klassen:
  - ManagedBeans im WebContainer
  - "Grüne" Assoziationen durch "Dependency Injection": Der Webcontainer stellt die Beans unter dem geforderten Namen bereit.

- DAO-Klassen und "rote" Assoziationen?
  - Vier Möglichkeiten

1. ManagedBeans mit ManagedProperties
2. EJBs mit Dependency Injection
3. EJBs mit JNDI
4. *Pojos in einem IoC-Container (z.B. Spring)*

IoC – Inversion of Control

- Allgemeineres Konzept:
  - nicht der Client erzeugt und verwaltet seine referierten Objekte
  - sondern die referierten Objekte werden autonom erzeugt und von außen an den Client gebunden.

- Dependency Injection ist wichtigste Umsetzung
  - Objekte werden von außen an die Referenzattribute gebunden
  - "Außen" ist immer irgendein Container.

Dependency Injection???

Hier nur grob:

- Die Client-Klasse besitzt ein Strukturattribut (oder Referenzattribut)
- benutzt es ohne Initialisierung
- Die referierte Instanz wird zur Laufzeit von außen "injiziert".

```
class A_Service {
     private A_DAO adao;   // hierher wird injiziert

     public A findA(int id)
     {       return adao.find(id);     }
```

…kennen wir schon…

Dependency Injection im JSF Data Binding

"Grüne" Beziehungen

- Der EL-Ausdruck "#{serviceBean.name}" in A_cr.jsp funktioniert aufgrund von Dependency Injection:

- Im aus A_cr.jsp erzeugten Servlet steht unter dem Attributnamen "serviceBean" automatisch eine Referenz auf das ManagedBean vom Typ A_Service zur Verfügung.
- Die ManagedBean wurde dazu vom JSF-Web-Container instanziiert und die Referenz an das Servlet übergeben.

```
class A_cr_Servlet extends Servlet {
        A_Service serviceBean;
        …
}
```

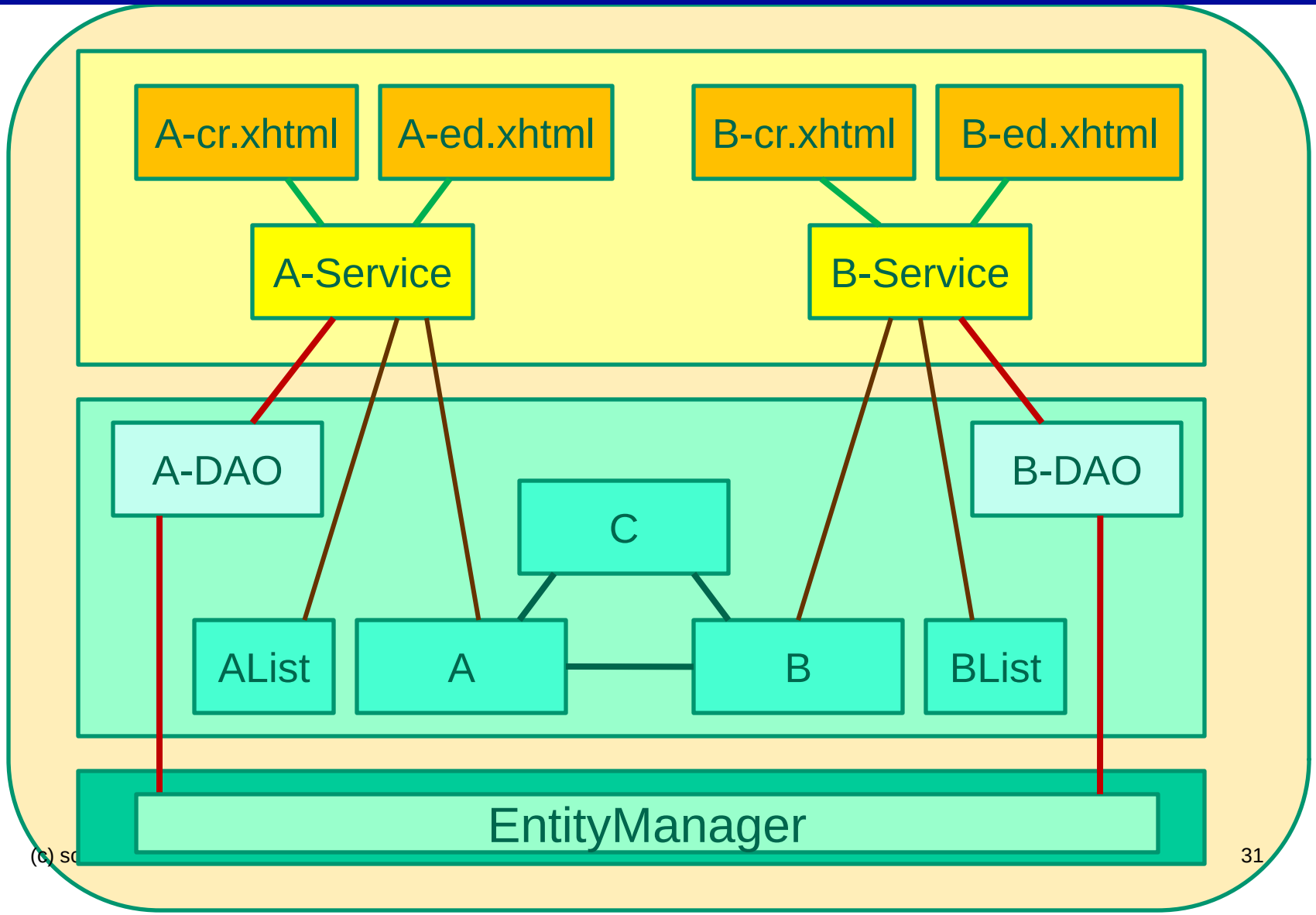
Schichten und Schichtenübergänge

- Im Architekturbild sind Schichten klar
- Rote Beziehungen müssen schichtenübergreifend hergestellt werden
  (schwarze entstehen durch DAO-Nutzung)

- **Ziel: Lose Kopplung** == Schichtentrennung
- Gegenteil "Harte Verdrahtung":
  die Assoziation ist fest einprogrammiert.

- Wir diskutieren 3 Möglichkeiten der losen Kopplung

Möglichkeit 1: Managed Properties



Möglichkeit 1: Managed Properties

- Alles läuft im (JSF-)Web-Container
- DAOs sind ManagedBeans
- Der EntityManager auch
  (genauer: eine EntityManagerFactory)

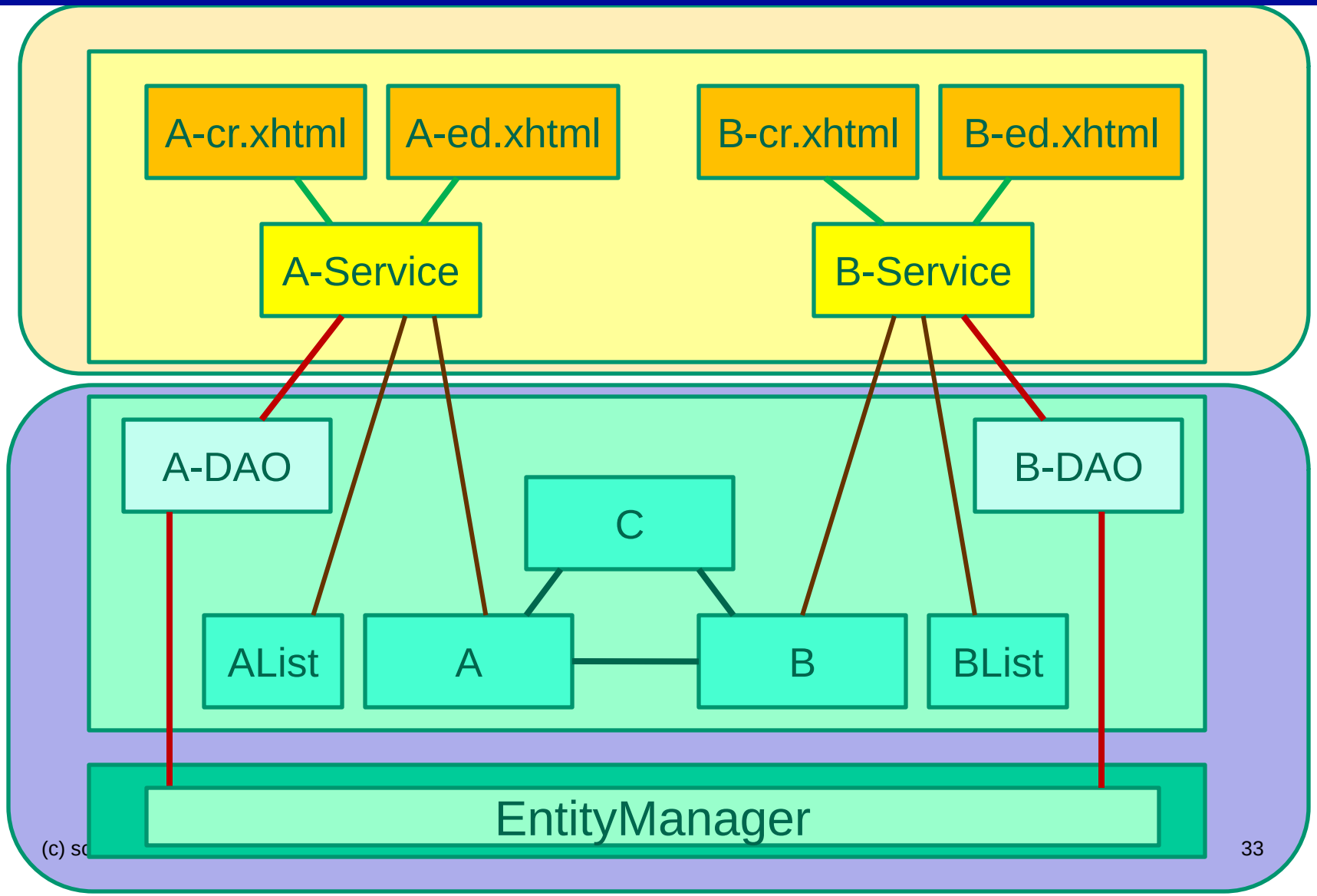
- Herstellung der roten Beziehungen als ManagedProperties

```
@ManagedBean(name="serviceBean")
class A_Service {
      @ManagedProperty(value="#{ADaoBean}"
      private A_DAO adao;   // hierher wird injiziert

      public A findA(int id)
      {       return adao.find(id);     }
}
```

```
@ManagedBean(name="ADaoBean")
class A_DAO {…}
```

Möglichkeit 2 und 3: EJBs



Möglichkeit 2: EJBs mit Dependency Injection

- DAOs sind EJBs, und zwar Stateless Session Beans
- DAOs laufen im EJB-Container

- Der EJB-Container bietet direkte Dependency Injection für den EntityManager

```
@ManagedBean(name="ServiceBean")
class A_Service {
      @EJB
      private A_DAO adao;   // hierher wird injiziert

      public A findA(int id)
      {       return adao.find(id);     }
}
```

```
@Stateless
class A_DAO {
     @PersistenceContext
     EntityManager em;  // injiziert
}
```

Möglichkeit 3: EJBs mit JNDI

- **JNDI** – Java naming and Directory Interface
- JNDI-Service ist Bestandteil von JEE

- Die Idee:
- ein Kontext exportiert Dienste per Namen
- ein anderer Kontext sucht und findet Dienste im Verzeichnis.

- Technisch benutzt man einen ServiceLocator als Schnittstelle, um zu einem Namen einen Dienst zu finden.

EntityManager und IoC?

- Abhängig vom JPA-Transaktions-Typ (eingestellt in der PersistenceUnit persistence.xml)

- RESOURCE_LOCAL
  - Standard in JSE-Anwendungen
  - Die Applikation muss EntityManager und Transaktionen selbst verwalten

- JTA oder TRANSACTION(Java Transaction API)
  - Standard in JEE-Anwendungen
  - Container verwaltet EntityManager und Transaktionen injiziert den EntitiyManager

RESOURCE_LOCAL

- RESOURCE_LOCAL

```
<persistence-unit name="myJSE_PU"
                  transaction-type="RESOURCE_LOCAL">
    <non-jta-data-source>myDatasource</non-jta-data-source>
    <class>model.Entity1</class>
 </persistence-unit>
```

  - Standard in JSE-Anwendungen
  - Die Applikation muss den EntityManager und die Transaktionen verwalten
  - EntityManagerFactory ist injizierbar

```
@PersistenceUnit  EntityManagerFactory emfactory;
EntityManager em = emfactory.createEntityManager();
```

  - Niemals mehrere EntityManager parallel benutzen!!

JTA

- JTA oder TRANSACTION(Java Transaction API)

```
<persistence-unit name="myJEE_PU"
                  transaction-type="JTA">
    <jta-data-source>myJTADatasource</jta-data-source>
    <class>model.Entity1</class>
  </persistence-unit>
```

  - Standard in JEE-Anwendungen
  - Container verwaltet EntityManager und Transaktionen
  - EntityManager ist direkt injizierbar, *keine Duplikate*

```
@PersistenceContext  EntityManager em;
```

IoC – Zusammenfassung

- Objektmanagement durch die Umgebung
  - IoC-Framework
  - Oder als IoC-Pattern implementiert

- Service-Konzept:
  - Anwendung "bestellt" Objekt: "*brauche DozentDAO*"
  - Framework liefert, gemäß Spezifikation,
    - neues Objekt
    - aktuell verfügbares Objekt
    - allgemein veröffentlichtes Objekt
  - kümmert sich um Instanziierung, Verwaltung, Thread-Safety,...

- Beliefern → Dependency Injection

Wichtige IoC-Frameworks

- Spring
  - Setter Injection (auch Constructor Injection möglich)
  - reichhaltige Bibliothek
  - Standard-Hibernate-Anschluss
  - Ausbaustufen für MVC, AOP
- PicoContainer
  - minimal, reiner IoC-Container
  - Constructor Injection
- Avalon
  - frühes Projekt, wird nicht mehr fortgesetzt
  - Interface Injection
- HiveMind

5. CRUD "aufbohren"

CRUD ist die Ausnahme

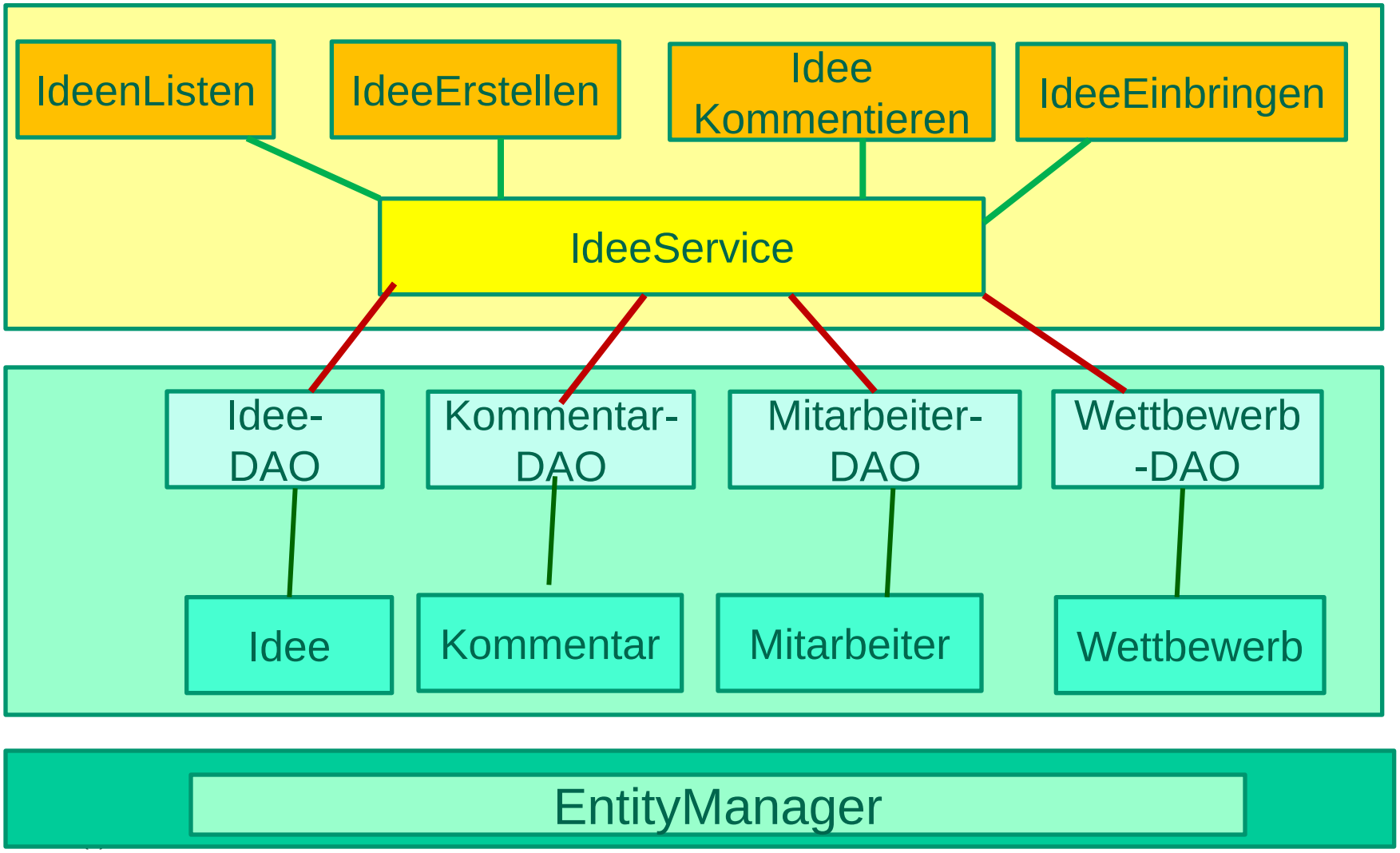
- CRUD → 1:1-Beziehung von Service zu DAO
  - Damit bleiben Managed Properties überschaubar.

- Realisitische Architektur:
  - Facelets richten sich nach den Anwendungsfällen
  - Services realisieren Anwendungsfälle
  - Services referieren viele Fachobjekte → viele DAOs

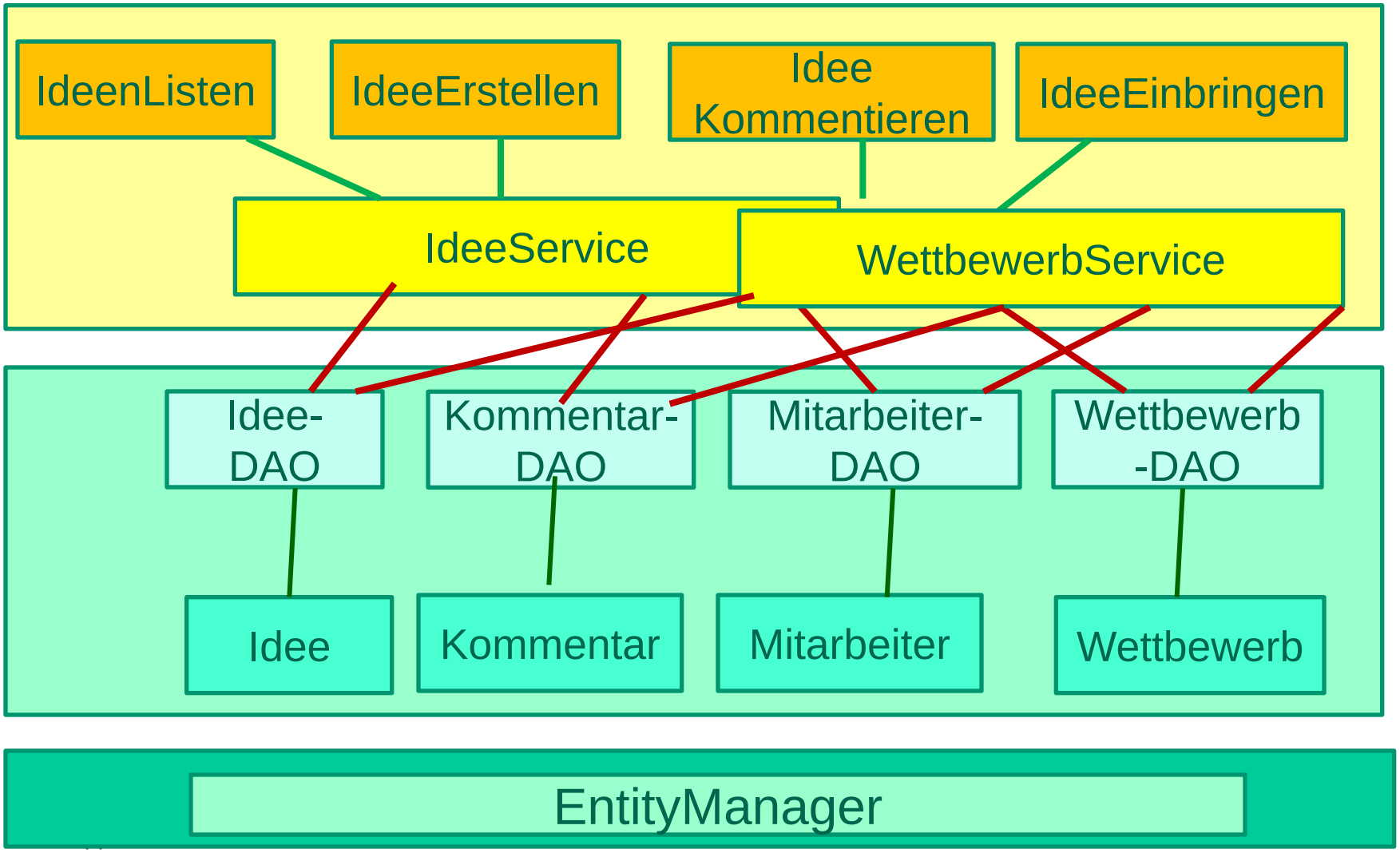
  → verschlungenes Netz von Managed Properties.

- Lösungen:
  - Fassade: Bündelung der Beziehungen
  - DAOs als EJBs: Dependency Injection oder JNDI

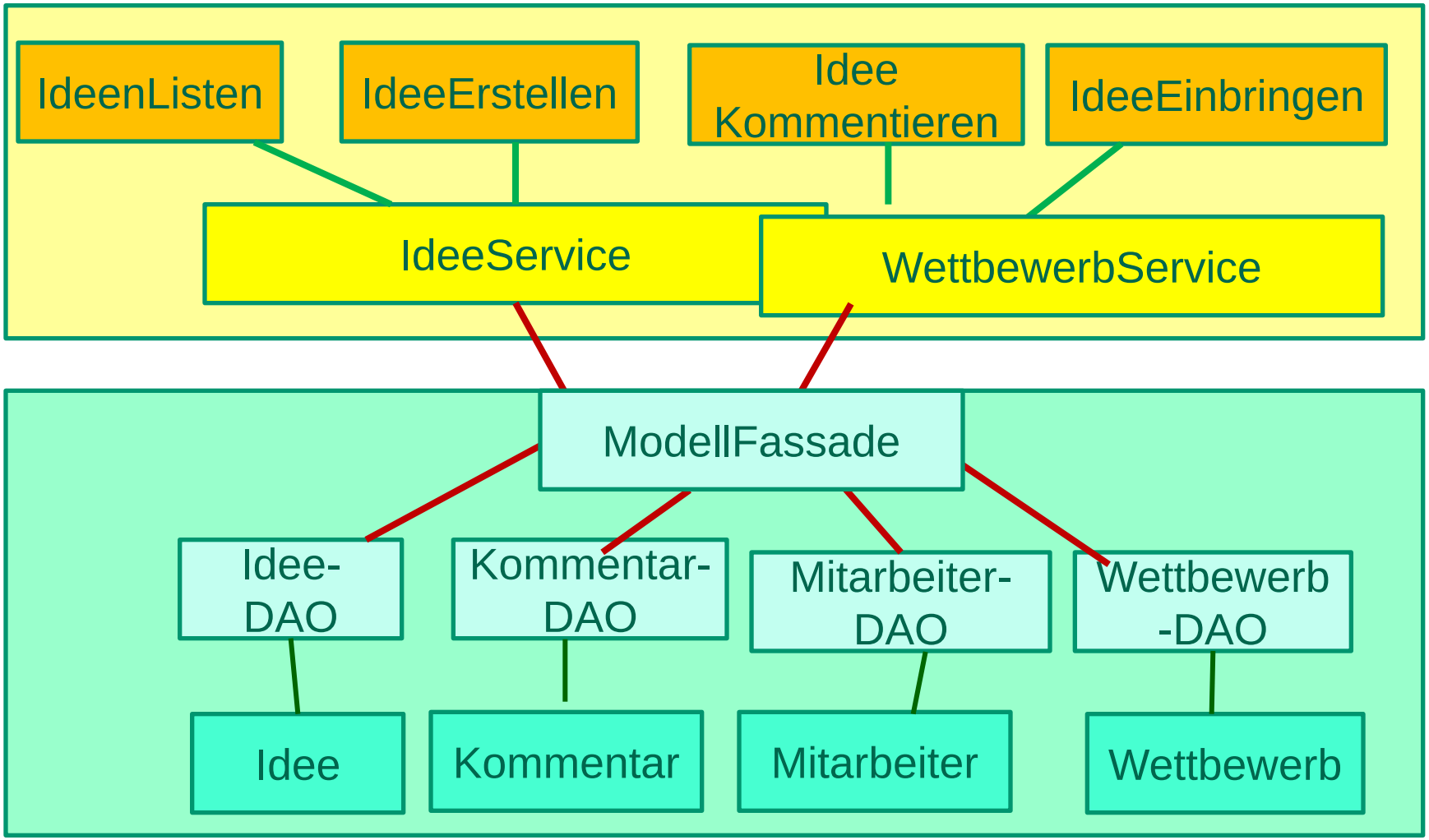
Architektur eines realen Systems (Ausschnitt)



Reales System (größerer Ausschnitt)



Fassade

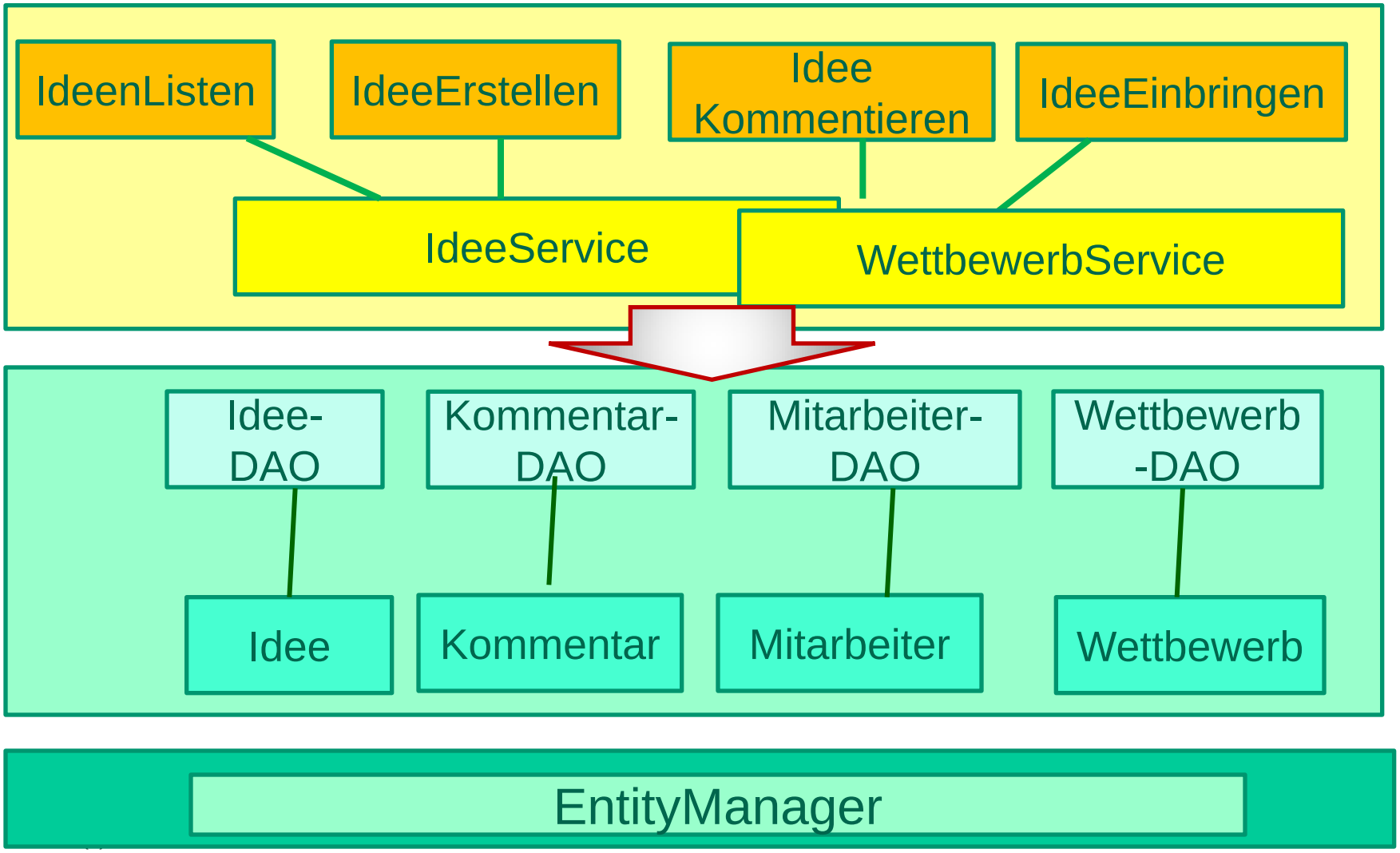

IdeenListen
IdeeErstellen
Idee Kommentieren
IdeeEinbringen
IdeeService
WettbewerbService
ModellFassade
Idee-DAO
Kommentar-DAO
Mitarbeiter-DAO
Wettbewerb-DAO
Idee
Kommentar
Mitarbeiter
Wettbewerb

DAOs als ManagedProperties in der Fassade

```
@ManagedBean(name="ModellFassadeBean")
public class ModellFassade {
   @ManagedProperty(value="#{IdeeDAOBean}")
   IdeeDAO ideeDao;
   @ManagedProperty(value=#{KommentarDAOBean}")
   KommentarDAO kommentarDao;
...

}
```

EJB-Injection



EJB-Injection

```
@ManagedBean
public class IdeeService {
@EJB IdeeDAO ideeDao;
@EJB KommentarDAO kommentarDao;
@EJB MitarbeiterDAO mitarbeiterDao;
…
}
```

Jetzt verstehen Sie hoffentlich, was der CRUD-Generator erzeugt!

Nächstes Mal beginnt die hohe Kunst der Klassenmodellierung!