Saxonia Systems

Wir lieben IT.

Saxonia Systems AG

Ihr Spezialist für IT Beratung, Software Entwicklung und Outsourcing Beratung

Dresden · Frankfurt/Main · Leipzig · München · Hamburg · Görlitz · Berlin

Saxonia Systems

Wir lieben IT.

www.saxsys.de

Daniel Röder

O/R Mapping mit der Java Persistence API

Dresden · Frankfurt/Main · Leipzig · München · Hamburg · Görlitz · Berlin

Meine Person

Vorstellung

- → Studium der Angewandten Informatik an der TU Chemnitz
- → seit über 4 Jahren als Software-Entwickler bei der Saxonia Systems AG
- → Projekterfahrung in den Branchen Banken, Energieversorger und Handel
- → Autor des Buches „JPA mit Hibernate – Java Persistence API in der Praxis“

Saxonia Systems AG (1)

Vorstellung

- Unabhängiges, mittelständisches IT-Beratungs- und Technologieunternehmen mit 3 Geschäftsbereichen:
  - IT- und Prozessberatung
  - Outsourcing-Beratung
  - Softwareentwicklung
- Gründung 1990 in Dresden durch Absolventen der TU Dresden
- 175 Mitarbeiter in der Unternehmensgruppe
- 90% der Mitarbeiter mit Hochschul- oder Fachhochschulabschluss
- Präsent mit 7 Standorten in Deutschland
- Langjährige Projekterfahrung, umfangreichen Projektreferenzen
- Stark in der Analyse und Methodik sowie in der Umsetzung
- Zertifizierung nach ISO 9001, ISO/IEC 15504 (SPICE)

Saxonia Systems AG (2)

Vorstellung

Als Technologieunternehmen fördern wir eine breite akademische Zusammenarbeit mit Hochschulen an verschiedenen Standorten, speziell in den Bereichen Informatik, Wirtschaftinformatik und vgl.

Angebote für Studenten (m/w)

- → Praktika
- → Projekteinsätze für Werkstudenten (m/w)
- → Abschlussarbeiten (Diplom, Bachelor, Master)
- → Förderung von Promotionen

Angebote für Absolventen (m/w)

- → Fachtrainee-Programm für Softwareentwickler und Prozessberater
- → Direkteinstieg an unseren Standorten in den Bereichen:
    - → Softwareentwicklung
    - → IT- und Prozessberatung
    - → Qualitätsmanagement/ Test

Gliederung

Einführung

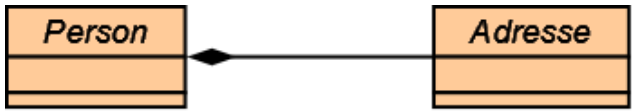
- → Vorstellung
- → Einführung
- → JPA praktisch
- → Aufbau und Mapping von Entities
- → Lebenszyklus einer Entity
- → Patterns für die Verwendung des EntityManagers
- → Datenbankabfragen mit JPQL

Motivation O/R Mapper (1)

Einführung

→ „Impedance Missmatch“ – Unterschied zwischen normalisierten, relationalen Datenbanken und objektorientierten Klassenhierarchien:

→ **Granularität** – objektorientiertes Modell sehr feingranular im Gegensatz zu relationalen Datenbanken



ID	Vorname	Nachname	...	Adresse_PLZ	Adresse_Stadt	...
1	Max	Mustermann	...	01234	Musterstadt	...
2	...	...	...	...	...	...

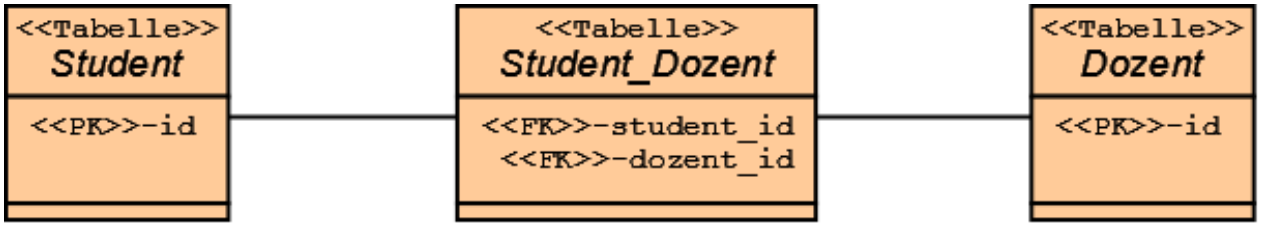
→ **Vererbung** – ist in objektorientierten Sprachen selbstverständlich, aber unbekannt bei relationalen Datenbanken

→ **Objektidentität** – in Java Unterschied zwischen Objektidentität (==) und Objektgleichheit (.equals()), bei Datenbanken erfolgt Identifikation eines Eintrags über dessen Inhalt -> Einführung eines Primärschlüssels

Motivation O/R Mapper (2)

Einführung

- **Beziehungen** – in Datenbanken Abbildung der Beziehungen über Fremdschlüssel (Verweis auf einen Primärschlüssel einer anderen Tabelle 1-zu-1 Beziehung), jedoch in der objektorientierten Welt auch 1-zu-n und n-zu-m Beziehungen



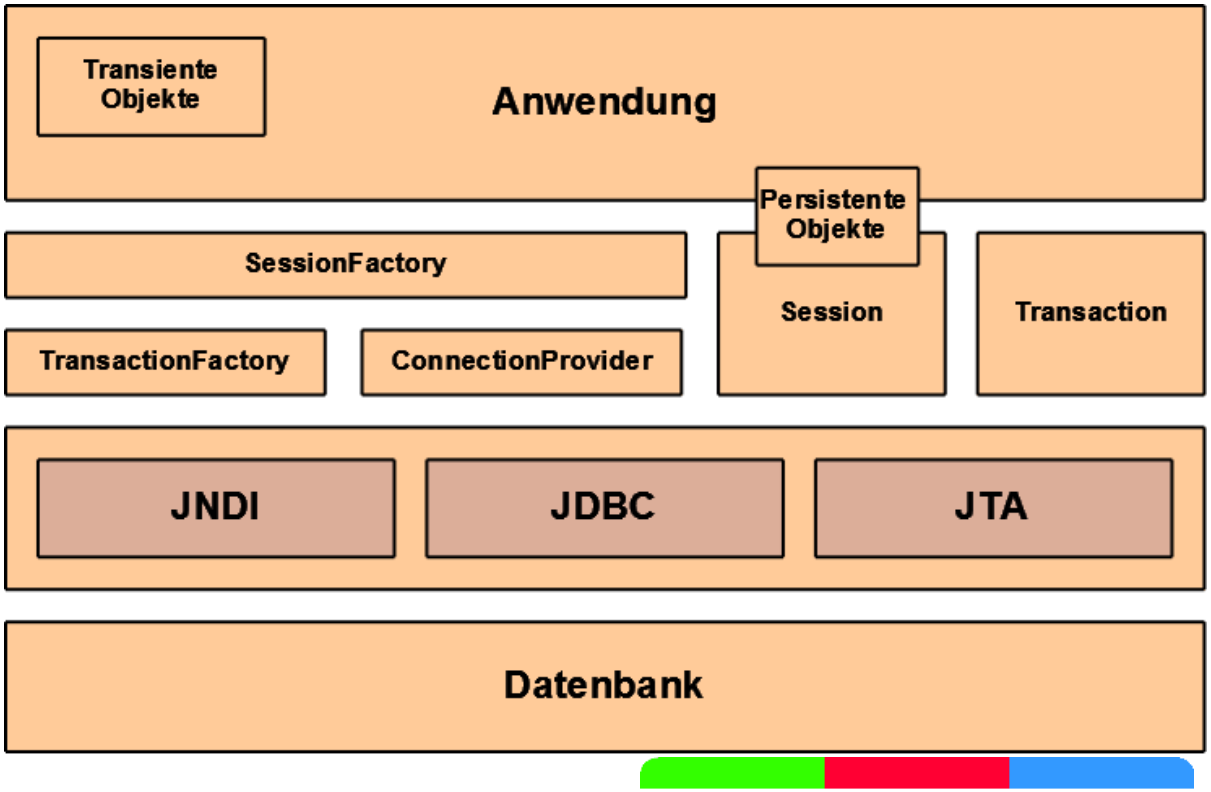
- **Graphennavigation** – eine Navigation über die Objekte ist in Java sehr einfach (*dozent.getVorlesungen()*), die Datenbank benötigt entweder mehrere Select-Statements oder ein Join (*select * from DOZENT left outer join VORLESUNG where ...*)

- **FAZIT:** Es gibt viele Unterschiede zwischen objektorientierter Programmierung und relationaler Datenbank → O/R Mapper soll Probleme lösen

Hibernate als O/R Mapper

Einführung

- Hibernate ist...
  - ...ein mächtiger und performanter O/R Mapper.
  - ...unterstützt nahezu alle DBMS.
  - ...Inspiration für JPA 1.0.
  - ...praxistauglich.
  - ...Open Source.
  - ...modular.



JPA – Java Persistence API

Einführung

→ JPA – **Java Persistence API** wurde als Teil der EJB3 Spezifikation (JSR 220) veröffentlicht und löste die EntityBeans aus J2EE ab
→Ziel von JPA ist eine Standardisierung des Basis-API sowie der Metadaten eines objektrelationalen Persistenzmechanismus für Java
→JPA ist KEIN fertiges Framework, lediglich eine Spezifikation

→Eigenschaften von **JPA 1.0**
- → die Entities sind einfache POJOs (Plain Old Java Objects)
- → objektorientierte Klassenhierarchien mit Vererbung, Assoziationen und Polymorphismus werden unterstützt
- → die objektorientierte Abfragesprache JPQL (Java Persistence Query Language)
- → API ist nicht nur in JEE Application Servern einsetzbar sondern auch mit Java SE

→Wichtigste Erweiterung von **JPA 2.0**
- → Collections von Basistypen
- → Eine Criteria API mit Metamodell
- → Ein Cache Interface
- → Erweiterungen der JPQL
- → Unterstützung der Bean Validation API (JSR 303)

JPA – Entity Manager und PersistenzKontext

Einführung

- **Entity Manager**
  - verwaltet die Entities
  - zentrales Interface von JPA für den Zugriff auf den Persistenzkontext
  - **container-managed EM** – Java EE Application Server verwaltet EM und PK
  - **application-managed EM** – erzeugen über EntityMangerFactory

- **Persistenzkontext**
  - eineindeutige Menge von Entities (d.h. jede mögliche Entity höchstens einmal enthalten)
  - pro EntityManager genau ein Persistenzkontext
  - **transaction-scoped** – Gültigkeitsdauer entspricht exakt der Dauer einer Transaktion
  - **extended** – Persistenzkontext auch außerhalb der Grenzen einer Transaktion gültig, muss daher manuell über EM geschlossen werden

Die erste Entity in der Datenbank

JPA praktisch

DEMO

Anforderungen an eine Entity

Aufbau und Mapping von Entities

- → einfaches POJO (Plain Old Java Object)
- → Markierung mit der Annotation @Entity
- → parameterloser Konstruktor mit Accessmodifikatior public oder protected
- → Top-Level-Klasse (kein Enum , kein Interface)
- → Deklaration von final weder für Entity noch für persistente Attribute erlaubt
- → Implementieren des Serializable-Interface
- → Primärschlüssel muss enthalten sein (kann auch von Superklasse geerbt werden)
- → zwei Möglichkeiten zum Annotieren der persistenten Felder

```
@Column
public String getName()
{…
```

→ zusätzlicher Code in Gettermethoden möglich

```
@Column
String name
```

→ unnötig Getter zu definieren, verhindern des Zugriffs von „außen“

→ seit JPA 2.0 auch Mischung von beiden Varianten mit @Access möglich

Benutzerdefinierte Tabellen und Spalten

Aufbau und Mapping von Entities

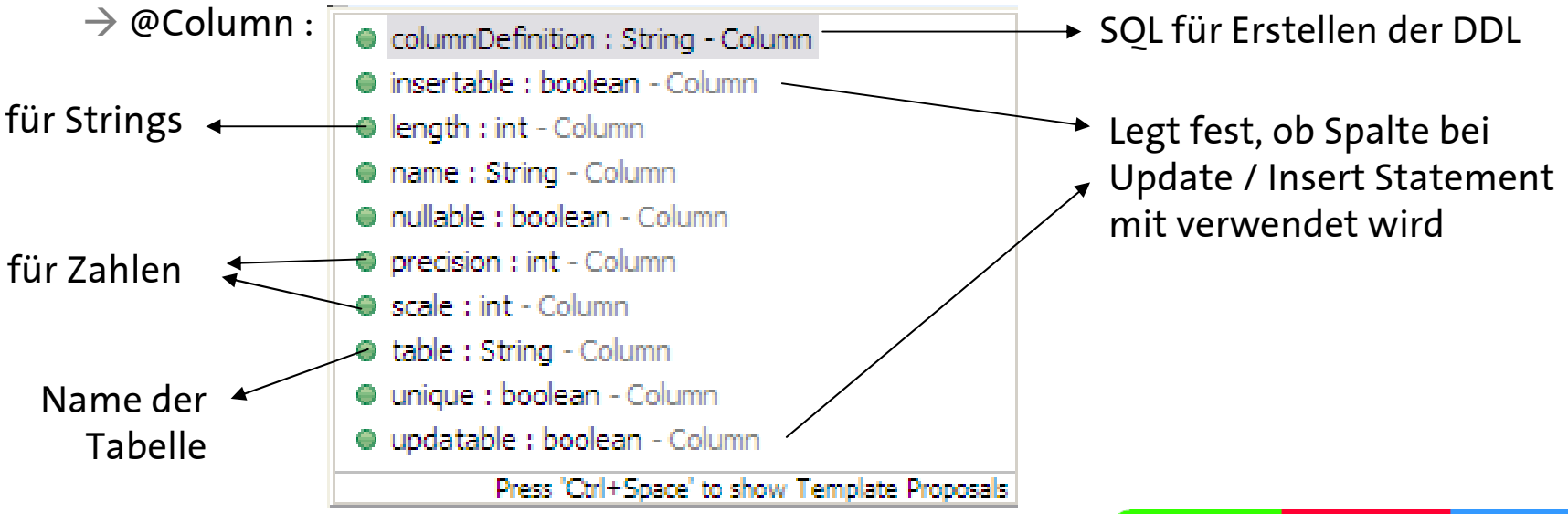
→ @Table : name, schema, uniqueconstraints

```
@Table(name="T_USER", schema="Test", uniqueConstraints={
        @UniqueConstraint(columnNames="name",name="unique_name"),
        @UniqueConstraint(columnNames="email",name="unique_email")})
```

→ @SecondaryTable(s) : Definition von mehreren Tabllen pro Entity

```
@SecondaryTables({ @SecondaryTable(name="T_USER_DETAIL"),
                   @SecondaryTable(name="T_USER_HIST")})
```

→ @Column :



Primärschlüssel

Aufbau und Mapping von Entities

- Anforderungen
  - nicht null
  - unveränderlich
  - pro Tabelle über alle Einträge eindeutig
  - mit der Annotation @Id zu markieren
  - erlaubte Datentypen: primitiver Typ , ein entsprechender Wrapper, java.lang.String, java.lang.Date, java.sql.Date (neu in JPA 2.0 BigInteger und BigDecimal)
- komplexe Primärschlüssel mit @EmbeddedId oder @IdClass
- „natürliche“ / fachliche Primärschlüssel vermeiden
- Generatoren für Primärschlüssel mit Annotation @GeneratedValue

```
@Id
@GeneratedValue
long id;
```

→ nutzt automatisch passenden Generator für verwendete DB

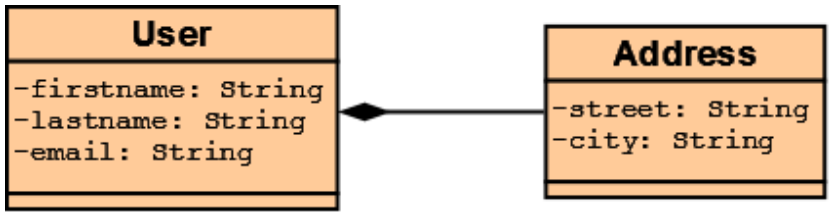
```
@Id @GeneratedValue(generator="uuid-gen")
@GenericGenerator(name="uuid-gen", strategy = "uuid")
String uuid
```

→ Verwendung von speziellen Hibernate Generator für UUID

Komponenten

Aufbau und Mapping von Entities

- Komponenten (Value-Types) ...
  - besitzen keine eigene Identität
  - „gehören“ einer Entity
  - ermöglichen feingranulare Objektstrukturen




```
@Embeddable
public class Address {
  private String street;
  private String city;
}
```

```
@Entity
public class User {
@Embedded
private Address address;
...
```

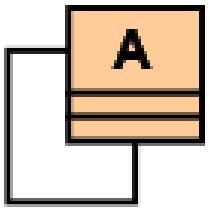
Id	Firstname	Lastname	Email	Address_Street	Address_City
1	Moritz	Muster	booksonline@...	Musterstr. 11	Musterhausen

Assoziationen – ein Überblick

Aufbau und Mapping von Entities

- Assoziationen ...
  - verbinden mindestens zwei Entities
  - ermöglichen die Navigation zwischen Entities
  - sind binär oder reflexiv
  - sind uni- oder bidirektional
  - haben eine Kardinalität die den Grad der Beziehung beschreibt
    - 1-zu-1; 1-zu-n; n-zu-1; n-zu-n

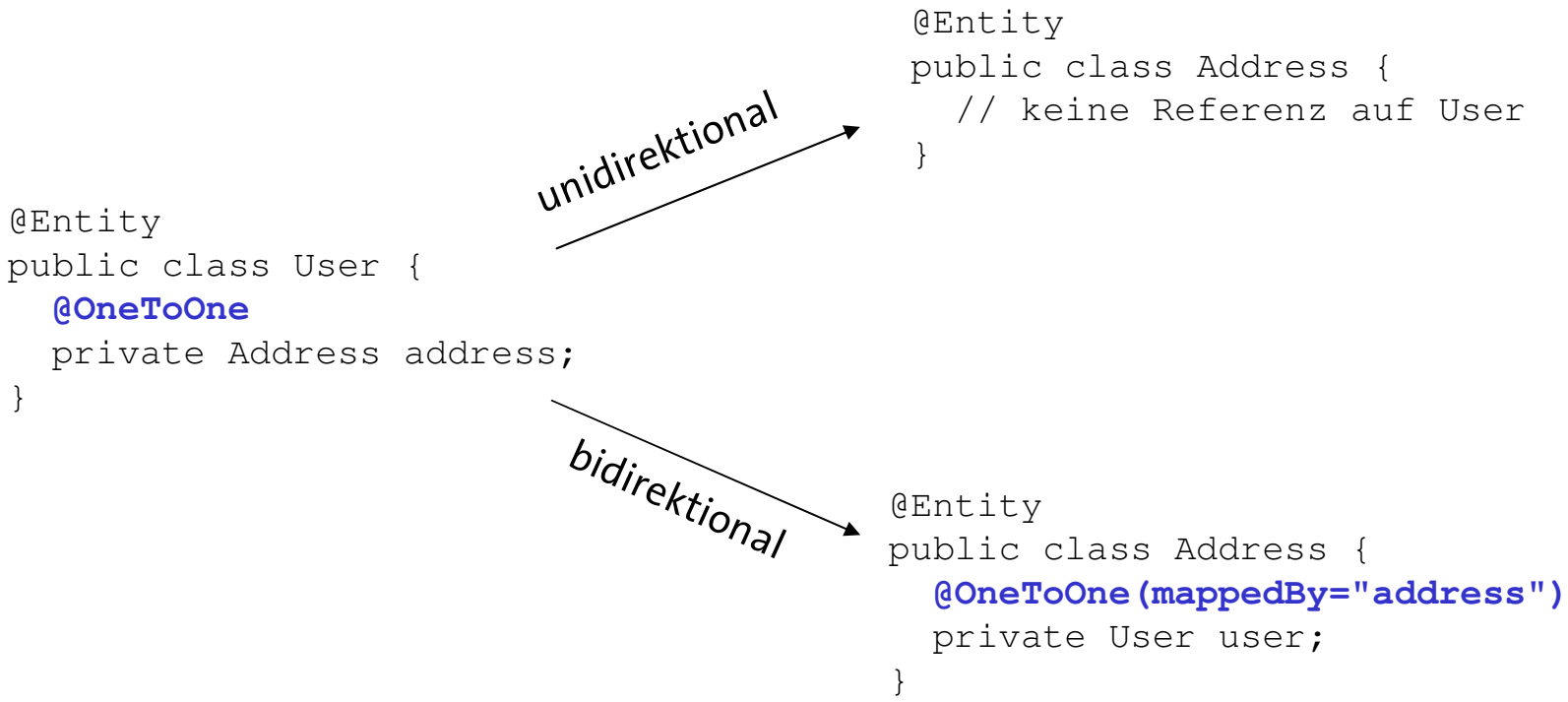
binäre Assoziation



reflexive Assoziation

Assoziationen – 1-zu-1 Beziehung

Aufbau und Mapping von Entities



Assoziationen – 1-zu-n / n-zu-1 Beziehung

Aufbau und Mapping von Entities

```
@Entity
public class User {
  @OneToMany
  private Set<LoginLog> ll;
}
```

unidirektional

```
@Entity
public class LoginLog {
  // keine Referenz auf User
}
```

Erzeugt eine Join-Tabelle USER_LOGINLOG mit zwei Fremdschlüsseln auf User.id und LoginLog.id

```
@Entity
public class User {
  @OneToMany(mappedBy="user")
  private Set<LoginLog> ll;
}
```

bidirektional

```
@Entity
public class LoginLog {
  @ManyToOne
  private User user;
}
```

Wichtig: Die „Rückreferenzen“ werden nicht automatisch gesetzt!

```
public void addLoginLog(LoginLog loginLog) {
    this.ll.add(loginLog);
    loginLog.setUser(this);
}
```

Assoziationen – n-zu-m Beziehung

Aufbau und Mapping von Entities

```
@Entity
public class User {
  @ManyToMany
  private Set<UserRole> roles;
}
```

uni- und bidirektional

```
@Entity
public class UserRole {
  // @ManyToMany(mappedBy="roles")
  // private Set<User> users;
}
```

Erzeugt eine Join-Tabelle USER_USERROLE mit zwei Fremdschlüsseln auf User.id und UserRole.id

Wichtig: Die „Rückreferenzen“ werden nicht automatisch gesetzt!

```
public void addUserRole(UserRole userRole) {
    this.roles.add(userRole);
    userRole.getUsers().add(this);
}
```

Assoziationen – Transitive Persistenz

Aufbau und Mapping von Entities

→ Transitive Persistenz erlaubt die Weitergabe von EntityManager-Operationen auf in Beziehung stehende Entities

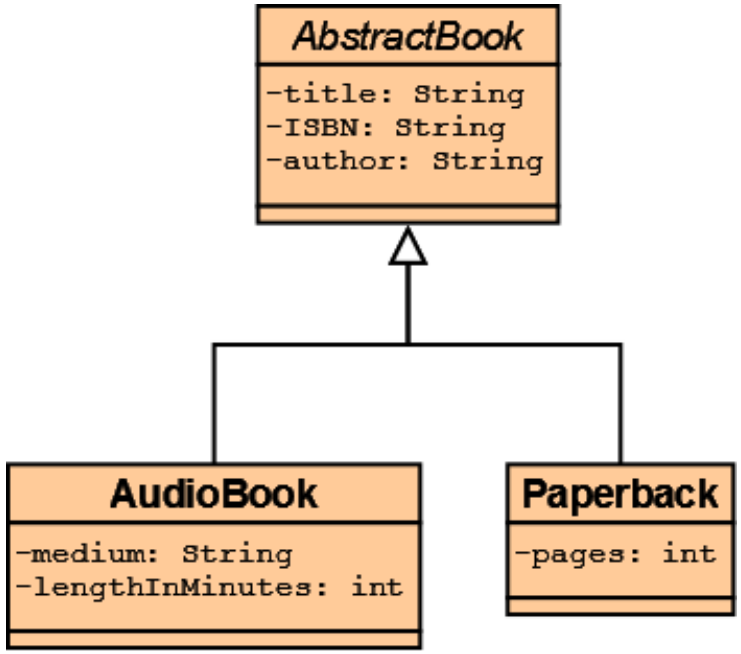
```
Set<LoginLog> ll = new HashSet<LoginLog>();
LoginLog loginLog = new LoginLog()
ll.add(loginLog);
user.setLl(ll);
em.persist(user);
```

→ per Default wird Entity LoginLog nicht mit der Entity User in der DB gespeichert
  a) Aufruf von em.persist(loginLog)
  b) transitive Persistenz: @OneToMany(**cascade = CascadeType.PERSIST**)
→ verwendbar für alle Beziehungstypen
→ mögliche CascadeType: PERSIST, MERGE, REMOVE, REFRESH, ALL

Vererbung – ein Überblick

Aufbau und Mapping von Entities

- JPA erlaubt die Abbildung von Vererbung
- drei mögliche Strategien:
  - SINGLE_TABLE – eine Tabelle für alle Klassen der Vererbungshierarchie
  - JOINED – für jede konkrete und abstrakte Klasse eine Tabelle
  - TABLE_PER_CLASS – für jede konkrete Klasse eine Tabelle.
- alle Strategien haben Vor- und Nachteile



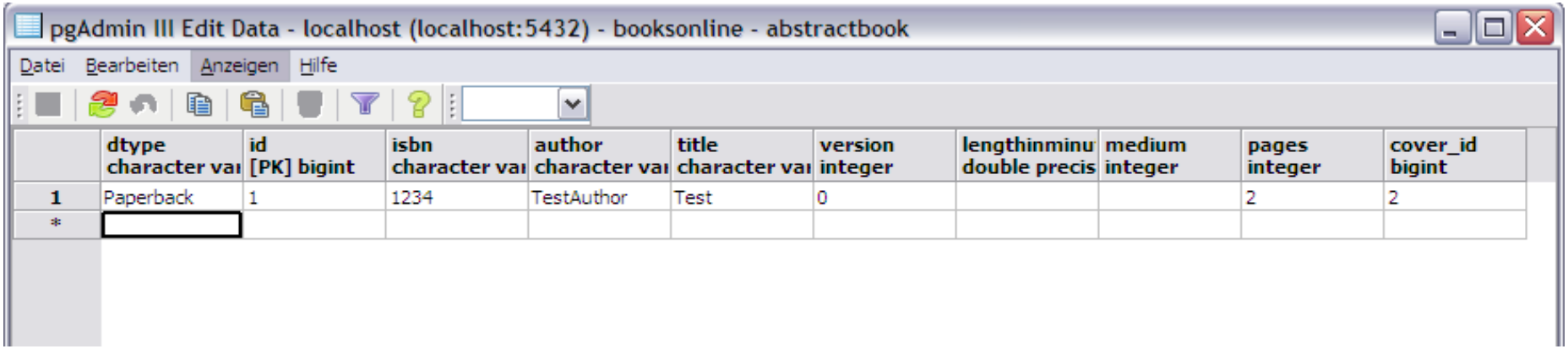
Vererbung – SINGLE_TABLE (1)

Aufbau und Mapping von Entities

→ Alle Klassen der Vererbungshierarchie werden in einer DB-Tabelle abgebildet
→ es wird ein Unterscheidungsfeld (*Discriminator)* benötigt

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
public abstract class AbstractBook implements Serializable {

@Entity
@DiscriminatorValue(value = "AudioBook")
public class AudioBook extends AbstractBook {
```



	dtype character vai	id [PK] bigint	isbn character vai	author character vai	title character vai	version integer	lengthinminu double precis	medium integer	pages integer	cover_id bigint
1	Paperback	1	1234	TestAuthor	Test	0			2	2
*										

Vererbung – SINGLE_TABLE (2)

Aufbau und Mapping von Entities

Vorteile

→ gute Performance, da für jegliche Abfrage (polymorph/konkret) nur ein Select benötigt wird

Nachteile

→ unnötige Spalten
→ bei großen Vererbungshierarchien entstehen sehr „breite“ Tabellen
→ verschlechterte Datenintegrität, da alle Attribute der abgeleiteten Entites nullable sein müssen

Fazit

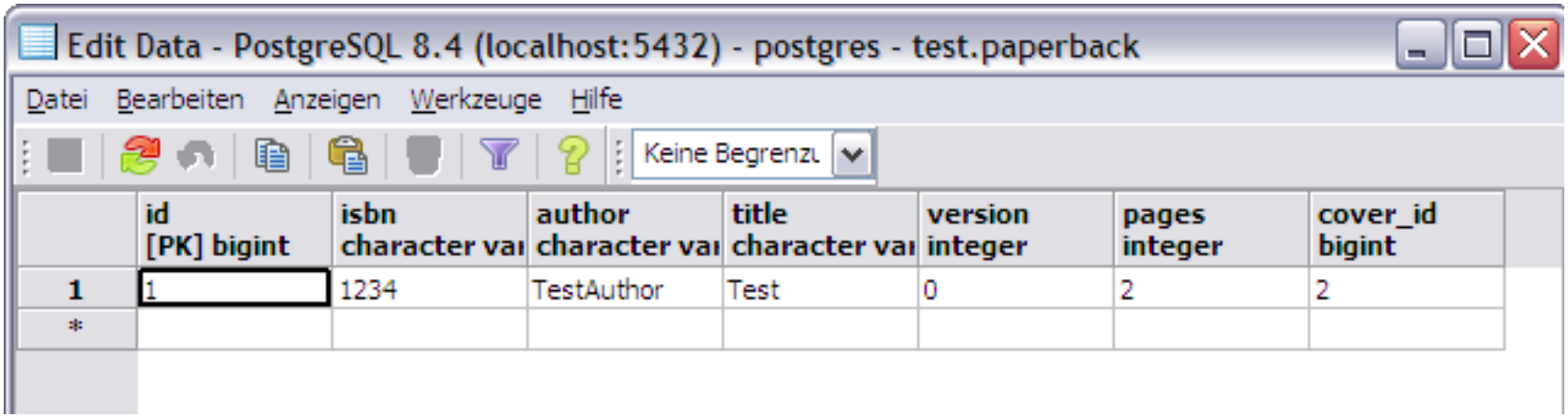
→ beste Wahl bei Vererbungshierarchien mit wenigen Attributen und Einsatz von polymorphen Abfragen
→ Alternative, bei nicht benötigter Polymorphie: TABLE_PER_CLASS

Vererbung – TABLE_PER_CLASS (1)

Aufbau und Mapping von Entities

→ Jede konkrete Klasse der Vererbungshierarchie wird in einer DB-Tabelle abgebildet
→ kein *Discriminator* zur Unterscheidung nötig

```
@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public abstract class AbstractBook implements Serializable {
```



Vererbung – TABLE_PER_CLASS (2)

Aufbau und Mapping von Entities

Vorteile

→ gute Performance bei Zugriff auf konkrete Entity-Typen
→ Datenintegrität kann gewährleistet werden (Verwendung von *not null*)

Nachteile

→ schlechte Performance bei polymorphen Abfragen
→ Änderung eines Attributs hat Auswirkungen auf alle Tabelle der Vererbungshierarchie

Fazit

→ beste Wahl bei Vererbungshierarchien ohne Einsatz von polymorphen Abfragen
→ unbrauchbar bei Assoziationen auf Superklasse

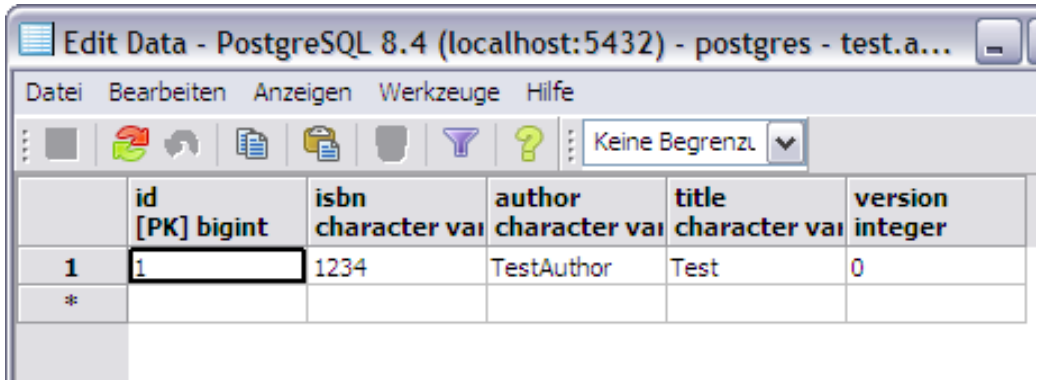
```
@OneToMany
private Set<AbstractBook> books;
```

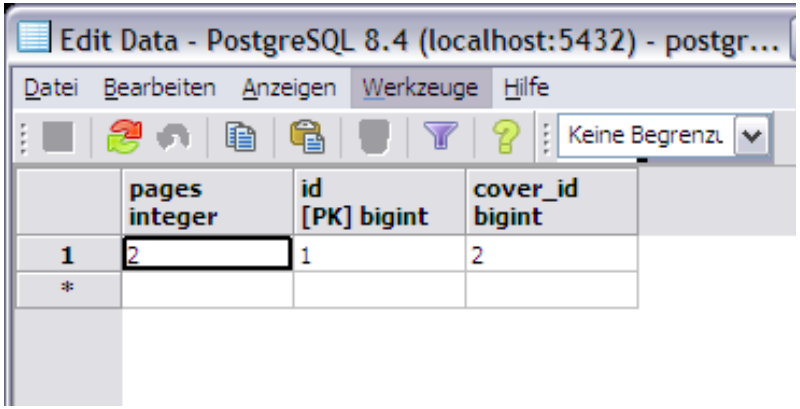
Vererbung – JOINED (1)

Aufbau und Mapping von Entities

→ Jede konkrete **und abstrakte** Klasse der Vererbungshierarchie wird in einer DB-Tabelle abgebildet
→ ein *Discriminator* zur Unterscheidung kann nötig sein (Hibernate braucht keinen)

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class AbstractBook implements Serializable {
```





Vererbung – JOINED (2)

Aufbau und Mapping von Entities

Vorteile

→ keine Verletzung der Datenintegrität (Verwendung von *not null*)
→ Assoziationen auf Superklasse möglich

Nachteile

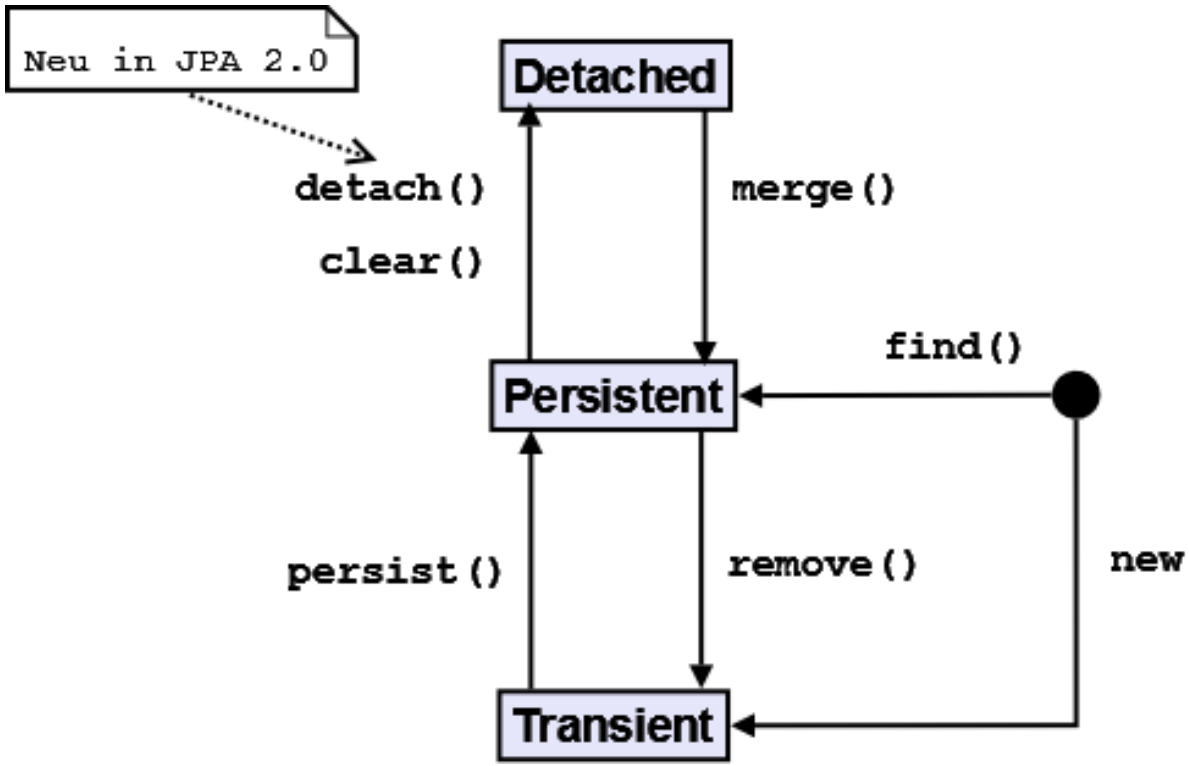
→schlechte Performance bei großen Vererbungshierarchien, da Verwendung von Joins bei Abfragen

Fazit

→ ausreichend bei polymorphen Abfragen durch Joins
→ geeignet bei Assoziationen auf Superklasse

Die Zustände einer Entity – ein Überblick

Lebenszyklus einer Entity



→ Zustände sind transparent, eine Entity kennt den eigenen Zustand nicht
→ EntityManger ist verantwortlich für korrekteZustandsübergänge der Entities

Die Zustände einer Entity – Transient

Lebenszyklus einer Entity

- eine transiente Entity ...
  - wird mit **new** erzeugt
  - wird nicht durch den EntityManager verwaltet
  - wird nicht persistent in der DB gespeichert
  - verhält sich wie jedes normale POJO (Garbage Collection)
  - bleibt unberührt von einem Rollback
  - hat noch keine automatisch generierten Primärschlüsselfelder
  - wird mit dem Aufruf von persist() auf dem EntityManager persistent, jede referenzierte Entity wird ebenfalls persistiert, je nach Kaskadierungsoption

Die Zustände einer Entity – Persistent

Lebenszyklus einer Entity

- → eine persistente Entity ...
  - → wird durch den EntityManager verwaltet (enthalten im Persistenzkontext)
  - → muss noch nicht in der DB existieren (erst mit commit() oder flush())
  - → hat auf jeden Fall einen Primärschlüssel
  - → ist Teil einer Transaktion, ein Rollback setzt den Zustand der Entity zurück
  - → wird automatisch bei Änderungen mit der DB synchronisiert (update/insert)
  - → kann mittels find() oder JPQL-Abfragen aus der DB geladen werden
  - → wird mit dem Aufruf von remove() auf dem EntityManager wieder transient

Die Zustände einer Entity – Detached

Lebenszyklus einer Entity

- eine detached Entity ...
  - entsteht beim Schließen des EntityManagers
  - entsteht beim Serialisieren und übertragen in einen anderen Prozess
  - wird nicht durch den EntityManager verwaltet
  - enthält persistente Daten, die veraltet sein können
  - kann mit dem Aufuf von merge() auf einem EntityManager wieder in den Persitenzkontext übernommen werden (Rückgabewert == persistente Entity)
  - kann beim Auflösen einer Referenz eine *LazyInitializationException* werfen

Die Zustände einer Entity – Callback Methoden

Lebenszyklus einer Entity

- Callback Methoden werden bei Zustandsänderungen aufgerufen
- Definition kann mit den folgenden Annotations direkt in der Entity erfolgen:
  - @PrePersist
  - @PostPersist
  - @PreRemove
  - @PostRemove
  - @PreUpdate
  - @PostUpdate
  - @PostLoad

```
@Entity
public class Login implements
Serializable {
...
@PrePersist
@PreUpdate
public void executePrePersist() {
System.out.println("2 - in @PrePersist");
}
// nur eine Annotation pro Typ pro Entity
// @PrePersist
// public void notAllowed() {
```

EntityManager per Request

Patterns für die Verwendung des EntityManagers

→ pro Nutzeranfrage wird ein EntityManager erzeugt

```
// Useranfrage trifft ein → EntityManager wird erzeugt
EntityManager em = ...
try {
   tx = em.getTransaction();
   tx.begin();// Transaktion wird gestartet
   ...
   // Datenbankoperationen für Request werden ausführen
   ...
   tx.commit(); // Transaktion wird geschlossen
} catch (Exception e) {
   if (tx!=null) tx.rollback(); //evtl. Abbruch der Transaktion
} finally {
   em.close(); // EntityManager wird geschlossen
}
```

→ gute Strategie für Multiuser-Anwendungen
→ bietet ausreichend hohe Skalierbarkeit und Performance

EntityManager per Conversation

Patterns für die Verwendung des EntityManagers

→EntityManager bleibt während Benutzerinteraktion geöffnet

```
EntityManager em = ... // EntityManager wird erzeugt
tx = em.getTransaction();
tx.begin();// Transaktion wird gestartet
//Laden einer Entity
User user = (User) em.find(User.class, new Long(42));
tx.commit(); // 1. Transaktion wird beendet
//Die Entity wird vom Benutzer verändert:
user.setFirstname("Max");
Transaction tx2 = em.getTransaction();
tx2.beginn(); // Start der zweiten Transaktion
em.lock(user, LockModeType.READ); // wurde die Entity von einer
                                  // parallelen Transaktion verändert?
em.flush(); // Änderungen in Datenbank übernehmen
tx2.commit(); // 2. Transaktion wird beendet.
em.close(); // EntityManager beenden
```

→ zu verwenden wenn während Benutzerinteraktion viele Entites transferiert werden

Antipattern

Patterns für die Verwendung des EntityManagers

- → EntityManager per Operation
  - → pro Operation eine Transaktion, dies entspricht AutoCommit
  - → keine Möglichkeit eines Rollback für zusammenhängende Operationen

- → EntityManager per Application
  - → EntityManager ist nicht Thread-safe → Synchronisation paralleler Zugriffe
  - → beim Auftreten einer Exception ist EM im inkonsistenten Zustand → Neustart der Anwendung
  - → EntityManager ist Cache für geladene Entities → große Datenmengen sammeln sich an

Das Query Interface

Datenbankabfragen mit JPQL

- das Query Interface ...
  - ist die zentrale Schnittstelle zum Erstellen und Ausführen von Datenbankabfragen mit JPQL (Java Persistence Query Language)
  - liefert bei Aufruf von getResultList() oder getSingleResult() die Ergebnissmenge
  - erfordert Casting, da die Rückgabewerte der Methoden nicht typisiert sind
  - wird ab JPA 2.0 durch ein TypedQuery Interface für typisierte Abfragen ergänzt
  - kann mit setMaxResults() die größe der Ergebnissmenge begrenzen
  - kann mit setFirstResult() die ersten n – Zeilen der Ergebnissmenge überspringen

```
EntityManager em = JpaUtil.getEntityManagerFactory().createEntityManager();
Query jQuery = em.createQuery("Select b from Book b");
```

Parameter Binding

Datenbankabfragen mit JPQL

→ mit dem Query Interface können Parameter den Abfragen übergeben werden

→ per Namen

```
jQuery = em.createQuery("Select b from Book b
                         where b.title = :title and b.ISBN = :isbn");
jQuery.setParameter("title", "Buch 1");
jQuery.setParameter("isbn", "1111");
```

→ per Index

```
jQuery = em.createQuery("Select b from Book b
                         where b.title = ?2 and b.ISBN = ?1");
jQuery.setParameter(2, "Buch 1");
jQuery.setParameter(1, "1111");
```

Definition benannter Abfragen

Datenbankabfragen mit JPQL

→ die Definition benannter Abfragen ist in den Metadaten möglich
→ erhöhen die Wartbarkeit und ermöglichen das mehrfache Verwenden von Abfragen

```
@Entity
@NamedQuery(name="booksonline.bo.Book.bookByIsbn",
query="Select b from Book b where b.ISBN=:isbn")
public class Book

...

jQuery = em.createNamedQuery("booksonline.bo.Book.bookByIsbn");
jQuery.setParameter("isbn", "1111");
System.out.println(((Book)jQuery.getSingleResult()).getTitle());
```

Grundaufbau der Abfragen

Datenbankabfragen mit JPQL

```
select b from booksonline.bo.Book as b
```

vollqualifizierter Name
und „as“ optional

```
select b from Book b
```

Navigation entlang der
Attribute möglich

```
select b.publisher.description from Book b
```

Angabe mehrerer Entities
ergibt kartesisches Produkt

```
select p,b from Publisher p, Book b
```

from → legt Wertebereich fest
select → definiert Rückgabewert
b → Identifikationsvariable

Einschränken der Ergebnismenge mit where

Datenbankabfragen mit JPQL

```
Select b from Book b where b.title = 'JPA mit Hibernate'
```

like Operator analog zu SQL

```
Select b from Book b where b.author like '%EN'
```

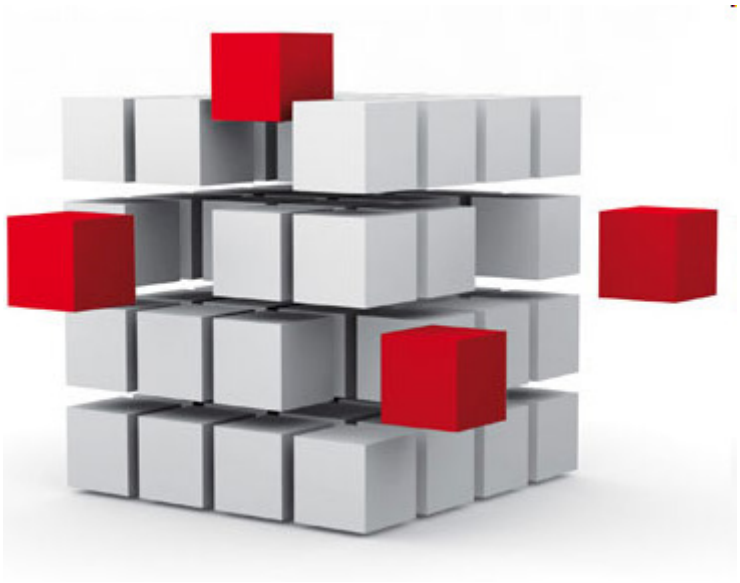
Mengenoperationen

```
Select b from Book b where b.ISBN in ('2222','4444')
```

Feature Liste

Datenbankabfragen mit JPQL

- Sortierung mit order by
- implizite, explizite und Fetch-Joins
- Aggregationsfunktionen
- Gruppieren mit group by
- Polymorphe Abfragen
- Subqueries
- Massen-Update und -Delete
- Natives SQL



Criteria Query API und Metamodell

Datenbankabfragen mit JPQL

- → Objekt-orientierte Abfragesprache
- → Dynamisches Erzeugen von Abfragen OHNE String Manipulation
- → Typ-Prüfungen beim Compilieren

mit Metamodell

+ jeder Knoten in Abfrage ist stark typisiert durch Generics
+ Code Vervollstädigung der Attribute möglich
- komplexer und technisch anspruchsvoller

ODER

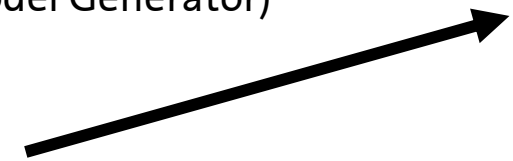
mit Strings

+ einfacher zu schreiben und zu lesen
+ keine Verwendung von Metamodell
- mögliche Schreibfehler

Statisches, kanonisches Metamodell

Datenbankabfragen mit JPQL

→ Generierung aus Annotationen (bspw. Hibernate Static Metamodel Generator)



Entity

```
@Entity public class Order {
  @Id Integer orderId;
  @ManyToOne Customer customer;
  @OneToMany Set<Item> lineitems;
  Address shippingAddress;
  BigDecimal totalCost;
  ...
}
```

Metamodell

```
@StaticMetamodel(Order.class)
public class Order_ {
public static volatile
  SingularAttribute<Order, Integer> orderId;
public static volatile
  SingularAttribute<Order, Customer> customer;
public static volatile
  SetAttribute<Order, Item> lineitems;
public static volatile
  SingularAttribute<Order, Address> shippingAddress;
public static volatile
  SingularAttribute<Order, BigDecimal> totalCost;
}
```

Dynamisches Metamodell

Datenbankabfragen mit JPQL

- Aufruf über em.getMetamodel oder emf.getMetamodel
- Zahlreiche Interfaces zur Darstellung von Entities und Attributen
  - EntityType, EmbeddableType, SingularAttribute, SetAttribute...
- Erweiterungen für objekt-relationales Mapping geplant

```
EntityManager em = ...;
Metamodel mm = em.getMetamodel();
EntityType<Employee> emp_ = mm.entity(Employee.class);
EmbeddableType<ContactInfo> cinfo_ = mm.embeddable(ContactInfo.class)
```

Criteria Query – Beispiele (1)

Datenbankabfragen mit JPQL

- Erzeugen von `CriteriaQuery` mit `CriteriaBuilder`
- `CriteriaBuilder` durch `em.getCriteriaBuilder()` oder `emf.getCriteriaBuilder()`
- `Root`s Basis der Abfrage, entsprechen `FROM` bei JPQL

```
CriteriaBuilder qb = em.getCriteriaBuilder;
CriteriaQuery<Customer> q = qb.createQuery(Customer.class);
Root<Customer> customer = q.from(Customer.class);
q.select(customer);
TypedQuery<Customer> tq = em.createQuery(q);
List<Customer> tq.getResultList();
```

Criteria Query – Beispiele (2)

Datenbankabfragen mit JPQL

```
SELECT i.name, p
FROM Item i JOIN i.photos p
WHERE KEY(p) LIKE '%egret%'
```

```
CriteriaQuery<Tuple> q = qb.createTupleQuery();
Root<Item> item = q.from(Item.class);
MapJoin<Item, String, Object> photo = item.join(Item_.photos);
q.multiselect(item.get(Item_.name), photo)
 .where(qb.like(photo.key(), "%egret%"));
```

Quellen

Anhang

- Literatur
  - Daniel Röder, JPA mit Hibernate – Java Persistence API in der Praxis
- Spezifikationen (www.jcp.org)
  - JSR 220: Enterprise JavaBeansTM 3.0
  - JSR 317: JavaTM Persistence 2.0
- Tutorial
  - http://download.oracle.com/javaee/5/tutorial/doc/

Der **Kontakt**

Anhang

→ **Dresden**

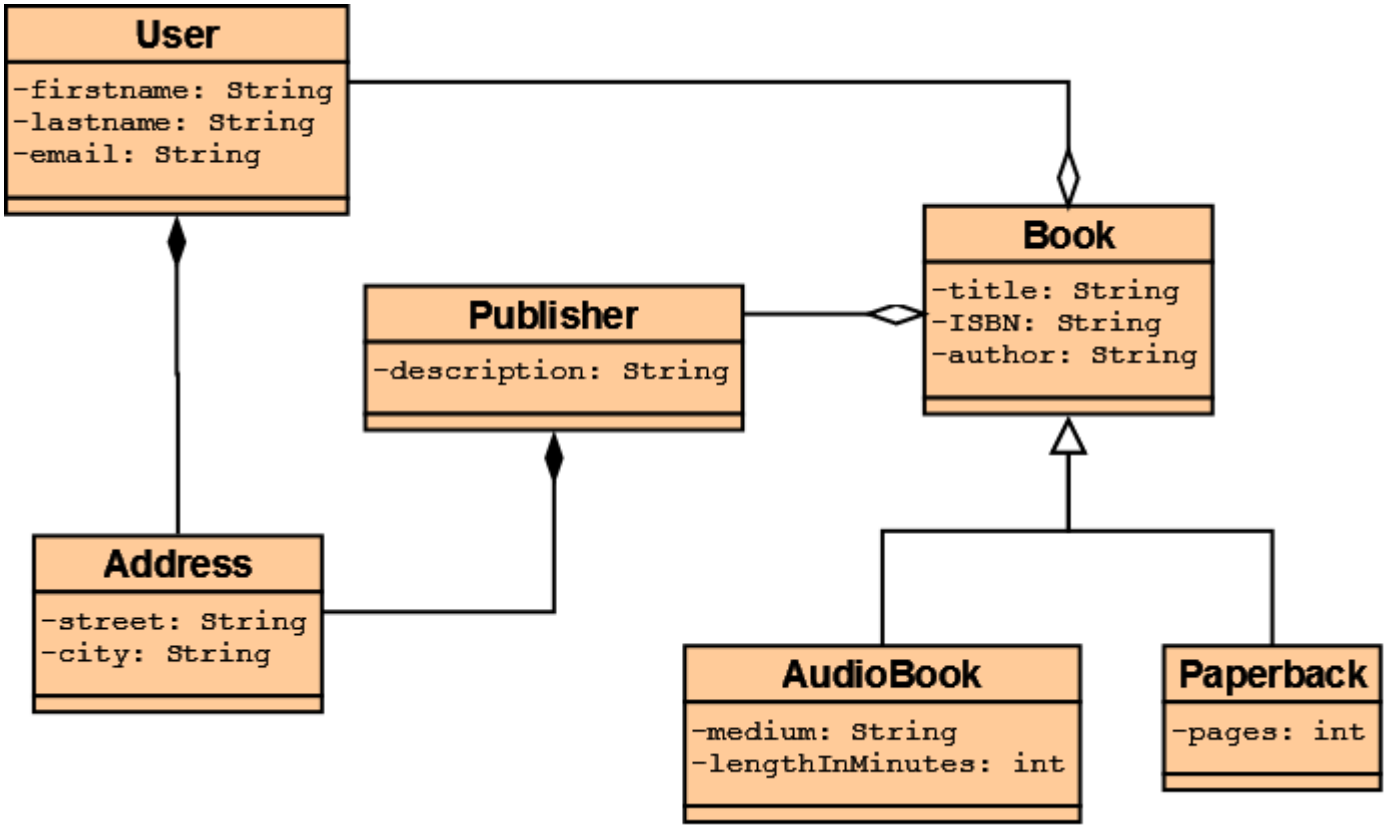
Fritz-Foerster-Platz 2, 01069 Dresden
Telefon: +49 (0)351 497 01-500
Telefax: +49 (0)351 497 01-589

→ **Mail** daniel.roeder@saxsys.de

→ **Görlitz**

Berliner Straße 63, 02826 Görlitz
Telefon: +49(0)3581 76 723-0
Telefax: +49(0)3581 76 723-29

Übung


User
-firstname: String
-lastname: String
-email: String
Book
-title: String
-ISBN: String
-author: String
Publisher
-description: String
Address
-street: String
-city: String
AudioBook
-medium: String
-lengthInMinutes: int
Paperback
-pages: int