Java Persistence with Hibernate

Version 1.1
2007-06
© s.metzler

Inhalt

1 Motivation 5

1.1 Impedance Mismatch 5
1.1.1 Granularität 5
1.1.2 Vererbung 5
1.1.3 Objektidentität 5
1.1.4 Beziehungen 6
1.1.5 Graphennavigation 6

2 OOA Thematik 7

2.1 Java 5 7
2.1.1 Annotations 7
2.1.2 Generics 10

3 Begriffserklärungen 13

3.1 Java Persistence API 13

3.2 EJB 3.0 13

3.3 Hibernate 14

4 Bestandteile eines O/R Mapping Frameworks 15

4.1 Entities 15
4.1.1 Primärschlüssel einer Entity 16
4.1.2 Embeddable Klassen 17
4.1.3 Lebenszyklus einer Entity 18

4.2 Relationships 21
4.2.1 bidirektionalen Beziehungen 22
4.2.2 Vererbung 23

4.3 Persistenzkontext 25
4.3.1 EntityManager 26
4.3.2 Entity Listener 30

4.4 Querys 32

5 Hibernate als Persistenzframework 35

5.1 Hibernate Annotations 35

5.2 Hibernate Validator 36

5.3 Hibernate EntityManager 37

6 Hibernate - Anwendungsbeispiel 38

6.1 Library als Beispielsanwendung 38
6.1.1 Anwendungsfälle 38
6.1.2 Klassendiagramm 38
6.1.3 Anforderungen an die Entwicklungsumgebung 39

6.2 Projektstruktur 39
6.2.1 Konfiguration 39
6.2.2 HibernateUtil Klasse 41
6.2.3 Logging mit Log4J 42

6.3 User Entity 42
6.3.1 Sichern 43
6.3.2 Laden 45
6.3.3 Aktualisieren 46
6.4 Hibernate XML-Mapping 46
6.5 Generatorstrategien 47
7 Architektur 52
7.1 Überblick 52
7.2 Services 52
7.3 Schnittstellen 53
7.3.1 SessionFactory 53
7.3.2 Session 53
7.3.3 Configuration 53
7.3.4 Transaction 53
7.3.5 ConnectionProvider 54
7.3.6 TransactionFactory 54
8 Lebenszyklus 55
8.1 Objektzustände 55
8.1.1 Transient 56
8.1.2 Persistent 56
8.1.3 Detached 57
8.2 Detached Entities 58
8.2.1 Limitationen von Detached Entities 59
8.2.2 Optimistisches Locking 59
8.2.3 Lazy Loading 60
9 Beziehungen 64
9.1 Komponenten 64
9.2 Assoziationen 69
9.2.1 1:1-Beziehungen 70
9.2.2 1:n und n:1-Beziehungen 73
9.2.3 n:m-Beziehungen 80
9.2.4 Transitive Persistenz 84
9.3 Vererbung 87
9.3.1 SINGLE_TABLE 88
9.3.2 TABLE_PER_CLASS 91
9.3.3 JOINED 92
10 Collections 95
10.1 Collection Mapping 96
10.2 Collections mit Index 99
10.3 Sortierte Collections 100
11 Hibernate Session und Transaktionen 102
11.1 Transaktion 102
11.2 Isolationsebenen 102
11.3 Transaktionen in Hibernate 104

11.4 Optimistisches Locking 105
11.5 Pessimistisches Locking 107

12 Datenbankabfragen 109
12.1 Query Interface 109
12.1.1 Methoden des Query Interface 110
12.1.2 Ausführen von Abfragen 110
12.1.3 dynamische Abfragen 111
12.1.4 namedQuery 112
12.2 HQL – Hibernate Query Language 113
12.2.1 from 113
12.2.2 where 113
12.2.3 order by 115
12.2.4 Verbundoperationen 115
12.2.5 select 116
12.2.6 Aggregat-Funktionen 117
12.2.7 group by 117
12.2.8 Polymorphe Abfragen 117
12.2.9 Subqueries 118
12.2.10 Criteria API 118
12.2.11 Native SQL Abfrage 122
12.2.12 Hibernate Filter 123

13 Hibernate Types 124
13.1 Hibernate Mapping Types 124
13.2 Benutzerdefinierte Mapping Types 125

14 Fetching und Caching 130
14.1 Fetching-Strategien 130
14.2 N + 1 Problematik 131
14.2.1 Batch-Fetching 132
14.2.2 Join-Fetching 133
14.2.3 Subselect-Fetching 134
14.3 Hibernate Query Cache 134
14.4 Second Level Cache 135
14.4.1 Strategien und Konfiguration 135
14.4.2 Second Level Cache Provider 136
14.5 Massen-Update und -Delete 136

15 Anhang 137
15.1 Hibernate Mapping Types 137
15.2 Hibernate SQL Dialects (`hibernate.dialect`) 138
15.3 Package org.hibernate 139
15.4 Literaturverzeichnis 142
15.5 Onlineverzeichnis 142

1 Motivation

- Schnittstelle zwischen relationalen Datenstrukturen und objektorientierten Klassenhirachien
- Definitionen der relationalen und objektorientierten Struktur sind grösstenteils identisch

1.1 Impedance Mismatch

Unterschied in der Struktur zwischen normalisierten relationalen Datenbanken und objektorientierten Klassenhierarchien:

- Granularität
- Vererbung
- Objektidentität
- Beziehungen
- Graphennavigation

1.1.1 Granularität

Beschreibt fein- und grobgranulare Architektur der Komponenten in den Computer-Architekturen:

- feingranularere Architektur ist klein und schnell bezüglich ihrer Abarbeitungszeit
- grobgranulare Architekturen haben lange Ausführungszeit, kleiner Datenaustausch, arbeiten selbständig.

Ein objektorientiertes Modell ist feingranular

Objekte können unterschiedlich granular sein, Tabellen hingegen sind bezüglich der Granularität beschränkt.

1.1.2 Vererbung

Vererbung ist in Programmiersprachen wie Java selbstverständlich.

Relationale Datenbanken kennen keine Vererbung

1.1.3 Objektidentität

Objektorientierte Umgebung

- zwei Objekte identisch, wenn beide dasselbe Objekt sind
- zwei Objekte haben identische Werte, sind sie gleich, aber nicht unbedingt identisch.

```
Object_A == object_B;          // Reference Sematitc
```

```
Object_A.equals(object_B);     // Value Sematic
```

Relationale Umgebung

- In relationalen Datenbanken wird ein Eintrag in einer Tabelle über die Daten identifiziert
  - damit lassen sich gleiche Datensätze finden
  - aber nicht sicherstellen, dass sie identisch sind

Um für ein Objekt den entsprechenden identischen Eintrag in der Datenbank zu finden, muss ein eindeutiger Primärschlüssel eingeführt werden.

1.1.4 Beziehungen

Mit einem Fremdschlüssel in der einen Tabelle wird ein Primärschlüssel in einer anderen Tabelle referenziert und somit die Tabellen in Beziehung gebracht.

In der objektorientierten Welt gibt es mehrere Arten von Beziehungen:

- 1 : 1
- 1 : n
- n : 1
- n : m

Diese können letztendlich alle mit Fremdschlüsseln abgebildet werden; bei der n:m Beziehung muss eine Beziehungstabelle (Join- Tabelle) eingeführt werden. Die Beziehungstabelle enthält zwei Fremdschlüssel, die jeweils auf eine Seite der Beziehung zeigen.

1.1.5 Graphennavigation

Über Java Objekte zu navigieren ist über die entsprechenden Methodenaufrufe implementiert.

- Eine Datenbankanfrage benötigt mehrere Zugriffe:
  - Abfrage Tabelle A
  - Beziehungen der Tabelle A (zu anderen Tabellen)

- Oder eine SQL-Join um die die Anzahl der Datenbankzugriffe zu reduzieren.

2 OOA Thematik

2.1 Java 5

hilfreiche und wichtige Neuerungen im Java-Standard:

2.1.1 Annotations

- Metadaten direkt im Sourcecode
- XML- oder Property-Dateien entfallen
- werden von Codegenerierungstools ausgelesen
- werden zur Laufzeit per Reflection abgefragt
- zentral im Sourcecode abgelegt (Entwickler Vorteil)
- Zuordnung zum Sourcecode ersichtlich

Verwendung von Annotations

- Annotations werden im Sourcecode vor dem zu markierenden Element eingefügt
- @ als Kennzeichnung vorangestellt (Leerzeichen sind erlaubt)
- Parameter in Klammern an den Annotationnamen angehängt

Annotationparameter werden als Name-Wert-Paar angegeben, wobei die Reihenfolge der Parameter keine Rolle spielt:

```
@MyAnnotation(parameter1="hello", parameter2="hibernate")
```

Hat die Annotation nur einen Parameter, kann der Name weggelassen werden. Bei parameterlosen Annotations ist die Angabe der Klammern optional.

Annotations können sich auf folgende Java-Elemente beziehen:

- Packages
- Klassen
- Interfaces
- Enumerations
- Methoden
- Variablen
- Methodenparameter

Annotations in der Java SE 5.0

- definiert sieben Annotations
  - Standard Annotation (normalen Verwendung beim Programmieren)
  - Meta Annotation (Definition neuer Annotationstypen)

Standard Annotations:

- @Deprecated:
  kennzeichnet Methoden und Klassen, die nicht mehr verwendet werden sollen.
- @Override:
  der Compiler stellt sicher, dass die Methode der Basisklasse überschrieben wird.
- @SuppressWarnings:
  Unterdrückt Compilerwarnungen. Warnungen werden als Parameter angegeben.

```
public class HelloWorld {
   @Deprecated
   public String sayHello() {
      return "Hello World";
   }

   @Override
   public String toString() {
      return "Hello World";
   }
}
```

- sayHello() ist veraltet und erzeugt Compilerwarnung
- toString() mit @Override stellt sicher, dass toString() aus Object überschrieben wird

Meta Annotations:

- @Documented:
  zur Erzeugung der Dokumentation bei Javadoc
- @Inherited:
  Annotation-Typ wird vererbt, auch rekursiv, falls keine Annotation in der aktuellen Klasse gefunden wird.
- @Retention:
  Definiert wie lange die Annotation verfügbar ist.
  - SOURCE – nur bis zur Compilezeit zur Verfügung
  - CLASS – werden in den Class-Dateien abgespeichert, aber nicht von der VM geladen.
  - RUNTIME – werden in der Class-Datei abgelegt und von der VM geladen und stehen somit zur Auswertung per Reflection zur Verfügung.
- @Target:

definiert Elemente (Klasse, Methode, Parameter etc.) denen eine Annotation zugeordnet werden kann.

Eigene Annotations

Beispiel:

```
package util;

public enum Datentyp {
   TEXT, NUMMER, DATUM
}
```

```
package util;

import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface Datenfeld {
   String bezeichnung() default "";
   Datentyp datentyp() default Datentyp.TEXT;
}
```

- @Retention(RetentionPolicy.RUNTIME) definiert die Verfügbarkeit
  - RUNTIME : Auswertung zur Laufzeit
  - CLASS : (DEFAULT) – Eintrag in der *.class Datei
  - SOURCE : verfügbar nur zur Compilezeit
- @Target({ElementType.FIELD}) definiert wo die Annotation eingesetzt werden kann
  - TYPE : auf Klassenebene
  - FIELD : für Attribute
  - METHODE : auf Methodenebene
- Parametrer
  - bezeichnung : optional, Default = ""
  - datentyp : optional, Defalut = Datentyp.TEXT

Verwendung:

Klasse Person annoniert Attribute als Datenfelder

```
package person;

import util.*;

public class Person implements java.io.Serializable {
```

```
  @Datenfeld(bezeichnung = "Nummer", datentyp = Datentyp.NUMMER)
  private int nr;
  @Datenfeld(bezeichnung = "Name", datentyp = Datentyp.TEXT)
  private String name;
  @Datenfeld
  private String beschreibung;
}
```

Klasse PersonClient liest die Annotationen aus.

```
package person;

import java.lang.reflect.Field;
import util.*;

public class PersonClient {

  public static void main(String[] args) {

    Person p = new Person();
    Field[] fields = p.getClass().getDeclaredFields();
    for (Field field : fields) {
      Datenfeld datenfeld = field.getAnnotation(Datenfeld.class);
      if (datenfeld != null) {
        System.out.println("\nannoiertes Attribut: "
            + field.getName());
        if (datenfeld.bezeichnung().length() != 0) {
          System.out.println("\tBEZ = "
            + datenfeld.bezeichnung());
        }
        System.out.println("\tTYP = " + datenfeld.datentyp());
      }
    }
  }
}
```

Ausgabe:

```
annoiertes Attribut: nr
  BEZ = Nummer
  TYP = NUMMER

annoiertes Attribut: name
  BEZ = Name
  TYP = TEXT

annoiertes Attribut: beschreibung
  TYP = TEXT
```

2.1.2 Generics

- erlauben die Abstraktion von Typen
- defineren Klassen und Methoden, die generisch, also unabhängig von einem konkreten Typ

Hibernate unterstützt bereits die Verwendung von Generics.

Verwendung von Generics

Collection Framework als Beispiel:

```
List v = new Vector();
v.add(new Double(1.0));
Double d = (Double)v.get(0);
```

- Cast auf Double ist notwendig denn der Rückgabewert von get(...) ist Object
- erlaubt inkompatible Typen in die Liste einzuführen

generischen Collections:

```
List<Double> v = new Vector<Double>();
v.add(new Double(1.0));
Double d = v.get(0);
```

- v ist eine Liste von Double
- Typ in spitzen Klammern definiert den konkreten Typparameter
- Verwendung wird durch Compiler sichergestellt
- unpassender Typ nicht mehr möglich
- Verstoss wird zur Compilezeit erkannt
- Casts entfallen, da Rückgabewert von get(...) nur Double
- mehrere Typparameter werden durch Komma getrennt (z. B. Hashtable<Long, String>)

Generics verbessert die Lesbarkeit und hilft durch die erhöhte Typsicherheit die Zuverlässigkeit des Codes zu erhöhen.

Wildcards und Bounds

- unabhängiger Code
- Implementierung von Such- oder Sortieralgorithmen
- Wildcardzeichen ist ?
- Vector<?> steht für einen Vector mit beliebigem Inhalt
- Bounds als Angabe von Schranken
- extends erlaubt den angegebenen Typ oder von dessen abgeleiteten Typ
- super erlaubt den angegebenen Typ oder dessen Supertyp

Beispiel:

```
Vector<?> v1;
```

```
Vector<String> v2 = new Vector<String>();
v1 = v2;
```

```
List<? extends Number> 100; // Number oder Subtyp
List<? super String> 2;     // String oder Supertyp
```

generische Typen

- Generics sind nicht auf die Verwendung bestehender Klassen beschränkt.
- eigene generische Typen können definiert werden.

Definition der generischen Klasse MyGeneric, welche zwei Typparameter (T und U) enthält. Die Typparameter können wie normale Typen bei der Definition der Klasse verwendet werden.

```
public class MyGeneric<T, U> {
   private T key;
   private U value;

   public MyGeneric(T key, U value) {
      this.key = key;
      this.value = value;
   }

   public T getKey() {
      return key;
   }

   public U getValue() {
      return value;
   }
}
```

Generische Methoden

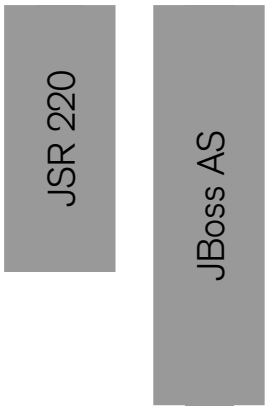
- Definition von generischen Methoden

Definition der generischen Methode sayHello(...), die zwei Typparameter enthält.

```
public class GenericMethod {
   public static <E, T> void sayHello(E pre, T post) {
      System.out.println(pre.toString() + " hello " +
        post.toString());
   }
   public static void main(String[] args) {
      sayHello("generic", new StringBuffer("world"));
   }
}
```

3 Begriffserklärungen

- JPA Java Persistence API
  - Spezifikation für das O/R Mapping
- EJB3
  - Spezifikation für Managed Persistence
- Hibernate
  - Impementation eines O/R Mapping Framework



3.1 Java Persistence API

- spezifiziert durch JSR 220
- spezifiziert Persistenz Framework
  - (Definition) aus Hibernate, TopLink und JDO entstanden
- nicht auf den Einsatz unter Java EE begrenzt
  - lauffähig unter Java SE
  - als normale Java Anwendungen
  - ausserhalb eines Java EE Containers
- Darstellung der Entitäten durch POJOs (Plain old Java Objects)
- Definition der Metadaten durch Annotations
  - XML (Deployment) Deskriptor-Dateien zur Angabe der Metadaten als Alternative
  - XML (Deployment) Deskriptor-Dateien überschreiben Annotationen

Ziel

- Java EE zu vereinfachen.

3.2 EJB 3.0

- Spezifikation für Managed Persistence
- Spezifikation standardisiert Java Persistenz API
- JSR 220 durch Java Community Process Programm: http://jcp.org/en/jsr/detail?id=220

Ziel

- Standardisierung der Basis-API
- Standardisierung von Metadaten eines objekt-relationalen Persistenz-Mechanismus
- definiert aber kein fertiges Framework, sondern ist Spezifikation

3.3 Hibernate

- Implementation eines O/R Mapping Framework
- Gavin King ist Gründer von Hibernate
- Hibernate Core
  - Definition mit XML
- Hibernate Annotation
  - Definition per Annotations
- Hibernate Entity Manager
  - Zugriff auf den Persistenzkontext

Hibernate ist als Open-Source ein objekt-relationaler Persistenz- und Query-Service:

> Hibernate is a powerful, high performance object/relational persistence and query service. Hibernate lets you develop persistent classes following object-oriented idiom- including association, inheritance, polymorphism, composition, and collections. Hibernate allows you to express queries in its own portable SQL extension (HQL), as well as in native SQL, or with an object-oriented Criteria and Example API.
>
> *http://www.hibernate.org*

4 Bestandteile eines O/R Mapping Frameworks

4.1 Entities

- leichtgewichtige, persistente Objekte
- zentraler Teil der Java Persistence API
- lösen die schwergewichtigen Entity Beans ab
- abstrakte wie auch konkrete Java-Klassen werden verwendet
- Vererbung, polymorphe Assoziationen und polymorphe Abfragen sind unterstützt
- nicht von einer bestimmten Basisklasse abgeleitet
- eigene Vererbungshierarchie kann verwendet werden

Entities und Java-Klassen können beliebig kombiniert werden; folgende Bedingungen müssen erfüllt sein:

- Entity-Klasse muss mit der Annotation @Entity markiert werden
- Entity-Klasse muss einen parameterlosen Konstruktor enthalten (public oder protected)
- Entity-Klasse muss eine Top-Level-Klasse sein, darf nicht als final deklariert sein
- Methoden oder persistente Attribute der Klasse dürfen nicht final sein
- Das Interface Serializable muss implementiert werden, falls Instanzen der Klasse "by-value" übertragen werden sollen
- Jede Entity muss einen Primärschlüssel enthalten

Innerhalb der Entity-Klassen können die persistenten Attribute durch Annotations mit Mapping- und anderen Metadaten versehen werden. Zwei Methoden stehen zur Verfügung:

- Direkte Markierung der Klassenattribute
- Markierung der Getter- Methoden

Die Markierung persistenter Attribute muss innerhalb einer Klasse identisch sein. Es dürfen also nicht beide Arten gleichzeitig innerhalb derselben Klasse verwendet werden. Ausserdem können keine Attribute als persistent markiert werden, die mit dem Javaschlüsselwort transient deklariert wurden. Als persistente Attribute einer Entity sind folgende Typen erlaubt:

- alle primitiven Javatypen und deren Wrapperklassen
- java.lang.String
- serialisierbare Javaklassen (z. B. java.math.BigInteger, java.math.BigDecimal, java.util.Date, java.util.Calendar, java.sql.Date, java.sql.Time, java.sql.Timestamp)
- selbstdefinierte serialisierbare Javaklassen
- byte[], Byte[], char[] und Character[]
- Enums

- java.util.Collection, java.util.Set, java.util.List, java.util.Map
- Collections von Entity-Klassen
- Embeddable Klassen

Beispiel Entity Person mit ManyToMany Beziehung zu Address:

```
package pojo;

import java.io.Serializable;
import java.util.*;

import javax.persistence.*;

@Entity
public class Person implements Serializable {

   @Id
   private Long id;
   private String name;

   @ManyToMany
   private Set<Address> address = new HashSet<Address>();

   public Person() {
   } // Default Konstruktor

   public Set<Address> getAddress() {
      return address;
   }

   public void setAddress(Set<Address> address) {
      this.address = address;
   }

   public Long getId() {
      return id;
   }

   public void setId(Long id) {
      this.id = id;
   }

   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }
}
```

4.1.1 Primärschlüssel einer Entity

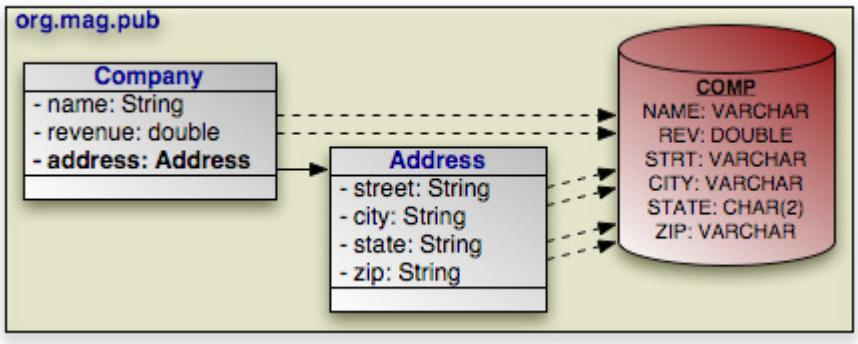
- mittels der Annotation @Id gekennzeichnet
- einfaches Attribut der Klasse
- alle primitiven Javatypen

- java.lang.String
- java.util.Date
- java.sql.Date
- Primärschlüsselklasse
  - die Klasse muss das Serializable Interface implementieren
  - die Methoden equals und hashcode müssen definiert werden
  - die Klasse muss public sein und einen parameterlosen Konstruktor enthalten
- zusammengesetzter Schlüssel
  - Instanzvariable vom Typ der Primärschlüsselklasse wird definiert (Embeddable Klasse)
  - Entity-Klasse enthält einzelne Felder des Primärschlüssels

Innerhalb einer Vererbungshierarchie von Entities darf nur ein Primärschlüssel definiert werden. Ausserdem muss er in der obersten Klasse der Hierarchie deklariert sein.

4.1.2 Embeddable Klassen

- dieselben Bedingungen wie die Entity-Klasse
- durch Annotation @Embeddable markiert (nicht @Entity)
- keinen Primärschlüssel, da ihre persistenten Attribute auf die gleiche Tabellenzeile gemappt werden wie die der sie umschliessenden Entity
- gehören direkt zur Entity und können nicht von mehreren Entity Objekten referenziert werden



http://openjpa.apache.org

Beispiel einer Embeddable Klasse

```
package util;

import java.io.Serializable;
import java.sql.Date;

import javax.persistence.*;

@Embeddable
```

```java
public class Zeitspanne implements Serializable {
   @Temporal(TemporalType.DATE) Date start;
   @Temporal(TemporalType.DATE) Date stop;
}
```

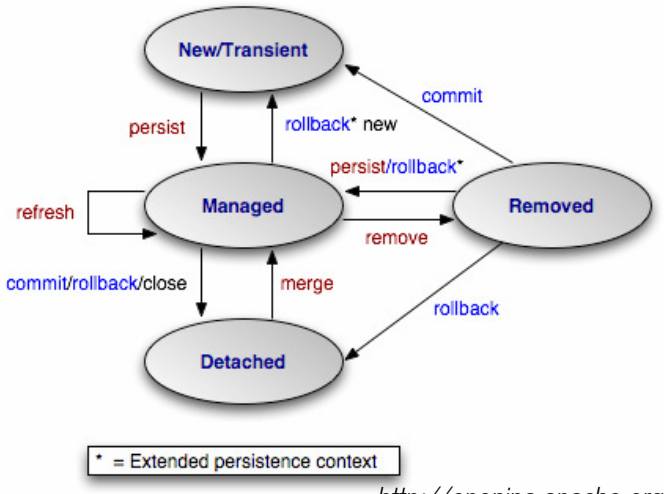
```java
package util;

import javax.persistence.*;

@Entity
public class Mitgliedschaft {
@Embedded
Zeitspanne zeitRaum;
}
```

4.1.3 Lebenszyklus einer Entity

- Verwaltung der Entity-Objekte durch Entity Manager
  - erzeugt persistente Entity-Instanzen
  - verwaltet persistente Entity-Instanzen
  - löscht persistente Entity-Instanzen
  - sucht mittels Queries persistente Entity-Instanzen

Eine Entity befindet sich immer in einem der folgenden vier Zustände:

- New / Transient
  - mittels new erzeugt
  - noch keinem Persistenzkontext zugeordnet
  - keine Repräsentation innerhalb der Datenbank
  - Eine eindeutige ID ist dem Objekt noch nicht zugewiesen worden
- Managed / Pesistent
  - besitzt entsprechende Repräsentation innerhalb der Datenbank
  - Instanz ist einem Persistenzkontext zugeordnet
  - wird von einem Entity Manager verwaltet



http://openjpa.apache.org

- Detached
  - hat eine Repräsentation innerhalb der Datenbank
  - ist keinem Persistenzkontext zugeordnet
  - Eine Entity befindet sich in diesem Zustand z. B. nachdem der Persistenzkontext in Folge eines Commits geschlossen wurde oder die Instanz per Remote Aufruf in eine andere VM übertragen wurde.
- Removed
  - hat eine Repräsentation innerhalb der Datenbank
  - ist einem Persistenz Kontext zugeordnet
  - ist zum Löschen freigegeben

Lifecycle Examples

Objekte persistieren:

```
// create some objects
Magazine mag = new Magazine("1B78-YU9L", "JavaWorld");
Company pub = new Company("Weston House");
pub.setRevenue(1750000D);
mag.setPublisher(pub);
pub.addMagazine(mag);
Article art = new Article("JPA Rules!", "Object Persistence");
art.addAuthor(new Author("Fred", "Hoyle"));
mag.addArticle(art);
// persist
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();
em.persist(mag);
em.persist(pub);
em.persist(art);
em.getTransaction().commit();
// or we could continue using the EntityManager...
em.close();
```

Objekte updaten:

```
Magazine.MagazineId mi = new Magazine.MagazineId();
mi.isbn = "1B78-YU9L";
mi.title = "JavaWorld";
// updates should always be made within transactions; note that
// there is no code explicitly linking the magazine or company
// with the transaction; JPA automatically tracks all changes
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();
Magazine mag = em.find(Magazine.class, mi);
mag.setPrice(5.99);
Company pub = mag.getPublisher();
pub.setRevenue(1750000D);
em.getTransaction().commit();
// or we could continue using the EntityManager...
em.close();
```

Objekte löschen:

```
// assume we have an object id for the company whose
// subscriptions we want to delete
Object oid = ...;

// deletes should always be made within transactions
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();
Company pub = (Company) em.find(Company.class, oid);
for (Subscription sub : pub.getSubscriptions())
   em.remove(sub);
pub.getSubscriptions().clear();
em.getTransaction().commit();
```

```
// or we could continue using the EntityManager...
em.close();
```

Objekte detachen und mergen: (Client-Server-Szenario)

```
//CLIENT:
//requests an object with a given oid
Record detached = (Record) getFromServer(oid);

...

//SERVER:
//send object to client; object detaches on EM close
Object oid = processClientRequest();
EntityManager em = emf.createEntityManager();
Record record = em.find(Record.class, oid);
em.close();
sendToClient(record);

...

//CLIENT:
//makes some modifications and sends back to server
detached.setSomeField("bar");
sendToServer(detached);

...

//SERVER:
//merges the instance and commit the changes
Record modified = (Record) processClientRequest();
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();
Record merged = (Record) em.merge(modified);
merged.setLastModified(System.currentTimeMillis());
merged.setModifier(getClientIdentityCode());
em.getTransaction().commit();
em.close();
```

4.2 Relationships

Die Java Persistenz API unterstützt Beziehungen (Relationships) zwischen Entities:

- 1 : 1 durch die Annotations @OneToOne abgebildet
- 1 : n durch die Annotations @ OneToMany abgebildet
- n : 1 durch die Annotations @ ManyToOne abgebildet
- n : m durch die Annotations @ ManyToMany abgebildet

Defaultmappings erlauben in den meisten Fällen um eine Beziehung zwischen zwei Entity-Klassen zu definieren.

Beispiel einer ManyToMany Beziehung Person-Address

```
@Entity
public class Person implements Serializable {

   @Id
   private Long id;
   private String name;

   @ManyToMany(targetEntity=Address.class)
   private Collection address;

   public Person() {
   } // Default Konstruktor

   public Collection<Address> getAddress() {
      return address;
   }

   public void setAddress(Collection<Address> address) {
      this.address = address;
   }
```

```
@Entity
public class Address {
   @ManyToMany(mappedBy = "address")
   Collection<Person> personen;
}
```

4.2.1 bidirektionalen Beziehungen

- unterscheiden zwischen "Besitzer" und der referenzierten Seite
- referenzierte Seite muss auf ihren Besitzer durch Angabe mappedBy der Relationship Annotation @OneToOne, @OneToMany oder @ManyToMany verweisen.
  - bei 1 : 1- Beziehungen enthält die Besitzerseite den Fremdschlüssel in der Datenbank.
  - "n"-Seite einer bidirektionalen Beziehung muss die "Besitzer"-Seite sein.
  - bei n : m-Beziehungen kann die Besitzerseite frei gewählt werden.

```
@Entity
public class Book {
   private Publisher publisher;

   @ManyToOne
   public Publisher getPublisher() {
      return publisher;
   }

   public void setPublisher(Publisher publisher) {
      this.publisher = publisher;
```

```
    }
}
```

```
@Entity
public class Publisher {
   private Collection<Book> books = new HashSet();

   @OneToMany(mappedBy = "book")
   public Collection<Book> getBooks() {
      return books;
   }

   public void setBooks(Collection<Book> books) {
      this.books = books;
   }
}
```

einfache 1:n Relation

4.2.2 Vererbung

- Polymorphe Assoziationen sind Beziehungen auf verschiedene konkrete Klassen innerhalb einer Vererbungshierarchie
- Polymorphe Abfragen liefern verschiedene konkrete Klassen einer Vererbungshierarchie zurück
- abstrakte Klassen können mit @Entity definiert werden
- abstrakte Klassen können nicht direkt instanziert werden
- Abfragen können sich auf abstrakte Entities beziehen, liefern Instanzen von konkreten Subklassen als Ergebnis

```
@Entity
@Table(name = "BOOK")
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class Book {
   @Id
   private Long id;
   private String title;
}
```

```
@Entity
@DiscriminatorValue(value = "Paperback")
public class Paperback extends Book {
   protected Integer priceDiscount;

   public Integer getPriceDiscount() {
      return priceDiscount;
```

```
    }
}
```

abstrakte Entity-Klasse

- Klasse Book ist als abstract deklariert
- Klasse Paperback ist eine konkrete Entity-Klasse, die von Book abgeleitet ist
- Klasse Paperback erbt alle Attribute von Book wie id und title

Eine Entity kann auch von einer Klasse abgeleitet sein, die selbst nicht als @Entity definiert ist und @MappedSuperclass und entsprechende Mapping-Annotations enthält.

- wird nicht direkt eine Tabelle zugeordnet
- Attribute werden auf die Tabellen der abgeleiteten Entity-Klassen gemappt
- kann nicht direkt mittels einer Abfrage gesucht werden
- Verwendung in einer Entity-Relationship ist nicht direkt möglich

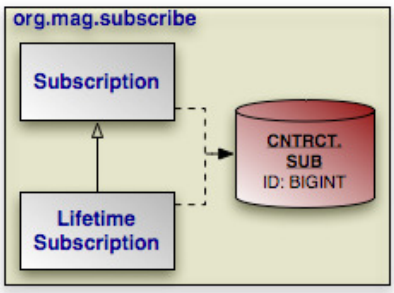
Das Ableiten von einer solchen nicht-Entity-Klasse ist z. B. dann sinnvoll, wenn mehrere Entities die gleichen Attribute enthalten sollen, denn die gemappten Attribute werden vererbt und sind Teil jeder Subklasse und werden dementsprechend beim Mapping aus der Datenbank berücksichtigt.

Entity-Klassen können auch von Klassen abgeleitet werden, die weder als @Entity noch als @MappedSuperclass markiert sind.

- werden nicht von einem Entity Manager verwaltet
- werden nicht persistent abgespeichert

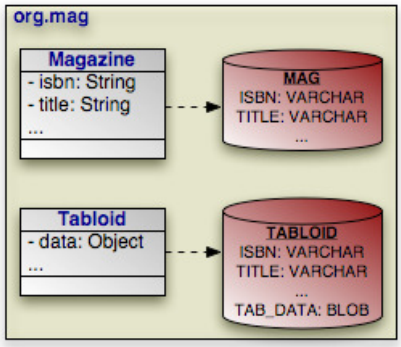
Für das Mapping der Entity-Hierarchien auf die konkreten Tabellen einer Datenbank stehen folgende Strategien zur Verfügung:

SINGLE_TABLE



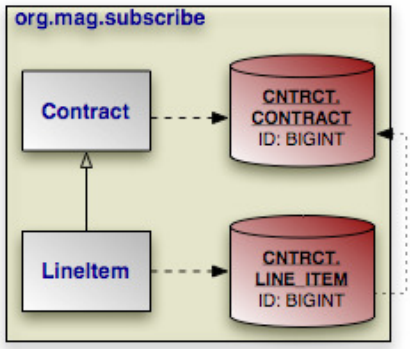
- eine Tabelle für die Abbildung einer Hierarchie
- Unterscheidungsmerkmal ist die Diskriminator Spalte
- Polymorphe Abfragen sind möglich

TABLE_PER_CLASS



- Eine Tabelle pro konkreter Entity-Klasse
- Unterstützung durch Implementierung optional
- jede Tabelle enthält ALLE Attribute einer Klasse inklusive der geerbten
- keine polymorphen Abfragen möglich
- SQL UNION als Behelf

JOINED

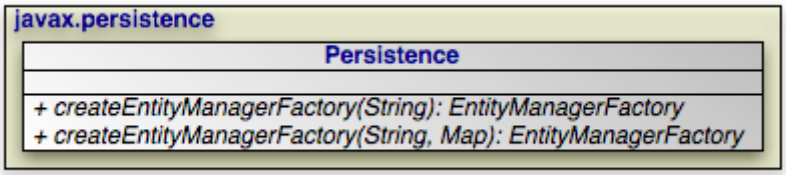


http://openjpa.apache.org

- eine Tabelle je Superklasse und eine Tabelle je abgeleitete Entity-Klasse
- eine Tabelle je Klasse einer Vererbungshirarchie
- jede Tabelle enthält nur spezifische Attribute der jeweiligen Klasse
- polymorphe Abfragen sind möglich (nicht sehr performant)

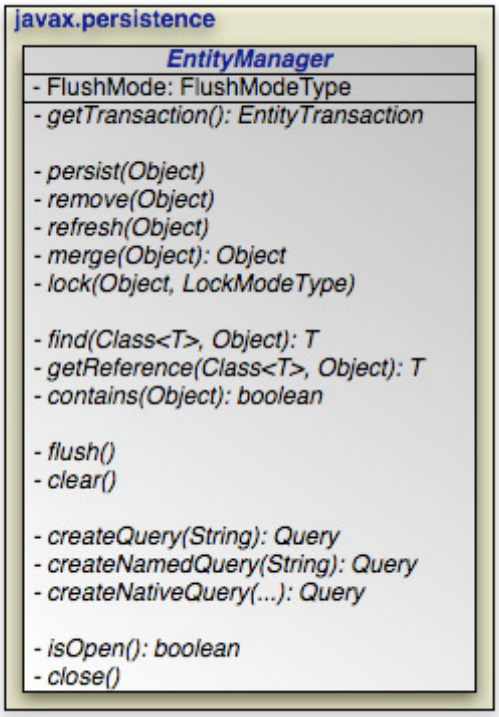
4.3 Persistenzkontext

- erzeugt EntityManager Factory
- definiert durch persistence.xml



http://openjpa.apache.org

4.3.1 EntityManager



http://openjpa.apache.org

- verwaltet die Entities
- stellt die API für den Zugriff auf einen Persistenzkontext bereit
- Persistenzkontext bezeichnet Menge von Entity-Objekten die innerhalb der Datenbank höchstens ein Java-Objekt im Kontext representieren
- einem Entity Manager ist immer genau ein Persistenzkontext zugeordnet, als:
  - Conntainer-Managed Entity Manager
  - Application-Managed Entity Manager

Container-Managed Entity Manager

- kommen innerhalb eines Java EE Containers zum Einsatz
- für das Erzeugen und Schliessen des Entity Managers verantwortlich
- der Java EE Container verwaltet JTA (Java Transaction API) Transaktionen
- Verwaltung geschieht transparent für die Anwendung
- Zugriff auf den Entity Manager erhält die Anwendung durch:
  - Dependency Injection
  - JNDI
- die Angabe der Annotation @PersistenceContext ist notwendig

Application-Managed Entity Manager

- Entity Manager wird von der Anwendung selbst verwaltet
- zum Erzeugen wird eine EntityManagerFactory verwendet
- Transaktionen durch:
  - JTA
  - lokale Transaktion (z. B. JDBC Transaktion)

Für die Gültigkeitsdauer eines Persistenzkontexts gibt es zwei Alternativen:

- transaction-scoped:
  - entspricht der Dauer einer Transaktion
  - nach Abschluss der Transaktion wird der Persistenzkontext geschlossen

- extended:
  - ist unabhängig von einer Transaktion
  - kann mehrere Transaktionen umfassen
  - der Persistenzkontext muss manuell (Entity Manager) geschlossen werden

Methoden der EntityManager API (Ausschnitt):

```
public void persist(Object entity);
```

Übergang zu managed Entity Instanz; commit() oder flush() persistiert Objekt in der Datensource.

- ist entity eine neue Entity wird diese zur managed Entity
- ist entity eine bestehende managed Entity so wird nichts geschehen
- ist entity eine removed Entity wird diese zur managed Entity
- ist entity eine detached Entity wird eine IllegalArgumentException geworfen
- gilt auch für relationale Felder von entity die mit CascadeType.PERSIST annotiert sind

```
public void remove(Object entity);
```

Übergang zu removed Entity Instanz; commit() oder flush() löscht Objekt in der Datensource.

- ist entity eine neue Entity so wird nichts geschehen
- ist entity eine bestehende managed Entity so wird diese gelöscht
- ist entity eine removed Entity so wird nichts geschehen
- ist entity eine detached Entity wird eine IllegalArgumentException geworfen
- gilt auch für relationale Felder von entity die mit CascadeType.REMOVE annotiert sind

```
public void refresh(Object entity);
```

Synchronisiert die Entity Instanz mit der Datensource. Änderungen innerhalb der Entity werden dabei überschrieben.

- ist entity eine neue Entity so wird nichts geschehen
- ist entity eine bestehende managed Entity so wird diese mit der Datensource synchronisiert
- ist entity eine removed Entity so wird nichts geschehen
- ist entity eine detached Entity wird eine IllegalArgumentException geworfen

- gilt auch für relationale Felder von entity die mit CascadeType.REFRESH annotiert sind

```
public void merge(Object entity);
```

Übergang von einer detached Entity zu einer managed Entity Instanz.

- ist entity eine detached Entity so wird diese in die existierende manged Instanz von entity kopiert oder eine neue Kopie einer manged entity Instanz erstellt
- ist entity eine neue Entity so wird eine neue managed Instanz entity* als Kopie erstellt
- ist entity eine bestehende managed Entity so wird nichts geschehen
- ist entity eine removed Entity wird eine IllegalArgumentException geworfen
- gilt auch für relationale Felder von entity die mit CascadeType.MERGE annotiert sind

```
public void lock(Object entity, LockModeType mode);
```

Sperrt die Entity Instanz:

- LockModeType.READ
  - andere Transaktionen können das Objekt lesen aber nicht ändern
- LockModeType.WRITE
  - andere Transaktionen können das Objekt weder lesen noch schreiben

```
public <T> T find(Class<T> entity, Object primaryKey);
```

Sucht ein Objekt:

- lädt die entity aus der Datenbank, welche dem Primärschlüssel zugeordnet ist
- liefert null, falls keine Entity gefunden wurde

```
public <T> T getReference(Class<T> entity, Object primaryKey);
```

Lädt ein Proxy (bzw. eine Referenz) einer Entity, welche durch den angegebenen Primärschlüssel eindeutig identifiziert wird

- übrige Felder der Entity können zu einem späteren Zeitpunkt nachgeladen werden
- Vorgehen ist sinnvoll, wenn eine Entity einer anderen zugeordnet werden soll und der

eigentliche Inhalt gar nicht benötigt wird

```
public boolean contains(Object entity);
```

Prüft ob sich die Entity im Persistenzkontext des Entity Managers befindet.

```
public void flush();
```

Synchronisiert die Datensource mit den Entity Objekten.

- weist den EntityManager an, den Zustand aller Entites die dem Persistenzkontext zugeordnet sind, mit der Datenbank zu synchronisieren
- Synchronisation geschieht dabei nur in eine Richtung; der Inhalt der Entity-Objekte im Persistenzkontext wird nicht aktualisiert, falls sich der Inhalt der entsprechenden Datenbankfelder geändert haben sollte.
- Synchronisation wird nur ausgeführt, wenn eine Transaktion aktiv ist

```
public void clear();
```

Leert den Persistenzkontext des Entity Managers.

- Persistenzkontext des Entity Managers wird geleert, alle verwalteten Entities werden detached Entities
- Synchronisation mit der Datenbank findet nicht statt

```
// emf = EntityManagerFactory ist Application Managed
// Neuen EntityManager erstellen
   EntityManager em = emf.createEntityManager();

// Transaktion starten
   em.getTransaction().begin();
   Book myBook = em.find(Book.class, new Long(1));
   myBook.setTitle("Hibernate 3.0");
   Book mySecondBook = new Book();
   mySecondBook.setId(2);
   mySecondBook.setTitle("Hibernate 3.2");
   em.persist(mySecondBook);
   em.getTransaction().commit();
// Ende der Transaktion
}
```

Verwendung des EntityManagers

- neuer EntityManager mit Hilfe der EntityManagerFactory wird erzeugt
- neue Transaktion wird begonnen
- find(...)-Methode lädt eine Instanz aus der Datenbank
- setTitle(...) weisst dieser Entity ein neuer Titel zu
- Änderungen sind aber noch nicht persistent in der Datenbank abgelegt
- neue Book-Instanz wird erzeugt und deren ID und Title gesetzt
- mit em.persist(...) definiert diese neue Entity zu einer verwalteten Entity
- mit em.getTransaction().commit() werden alle Änderungen in der Datenbank abgespeichert

Die Ausführung ist ausserhalb eines Java EE Containers ausgelegt, da die Erzeugung eines EntityManager sowie das manuelle Starten und Beenden einer Transaktion innerhalb eines Containers von diesem übernommen würde.

4.3.2 Entity Listener

Die neue Java Persistenz API bietet die Möglichkeit, über bestimmte Ereignisse während des Lebenszyklus einer Entity informiert zu werden. Hierfür sind zwei Mechanismen definiert:

deklarieren von Callback-Methoden

- Methoden werden unmittelbar nach oder vor dem jeweiligen Ereignis automatisch aufgerufen
- eignet sich um den Persistenzmechanismus durch beliebige eigene Funktionalitäten zu erweitern

Folgende Annotations stehen zur Verfügung:

- @PrePersist:
  - wird vor der Ausführung der persist Operation des EntityManagers aufgerufen
- @PreRemove:
  - wird vor der Ausführung von remove aufgerufen
- @PostPersist:
  - wird nach der Ausführung von persist aufgerufen
  - wird auch ausgeführt, nachdem die jeweiligen INSERT Statements innerhalb der Datenbank durchgeführt wurden
- @PostRemove:
  - wird nach der Ausführung von remove aufgerufen
- @PreUpdate:
  - wird aufgerufen, bevor ein UPDATE innerhalb der Datenbank ausgeführt wird
- @PostUpdate:
  - wird nach einem UPDATE-Statement ausgeführt

- @PostLoad:
  - wird aufgerufen, nachdem eine Entity in den aktuellen Persistenzkontext geladen wurde sowie nach jedem refresh der Entities

zuweisen einer Entity-Listener-Klassen

- mit Annotation @EntityListener
- Methoden werden automatisch aufgerufen

aufzurufende Methoden müssen folgende Bedingungen erfüllen:

- Callback-Methode muss die folgende Signatur haben

```
void <METHODENNAME>()
```

- Methode einer Entity-Listener-Klasse muss die folgende Signatur haben

```
void <METHODENNAME> (<KLASSE> o)
```

  - Typ des Parameters o muss der Klasse der Entity entsprechen, auf die sich die Entity-Listener- Klasse bezieht.

Definitionen für die Verwendung von Callback Methoden

- mehrere Methoden mit verschiedenen Callback Annotations können markiert werden
- mehrfache Markierung einer Methode durch verschiedene Callback Annotations ist zulässig
- mehrere Methoden können nicht mit der gleichen Annotation versehen werden, d. h., es kann immer nur eine Methode pro Ereignis definiert werden
- aber mehrere Listener pro Entity können definiert werden, Reihenfolge der Aufrufe hängt von der Reihenfolge der Definition innerhalb der @EntityListener Annotation ab
- gleichzeitige Verwendung von Entity Listener und Callback-Methoden gemeinsam in einer Entity ist möglich, Methoden der Entity Listener werden vor den Callback-Methoden aufgerufen

```
@Entity
public class Book1 {
   @PostPersist
   protected void postPersistBook() { ... }
}

@Entity
@EntityListeners(
```

```
    PaperbackListener.class, PaperbackListener2.class)
  public class Paperback extends Book { ... }

  public class PaperbackListener {
    @PostPersist
    protected void postPersistMethod1(Paperback book) {...}
  }

  public class PaperbackListener2 {
    @PostPersist
    protected void postPersistMethod2(Paperback book) {...}
}
```

Verwendung von Callback-Methoden und Entity Listenern

Das Beispiel zeigt die Verwendung von Callback-Methoden und Entity Listenern innerhalb einer Entity-Vererbungshierarchie. Wird ein Paperback mittels persist() in der Datenbank abgespeichert, werden die Listenermethoden in der folgenden Reihenfolge ausgeführt:

1. postPersistMethod1(...)
2. postPersistMethod2(...)
3. postPersistBook()

4.4 Querys

- können in SQL formuliert werden
- durch EJB Query Language (EJB QL) definiert (erweiterte Features)
  - Massen-Updates und –Deletes
  - Joins
  - Group By
  - Subqueries auszuführen
- EJB QL ist portable, da Datenbank unabhängig (gegenüber SQL)

Definition und Verwendung von Queries:

```
public Query createQuery(String ejbqlString)
```

- Erzeugt ein Query-Objekt, mit dem die Abfrage ausgeführt werden kann
- übergebene Query-String muss als EJB-QL-Abfrage formuliert sein

```
public Query createNamedQuery(String name)
```

- Erzeugt ein Query-Objekt aus einer so genannten Named Query
- vordefinierte Abfrage in EJB QL oder SQL, welche in den Metadaten (XML Dateien oder Annotations) hinterlegt wurde

```
public Query createNativeQuery(String sqlString)
```

- Erzeugt ein Query-Objekt aus einem in nativ SQL formulierten Query-String

```
public Query createNativeQuery(String sqlString, Class resultClass)
```

- Erzeugt ein Query-Objekt aus einem in nativ SQL formulierten Query-String
- Class Parameter resultClass bestimmt den Typ der Ergebnisse, welche von dieser Query zurückgeliefert werden

```
public Query createNativeQuery(String sqlString, String resultSetMapping)
```

- Erzeugt ein Query-Objekt aus einem in nativ SQL formulierten Query-String
- Mapping definiert die JDBC Ergebnisse auf die jeweiligen Entities zu mappen

Das Query Interface enthält eine Reihe von Methoden zur Angabe von Parametern sowie zur Durchführung der durch das Query-Objekt repräsentierten Abfrage. Queries können auch als so genannte "Named Queries" statisch innerhalb der XMLMetadaten oder als Annotation definiert werden:

```
@NamedQuery(
  name="findAllBooksWithTitle",
  query="SELECT b FROM Book b WHERE b.title LIKE :bookTitle"
)
```

Eine so definierte Query kann folgendermassen verwendet werden:

```
EntityManager em = ...
List books = em.createNamedQuery("findAllBooksWithTitle")
  .setParameter("bookTitle", "Hibernate").getResultList();
```

- Queries sind polymorph:
  - Abfrage liefert nicht nur die Instanzen der Entity-Klasse, sondern auch Instanzen von abgeleiteten Entity-Klassen, wenn diese die Abfragebedingungen erfüllen

- Queries in SQL formuliert:
  - wenn EJB QL nicht ausreicht
  - datenbankspezifische SQL Features

- nicht mehr datenbankunabhängig
- Ergebnis einer SQL formulierten Abfrage sind skalierbar

- SqlResultSetMapping deklariert die Javaklassen der Entities:

```
Query q = em.createNativeQuery(
  "SELECT b.id, b.title, b.publisher, p.id, p.name" +
  "FROM Book b, Publisher p " +
  "WHERE (b.title LIKE 'Hibernate%') " +
  "AND (b.publisher = p.id)" +
  ,"BookPublisherResults"
);

@SqlResultSetMapping(name="BookPublisherResults",
  entities={
     @EntityResult(entityClass=Book.class),
     @EntityResult(entityClass=Publisher.class)
  }
)
```

5 Hibernate als Persistenzframework

- implementiert die standardisierten Annotations
- deklariert Bedingungen durch Validators
- stellt mit dem EntityManager eine Persistenzengine zur Verfügung

5.1 Hibernate Annotations

- Verwendung der Hibernate Annotations (neu)
- Metadaten im XML-Format
- als XDoclet-Kommentare

Die Hibernate Annotations fügen eine weitere Möglichkeit hinzu, um für Hibernate relevante Metadaten zu deklarieren, welche sich auch mit den bisherigen Lösungen mischen lassen.

hibernate-spezifischen Annotations (zusammen mit EJB Annotations):

- @org.hibernate.annotations.Entity
  erlaubt Einstellungen, die über den Umfang der EJB3-Spezifikation hinausgehen. Ist kein Ersatz der Annotation @javax.persistence.Entity, sondern kann zusätzlich angegeben werden.
  - mutable definiert ob die Entity veränderbar ist
- selectBeforeUpdate
  definiert das Hibernate nie ein SQL UPDATE ausführt, ohne zu überprüfen, ob sich eine Entity tatsächlich verändert hat
- optimisticLock
  erlaubt die Festlegung der Lock Strategie:
  - OptimisticLockType.VERSION
  - OptimisticLockType.NONE
  - OptimisticLockType.DIRTY
  - OptimisticLockType.ALL
- @org.hibernate.annotations.GenericGenerator
  ermöglicht die Angabe eines Hibernate ID Generators
  mit der EJB3 Annotation @Id lassen sich so alle in Hibernate verfügbaren ID Generierungsmechanismen nutzen
- @org.hibernate.annotations.AccessType
  legt die Art, wie die Persistenz-Engine auf Attribute einer Entity zugreift, bis auf Attribute-Ebene individuell fest.
  z. B. kann ein bestimmtes Attribut definiert werden, auf das nur über die Getter- und Setter-Methoden zugegriffen wird, während auf die übrigen Attribute innerhalb der selben Entity direkt, also auf die Instanzvariablen, zugegriffen wird.

Die hier vorgestellten Annotations sind nur ein kurzer Auszug aus den Erweiterungen, die das Hibernate Annotations Projekt bereithält. So sind unter anderen auch zusätzliche Annotations für den Umgang mit Vererbung, Assoziationen, Collections und Queries enthalten.

5.2 Hibernate Validator

- ermöglicht Deklaration von Bedingungen, die eine gültige Entity erfüllen muss
- Constraints, werden als Annotations direkt in der betroffenen Entity deklariert, wobei sowohl Methoden wie auch Instanzvariablen markiert werden können
- entsprechende Validator Implementierung führt die tatsächliche Überprüfung aus
- Deklaration zentral, Überprüfung innerhalb einer mehrschichtigen Anwendung
- Constraints weren direkt über die Entity- Objekte überprüft, z. B. in Form von Entity Listenern, die vor einem Insert oder Update aufgerufen werden.

Die Hibernate Validator enthaltenen Constraints decken ein breites Spektrum ab:

- @Length: Überprüfung von Stringlängen
- @Max: Maximalwert für numerische Werte
- @Min: Minimalwert für numerische Werte
- @NotNull: Überprüfung auf NULL
- @Past: Überprüft, ob ein Datum in der Vergangenheit liegt
- @Future: Überprüft, ob ein Datum in der Zukunft liegt
- @Pattern: Überprüft, ob ein Attribut einem regulären Ausdruck entspricht
- @Range: Wertebereich für numerische Werte
- @Size: Überprüfung der Länge von Collections, Arrays und Maps
- @AssertFalse: Überprüft boolean Werte auf false
- @AssertTrue: Überprüft boolean Werte auf true
- @Valid: Führt eine Überprüfung des markierten Objekts durch

Definition eines neuen Constraints:

- Definition einer Annotation
- Implementierung einer Validatorklasse

```
@Entity
public class Book implements Serializable {

  @Id
  private Long id;
```

```
    @Length(max=100)
    @NotNull
    private String title;
    ...
}
```

Hibernate Validator zur Überprüfung einer Book-Entity

Der Titel eines Buches darf maximal 100 Zeichen lang sein und darf nicht null sein.

```
ClassValidator bookValidator = new ClassValidator( Book.class );
InvalidValue[] validationMessages = addressValidator.getInvalidValues(book);
```

Definition von Constraints

Für jede verletzte Bedingung liefert ein so genannter ClassValidator ein InvalidValue- Objekt, welches Details über die nicht erfüllte Bedingung enthält. Die ClassValidator Instanz sollte normalerweise nur einmal pro Anwendung erzeugt und danach gecached werden.

5.3 Hibernate EntityManager

- implementiert durch die Java Persistence API
- Einsatz innerhalb eines Java EE Containers
- Einsatz als Standalone Lösung ausserhalb eines Containers

Hibernate EntityManager setzt auf den Hibernate Core auf und verwendet diesen als Persistenzengine.

6 Hibernate - Anwendungsbeispiel

- Architektur von Hibernate
- wichtigste Klassen bzw. Schnittstellen von Hibernate
- Onlinebibliothek zum Verwalten von Medien
- Referenzbeispiel aus "Hibernate und die Java Persistence API"

6.1 Library als Beispielsanwendung

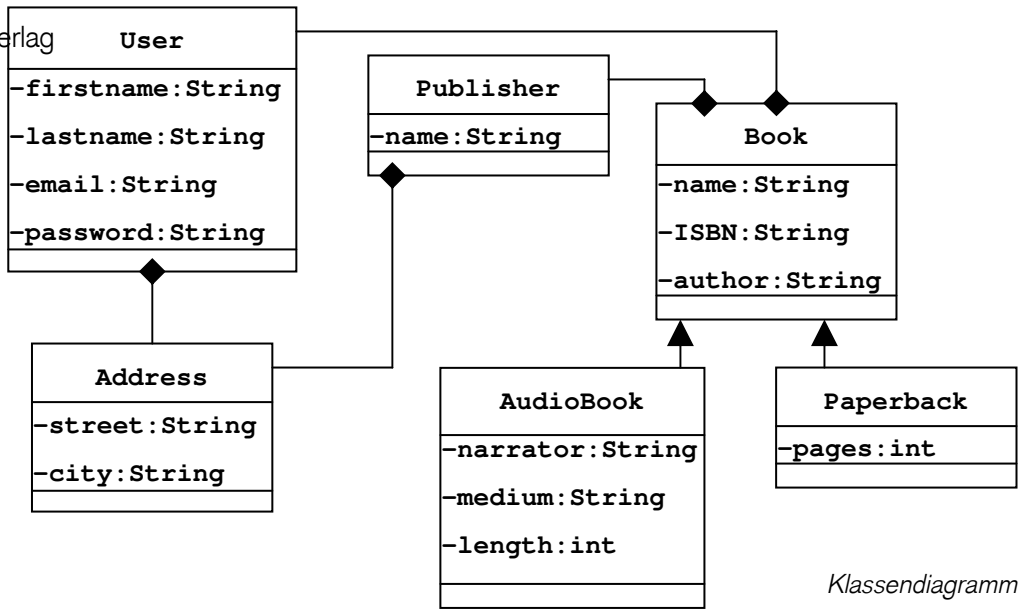
- Bücher verwalten
- Benutzer anlegen
- Suchfunktionen

6.1.1 Anwendungsfälle

- Grundlegende Funktionen der Anwendung:
  - Registrierung eines Benutzers
  - Anmelden eines Benutzers
  - Erfassen eines Buches (Buchname, ISBN, Verlag, Autor)
  - Suche nach Büchern
  - Ausleihe und Rückgabe eines Buches (nur als Benutzer)

6.1.2 Klassendiagramm

- Klassen bzw. Entities:
  - Benutzer mit Adresse (Email-Adresse als eindeutiger Identifikator)
  - Buch (eindeutige ISBN) als Hörbuch oder Taschenbuch
  - Autor
  - Verlag



Klassendiagramm

6.1.3 Anforderungen an die Entwicklungsumgebung

- Java Standard Edition (SE) 5.0
  http://java.sun.com/javase/downloads/index_jdk5.jsp
- Ant als Build-Tool
  http://ant.apache.org
- Eclipse als Entwicklungsumgebungen
  http://www.eclipse.org
- MySQL als Datenbank
  http://www.mysql.org
- JDBC-Treiber (Java Database Connectivity API)
  http://java.sun.com/products/jdbc/overview.html
- das Hibernate Annotations Package
  http://www.hibernate.org
- das Hibernate Core Package
  http://www.hibernate.org

6.2 Projektstruktur

Die Verzeichnisstruktur sieht wie folgt aus:

```
/library/lib
/library/src
/library/bin
```

JARs aus Hibernate Annotations (/lib-Verzeichnis):

- hibernate-annotations.jar
- ejb3-persistence.jar

JARs aus Hibernate Core (/lib-Verzeichnis):

- Readme im /lib-Verzeichnis von Hibernate Core

JDBC-Treiber (/lib-Verzeichnis):

- http://www.mysql.org

6.2.1 Konfiguration

- hibernate.properties
- hibernate.cfg.xml

Hibernate XML-Konfigurations Datei

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-
3.0.dtd">

<hibernate-configuration>
  <session-factory>
    <!-- Database connection settings -->
    <property name="connection.driver_class">
      com.mysql.jdbc.Driver</property>
    <property name="connection.url">
      jdbc:mysql://localhost:3306/library</property>
    <property name="connection.username">root</property>
    <property name="connection.password"></property>
    <!-- SQL dialect -->
    <property name="dialect">
      org.hibernate.dialect.MySQLDialect</property>
    <!-- Disable the second-level cache  -->
    <property name="cache.provider_class">
      org.hibernate.cache.NoCacheProvider</property>
    <!-- Echo all executed SQL to stdout -->
    <property name="show_sql">true</property>

    <!-- Create the database schema on startup -->
    <property name="hbm2ddl.auto">update</property>

    <!-- Drop and re-create the database schema on startup -->
    <!--property name="hbm2ddl.auto">create</property>-->

    <!-- Mapping beschrieben durch XML -->
    <!--<mapping resource="library/User.hbm.xml"/>-->

    <!-- Mapping beschrieben durch Annotations -->
    <mapping class="library.User"/>

  </session-factory>
</hibernate-configuration>
```

hibernate.cfg.xml

hibernate.cfg.xml definiert:

- JDBC-Treiber-Klasse
- Datenbank-URL mit Username, Passwort
- SQLDialekt, zum Beispiel für
  - PostgreSQL: org.hibernate.dialect.PostgreSQLDialect
  - MySQL (InnoDB): org.hibernate.dialect.MySQLInnoDBDialect
  - Oracle 9i/10g: org.hibernate.dialect.Oracle9Dialect
  - HypersonicSQL: org.hibernate.dialect.HSQLDialect
- Cache Provider (Caching-Strategie)
- Mapping beschrieben durch:

- XML Datei : z.B. User.hbm.xml
- Annotations : z.B. User.java (POJO)

- Datenbankeinstellungen:
  - show_sql = true : generierten SQLs auf Konsole
  - hbm2ddl.auto = update : Anwendungsstart generiert die Tabellen
  - hbm2ddl.auto = create : löscht Tabellen vor dem Erstellen

hibernate.properties ist im /etc von Hibernate Core

Hibernate.cfg.xml und hibernate.properties müssen im Klassenpfad sein.

6.2.2 HibernateUtil Klasse

Session aus SessionFactory wird benötigt um:

- Hibernate Objekt zu sichern
- Hibernate Objekt zu laden
- Hibernate Objekt zu aktualisieren
- ermöglicht den Zugriff auf Datenbankverbindungen
- erlaubt Transaktionssteuerung und Transaktionskontrolle
- bietet CRUD-Operationen (create, read, update und delete)

Session-Factory :

- aus Hibernate Konfiguration
- erzeugt beim Start der Anwendung
- zur gesamten Laufzeit der Anwendung erreichbar

Damit zur Laufzeit der Zugriff auf eine Session möglichst leicht ist, wird eine Klasse HibernateUtil erstellt, die mit einer statischen Methode eine Session zurückgibt.

```
package util;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.AnnotationConfiguration;
public class HibernateUtil {
  private static final SessionFactory sessionFactory;
  static {
    try {
      // Create the SessionFactory hibernate.cfg.xml
      sessionFactory = new AnnotationConfiguration().configure()
          .buildSessionFactory();
    } catch (Throwable ex) {
      System.err
          .println("Initial SessionFactory creation failed." + ex);
      throw new ExceptionInInitializerError(ex);
    }
```

```
    }
    public static SessionFactory getSessionFactory() {
      return sessionFactory;
    }
}
```

HibernateUtil.java

6.2.3 Logging mit Log4J

- Open Source Logging Framework Log4j
- (http://logging.apache.org/log4j/docs/index.html)
- LOGgausgaben von Hibernate im Debug-Level in der Datei library.log

```
log4j.rootLogger=DEBUG

log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d %5p [%t] (%F:%L)
- %m%n

log4j.appender.LIBRARY=org.apache.log4j.RollingFileAppender
log4j.appender.LIBRARY.File=library.log
log4j.appender.LIBRARY.MaxFileSize=512KB
log4j.appender.LIBRARY.MaxBackupIndex=5
log4j.appender.LIBRARY.layout=org.apache.log4j.PatternLayout
log4j.appender.LIBRARY.layout.ConversionPattern=%d %p [%t] %c -
%m%n

log4j.logger.org.hibernate=DEBUG, LIBRARY
log4j.logger.booksonline=DEBUG, LIBRARY
```

log4j.properties

6.3 User Entity

```
package library;
import javax.persistence.*;
@Entity
public class User {
   private Long id;
   private String firstname;
   private String lastname;
   private String email;
   private User() {}
   public User(String firstname, String lastname, String email) {
      this.firstname = firstname;
      this.lastname = lastname;
      this.email = email;
   }
   public Long getId() {
      return id;
   }
   private void setId(Long id) {
```

```
    this.id = id;
  }
  public String getFirstname() {
    return firstname;
  }
  public void setFirstname(String firstname) {
    this.firstname = firstname;
  }
  public String getLastname() {
    return lastname;
  }
  public void setLastname(String lastname) {
    this.lastname = lastname;
  }
  public String getEmail() {
    return email;
  }
  public void setEmail(String email) {
    this.email = email;
  }
}
```

User.java

einfaches POJO (Plain Old Java Object) mit:

- Annotations der Java Persistence API
- ohne externen Abhängigkeiten
- Getter- und Setter-Methoden
- Java- Beans-Nameskonventionen
- id als eindeutigen Bezeichner für persistente Klasse
- Standardkonstruktor (auch private)
- @Entity vor der Klassendefinition
- @Id vor dem Getter für id (Primärschlüssel)
- @Transient für nicht persistente Felder

6.3.1 Sichern

Klasse User erweitern mit:

- @Table (auf Klassenebene)
- @GeneratedValue (Strategie zur Generierung des Primärschlüssel)

  Veranwortung für das Generieren des PrimaryKey an Hibernate delegiert!

```
package library;
import javax.persistence.*;
@Entity
@Table(name = "T_USER")
public class User {
  private Long id;
  private String firstname;
```

```
   private String lastname;
   private String email;
   private User() {}
   public User(String firstname, String lastname, String email) {
      this.firstname = firstname;
      this.lastname = lastname;
      this.email = email;
   }
   @Id
   @GeneratedValue
   public Long getId() {
      return id;
...
```

User.java

SaveUser persistiert die Entity User:

- neuer User wird mit new angelegt
- eine Session wird über die Hilfsklasse HibernateUtil geöffnet
- eine Transaktion wird gestartet
- das Objekt der Session übergeben mit session.persist(user)
  Hibernate generiert Datensatz, setzt die Id und schreibt das Objekt in die Datenbank
- Transaktion wird mit commit beendet und der neue User persisitiert

```
package library;
import org.hibernate.Session;
import org.hibernate.Transaction;
public class SaveUser {
   public static void main(String[] args) {
      User user = new User("my_First", "my_Last",
           "my_Email@email.com");
      Session session = HibernateUtil.getSessionFactory()
           .openSession();
      Transaction tx = session.beginTransaction();
      session.persist(user);
      tx.commit();
      session.close();
   }
}
```

SaveUser.java

entsprechendes SQL (generiert durch Hibernate):

```
Hibernate:
insert into USER_TABLE (firstname, lastname, email) values (?,?,?)
```

Insert-Statement wird ausgeführt durch :

- tx.commit()
- als Alternative :

library:t_user

id	firstname	lastname	email
1	my_First	my_Last	my_Email@email.com

- Session.save(Object obj)
- oder : Session.saveOrUpdate(Object ojb), zum Aktualisieren eines Objekts

6.3.2 Laden

- Session öffnen
- Transaktion starten
- User Entity laden mit:
  - Session session.get(Class class, Serializable id)
  - oder: Session session.load(Class class, Serializable id)
- beide Methoden benötigen die Id des zu ladenden Datensatzes

```
package library;
import org.hibernate.Session;
import org.hibernate.Transaction;
public class LoadUser {
  public static void main(String[] args) {
     Session session = HibernateUtil.getSessionFactory()
          .openSession();
     Transaction tx = session.beginTransaction();
     User user = (User) session.get(User.class, new Long(1));
     System.out.println("User gefunden : " + user.getFirstname());
     tx.commit();
     session.close();
  }
}
```

LoadUser.java

get() vs load()

- falls das Objekt nicht existiert
  - get() gibt immer null zurück
  - load() wirft eine Exception

- Zugriff über Proxy
  - load() kann einen Proxy zurückgeben, Platzhalter für das eigentliche Objekt
  - sobald Zugriff auf Proxy erfolgt, wird eigentliches Objekt nachgeladen
  - erst dann erfolgt Datenbankzugriff, falls nötig
  - falls Proxy aus Cache gebildet wird, kann beim Nachladen festgestellt werden, ob Objekt existiert oder nicht (Exception)

- Objekt sicher existent
  - load() genügt
  - sonst sicherer mit get()

6.3.3 Aktualisieren

- session.update(Object entity) aktualisiert Objekt
- falls Entity Objekt in die Session geladen wurde:
  - Hibernate aktualisiert Objekt
  - sonst muss Objekt mit update() der Session hinzugefügt werden

```
package library;
import org.hibernate.Session;
import org.hibernate.Transaction;
public class UpdateUser {
   public static void main(String[] args) {
      Session session = HibernateUtil.getSessionFactory()
           .openSession();
      Transaction tx = session.beginTransaction();
      User user = (User) session.get(User.class, new Long(1));
      System.out.println("User geändert : " + user.getFirstname());
      user.setEmail("new_email@email.com");
      session.update(user);
      tx.commit();
      session.close();
   }
}
```

UpdateUser.java

durch Hibernate generiertes Update-Statement:

```
Hibernate:
update USER_TABLE set firstname=?, lastname=?, email=? where id=?
```

6.4 Hibernate XML-Mapping

Mapping von Entities auf Tabellen :

- XML-Mapping (in Hibernate Versionen vor 3.1 ausschliesslich)
- XDoclet aus speziellen Tags im JavaDoc
  (http://xdoclet.sourceforge.net/xdoclet/index.html)
- eigene XML-Datei mit der Endung .hbm.xml

Das XML-Mapping für die Klasse User:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="library">
  <class name="User" table="T_USER">
    <id name="id">
      <generator class="native"/>
    </id>
    <property name="firstname"/>
    <property name="lastname"/>
    <property name="email"/>
  </class>
</hibernate-mapping>
```

User.hbm.xml

- beginnt mit <hibernate-mapping>(optionale package Deklaration)
  - **oder**: vollqualifizierte Klassenname in <class>
- Generator native ist Default-Generator (auch bei Annotation @GeneratedValue)
  - Strategie zur Generierung der Primärschlüssel
  - z.B. Sequences bei Oracle
- persistent Felder als <property>
  (Annotation @Transient bezeichnet Felder die nicht persistent sind)

Ändern der Mapping Informationen (hibernate.cfg.xml)

```
    <!-- Mapping beschrieben durch XML -->
    <mapping resource="library/User.hbm.xml"/>
```

hibernate.cfg.xml

6.5 Generatorstrategien

- persistente Datenbankobjekte haben Datenbankidentität
- zwei gleiche Objekte besitzen denselben Primärschlüssel
- Verwalten der Primärschlüssel übernimmt Hibernate (Normalfall)
- oder definieren des Primärschlüssel (natürliche Schlüsseln, z.B. Personalnummer)

Anforderungen an Primärschlüssel

Entity User definiert Primärschlüssel vom Type Long.

Grundsätzliche Anforderungen an den Primärschlüssel bestehen:

- darf nicht null sein
- kann nie verändert werden (natürlicher Schlüssel ?)
- ist in einer Tabelle über alle Einträge eindeutig

Datenbankidentität, Objektidentität, Objektgleichheit

- Objektidentität (JAVA : ==) Referenz-Semantic :
  - zwei Objekte sind identisch, wenn sie dieselbe Adresse im Speicher der Java VM (Virtual Machine) haben.
- Objektgleichheit (JAVA : equals()) Value-Semantic :
  - zwei Objekt sind gleich, wenn sie denselben Inhalte wie in der Methode equals() definiert haben. Falls equals() aus der Klasse java.lang.Object nicht überschrieben wurde, wird die Objektidentität zurückgegeben.

Praxis

- equals() und hashcode() immer überschreiben
  - equals() soll geschäftsrelevante, eindeutige Daten vergleichen, z.B. Email
  - hascode() soll gute Verteilung definieren (mehr, aber echte Daten)

Problematisch bei

- Objekte in Collections (Sets, Maps, etc.)
  - da Implementation durch java.util.HashSet
- einbinden von Detached Objekten
  - bestehendes Objekt wird mit neuer Session verlinkt

(mögliche) Implementierung von equals() und hashcode() für die Klasse User:

```
@Override
public boolean equals(Object obj) {
   if (this == obj) return true;
   if (!(obj instanceof User)) return false;
   final User user = (User) obj;
   return !(email != null ? !email.equals(user.email)
        : user.email != null);
}
@Override
public int hashCode() {
   return 29 * (email != null ? email.hashCode() : 0);
}
```

User.java (equals() und hashcode())

Primärschlüssel ist für Objektgleichheit nicht geeignet:

- User hat n Adressen
- zwei Adressen (Rechnungsadresse und Lieferadresse) werden hinzugefügt (über Collections!)
- Objektgleichheit über Primärschlüssel bestimmt überschreibt die erste Adresse beim Hinzufügen der zweiten Adresse

- Primary Key wird generiert durch die Datenbank
- Primary Key ist NULL solange das Objekt nicht persistiert ist

- beide Adressen sind transient und Primärschlüssel null

```
    // erste Adresse hinzufügen
    user.addAddress(new Address("Street 1", "City 1"));
    // zweite Adresse hinzufügen
    user.addAddress(new Address("Street 2", "City 2"));
...
    // erzeugt nur eine Adresse für den User
    session.update(user);
```

Generators für die Primärschlüssel

- Annotation @Id definiert Feld als Primärschlüssel
- Annotation @GeneratedValue wählt ein Id-Generator

Generatorname	Beschreibung
AUTO	wählt entsprechend der darunterliegenden Datenbank eine Strategie (TABLE, IDENTIY, SEQUENCE). Entspricht der Hibernate-Generatorstrategie **native**.
TABLE	IDs sind in einer eigenen Tabelle. Entspricht der Hibernate-Generatorstratgie **hilo**. (Hi/Lo Algorithmus)
IDENTITY	unterstützt Identity Columns, die es beispielsweise in MySQL, HSQLDB, DB2 und MS SQL Server gibt. Entspricht der Hibernate-Generatorstrategie **identity**.
SEQUENCE	unterstützt Sequences, die es beispielsweise in PostgreSQL, Oracle und Firebird gibt. Entspricht der Hibernate-Generatorstrategie **sequence.**

ID-Generatoren der Java Persistence API

Hibernate bietet Hibernate-eigene Annotation (@org.hibernate.annotations.GenericGenerator) für die Definition weiterer Generatoren an.

```
@Id
@GeneratedValue(generator = "useridgen")
@GenericGenerator( name = "useridgen",
                   strategy = "seqhilo",
                   parameters = {
     @Parameter( name = "max_lo", value = "5"),
     @Parameter(name = "sequence", value = "mysequence") })
public Long getId() {
   return this.id;
}
```

@GenericGenerator parametrisiert:

- einen eindeutigen Namen, identisch zu Annotation @GeneratedValue
- eine Generatorstrategie
- ev. weitere Parameter …

Strategie	Parameter	Beschreibung
native	–	Wählt entsprechend der darunterliegenden Datenbank eine Strategie (identity, sequence oder hilo). Welche Strategie für eine Datenbank gewählt wird, ist in den Dialekten definiert.
uuid	separator	Gibt einen String mit Länge von 32 ausschliesslich hexadezimalen Zeichen zurück. Optional kann ein Separator zwischen den UUID-Komponenten mit generiert werden.
hilo	table column max_lo	Dieser Generator nutzt einen Hi/Lo-Algorithmus, um numerische IDs (Long, long, int, ...) zu erzeugen. Optional können die Spaltennamen für die Hi/Lo- Tabelle angegeben werden (Defaultwerte: hibernate_unique_key und next_hi). Mit max_lo kann die Anzahl der IDs bestimmt werden, die erzeugt werden, bevor wieder ein Datenbankzugriff erfolgt, um den Max-Value zu erhöhen. Der Generator kann nicht mit einer eigenen Connection oder eine über JTA (Java Transaction API) erhaltene Connection verwendet werden, da Hibernate in der Lage sein muss, den "hi"-Value in einer neuen Transaktion zu holen.
seqhilo	sequence max_lo parameters	Dieser Generator kombiniert einen Hi/Lo-Algorithmus mit einer darunterliegenden Sequence, die die Hi-Values generiert. Die Datenbank muss Sequences unterstüzen, wie zum Beispiel Oracle und PostgreSQL. Der Parameter parameters wird dem Create Sequence Statement hinzugefügt, beispielsweise "INCREMENT BY 1 START WITH 1 MAXVALUE 100 NOCACHE". Mit sequence kann ein Name für die Sequence vergeben werden, Default ist "hibernate_sequence".
identity	–	Dieser Generator unterstützt Identity Columns bzw. autoincrement, die es beispielsweise in MySQL, HSQLDB, DB2 und MS SQL Server gibt.
select	key (required)	Die ID wird über ein Select mit einem eindeutigen key erhalten. Der Primärschlüssel wird von der Datenbank, zum Beispiel mit einem Trigger, vergeben.

Strategie	Parameter	Beschreibung
sequence	sequence parameters	Dieser Generator unterstützt Sequences, die es beispielsweise in PostgreSQL, Oracle und Firebird gibt. Die ID kann vom Type Long, long, int etc. sein. Der Parameter parameters wird dem Create Sequence Statement hinzugefügt, beispielsweise "INCREMENT BY 1 START WITH 1 MAXVALUE 100 NOCACHE". Mit sequence kann ein Name für die Sequence vergeben werden, Default ist "hibernate_sequence".
assigned	–	Bei assigned muss die ID selbst gesetzt werden, vor dem Aufruf von save(). Nützlich bei natürlichen Keys. Das ist zugleich auch die Default-Strategie, falls die Annotation @GeneratedValue nicht angegeben wurde.
increment	–	Der maximale Primärschlüssel-Wert einer Tabelle wird beim Start der Anwendung gelesen und dann jedes Mal erhöht, wenn ein Insert erfolgt. Sollte nicht in einer Cluster-Umgebung benutzt werden.
foreign	property (required)	Benutzt die ID eines assoziierten Objekts, welches beim Parameter property angegeben werden muss. Wird gewöhnlich in Verbindung mit 1-zu-1-Beziehungen verwendet.
guid	–	Nutzt die von MS SQL Server oder MySQL generierte GUID.

Hibernate ID-Generatoren

weitere Generatoren können selbst erstellt werden:

- Interface IdentifierGenerator muss implementiert werden

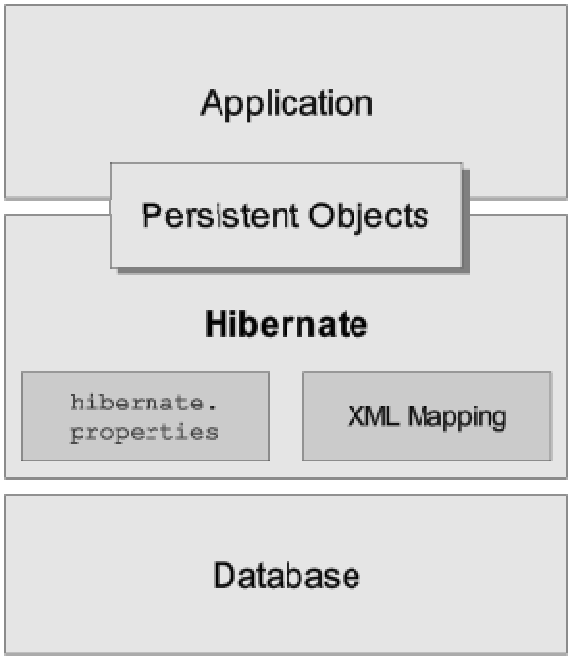
seqhilo-Generators als Beispie User.hbm.xml:

```
<id name="id" type="long" column="id">
  <generator class="hilo">
    <param name="table">hi_value</param>
    <param name="column">next_value</param>
    <param name="max_lo">100</param>
  </generator>
</id>
```

7 Architektur

7.1 Überblick

- stellt persistente Objekte der Applikation zur Verfügung
- durch Konfiguration
  - hibernate.properties
  - XML Mapping
- ist sehr flexibel
- unterschiedlichste Einsatzszenarien
- Datenbankanbindung durch Anwendung
- Anwendung übernimmt Transaktionen und weitere Dienste

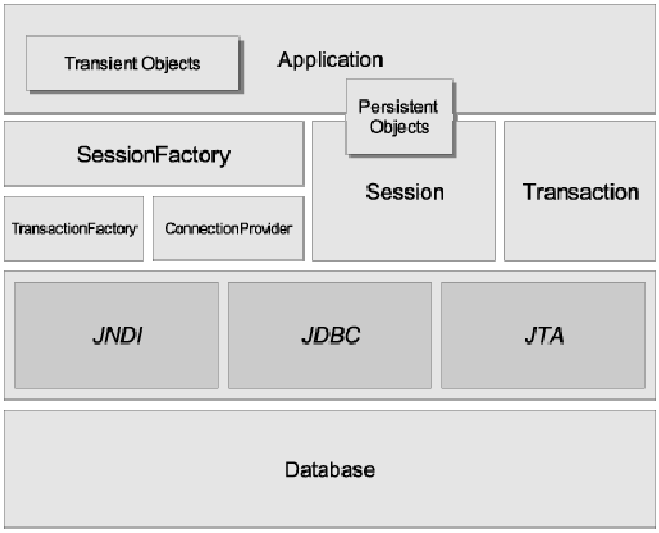


oder:

- Hibernate übernimmt JDBC, JTA, JNDI etc.

7.2 Services

- Java Naming and Directory Interface, Zugriff auf Namens- und Verzeichnisdienste
  http://java.sun.com/products/jndi/
- Java Database Connectivity, Zugriff auf Datenquellen
  http://java.sun.com/javase/technologies/database/index.jsp
- Java Transaction API, erlaubt die Steuerung der Transaktionsverwaltung
  http://java.sun.com/products/jta/



Hibernate kapselt JDBC, JTA und JNDI

7.3 Schnittstellen

7.3.1 SessionFactory

- Cache enthält generierte SQL Statements
- Cache enthält Mappingdaten
- erzeugen Session-Instanzen für Anwendung
- als Singelton
- schwergewichtig, sehr teuer in der Erstellung
- während der Initialisierung der Anwendung erzeugt
- eine SessionFactory pro Datenbank

7.3.2 Session

- Hibernate Interface zur Anwendung
- leichtgewichtig, billig zu erzeugen
- viele Sessions pro Anwendung erzeugen und löschen
- normalerweise bei jeder Anfrage oder für jeden Anwendungsfall (Use-Case)
- kapselt eine Datenverbindung
- verwaltet die Transaktion
- definiert den First-Level-Cache
- realisiert Änderungen an Objekten
- Session ist nicht "Thread-Safe", Entities einer Session dürfen nur im Thread dieser Session verwendet werden.

7.3.3 Configuration

- spezifiziert Konfiguration beim Erzeugen der SessionFactory
- spezifiziert Mappings beim Erzeugen der SessionFactory
- nur zur Initialisierungszeit von Hibernate relevant
- Beziehung zur Configuration wird nicht aufrecht erhalten

7.3.4 Transaction

- Abstrahiert die Anwendung der unterliegenden
  - JDBC Transaktion
  - JTA Transaktion
  - Corba Transaktion
- Transaction ist "Single-Threaded"
- kurzlebiges Objekt

- wird von der Anwendung benutzt
- spezifiziert atomare Folge von Abläufen

7.3.5 ConnectionProvider

- Factory für Datenbankverbindungen
- abstrahiert die Anwendung von einer Datasource oder einem DriverManager
- wird intern von Hibernate verwendet
- ermöglicht definierte Datenverbindungen

7.3.6 TransactionFactory

- optionale Factory für Instanzen von Transactions
- konkrete Implementierungen:
  - CMT-TransactionFactory (Container Manged Transaction)
  - JDBC-TransactionFactory (Java Database Connectivity)
  - JTA-TransactionFactory (Java Transaction API)
- gewünschte Implementierung durch Konfiguration

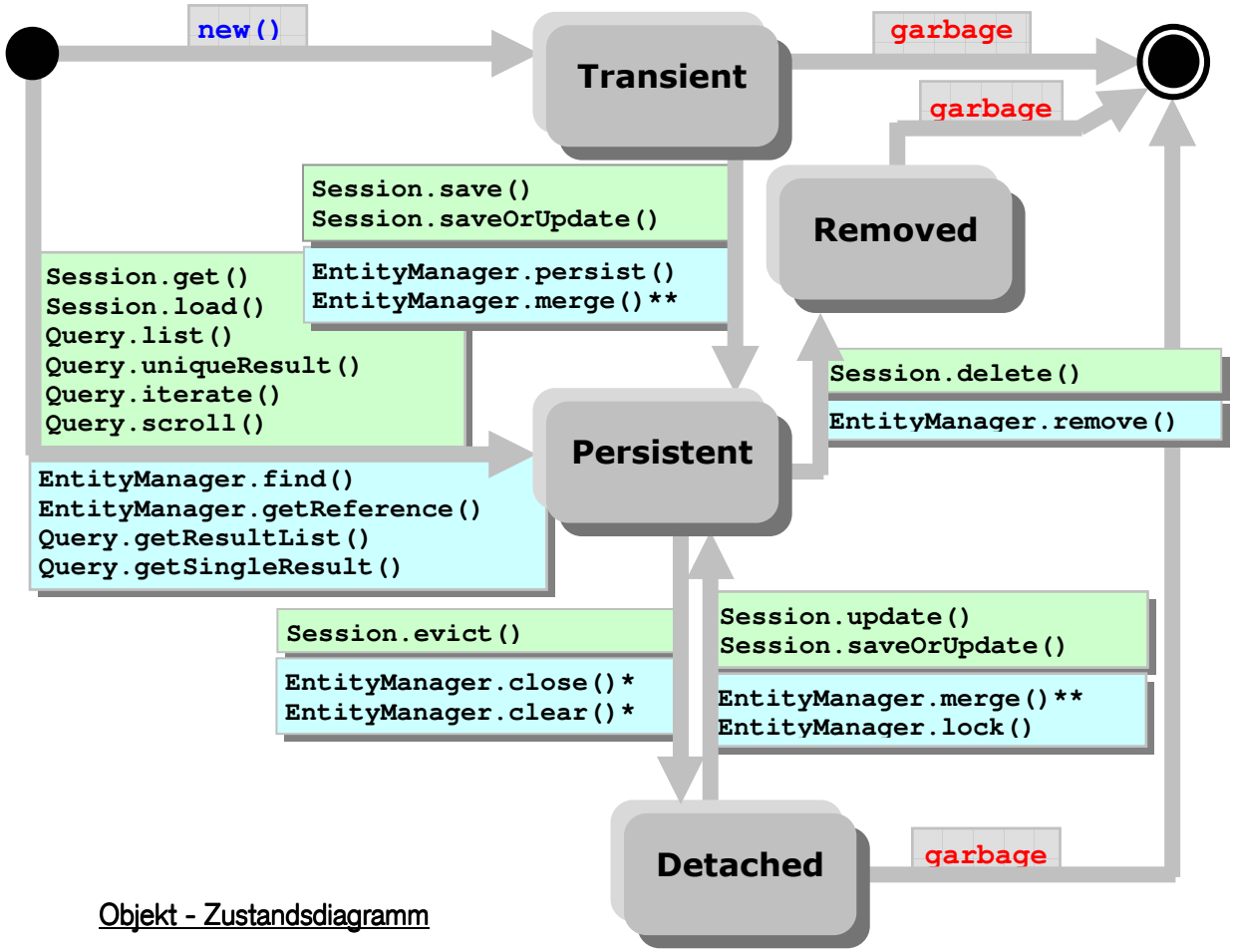
8 Lebenszyklus

- Hibernate verwaltet die Zustände eines persistemtem Objektes transparent
  - Hibernate ist transparentes O/R-Mapping Framework
  - Entities ist ihr eigener Zustand nicht bekannt
  - es ist nicht direkt möglich, eine Entity nach ihrem aktuellen Zustand zu fragen

8.1 Objektzustände

Eine Entity in Hibernate definiert vier verschiedene Zustände:

- Transient / Removed
- Persistent
- Detached



Objekt - Zustandsdiagramm

- Zustandsänderung durch Operationen des Persistenzmanager
  - ☐ Hibernate Session (superset der JPA Spezifikation)
  - ☐ EntityManager (standardisiert durch JPA Sezifikation)

* betrifft alle Instanzen des Persistenzkontext (Hibernate & JPA)

** erzeugt persistente Instanz, Originalobjekt bleibt erhalten

8.1.1 Transient

- durch new erzeugtes Entity-Objekt
- keine Verbindung zwischen der Entity und Hibernate Session
- Zustand der Entity nicht persistent abgespeichert
- normales Java-Objekt POJO
- Garbage Collector entsorgt Objekte ohne Referenz
- keine Repräsentation der Entity innerhalb der Datenbank
- alle Daten der Entity sind flüchtig und gehen verloren (Garbage Collector)
- Entities sind transient, nicht Teil einer Transaktion
- Rollback nicht anwendbar (in Transaktion)
- Primärschlüsselfelder, von Hibernate erzeugt, sind noch nicht gesetzt

Übergang von Transient zu Persistent mit:

- save(...)
- saveOrUpdate(...)
- persist(...)
- Referenzierung durch persistente Entity (abhängig von Kaskadierungseinstellungen der Assoziation)

8.1.2 Persistent

- Entity ist einer Hibernate Session zugeordnet
- Entity hat Primärschlüssel zugewiesen
- (auch ohne) Repräsentation innerhalb der Datenbank
- save(...) veranlasst Hibernate nicht Datenbankaufrufe durchzuführen
- Entities im persistenten Zustand sind Teil einer Transaktion
- Änderungen können durch Rollback rückgängig gemacht werden
- Änderung von Attributen wird von Hibernate erkannt
  - UPDATE innerhalb der Datenbank
  - INSERT innerhalb der Datenbank
- get(), load(), find(), iterate() – erzeugt persistentes Entity
  - ohne Übergang von Transient nach Persistent

Übergang von Persistent zu Transient:

- session. delete(...)

Achtung: delete() löscht die Daten der Entity innerhalb der Datenbank
Java-Objekt bleibt verfügbar, vorausgesetzt es wird noch referenziert

8.1.3 Detached

- schliessen der Session mit close(...)
- Zuordnung der Entities zu Session endet
- Entities nicht mehr Teil einer Transaktion
- Änderungen werden nicht mehr mit Datenbank synchronisiert
- Java-Objekte enthalten trotzdem (veraltete) persistente Daten
- Serialisieren (und Übertragen) einer Entity in einen anderen Prozess
  - Transferieren des POJO in eine Remote-Anwendung, (Client-Request)
- Transfer-Objekte zwischen Präsentationsschicht und Datenbankschicht

Übergang von Detached zu Persistent:

- public void **update**(Object object):
  - übergebene Entity wird an die Session gebunden
  - Änderungen an den Daten der Entity werden von Hibernate erkannt
  - spielt keine Rolle, ob Entity vor oder nach update(...) verändert wurde
  - existiert Primärschlüssel in Session wird Exception geworfen
  - nur verwenden falls Entity nicht bereits in der Session existiert

- public void **saveOrUpdate**(Object object):
  - Convenience-Methode (speichern oder aktualisieren)
    - identisch zu update(...) bei detached Entities
    - identisch zu save(...) bei transienten Entities
    - keine Auswirkung bei persistenten Entities
  - existiert Primärschlüssel in Session wird Exception geworfen

- public Object **merge**(Object object):
  - Zustand der übergebenen Entity wird in Session übertragen
  - Kopie der Entity wird in Session aufgenommen
  - existiert Primärschlüssel in Session werden Daten der Entity kopiert
  - falls keine Entity in Session ist:
    - neue Entity wird aus der Datenbank geladen und die Daten kopiert
    - neue Entity wird erstellt und die Daten kopiert

- gibt Referenz auf die persistente Entity zurück

- public void **lock**(Object object, LockMode lockMode):
  - bindet Detached Entity wieder an Session
  - Detached Entity darf nicht verändert worden sein
  - Parameter lockMode definiert verschiedene Lock-Modi, welche in Verbindung mit Transaktionen eine Rolle spielen

8.2 Detached Entities

- Remote Zugriff auf die Klasse User
- Serverseitiges Laden eines User

```
Session session = HibernateUtil.getSessionFactory().openSession();
Transaction tx = session.beginTransaction();
User user = (User) session.get(User.class, (long) 1);
tx.commit();
session.close();
```

- Schliessen der Session führt Entity in den detached Zustand
- Entity zur Remoteanwendung übertragen, z. B. mittels RMI

- Remoteanwendung verändert die Entity durch Getter/Setter-Methoden

```
user.setEmail("new_email@email.com");
```

- Entity wird an Serveranwendung übergeben

```
Session session = HibernateUtil.getSessionFactory().openSession();
Transaction tx = session.beginTransaction();
session.update(user);
tx.commit();
session.close();
```

- neue Session wird erzeug
- update(...) ordnet Entity der neuen Session zu
- Änderungen werden mit commit() der Transaktion in die Datenbank übernommen
- Verwendung bei Objektgraphen
- minimieren von teuren Operationen
- Vorteil bei Kaskadierung

8.2.1 Limitationen von Detached Entities

- NonUniqueObject

```
package library;
import org.hibernate.*;
public class DetachedUpdateUser {
   public static void main(String[] args) {
      Session session;
      Transaction tx;
      session = HibernateUtil.getSessionFactory().openSession();
      tx = session.beginTransaction();
      User user = (User) session.get(User.class, new Long(1));
      tx.commit();
      session.close();
      // Detached Entity
      System.out.println("User detached : " + user.getFirstname());
      session = HibernateUtil.getSessionFactory().openSession();
      tx = session.beginTransaction();
      User other_user = user;
      user = (User) session.get(User.class, new Long(1));
      session.update(other_user); // NonUniqueObjectException
      tx.commit();
      session.close();
   }
}
```

- update(...) wirft org.hibernate.NonUniqueObject- Exception
- Entity mit gleicher ID im Kontext der Session vorhanden
- daher Verwendung von merge(...) ist egal, ob sich bereits eine Entity im Session-Kontext befindet, der Inhalt wird auf jeden Fall übernommen

REGEL: Update nur verwenden, wenn sichergestellt ist, dass sich die entsprechenden Entities nicht schon in der Session befinden.

Da bei komplexen Objektgraphen schwierig, update(...) nur unmittelbar nach dem Öffnen einer "frischen" Session verwenden, da dieser mit Sicherheit keine Entities zugeordnet sind.

8.2.2 Optimistisches Locking

- Lost Updates kommen bei parallelen Zugriffen vor
- letztes Update überschreibt vorhergehende Änderungen

Version Attribut

```
@Entity
@Table(name = "T_USER")
public class User {
   private Long id;
```

```
  private int version;
  ...

  }
  @Version
  public int getVersion() {
     return version;
  }
  public void setVersion(int version) {
     this.version = version;
  }
```

Entity User mit Versionsfeld

- Versionsfeld mit Annotation @Version markiert
  - oder in XML-Mappingdatei mit <version name="version"/>

- Hibernate verwendet version als einfacher Zähler
- jede Datenänderung incrementiert den Zähler
- jeder Update vergleicht den Zählerstand

Hibernate UPDATE Statement um User-Entity zu aktualisieren:

```
Hibernate:
update T_USER
set firstname=?, version=3, lastname=?, email=?
where id=? and version=2
```

- Hibernate signalisiert Lost Updates durch eine org.hibernate.StaleObjectStateException beim Beenden der Transaktion
  - StaleStateException
    Ein Objekt hat einen veralteten Zustand. Mögliche Ursachen: Es wurde versucht, mit einer optimistischen Sperrstrategie eine Objekt zu speichern, dass bereits von einem änderen Benutzer gespeichert wurde; es wurde versucht, einen nicht existierenden Datensatz zu löschen oder zu aktualisieren.
  - StaleObjectStateException
    Wie StaleStateException, aber mit zusätzlichen Informationen über das problematische Objekt.

8.2.3 Lazy Loading

- Lazy Loading kann in Hibernate eingesetzt werden, um Assoziationen zwischen Entities erst später aufzulösen
- Daten der referenzierten Entities werden erst bei Bedarf geladen

- jedoch fehleranfällig

Klasse User wird mit Assoziation zur Klasse Address erweitert:

```
package library;
import javax.persistence.*;
@Entity
@Table(name = "T_USER")
public class User {
   private Long id;
   private Address address;
   private String firstname;
   private String lastname;
   private String email;
   private User() {}
   public User(String firstname, String lastname, String email) {
      this.firstname = firstname;
      this.lastname = lastname;
      this.email = email;
   }
   @Id
   @GeneratedValue
   public Long getId() {
      return id;
   }
   private void setId(Long id) {
      this.id = id;
   }
   @OneToOne(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
   @JoinColumn(name = "address_id", unique = true)
   public Address getAddress() {
      return address;
   }
   public void setAddress(Address address) {
      this.address = address;
   }
```

Die Klasse User mit einer Assoziation zur Klasse Address

- fetch = FetchType.LAZY aktiviert Lazy Loading
- Klasse Address ist wie User eine normale Entity (POJO)

```
package library;
import util.HibernateUtil;
import org.hibernate.*;
public class SaveUserAddress {
   public static void main(String[] args) {
      User user = new User("my_First", "my_Last",
           "my_Email@email.com");
      user.setAddress(new Address(null, "my_Street", "my_City"));
      Session session = HibernateUtil.getSessionFactory()
           .openSession();
      Transaction tx = session.beginTransaction();
      session.persist(user);
      tx.commit();
      session.close();
```

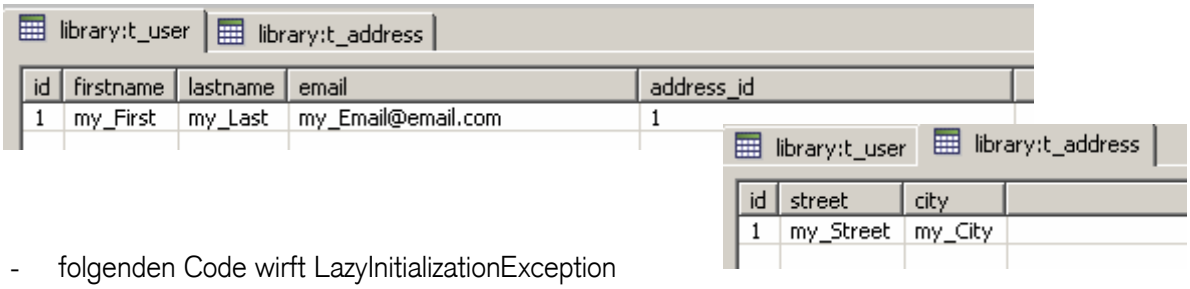
```
    }
}
```

SaveUserAddress.java

Hibernate generiert SQL:

```
insert into T_ADDRESS (street, city) values (?, ?)
insert into T_USER (address_id, firstname, lastname, email)
values (?, ?, ?, ?)
```

generierte Tabellen:



- folgenden Code wirft LazyInitializationException

```
Session session = HibernateUtil.getSessionFactory()
        .openSession();
Transaction tx = session.beginTransaction();

User user = (User) session.get(User.class, new Long(1));
tx.commit();
session.close();

Address address = user.getAddress();
address.getCity(); // LazyInitializationException
```

- adress.getCity() wirft org.hibernate.LazyInitializationException
- Daten der Entity address stehen nicht zur Verfügung
- Session bereits geschlossen, Entity user ist im Zustand Detached
- Daten von address können nicht nachgeladen werden

REGEL: alle Daten einer Entity laden, bevor die Entity in den Zustand Detached übergeht

- besser Verzicht auf Lazy Loading
  - durch entsprechend formulierte Queries
  - durch die Verwendung von Hibernate.initialize(Object proxy):

```
Session session = HibernateUtil.getSessionFactory()
          .openSession();
Transaction tx = session.beginTransaction();

User user = (User) session.get(User.class, new Long(1));
Hibernate.initialize(user.getAddress());
```

- Hibernate.initialize(...) initialisiert nicht referenzierte Entities innerhalb der übergebenen Entity
  - nicht für grössere Objektgraphen geeignet
  - gesamter Graph muss durchlaufen werden
  - alle Elemente werden initialisieren
  - viele einzelne Abfragen an die Datenbank sind notwendig

Der sinnvollste Weg, das Problem von Detached Entities in Verbindung mit Lazy Loading zu vermeiden, ist die Verwendung von entsprechenden Queries, in denen sichergestellt ist, dass alle benötigten Daten geladen werden.

9 Beziehungen

- Beziehungen zwischen Klassen – Relation zwischen Tabellen
- Abbilden eines Objektmodell (Vererbung) durch Relationen in einem Datenmodel
- Java Objekte werden in Relationen abgebildet
  - als Komponenten
  - durch Assoziationen
- uni-direktional
- bi-direktional (@mappedBy gibt führendes Ende an)
- N Seite als Collection
  - java.lang.List (Reihenfolge der Elemente nicht gegeben ▶ @OrderBy)
  - java.lang.Set
  - java.lang.Map
  - java.lang.Bag

9.1 Komponenten

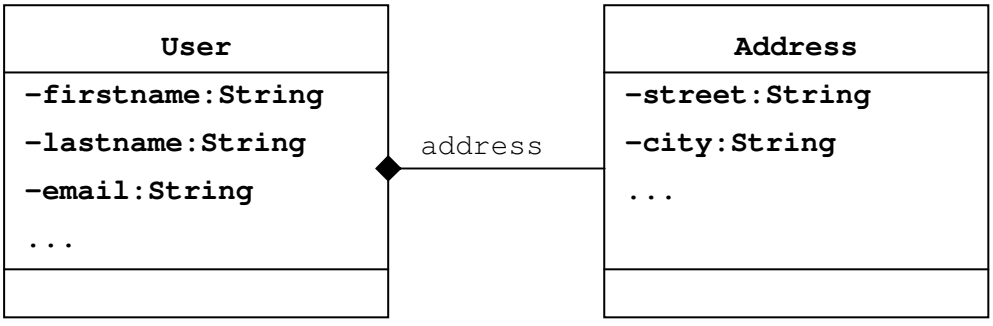
Entity-Komponente

- haben einen Primärschlüssel
- haben einen Lebenszyklus

Value-Komponente

- haben keine Datenbankidentität
- haben keinen Primärschlüssel
- gehören zu einer Entity und ihr Zustand wird innerhalb der Tabelle der dazugehörigen Entity gesichert
- Ausnahme sind Collections von Value-Typen
- typisch sind einfache Objekte vom Typ String
- Lebensdauer eines Value-Typ ist immer an den Lebenszyklus der enstpr. Entity gebunden

Komponenten ermöglichen die Abbildung mehrerer Klassen auf eine Tabelle.



Komposition

- strengere Form der Aggregation
- die Lebensdauer eines Objektes Address ist dabei an das Objekt User gebunden
- eine Address wird also immer mit oder nach dem User erzeugt
- und mit dem User zerstört
- Komposition wird in UML (http://www.omg.org) mit einer gefüllten Raute dargestellt

Aggregation

- dargestellt mit einer leeren Raute
- beschreibt eine schwache Beziehung zwischen Objekten
- Objekt ist ein Teil eines anderen Objekts
- kann aber im Gegensatz zur Komposition auch alleine existieren

Die Implementierung mit Java macht keinen Unterschied. Aber Objekt Address ist für Hibernate ein Value-Typ, hat keinen Primärschlüssel.

```
package library;
import javax.persistence.*;
@Embeddable
public class Address {
   private String street;
   private String city;
   private Address() {}
   public Address(String street, String city) {
      this.street = street;
      this.city = city;
   }
   public String getStreet() {
      return street;
   }
   public void setStreet(String street) {
      this.street = street;
   }
   public String getCity() {
      return city;
   }
   public void setCity(String city) {
      this.city = city;
   }
   @Override
   public String toString() {
      return " " + this.street + " " + this.city;
   }
}
```

Komponenten Implementierung mit Annotationen der Klasse Address.java

```java
package library;
import javax.persistence.*;
@Entity
@Table(name = "T_USER")
public class User {
  private Long id;
  @Embedded // optional
  private Address address;
  private String firstname;
  private String lastname;
  private String email;
  private User() {}
  public User(String firstname, String lastname, String email) {
    this.firstname = firstname;
    this.lastname = lastname;
    this.email = email;
  }
  @Id
  @GeneratedValue
  public Long getId() {
    return id;
  }
  private void setId(Long id) {
    this.id = id;
  }
  public Address getAddress() {
    return address;
  }
  public void setAddress(Address address) {
    this.address = address;
  }
  public String getFirstname() {
    return firstname;
  }
  public void setFirstname(String firstname) {
    this.firstname = firstname;
  }
  public String getLastname() {
    return lastname;
  }
  public void setLastname(String lastname) {
    this.lastname = lastname;
  }
  public String getEmail() {
    return email;
  }
  public void setEmail(String email) {
    this.email = email;
  }
  @Override
  public boolean equals(Object obj) {
    if (this == obj) return true;
    if (!(obj instanceof User)) return false;
    final User user = (User) obj;
    return !(email != null ? !email.equals(user.email)
         : user.email != null);
  }
  @Override
  public int hashCode() {
    return 29 * (email != null ? email.hashCode() : 0);
  }
```

```
  @Override
  public String toString() {
    return " " + this.firstname + " " + this.lastname + " "
        + this.email;
  }
}
```

Komponente Address in der Klasse User.java

- Annotation @Embeddable definiert Komponente auf Klassenebene
- Entity User mit dem Attribut address muss nicht gekennzeichnet werden
  - @Embedded ist optional

- Komponente Address bei XML-Mapping im User.hbm.xml definiert
  - <component name="..." class="..."/> definiert Komponente innerhalb des Elements <class .../>
  - Eigenschaften der Komponente werden innerhalb von <component/> mit <property name="..."/> angegeben:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="booksonline.bo">
  <class name="User" table="T_USER">
    <id name="id">
      <generator class="native"/>
    </id>
    <property name="firstname"/>
    <property name="lastname"/>
    <property name="email"/>
    <property name="password"/>
    <component name="address" class="Address">
      <property name="street"/>
      <property name="city"/>
    </component>
  </class>
</hibernate-mapping>
```

User.hbm.xml mit Komponente Address

Speichern von User mit Adresse in einer Tabelle:

- mit setAddress() dem User seine Adresse definieren
- Komponente Address ist Teil der Entity User
- session.persist() speichert User mit der Address persistent in Datenbank

```
package library;
```

```
import util.HibernateUtil;
import org.hibernate.*;
public class SaveUserAddress {
   public static void main(String[] args) {
      User user = new User("my_First", "my_Last",
           "my_Email@email.com");
      user.setAddress(new Address("my_Street", "my_City"));
      Session session = HibernateUtil.getSessionFactory()
           .openSession();
      Transaction tx = session.beginTransaction();
      session.persist(user);
      tx.commit();
      session.close();
   }
}
```

SaveUserAddress.java

Datenbanktabelle USER_TABLE

library:t_user

id	street	city	firstname	lastname	email
1	my_Street	my_City	my_First	my_Last	my_Email@email.com

- Hibernate definiert als Spaltennamen für Komponenten den Attributnamen
  - Annotation @Column definiert den Spaltennamen benutzerspezifisch
  - name, unique, nullable, insertable, updatable, lenght, precusion, scale, etc.

```
   @Column(name = "address_street")
   public String getStreet() {
      return street;
   }
   public void setStreet(String street) {
      this.street = street;
   }
   @Column(name = "address_city", length = 50, insertable = false)
   public String getCity() {
      return city;
   }
```

- Annotationen @AttributeOverrides und @AttributeOverride überschrieben die Attribute der Komponente von der Entity aus
  - ermöglicht Wiederverwendung der Komponente

```
package library;
import javax.persistence.*;
@Entity
@Table(name = "T_USER")
public class User {
   private Long id;
   @Embedded
```

```
    private Address address;
    ...

    @Id
    @GeneratedValue
    public Long getId() {
       return id;
    }
    private void setId(Long id) {
       this.id = id;
    }
    @Embedded
    @AttributeOverrides( {
         @AttributeOverride(
           name = "street",
           column = @Column(name = "userstreet")),
         @AttributeOverride(
           name = "city",
           column = @Column(name = "usercity", length = 50)) })
    public Address getAddress() {
       return address;
    }
    public void setAddress(Address address) {
       this.address = address;
    }
...
```

Attribute überschreiben mit @AttributeOverrides

9.2 Assoziationen

- Verbindung mindestens zweier Klassen
  - erlaubt das Navigieren von der einen zur anderen Klasse
- binäre Assoziation ist eine Beziehung zwischen zwei Klassen
- reflexive Assoziation definiert Beziehung zu sich selbst

Kardinalität beschreibt den Grad einer Beziehung zwischen zwei Klassen:

Eins-zu-Eins (1:1)

- eine Klasse steht mit einer anderen Klasse in Beziehung

Eins-zu-Viele/Viele-zu-Eins (1:n/n:1)

- eine Klasse steht mit mehreren anderen Klassen in Beziehung

Viele-zu-Viele (n:n)

- mehrere Klassen stehen mit mehreren anderen Klassen in Beziehung

Bei allen Assoziationen unterscheidet man:

- unidirektionale Beziehungen:
  - Person führt zu Adresse

- über Adresse kann keine Person definiert werden
- nur von User kann zu Address navigiert werden

- bidirektionale Beziehungen:
  - Person führt zu Adresse
  - Adresse führt zu Person
    - Entity Address muss Getter und Setter für den User definieren

9.2.1 1:1-Beziehungen

- bei 1:1 Beziehungen haben die beiden Tabellen den gleichen Primary-Key:
  - Primary-Key (Relation) selbst erzeugen
  - Generatorstategie verwenden

<u>identischer Primärschlüssel selbst erzeugen</u>

- Annotation @OneToOne kennzeichnet Address als 1-zu-1-Beziehung
- Annotation @PrimaryKeyJoinColumn
  - beiden in Beziehung stehenden Objekte haben immer denselben Primärschlüssel

```
package library;
import javax.persistence.*;
@Entity
@Table(name = "T_USER")
public class User {
   private Long id;
...
   }
   @Id
   @GeneratedValue
   public Long getId() {
      return id;
   }
   private void setId(Long id) {
      this.id = id;
   }
   @OneToOne
   @PrimaryKeyJoinColumn
   public Address getAddress() {
      return address;
   }
   public void setAddress(Address address) {
      this.address = address;
   }
   ...
}
```

```
package library;
import javax.persistence.*;
@Entity
@Table(name = "T_ADDRESS")
public class Address {
   private Long id;
   private String street;
   private String city;
   private Address() {}
   public Address(Long id, String street, String city) {
      this.id = id;
      this.street = street;
      this.city = city;
   }
   @Id
   public Long getId() {
      return id;
   }
   public void setId(Long id) {
      this.id = id;
   }
   ...
}
```

Unidirektionale 1-zu-1-Beziehung mit gemeinsamem Primärschlüssel

- Primärschlüssel für die Entity Address muss selbst gesetzt werden
- session.persist(user) erzeugt einen Primärschlüssel für den User

```
package library;
import org.hibernate.*;
import util.HibernateUtil;
public class SaveOneToOneUserAddress {
   public static void main(String[] args) {
      User user = new User("my_First", "my_Last",
           "my_Email@email.com");
      Session session = HibernateUtil.getSessionFactory()
           .openSession();
      Transaction tx = session.beginTransaction();
      session.persist(user);
      Address address = new Address(user.getId(), "my_Street",
           "my_City");
      user.setAddress(address);
      session.persist(address);
      tx.commit();
      session.close();
      session = HibernateUtil.getSessionFactory().openSession();
      tx = session.beginTransaction();
      user = (User) session.get(User.class, new Long(1));
      System.out.println("Address: " + user.getAddress());
      tx.commit();
      session.close();
   }
}
```

SaveOneToOneUserAddress.java

Generatorstrategie foreign

- 1-zu-1-Beziehung mit gemeinsamem Primärschlüssel
- beide Entities haben denselben Primärschlüssel
- Parameter property referenziert Komponente
- Primärschlüssel in der Entity Address wird mit der Generatorstrategie forein generiert

```
package library;
import javax.persistence.*;
import org.hibernate.annotations.GenericGenerator;
import org.hibernate.annotations.Parameter;
@Entity
@Table(name = "T_USER")
public class User {
   private Long id;
   private Address address;
   private String firstname;
   private String lastname;
   private String email;
   private User() {}
   public User(String firstname, String lastname, String email) {
      this.firstname = firstname;
      this.lastname = lastname;
      this.email = email;
   }
   @Id
   @GeneratedValue(generator = "foreign")
   @GenericGenerator( name = "foreign",
                      strategy = "foreign",
                      parameters = {
      @Parameter(name = "property", value = "address") })
   public Long getId() {
      return id;
   }
   private void setId(Long id) {
      this.id = id;
   }
   @OneToOne
   @PrimaryKeyJoinColumn
   public Address getAddress() {
      return address;
   }
   public void setAddress(Address address) {
      this.address = address;
   }
   ...
```

Unidirektionale 1:1-Beziehung mit gemeinsamem Primärschlüssel und forein als GeneratorStrategie

- Generatorstrategie foreign erzeugt Address Primärschlüssel automatisch
  - id in Address darf nicht mehr gesetzt werden
  - persist() sichert User (inkl. Addresse), wenn User mit Address defiert sind

```
package library;
import org.hibernate.*;
```

```
import util.HibernateUtil;
public class SaveOneToOneUserAddress {
   public static void main(String[] args) {
      Session session = HibernateUtil.getSessionFactory()
           .openSession();
      Transaction tx = session.beginTransaction();
      User user = new User("my_First", "my_Last",
           "my_Email@email.com");
      user.setAddress(new Address("my_Street", "my_City"));
      session.persist(user);
      tx.commit();
      session.close();
      session = HibernateUtil.getSessionFactory().openSession();
      tx = session.beginTransaction();
      user = (User) session.get(User.class, new Long(1));
      System.out.println("Address: " + user.getAddress());
      tx.commit();
      session.close();
   }
}
```

bi-direktionale 1:1 Beziehung

- Klasse Address kann User referenzieren
- getUser()in der Entity Address mit Annotation @OneToOne

```
@Entity
@Table(name = "T_ADDRESS")
public class Address {
   private Long id;
   private User user;
…
   @OneToOne(mappedBy = "address")
   public User getUser() {
      return user;
   }
```

- Attribut mappedBy definiert die "owning"-Seite der Beziehung
- XMLMapping benutzt Attribut inverse
  - inverse = true definiert das nur Änderungen an die Datenbank weitergegeben werden, die am anderen Ende, also dem User, gemacht wurden.

9.2.2 1:n und n:1-Beziehungen

- jedes Buch hat einen Verlag
- ein Verlag kann kein, eins oder mehrere Bücher haben

1-zu-n bzw. n-zu-1-Beziehung zwischen Book und Publisher

```java
package library;
import org.hibernate.annotations.GenericGenerator;
import javax.persistence.*;
@Entity
@Table(name = "T_BOOK")
public class Book {
   private Long id;
   private String name;
   private String ISBN;
   private Publisher publisher;
   public Book(String name, String isbn) {
      super();
      this.name = name;
      ISBN = isbn;
   }
   protected Book() {}
   @Id
   @GeneratedValue(generator = "increment")
   @GenericGenerator(name = "increment", strategy = "increment")
   public Long getId() {
      return id;
   }
   protected void setId(Long id) {
      this.id = id;
   }
   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }
   public String getISBN() {
      return ISBN;
   }
   public void setISBN(String ISBN) {
      this.ISBN = ISBN;
   }
   @ManyToOne(cascade = CascadeType.ALL)
   @JoinColumn(nullable = false)
   public Publisher getPublisher() {
      return publisher;
   }
   public void setPublisher(Publisher publisher) {
      this.publisher = publisher;
   }
}
```

Book.java mit n-zu-1-Beziehung zu Publisher

- @JoinColumn erlaubt Definition des Fremdschlüssels
- nullable=false definiert Fremdschlüssel nie null

- Klasse Publisher ist eine Entity mit:
  - Annotation @Entity
  - Primärschlüssel mit Annotation @Id
  - 1-zu-n-Beziehung zu Book mit Annotation @OneToMany

- Attribut mappedBy definiert "owning"-Seite der Beziehung

```
package library;
import java.util.HashSet;
import java.util.Set;
import javax.persistence.*;
@Entity
@Table(name = "T_PUBLISHER")
public class Publisher {
   private Long id;
   private Set<Book> books = new HashSet<Book>();
   private String name;
   @Id
   @GeneratedValue
   public Long getId() {
      return id;
   }
   public void setId(Long id) {
      this.id = id;
   }
   @OneToMany(mappedBy = "publisher")
   public Set<Book> getBooks() {
      return books;
   }
   public void setBooks(Set<Book> books) {
      this.books = books;
   }
   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }
}
```

Publisher.java mit 1-zu-n-Beziehung zu Book

- Name des Fremdschlüssels kann angegeben werden mit referencedColumnName
- oder per Definition übernommen
  - Primärschlüsselfeld getrennt mit einem Unterstrich zusammengesetzt : PUBLISHER_TABLE.publisher_id

Die Attribute von @ManyToOne können die Performance erheblich beeinflussen.

@ManyToOne Attribute

- cascade [ALL, MERGE, PERSIST, REFRESH, REMOVE,..]
  - update( ), persist( ) usw. wird an die referenzierten Objekte durchgereicht
  - Default = kein durchreichen

- fetch [EAGER, LAZY]
  - EAGER : Fetch-Strategie lädt die Beziehungsobjekte sofort mit
  - LAZY : lädt erst beim Zugriff auf Beziehung das entsprechende Objekt nach

- referencedColumnName defniert Name des referenzierten Feldes

```
@ManyToOne(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
@JoinColumn(referencedColumnName = "myId", nullable = false)
public Publisher getPublisher() {
   return publisher;
}
```

Beziehungen mit Verbindungstabelle

- 1:n und/oder n:1 Beziehung mit Verbindungstabelle
  - Annotation @JoinTable, @JoinColumns definiert Tabellen-, Feldnamen
  - interessant bei bestehendem DB-Schema

```
@ManyToOne(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
@JoinTable( name = "T_BOOK_PUBLISHER",
            joinColumns = {@JoinColumn(name = "Buch")},
            inverseJoinColumns = {@JoinColumn(name = "Verlag")})
public Publisher getPublisher() {
   return publisher;
}
```

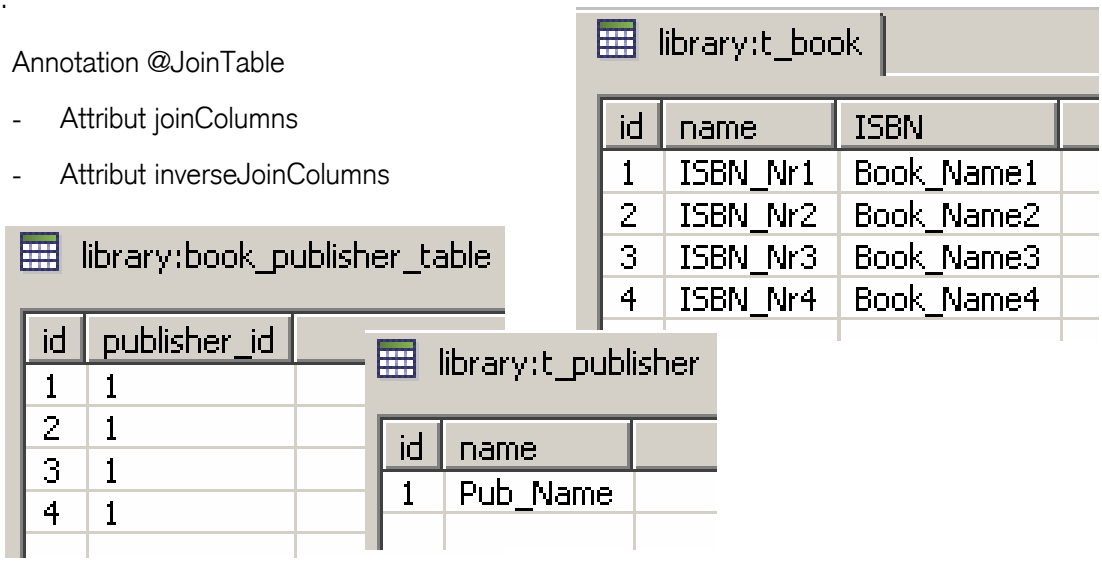
viele-zu-eins mit Verbindungstabelle BOOK_TABLE_PUBLISHER

Hibernate generiert in diesem Fall drei Tabellen:

- BOOK_TABLE mit Name, ISBN, etc.,
- PUBLISHER_TABLE
- BOOK_PUBLISHER_TABLE mit zwei Fremdschlüsseln auf BOOK_TABLE.ID und PUBLISHER_TABLE.ID

Die Join-Tabelle wird per Default aus den Namen der beiden Tabellen zusammengesetzt, oder mit:

- Annotation @JoinTable
  - Attribut joinColumns
  - Attribut inverseJoinColumns



Die neuen Entities Book und Publisher müssen der Konfigurationsdatei hibernate.cfg.xml hinzugefügt werden:

```
    <!-- Mapping beschrieben durch Annotations -->
    <mapping class="library.Publisher"/>
    <mapping class="library.Book"/>
```

- falls keine Kaskadierung für die Beziehung zwischen Book und Publisher wurde
  - expliziter Aufruf von session.persist(publisher) ist notwendig

```
package library;
import util.HibernateUtil;
import org.hibernate.Session;
import org.hibernate.Transaction;
public class SaveBookPublisherAssociation {
  public static void main(String[] args) {
     Publisher publisher = new Publisher();
     publisher.setName("Pub_Name");
     Book book1 = new Book("ISBN_Nr3", "Book_Name3");
     Book book2 = new Book("ISBN_Nr4", "Book_Name4");
     Session session = HibernateUtil.getSessionFactory()
          .openSession();
     Transaction tx = session.beginTransaction();
     session.persist(publisher);
     book1.setPublisher(publisher);
     book2.setPublisher(publisher);
     session.persist(book1);
     session.persist(book2);
     tx.commit();
     session.close();
  }
}
```

SaveBookPublisherAssociation.java

XML-Mapping

Die beiden genannten Varianten als XML-Mapping defineren sich wie folgt:

```
<hibernate-mapping package="library">
  <class name="Book" table="T_BOOK">
    <id name="id" column="bookId">
      <generator class="native"/>
    </id>
    <discriminator column="TYPE" type="string"/>
    <property name="name"/>
    <property name="ISBN"/>
    <!-- Publisher als many-to-one -->
    <many-to-one name="publisher"
                 column="publisherId"
                 not-null="true"/>
    <!-- Publisher als many-to-one mit Join-Tabelle -->
    <join table="T_BOOK_PUBLISHER">
      <key column="bookId" unique="true"/>
      <many-to-one name="publisher"
                   column="publisherId"
                   not-null="true"/>
    </join>
...
```

Ausschnitt aus Book.hbm.xml

```
<hibernate-mapping package="library">
  <class name="Publisher" table="T_PUBLISHER">
    <id name="id" column="publisherId">
      <generator class="native"/>
    </id>
    <property name="name"/>
    <!-- Address als Component -->
    <component name="address" class="Address">
      <property name="street"/>
      <property name="city"/>
    </component>
    <set name="books" inverse="true" batch-size="5">
        <key column="publisherId"/>
        <one-to-many class="Book"/>
    </set>
  </class>
```

Ausschnitt aus Publisher.hbm.xml

- Set beim XML-Mapping entspricht Collection bei Annotationen
- Attribut inverse entspricht der mappedBy Annotationen
  - inverse = true wird die "inverse"-Seite definiert
  - andere Seite ist die "owning"-Seite

HibernateUtil.java:

```
    private static final SessionFactory sessionFactory;
```

```
  static {
    try {
      // Create the SessionFactory for XML Mapping
      sessionFactory = new Configuration().configure()
           .buildSessionFactory();
    } catch (Throwable ex) {
      System.err.println("SessionFactory creation failed." + ex);
      throw new ExceptionInInitializerError(ex);
    }
  }
```

hibernate.cfg.xml:

```
<!-- Mapping beschrieben durch XML -->
<mapping resource="library/Book.hbm.xml"/>
<mapping resource="library/Publisher.hbm.xml"/>
```

Die XML-Dateien *.hbm.xml müssen unter /library im Klassenpfad sein.

bi-direktionale One-To-Many Beziehung

- setzten der Beziehungen auch auf der referenzierten Seite
  - addBook für die Entity Publisher

```
package library;
import java.util.HashSet;
import java.util.Set;
import javax.persistence.*;
@Entity
@Table(name = "T_PUBLISHER")
public class Publisher {
  private Long id;
  private Set<Book> books = new HashSet<Book>();
  private String name;
  @Id
  @GeneratedValue
  public Long getId() {
    return id;
  }
  public void setId(Long id) {
    this.id = id;
  }
  @OneToMany(mappedBy = "publisher", cascade = { CascadeType.ALL
})
  public Set<Book> getBooks() {
    return books;
  }
  public void setBooks(Set<Book> books) {
    this.books = books;
  }
  public void addBook(Book book) {
```

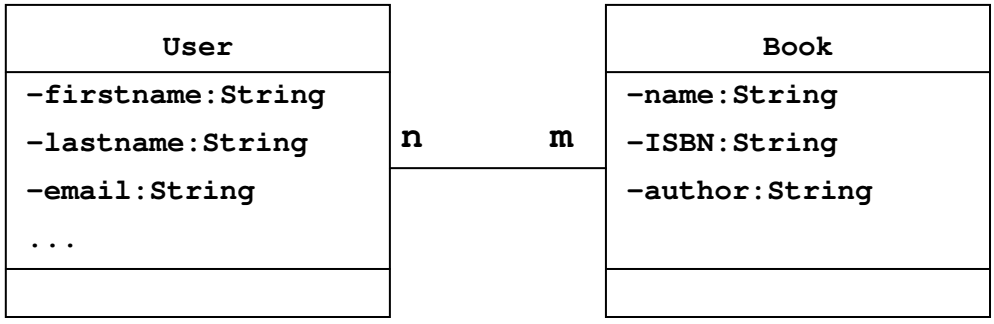
```
      this.books.add(book);
      book.setPublisher(this);
   }
   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }
}
```

- Speichern einer Publisher Book Beziehung

```
package library;
import util.HibernateUtil;
import org.hibernate.Session;
import org.hibernate.Transaction;
public class SavePublisherBookAssociation {
   public static void main(String[] args) {
      Publisher publisher = new Publisher();
      publisher.setName("Pub_Name");
      Book book1 = new Book("ISBN_Nr1", "Book_Name1");
      Book book2 = new Book("ISBN_Nr2", "Book_Name2");
      Session session = HibernateUtil.getSessionFactory()
           .openSession();
      Transaction tx = session.beginTransaction();
      publisher.addBook(book1);
      publisher.addBook(book2);
      session.persist(publisher);
      tx.commit();
      session.close();
   }
}
```

9.2.3 n:m-Beziehungen

- immer mit einer Verbindungstabelle (Assoziationstabelle)
- Annotation @ManyToMany
- Beispiel : User und Book



```
package library;
import javax.persistence.*;
import java.util.HashSet;
import java.util.Set;
@Entity
@Table(name = "T_USER")
public class User {
   private Long id;
   private String firstname;
   private String lastname;
   private String email;
   private Address address;
   private Set<Book> books = new HashSet<Book>();
   private User() {}
   public User(String firstname, String lastname, String email) {
      this.firstname = firstname;
      this.lastname = lastname;
      this.email = email;
   }
   @Id
   @GeneratedValue
   public Long getId() {
      return id;
   }
   private void setId(Long id) {
      this.id = id;
   }
   public String getFirstname() {
      return firstname;
   }
   public void setFirstname(String firstname) {
      this.firstname = firstname;
   }
   public String getLastname() {
      return lastname;
   }
   public void setLastname(String lastname) {
      this.lastname = lastname;
   }
   public String getEmail() {
      return email;
   }
   public void setEmail(String email) {
      this.email = email;
   }
   @Embedded
   @AttributeOverrides( {
        @AttributeOverride(name = "street", column = @Column(
          name = "userstreet")),
        @AttributeOverride(name = "city", column = @Column(
          name = "usercity", length = 50)) })
   public Address getAddress() {
      return address;
   }
   public void setAddress(Address address) {
      this.address = address;
   }
   @ManyToMany
   public Set<Book> getBooks() {
      return books;
```

```
    }
    public void setBooks(Set<Book> books) {
      this.books = books;
    }
    @Override
    public boolean equals(Object obj) {
      if (this == obj) return true;
      if (!(obj instanceof User)) return false;
      final User user = (User) obj;
      return !(email != null ? !email.equals(user.email)
          : user.email != null);
    }
    @Override
    public int hashCode() {
      return 29 * (email != null ? email.hashCode() : 0);
    }
    @Override
    public String toString() {
      return " " + this.firstname + " " + this.lastname + " "
          + this.email;
    }
}
```

User.java mit n-zu-m-Beziehung zu Book

Book Entity definiert die "owning"-Seite der bidirektionalen Beziehung mit mappedBy:

```
package library;
import java.util.HashSet;
import java.util.Set;
import org.hibernate.annotations.GenericGenerator;
import javax.persistence.*;
@Entity
@Table(name = "T_BOOK")
public class Book {
  private Long id;
  private String name;
  private String ISBN;
  private Publisher publisher;
  private Set<User> users = new HashSet<User>();
  public Book(String name, String isbn) {
    super();
    this.name = name;
    ISBN = isbn;
  }
  protected Book() {}
  @Id
  @GeneratedValue(generator = "increment")
  @GenericGenerator(name = "increment", strategy = "increment")
  public Long getId() {
    return id;
  }
  protected void setId(Long id) {
    this.id = id;
  }
  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
```

```
    }
    public String getISBN() {
       return ISBN;
    }
    public void setISBN(String ISBN) {
       this.ISBN = ISBN;
    }
    @ManyToOne(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
    public Publisher getPublisher() {
       return publisher;
    }
    public void setPublisher(Publisher publisher) {
       this.publisher = publisher;
    }
    @ManyToMany(mappedBy = "books")
    public Set<User> getUsers() {
       return users;
    }
    public void setUsers(Set<User> users) {
       this.users = users;
    }
    public void addUser(User user) {
       this.users.add(user);        // weist Book dem User-Set zu
       user.getBooks().add(this);   // weist User dem Book-Set zu
    }
}
```

Book.java referenziert n:m mit Attribut books

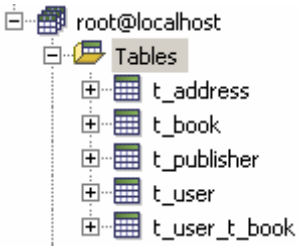
- Setzen der bi-directionalen Beziehungen:
  - User.setBooks(Set<Book> books)
  - Book.addUser(User user)

```
    public void addUser(User user) {
       this.users.add(user);        // weist Book dem User-Set zu
       user.getBooks().add(this);   // weist User dem Book-Set zu
    }
```

- n-zu-m-Beziehung zwischen User und Book resultiert in folgenden Tabellen:
  - Join-Table : t_user_t_book

library:t_user_t_book

users_id	books_id
1	1



Tabellen der n-zu-m-Beziehung zwischen User und Book

9.2.4 Transitive Persistenz

- Alle Objekte, die von einem persistenten Objekt erreichbar sind werden selbst auch persistent.
  - erlaubt die Weitergabe von Entity-Operationen : persist(), merge(), delete()
  - mit Annotationen : CascadeType
  - bei Beziehungen : @OneToOne, @OneToMany / @ManyToOne, @ManyToMany

```
@ManyToOne(cascade = CascadeType.PERSIST)
```

- A.persist() persistiert auch alle an A hängenden Objekte B

- weitere CascadeTypes überschreiben die CascadeTypes der Java Persistence API
  - Annotation @org.hibernate.annotations.Cascade
  - definiert CascadeType mit Annotation @org.hibernate.annotations.CascadeType

```
@OneToMany
@org.hibernate.annotations.Cascade( {
     @org.hibernate.annotations.CascadeType.SAVE_UPDATE,
     @org.hibernate.annotations.CascadeType.MERGE })
public List<B> getB() {
   return b;
}
```

Übersicht der CascadeTypes der Java Persistence API

Übersicht der möglichen CascadeTypes von JPA und HIbernate

JPA	HIBERNATE	Beschreibung
javax.persistence.CascadeType	org.hibernate.annotations.CascadeType	
PERSIST	PERSIST	Übergang zu managed Entity Instanz; commit() oder flush() persistiert Objekt in der Datensource.
MERGE	MERGE	Übergang von einer detached Entity zu einer managed Entity Instanz.
REMOVE	REMOVE	Übergang zu removed Entity Instanz; commit() oder flush() löscht Objekt in der Datensource.
REFRESH	REFRESH	Synchronisiert die Entity Instanz mit der Datensource. Änderungen innerhalb der Entity werden dabei überschrieben.
	DELETE	Entspricht javax.persistence.REMOVE

JPA	HIBERNATE	Beschreibung
javax.persistence.CascadeType	org.hibernate.annotations.CascadeType	
ALL		Enspricht cascade = { PERSIST, MERGE, REMOVE, REFRESH }.
	SAVE_UPDATE	save(Object entity) oder update(Object entity) wird an die in Beziehung stehenden Objekte mit diesem CascadeType durchgereicht
	REPLICATE	replicate(Object entity, ...) Objekt wird unter Verwendung der existierenden Id in der Datenbank abgelegt
	DELETE_ORPHAN	Alle persistenten Entities, die aus einer Beziehung zu einer persistenten Entity, beispielsweise aus einer Collection, genommen wurden (Objekte, die nicht mehr referenziert werden), werden automatisch gelöscht
	LOCK	lock(Object entity, ... Objekt wird mit angegebenem LockMode versehen, um beispielsweise Versionschecks oder pessimistisches Locking durchzuführen
	EVICT	evict(Object entity Objekt wird aus der Session gelöscht und damit verhindert, dass Änderungen in die Datenbank geschrieben werden
	ALL	Beinhaltet … und SAVE_UPDATE, DELETE, EVICT und LOCK.

Übersicht der CascadeType Annotations

Beispiel transitiver Persistenz

- Publisher und Book Beziehung definiert CascadeType.Persist und CascadeType.Merge
  - keine explizierter Aufruf durch persist(book) notwendig

```
@OneToMany(mappedBy = "publisher", cascade = {
     CascadeType.PERSIST, CascadeType.MERGE })
public Set<Book> getBooks() {
   return books;
}
```

Entity Publisher : CascadeType.PERSIST und CascadeType.MERGE

```
@ManyToOne
public Publisher getPublisher() {
   return publisher;
}
```

Entity Book : Beziehung zu Publisher

- bestehendem Publisher wird ein transientes Book hinzugefügt
- Publisher und Book werden detached

- Book Name wird geändert
- Publisher wird der Session hinzugefügt mit merge() und pesistiert
  - explizieter Aufruf von persist oder merge des transienten Books ist nicht notwendig

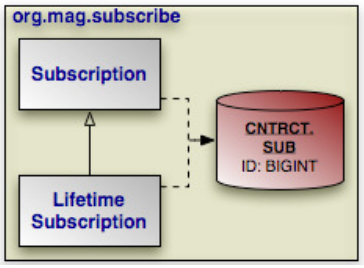
```
package library;
import util.HibernateUtil;
import org.hibernate.Session;
import org.hibernate.Transaction;
public class SavePublisherBookAssociation {
  public static void main(String[] args) {
    // jpa09 creates publisher
    Book book = new Book();
    book.setISBN("ISBN_Nr3");
    book.setName("Book_Name3");
    // Session holen und Transaktion beginnen
    Session session = HibernateUtil.getSessionFactory()
        .openSession();
    Transaction tx = session.beginTransaction();
    // Bereits persistenten Publisher aus Datenbank holen
    Publisher publisher = (Publisher) session.get(
        Publisher.class, (long) 1);
    // Nicht persistentes Book dem Publisher hinzufügen
    publisher.addBook(book);
    // Ein expliziter Aufruf von persist(book) ist durch
    // CascadeType.PERSIST nicht nötig
    tx.commit();
    session.close();
    // Ausserhalb der Session, z.B. beim Client
    // Publisher und Book sind nun "detached Objects".
    // Book wird am Client geändert.
    book.setName("Book_Name4");
    // (neue) Session holen und (neue) Transaktion beginnen
    Session newsession = HibernateUtil.getSessionFactory()
        .openSession();
    Transaction newtx = newsession.beginTransaction();
    // Publisher wird der Session wieder hinzugefügt,
    // diese erkennt die Änderungen am in der Beziehung
    // stehenden Book und wird diese mit der Datenbank
    // mit merge() synchronisieren
    newsession.merge(publisher);
    newtx.commit();
    newsession.close();
  }
}
```

transitive Persistenz : SavePublisherBookAssociation.java

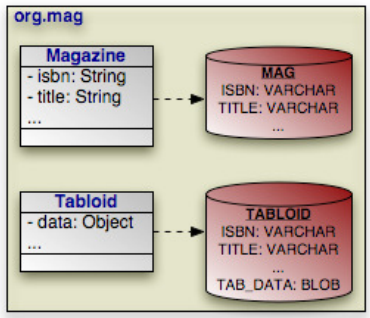
9.3 Vererbung

- Hibernate unterstützt Vererbung in relationale Datenbanken
- drei verschiedene Vererbungsstrategien:

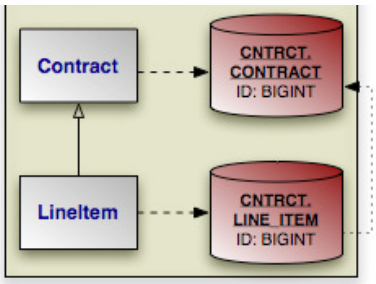
  - SINGLE_TABLE
    - eine Tabelle für die Abbildung einer Hierarchie
    - Unterscheidungsmerkmal ist die Diskriminator Spalte
    - Polymorphe Abfragen sind möglich



  - TABLE_PER_CLASS
    - eine Tabelle pro konkreter Entity-Klasse
    - Unterstützung durch Implementierung optional
    - jede Tabelle enthält ALLE Attribute (auch Superclass)
    - keine polymorphen Abfragen möglich
    - SQL UNION als Behelf

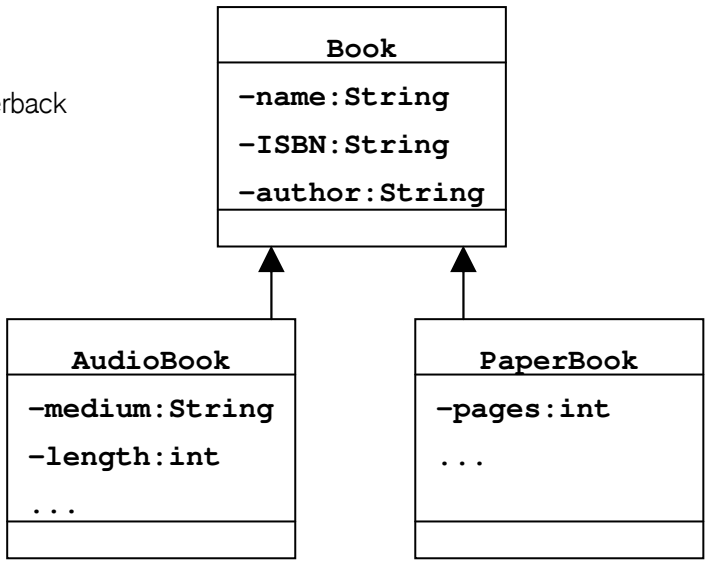


  - JOINED
    - eine Tabelle je Superklasse und abgeleitete Klasse
    - eine Tabelle je Klasse einer Vererbungshirarchie
    - jede Tabelle enthält nur spezifische Attribute der Klasse
    - polymorphe Abfragen sind möglich (nicht performant)



http://openjpa.apache.org

- Vererbungshierarchie von Book, Audiobook und Paperbook
  - Entities AudioBook und Paperback erben von der Entity Book



Vererbungshierarchie aus dem Beispielprojekt

9.3.1 SINGLE_TABLE

- einfachste Möglichkeit
- Defaultstrategie
- Klassen werden in einer einzigen Tabelle abgebildet
- Annotation @Inheritance bestimmt Vererbungsstrategie
- Annotation @DiscriminatorValue definiert mapping zwischen Objekt und Tabellen-Eintrag
- Annotation @DiscriminatorColumn definiert Typ und der Name der Discriminatorspalte
  - beschreibt Vererbung, z.B: AudioBook oder PaperBack
  - Defaultwerte : Name (DTYPE), Typ (String)

Beispiel mit SINGLE_TABLE

- abstrakte Klasse Book bestimmt Vererbungsstrategie mit InheritanceType
- konkrete Klassen AudioBook und PaperBack definieren Discriminatorvalue

```
package library;
import java.util.HashSet;
import java.util.Set;
import org.hibernate.annotations.GenericGenerator;
//import org.hibernate.annotations.CascadeType;
import javax.persistence.*;
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@Table(name = "T_BOOK")
public abstract class Book {
   private Long id;
   private String name;
   private String ISBN;
   private Publisher publisher;
   private Set<User> users = new HashSet<User>();
   public Book(String name, String isbn) {
      super();
      this.name = name;
      ISBN = isbn;
   }
   protected Book() {}
   @Id
   @GeneratedValue(generator = "increment")
   @GenericGenerator(name = "increment", strategy = "increment")
   public Long getId() {
      return id;
   }
   protected void setId(Long id) {
      this.id = id;
   }
   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }
   public String getISBN() {
```

```
    return ISBN;
  }
  public void setISBN(String ISBN) {
    this.ISBN = ISBN;
  }
  @ManyToOne
  public Publisher getPublisher() {
    return publisher;
  }
  public void setPublisher(Publisher publisher) {
    this.publisher = publisher;
  }
  @ManyToMany(mappedBy = "books")
  public Set<User> getUsers() {
    return users;
  }
  public void setUsers(Set<User> users) {
    this.users = users;
  }
  public void addUser(User user) {
    this.users.add(user); // weist Book dem User-Set zu
    user.getBooks().add(this); // weist User dem Book-Set zu
  }
}
```

Book.java mit Vererbungsstrategie SINGLE_TABLE

```
package library;
import javax.persistence.*;
@Entity
@DiscriminatorValue(value = "AudioBook")
public class AudioBook extends Book {
  private String narrator;
  private String medium;
  private int lengthInMinutes;
  public AudioBook(String name, String isbn, String narrator,
      String medium, int lengthInMinutes) {
    super(name, isbn);
    this.narrator = narrator;
    this.medium = medium;
    this.lengthInMinutes = lengthInMinutes;
  }
  public String getNarrator() {
    return narrator;
  }
  public void setNarrator(String narrator) {
    this.narrator = narrator;
  }
  public String getMedium() {
    return medium;
  }
  public void setMedium(String medium) {
    this.medium = medium;
  }
  public int getLengthInMinutes() {
    return lengthInMinutes;
  }
  public void setLengthInMinutes(int lengthInMinutes) {
    this.lengthInMinutes = lengthInMinutes;
  }
```

```
}
```

AudioBook.java mit DiscriminatorValue

```
package library;
import javax.persistence.*;
@Entity
@DiscriminatorValue(value = "Paperback")
public class PaperBack extends Book {
   private int pages;
   public PaperBack(String name, String isbn, int pages) {
      super(name, isbn);
      this.pages = pages;
   }
   public int getPages() {
      return pages;
   }
   public void setPages(int pages) {
      this.pages = pages;
   }
}
```

PaperBack.java mit DiscriminatorValue

- AudioBook und PaperBack in Single_Table erzeugen

```
package library;
import util.HibernateUtil;
import org.hibernate.Session;
import org.hibernate.Transaction;
public class SaveBookAsSingleTable {
   public static void main(String[] args) {
      Book audioBook = new AudioBook("ab_name1", "ab_isdn1",
           "ab_narriator1", "ab_medium1", 1);
      Book paperBook = new PaperBack("pb_name1", "pb_isdn1", 100);
      Session session = HibernateUtil.getSessionFactory()
           .openSession();
      Transaction tx = session.beginTransaction();
      // Nicht persistente Book(s) als Single Table persistieren
      session.persist(audioBook);
      session.persist(paperBook);
      tx.commit();
      session.close();
   }
}
```

SaveBooksAsSingleTable.java für Vererbungsstrategie SINGLE_TABLE

Hibernate erstellt folgende Tabelle:

library:t_book

DTYPE	id	name	ISBN	narrator	medium	lengthInMinutes	pages	publisher_id
AudioBook	1	ab_name1	ab_isdn1	ab_narriator1	ab_medium1	1	<NULL>	<NULL>
Paperback	2	pb_name1	pb_isdn1	<NULL>	<NULL>	<NULL>	100	<NULL>

Vorteil der SINGLE_TABLE-Strategie

- gute Performance, da polymorphe Abfragen nur immer eine Tabelle betreffen
  - geeignet für wenig Attributte

- polymorphe Abfragen

Nachteil der SINGLE_TABLE-Strategie:

- grosse Vererbungshierarchien erzeugen sehr grossen Tabelle
- vielen redundante Spalten
- schlechte Datenintegrität da Felder der abgeleiteten Klassen NULL Werte haben

9.3.2 TABLE_PER_CLASS

- jede konkrete Klasse in eigene Tabelle
- Tabellen eindeutig den Entities zugeordnet
- kein Unterscheidungsfeld (Discriminator) notwendig

Die Persistenzprovider sind nach der Spezifikation der Java Persistence API (EJB 3.0) nicht verpflichtet, die TABLE_PER_CLASS-Strategie bereitzustellen.

Beispiel mit TABLE_PER_CLASS

- abstrakte Klasse Book bestimmt Vererbungsstrategie
- Tabellen für konkrete Entitäten erhalten auch die Attribute der geerbten Klasse

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
@Table(name = "T_BOOK")
public abstract class Book {
   ...
```

abstract Book.java mit Vererbungsstrategie TABLE_PER_CLASS

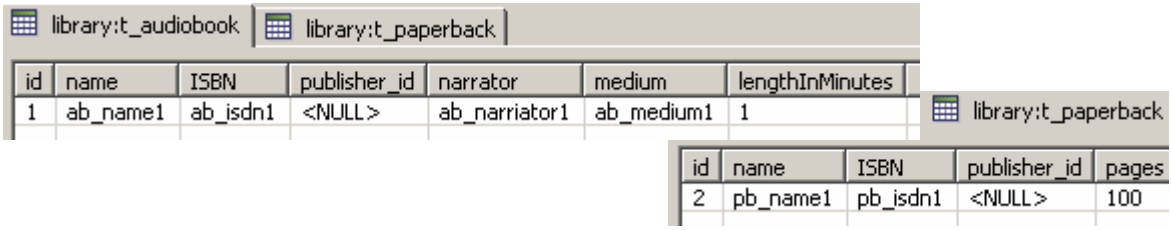
```
@Entity
@Table(name = "T_AUDIOBOOK")
public class AudioBook extends Book {
...
```

AudioBook.java

```
@Entity
@Table(name = "T_PAPERBACK")
public class PaperBack extends Book {
...
```

PaperBack.java

Hibernate erstellt folgende Tabelle:



Datenbanktabellen bei Vererbungsstrategie TABLE_PER_CLASS

Vorteil der TABLE_PER_CLASS –Strategie

- geeignet für Veerbungshierarchien ohne polymorphe Abfragen und Beziehungen
- Abfragen auf konkrete Klassen sehr einfach und performant:

```
SELECT id, name, ..., pages FROM Paperback
```

Nachteil der TABLE_PER_CLASS -Strategie:

- Tabellenstruktur ist abhängig von den Superklassen
- polymorphe Abfragen nicht optimal unterstützt
  - Abfragen über alle Books:

```
SELECT id, name, narrator, medium, length
FROM AudioBook
WHERE ...

SELECT id, name, pages
FROM Paperback
WHERE ...
```

- Hibernate unterstützt SQL-Unions:

```
SELECT * FROM (
  SELECT id, name, narrator, ..., 'A' AS type
  FROM AudioBook
  UNION
  SELECT id, pages, 'P' AS type
  FROM Paperback )
WHERE ...
```

9.3.3 JOINED

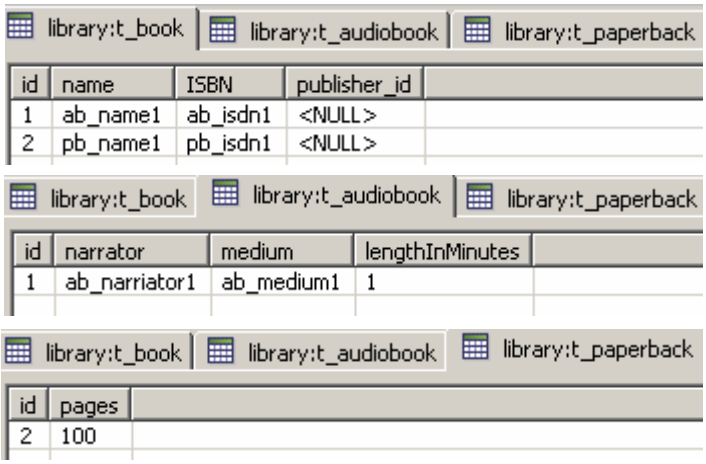
- jede abstrakte und jede konkrete Klasse hat eigene Tabelle

```java
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
@Table(name = "T_BOOK")
public abstract class Book {
```

abstract Book.java mit Vererbungsstrategie JOINED

Hibernate erstellt folgende Tabelle:

- t_book
- t_audiobook
- t_paperback



library:t_book

id	name	ISBN	publisher_id
1	ab_name1	ab_isdn1	<NULL>
2	pb_name1	pb_isdn1	<NULL>

library:t_audiobook

id	narrator	medium	lengthInMinutes
1	ab_narriator1	ab_medium1	1

library:t_paperback

id	pages
2	100

Datenbanktabellen bei Vererbungsstrategie JOINED

- jede Tabelle hat einen Primärschlüssel id
- Primärschlüssel id ist auch Fremdschlüssel zur Superklasse t_book
- JOINED-Strategie benötigt keinen Discriminator
  - optionale Definition möglich für Implementierungen anderer Persistenzprovider

Vorteil der JOINED-Strategie:

- Datenbankintegrität wird nicht verletzt
  - Attribute in den Subklassen haben keine NULL Werte
- polymorphen Abfragen und Assoziationen sind möglich

Nachteil der JOINED-Strategie:

- komplexe Abfragen
- schlechte Performance
- Abfrage auf eine konkrete Klasse benötigt Inner-Joins

```sql
SELECT b.id, b.name, p.pages
FROM t_paperback p
INNER JOIN t_book b ON b.id = p.id
WHERE ...
```

- für Abfragen über alle Books sind Outer Joins notwendig

```
SELECT b.id, b.name,  b.ISBN, p.pages, a.narrator, a.medium,
a.lengthinminutes
FROM t_book AS b
LEFT OUTER JOIN t_paperback AS p ON b.id = p.id
LEFT OUTER JOIN t_audiobook AS a ON b.id = a.id
WHERE ...
```

10 Collections

Assoziationen zwischen Entities werden als Collections umgesetzt.

- auch Verwendung von Value Types
  - repräsentieren keine eigenständigen Entities innerhalb der Datenbank

- Collections als Attribute von persistenten Klassen
  - durch Deklaration des Typ als Interface

Folgende Interfaces sind möglich:

- java.util.Set
- java.util.Collection
- java.util.List
- java.util.Map
- java.util.SortedSet
- java.util.SortedMap
- oder selbst definierte Interfaces durch die Implementation von org.hibernate.usertype.UserCollectionType

> Die Verwendung von Collections in persistenten Klassen erlaubt nur die erwähnten Interfaces

- folgender Code wirft eine Laufzeit-Exception:

```
Publisher publisher = new Publisher();
Set books = new HashSet<String>();
Session session = HibernateUtil.getSessionFactory().openSession();
Transaction tx = session.beginTransaction();
publisher = (Publisher) session.get(Publisher.class,(long) 1);
books = publisher.getBooks(); // ok
HashSet my_books = (HashSet) publisher.getBooks(); // Exception
```

Falsche Verwendung von Collections

- Hibernate ersetzt HashSet während persist(...) durch eigene Implementierung
  - durch Entity referenzierte Collection-Instanzvariablen werden persistiert
  - referenzlose Collection-Instanzvariablen werden gelöscht
- zwei Entities dürfen nocht auf gleiche Collection-Instanz verweisen
  - Hibernate unterscheidet nicht zwischen einer Null-Referenz auf eine Collection und einer leeren Collection

10.1 Collection Mapping

In Hibernate stehen folgende Mappings zur Umsetzung von Collections zur Verfügung:

- <set>
- <list>
- <map>
- <bag>
- <array>
- <primitive-array>

folgende Parameter sind alle optional (ausser name):

Parameter	Beschreibung
name	Name der Entity Instanzvariable vom Typ Set
access	Gibt an, ob Hibernate direkt auf das Collection Attribut zugreifen soll oder eine Getter-Methode verwendet werden soll (Default = Zugriff per Getter)
table	Name der Collection Tabelle (Default = Name der Instanzvariable)
schema	Name des Datenbankschemas. Überschreibt die Angabe im übergeordneten <class>-Element (Default = Einstellung aus <class>)
lazy	Kann auf false gesetzt werden, um Lazy Fetching zu deaktivieren. Die Elemente der Collection werden stets zusammen mit der Entity geladen (Default = true)
sort	Erlaubt die Definition einer sortierten Collection. Mit natural wird die normale Reihenfolge der Collectionelemente als Sortierkriterium verwendet. Alternativ kann eine Comparator-Klasse angegeben werden (Default = keine Sortierung)
inverse	Mit true wird die Collection als inverses Ende einer bidirektionalen Assoziation gekennzeichnet (Default = false)
mutable	Durch Setzen auf false wird definiert, dass sich der Inhalt der Collection niemals ändert (Default = true)
cascade	Ermöglicht die Kaskadierung von z. B. Löschoperation auf Kind-Elemente innerhalb der Collection (Default = none)
order-by	Durch Angabe einer Tabellenspalte können die Elemente der Collection sortiert werden. Durch die optionale Angabe von asc oder desc kann die Reihenfolge auf aufsteigende oder absteigende Reihenfolge festgelegt werden (Default = keine Sortierung)
where	Durch Angabe einer SQL-WHERE-Anweisung können die Elemente der Collection eingeschränkt werden (Default = keine Einschränkungen)
batch-size	Definiert die Batchgrösse in Bezug auf lazy Fetching, also die Anzahl der Collectionelemente, die gleichzeitig aus der Datenbank geladen werden (Default = 1)
fetch	Auswahlmöglichkeit zwischen outer-join-Fetching, Fetching durch sequenzielle Selects und Fetching durch sequenzielle Subselects (Default = select)
optimistic-lock	Ermöglicht Optimistic Locking, d. h., bei Zustandsänderungen der Collection wird die Version der Entity erhöht (Default = true)

Innerhalb von Set können folgende Subelemente verwendet werden:

- <key>
- <element>
- <composite-element>
- <one-to-many>
- <many-to-many>
- <many-to-any>

- Collection Mappings mittels Annotation
  - @OneToMany, @ManyToMany, @ManyToOne, @OneToOne
  - EJB-3.0 benützt hibernate-spezifische Annotation @org.hibernate.annotations.CollectionOfElements

Beispiel : Set<String> Email-Adresse pro User

- @CollectionOfElements deklariert Instanzvariable emailAddresses als Collection von Value-Typen
- @JoinTable gibt die Tabelle der Email-Adressen an
- @JoinColumn enthält Tabellenspalte mit Fremdschlüssel der Entity
- @Column definiert den Name der Tabellenspalte der Email-Adressen

```
package library;
import java.util.Set;
import javax.persistence.*;
import org.hibernate.annotations.CollectionOfElements;
import org.hibernate.annotations.GenericGenerator;
import org.hibernate.annotations.Parameter;
@Entity
@Table(name = "T_USER")
public class User {
  private Long id;
  private Address address;
  private String firstname;
  private String lastname;
  private Set<String> emailAddresses;
  private User() {}
  public User(String firstname, String lastname,
       Set<String> emailAddresses) {
    this.firstname = firstname;
    this.lastname = lastname;
    this.emailAddresses = emailAddresses;
  }
  @Id
  @GeneratedValue(generator = "foreign")
  @GenericGenerator(name = "foreign", strategy = "foreign",
parameters = { @Parameter(name = "property", value = "address") })
```

```
  public Long getId() {
    return id;
  }
  private void setId(Long id) {
    this.id = id;
  }
  @OneToOne
  @PrimaryKeyJoinColumn
  public Address getAddress() {
    return address;
  }
  public void setAddress(Address address) {
    this.address = address;
  }
  public String getFirstname() {
    return firstname;
  }
  public void setFirstname(String firstname) {
    this.firstname = firstname;
  }
  public String getLastname() {
    return lastname;
  }
  public void setLastname(String lastname) {
    this.lastname = lastname;
  }
  @CollectionOfElements
  @JoinTable( name = "T_EMAIL",
              joinColumns = @JoinColumn(name = "USER_ID"))
  @Column(name = "email", nullable = false)
  public Set<String> getEmailAddresses() {
    return emailAddresses;
  }
  public void setEmailAddresses(Set<String> emailAddresses) {
    this.emailAddresses = emailAddresses;
  }
}
```

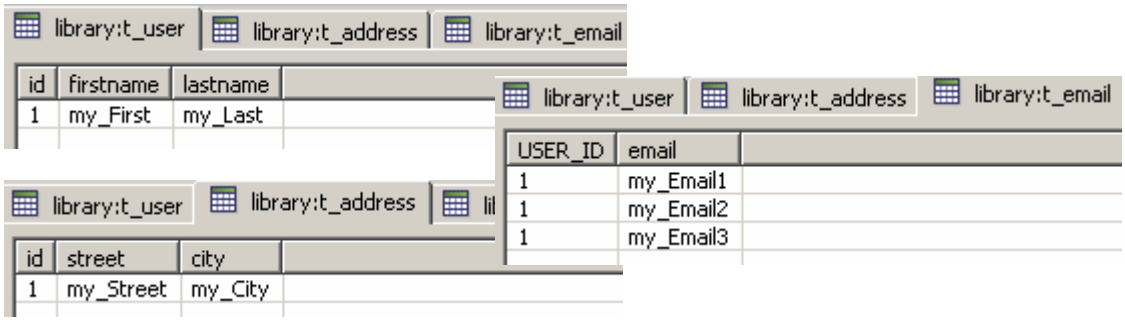
Klasse User mit mehreren Email-Adressen

- User mit mehreren Email speichern:

```
package library;
import java.util.HashSet;
import java.util.Set;
import org.hibernate.*;
import util.HibernateUtil;
public class SaveUserWithEmails {
  public static void main(String[] args) {
    Session session = HibernateUtil.getSessionFactory()
        .openSession();
    Transaction tx = session.beginTransaction();
    Set<String> emails = new HashSet<String>();
    emails.add("my_Email1");
    emails.add("my_Email2");
    emails.add("my_Email3");
    User user = new User("my_First", "my_Last", emails);
    user.setAddress(new Address("my_Street", "my_City"));
    session.persist(user);
    tx.commit();
```

```
      session.close();
      session = HibernateUtil.getSessionFactory().openSession();
      tx = session.beginTransaction();
      user = (User) session.get(User.class, new Long(1));
      System.out.println("Emails: " + user.getEmailAddresses());
      tx.commit();
      session.close();
    }
  }
```

- folgende Email Tabelle wird dabei generiert:



10.2 Collections mit Index

- Indexierte Collections z. B. List, Map oder Arrays
  - erlauben Zugriff über Index
- müssen Indexspalte (bzw. ein Key bei Maps) definieren
  - enthält den Index der Collection
  - List und Array erlauben nur Integer
  - Map erlaubt beliebiger Basistyp

Beispiel

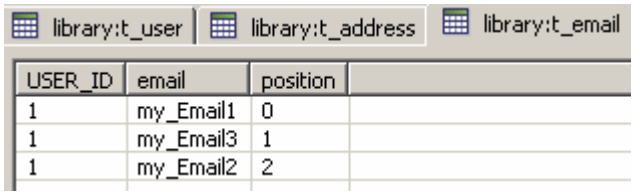
- Typ der Instanzvariable emailAddresses auf List<String> ändern
  - Liste von E-Mails haben feste Reihenfolge
  - List Index wird mit @IndexColumn auf Tabellenspalte position gemappt

```
@Entity
@Table(name = "T_USER")
public class User {
   ...
   private List<String> emailAddresses;
   public User(String firstname, String lastname,
        List<String> emailAddresses) {
     this.firstname = firstname;
     this.lastname = lastname;
     this.emailAddresses = emailAddresses;
```

```
    }
    ...
    @CollectionOfElements
    @JoinTable( name = "T_EMAIL",
                joinColumns = @JoinColumn(name = "USER_ID"))
    @Column(name = "email", nullable = false)
    @IndexColumn(name = "position")
    public List<String> getEmailAddresses() {
       return emailAddresses;
    }
    public void setEmailAddresses(List<String> emailAddresses) {
       this.emailAddresses = emailAddresses;
    }
...
```

Klasse User mit Liste von Email-Adressen

erzeugte Tabelle t_email mit List-Index position:



USER_ID	email	position
1	my_Email1	0
1	my_Email3	1
1	my_Email2	2

10.3 Sortierte Collections

- sortierte Collections in Hibernate implemntieren java.util.SortedMap und java.util.SortedSet
  - Annotation @org.hibernate.annotations.Sort ist Mapping Parameter
  - Art der Sortierung (NATURAL, UNSORTED, COMPARATOR)
  - COMPARATOR verlangt Benutzerspezifische Comparatorklasse:

```
@Sort(type = SortType.COMPARATOR, comparator = Comparator.class)
```

- normale Collections beim Laden der Daten durch Datenbank sortiert
  - durch Annotation @javax.persistence.OrderBy (EJB3 Spezifikation)
  - Beispiel : @javax.persistence.OrderBy("title asc, id desc)
  - Achtung : @org.hibernate.annotations.OrderBy definiert SQL-Anweisung

Beispiel:

- Sortieren des Set<String> der Email Adressen beim Laden
  - verhält sich wie ein LinkedHashSet

```
    @CollectionOfElements
    @JoinTable( name = "T_EMAIL",
                joinColumns = @JoinColumn(name = "USER_ID"))
```

```
@Column(name = "email", nullable = false)
@org.hibernate.annotations.OrderBy(clause ="email")
public Set<String> getEmailAddresses() {
   return emailAddresses;
}
public void setEmailAddresses(Set<String> emailAddresses) {
   this.emailAddresses = emailAddresses;
}
```

Klasse User mit Email-Adressen, die beim Laden sortiert werden

11 Hibernate Session und Transaktionen

- verschiedene Anwendungsszenarien (Patterns) für transaktionale Operationen

11.1 Transaktion

- ACID-Eigenschaften
  - Atomic (atomar)
    Eine Transaktion ist Reihe von "primitiven" unteilbaren Operationen. Es werden entweder alle Operationen innerhalb einer Transaktion ausgeführt oder gar keine.
  - Consistent (konsistent)
    Die Änderungen einer Transaktion hinterlassen die Datenbank in einem konsistenten Zustand.
  - Isolated (isoliert)
    Gleichzeitige Transaktionen beeinflussen sich nicht.
  - Durable (dauerhaft)
    Abgeschlossen Transaktion sind dauerhaft.

- beenden einer Transaktionen
  - commit
    Erfolgreiche Beendigung einer Transaktion, die Änderungen sind dauerhaft in der Datenbank abgespeichert und für alle sichtbar.
  - rollback
    Der Abbruch einer Transaktion führt Änderungen am Datenbestand zum Zustand des letzten commit zurück.

11.2 Isolationsebenen

- komplette Serialisierung der transaktionalen Operationen
  - sicher aber teuer
- Isolationsebene ist Einschränkung im Vergleich zu idealen ACID Transaktionen
  - (nach ANSI-92 SQL):

Isolation Level	Beschreibung
Read Uncommitted	Die Transaktion kann alle Daten lesen, unabhängig davon, ob Write Locks vorhanden sind. Zusätzlich wird die Transaktion in diesem Modus keine eigenen Read Locks absetzen.
Read Committed	Die Transaktion kann nur Daten lesen, welche bereits committed wurden, d.h. es muss gewartet werden, bis offene Write Locks abgeschlossen sind. Für das Lesen von Daten werden neue Read Locks

Isolation Level	Beschreibung
	abgesetzt, welche aber bereits nach Abschluss des SQL Statements (d.h. vor Ende der Transaktion!) wieder freigegeben werden.
Repeatable Read	Analog zu Read Committed mit der Ausnahme, dass die Read Locks bis ans Ende der Transaktion aufrecht erhalten werden.
Serializable	Analog zu Repeatable Read wobei zusätzlich sichergestellt wird, dass selbst bei mehrmaligem Ausführen von ein und demselben SQL Statement immer dieselben Resultate gesehen werden.

- Probleme bei parallelen Datenbankoperationen
  - Dirty Read
    Innerhalb einer Transaktion (T1) wird ein Datensatz verändert. Dieser veränderte Datensatz wird innerhalb einer zweiten Transaktion (T2) gelesen, bevor T1 abgeschlossen wurde (Commit). Wird nun T1 mit einem Rollback abgebrochen, arbeiten die Operationen innerhalb von T2 mit einem ungültigen Wert.
  - Non Repeatable Read
    Innerhalb einer Transaktion (T1) wird ein bestimmter Datensatz gelesen. Direkt nach dem Lesen von T1, aber noch vor einem Commit von T1, verändert eine zweite Transaktion (T2) diesen Datensatz und wird mit einem Commit beendet. Liest nun T1 diesen Datensatz erneut, wird ein anderer Inhalt zurückgegeben als zu Beginn, obwohl aus der Sicht von T1 der Datensatz nicht verändert wurde.
  - Phantom Read
    Innerhalb einer Transaktion (T1) wird eine Abfrage an die Datenbank gestellt, welche eine bestimmte Anzahl von Ergebnisdatensätzen liefert. Eine zweite Transaktion (T2) ändert den Inhalt der Datenbank, indem es neue Datensätze einfügt und wird mit einem Commit beendet. Führt nun T1 die gleiche Abfrage erneut aus, so werden mehr Ergebnisdatensätze als beim ersten Mal gefunden.

Probleme bei Isolationsebenen:

Isolationsebene	Dirty Read	Non Repeatable Read	Phantom Read
Read Uncommitted	✓	✓	✓
Read Committed	✗	✓	✓
Repeatable Read	✗	✗	✓
Serializable	✗	✗	✗

Mögliche Probleme bei verschiedenen Isolationsebenen

- mit zunehmendem Isolationsgrad
  - sicherere Transaktion
  - schlechtere Performance, da mehr Sperren innerhalb der Datenbank

11.3 Transaktionen in Hibernate

- basieren auf der Verwendung von
  - JDBC Connections
  - Java Transaction API (JTA)
- optimistischen Locking
  - beim Schreiben wird geprüft ob sich die Daten verändert haben
  - implementiert durch die Verwendung von Versionsattributen @Version
- pessimistisches Locking
  - lesender Client sperrt Daten
  - implementiert durch API die SELECT FOR UPDATE der Datenbank umsetzt
- Repeatable-Read-Semantik definert Caching innerhalb Hibernate Session
  - dabei muss jede Datenbankoperation innerhalb einer Transaktion laufen
  - auch reine Leseoperationen
  - aktiver Auto-Commit-Modus wird durch Hibernate deaktiviert
    - sollte generell nicht in Verbindung mit Hibernate eingesetzt werden

Verwendung von Transaktionen in Hibernate unterscheidet:

- managed : innerhalb eines Java EE Application Servers
  - BMT (Bean Managed Transactions)
  - CMT (Container Managed Transactions)
- non-managed : in einer Java-SE-Anwendung

Interface org.hibernate.Transaction verwaltet BMT:

```
Session mySession = factory.openSession(); // Session erzeugen
Transaction tx = null;
try {
  tx = mySession.beginTransaction();       // Transaktions Start
  ...                                      // CRUD Create,Read …
  tx.commit();                             // Transaktions Ende
} catch (RuntimeException e) {
  if (tx != null) tx.rollback();           // Transaktion ROLLBACK
  throw e;
}
```

Starten und Beenden einer Transaktion über die Hibernate API

- Code als managed sowie unmanaged Transaktion lauffähig
  - verwendet nur Hibernate API
  - innerhalb eines Application Servers
    - als JTA (Java Transaction API)
      Verwaltung der Transaktionen sowie Datasource geschieht über Application Server
    - als UserTransaction
      verwendet mit sessionfactory.getCurrentSession() aktuelle Session
  - ausserhalb eines Application Servers meistens als JDBC-Transaktionen

UserTransaction über JNDI lookup()

```
try {
   UserTransaction utx = (UserTransaction)
   new InitialContext().lookup("java:comp/UserTransaction");
   utx.begin();
   ...                                          // CRUD Create,Read ...
   factory.getCurrentSession().load(...);
   ...
   utx.commit();
} catch (RuntimeException e) {
   utx.rollback();
   throw e;
}
```

Verwendung einer JTA UserTransaction

11.4 Optimistisches Locking

"optimistisches" Locking bedeutet: "meistens geht alles gut"

- hohe Performance
- gute Skalierbarkeit
- keine Überschneidungen von Änderungen durch parallele Transaktionen
  - Auftreten einer Überschneidung wird durch aufwändige, manuelle Korrektur behoben
    - z.B. durch Versionierung der Datensätze
  - oder auch Last Commit Wins
    - Verwirrungen seitens der User

Die aufwändigste Verwendung ist die manuelle Überprüfung der Versionen:

```
package library;
import org.hibernate.*;
import util.HibernateUtil;
public class LoadUserWithVersioning {
   public static void main(String[] args) {
```

```
        Session s;
        Transaction t;
        s = HibernateUtil.getSessionFactory().openSession();
        t = s.beginTransaction();
        User user = new User(null, "original");
        s.persist(user);
        t.commit();
        s.close();
        // new user erzeugt
        s = HibernateUtil.getSessionFactory().openSession();
        t = s.beginTransaction();
        User user_A = (User) s.get(User.class, (long) 1);
        t.commit();
        s.close();
        // user_A detached
        s = HibernateUtil.getSessionFactory().openSession();
        t = s.beginTransaction();
        User user_B = (User) s.get(User.class, new Long(1));
        user_B.setLastname("new");
        s.persist(user_B);
        t.commit();
        s.close();
        // user auf DB durch user_B geändert
        s = HibernateUtil.getSessionFactory().openSession();
        t = s.beginTransaction();
        // Versionsvergleich
        if (user_A.getVersion() !=
           ((User) s.get(User.class, user_A.getId())).getVersion())
           throw new StaleObjectStateException(null, s);
        user_A.setLastname("newer");
        s.merge(user_A);
        t.commit();
        s.close();
    }
}
```

Manuelle Überprüfung von Versionen bei optimistischem Locking

- manuelle Überprüfung mittels Versionsvergleich für jede Entity
- Versionsattribut der Entities wird automatisch beim Commit hochgezählt
- manuelle Ansatz nur bei kleinen, trivialen Persistenzlösungen praktikabel
  - manuelle Überprüfung eines komplexen Objektgraphen nicht einfach

- Hibernate bietet automatische Versionsüberprüfung
  - für erweiterte Sessions
    - Session wird nach Beenden einer Transaktion nicht geschlossen
    - Entities bleiben erhalten (Session ist ein Cache für die Entities)
  - für Detached Objects
  - mit lock(Object object, LockMode READ) auch für Entities die nicht innerhalb der Transaktion verändert wurden
  - wirft eine Stale-ObjectStateException

- Mappingattribut optimistic-lock **false** deaktiviert Versionsüberprüfung
- Mappingattribut optimistic-lock **all** vergleicht alle Attribute
- Mappingattribut optimistic-lock **dirty** vergleicht nur Attribute der aktuellen Transaktion

11.5 Pessimistisches Locking

häufige Kollisionen zwischen konkurrierenden Transaktionen

- restriktives Vorgehen verlangt
- Tabellenzeilen werden explizit gesperrt
  - konkurrierende Transaktion erhält Fehlermeldung
  - konkurrierende Transaktion muss warten
- verschlechtert Gesamtperformance des Systems aufgrund wartenden Transaktionen

Locking-Mechanismus

- um Deadlocks innerhalb der Datenbank zu vermeiden
- Hibernate nutzt für pessimistischem Locking die Datenbank Funktionalität
- Hibernate setzt nie Locks auf Entity-Objekte im Speicher!
- Klasse LockMode definiert verschiedene Lock-Modi
  - es stehen nicht immer alle Modi zur Verfügung (Datenbankabhängig)
  - falls gewünschter Lock-Modus nicht unterstützt wird wählt Hibernate automatisch einen verwandten, verfügbaren Modus

Folgende Lock-Modi werden in LockMode definiert:

LockMode	Beschreibung
NONE	Es wird kein Lock auf Zeilen in der Datenbank gesetzt. Mit diesem Modus wird beim Lesen eines Datensatzes nur auf die Datenbank zugegriffen, wenn sich die entsprechende Entity nicht im Cache befindet.
READ	Dieser Lockmodus weist Hibernate an, direkt auf die Datenbank zuzugreifen und eine Versionsüberprüfung der betroffenen Entities durchzuführen. Sinnvoll z. B. bei Verwendung von Detached Entities. Dieser Lockmodus wird von Hibernate automatisch verwendet, wenn als Isolationsebene Repeatable Read oder Serializable ausgewählt wurde.
UPGRADE	Wird für pessimistisches Locking verwendet. Es wird mit SELECT ... FOR UPDATE ein Lock auf die Tabellenzeilen gesetzt, wenn die Datenbank dieses Feature unterstützt.

LockMode	Beschreibung
UPGRADE_NOWAIT	Verhält sich wie UPGRADE. Durch ein SELECT ... FOR UPDATE NOWAIT, welches in Oracle Datenbanken verfügbar ist, wird die DB angewiesen, nicht darauf zu warten, falls die selektierte Tabellenzeile bereits gelockt ist, sondern eine Locking Exception zu werfen.
WRITE	Ein "interner" Modus, der automatisch von Hibernate verwendet wird, wenn ein Datensatz aktualisiert oder eingefügt wird. Dieser Modus kann nicht explizit durch den User verwendet werden.

- Lock-Modi erlauben pessimistisches Locking auf feingranularen Ebene bei:
  - Session.load( ), Session. lock( ), Session.get( ) und Query.setLockMode( )

Beispiel:

- Entity vom Typ User durch Übergabe von Lock-Mode.UPGRADE in der Sessionmethode get(...) wird innerhalb der Datenbank gelockt.
- parallele schreibende Transaktion auf dieselbe Entity wirft Exception

```
package library;
import org.hibernate.*;
import util.HibernateUtil;
public class LoadUserWithLockMode {
  public static void main(String[] args) {
     Session s1;
     Session s2;
     Transaction t1;
     Transaction t2;
     s1 = HibernateUtil.getSessionFactory().openSession();
     t1 = s1.beginTransaction();
     User user = new User(null, "original");
     s1.persist(user);
     t1.commit();
     s1.close();
     // User erzeugt new user
     s1 = HibernateUtil.getSessionFactory().openSession();
     t1 = s1.beginTransaction();
     User user_A = (User) s1.get(User.class, (long) 1,
          LockMode.UPGRADE);
     // User_A setzt Lock auf user
     s2 = HibernateUtil.getSessionFactory().openSession();
     t2 = s2.beginTransaction();
     User user_B = (User) s2.get(User.class, (long) 1);
     user_B.setLastname("new");
     s2.persist(user_B);
     // User_B möchte user ändern
     t1.commit();
     s1.close();
     t2.commit();
     s2.close();
  }
}
```

Locking einer Entity auf Datenbank

12 Datenbankabfragen

- Entities in einer Datenbank mittels Abfragen finden
- Query Interface
  - zentrale Schnittstelle zur Ausführung von Abfragen
- Hibernate Query Language (HQL)
  - starke, komplett objektorientierte Abfragesprache
- Criteria API
  - Abfragen durch Aufruf einer normalen API formuliert
- Abfragen per SQL formuliert

12.1 Query Interface

- zentrale Schnittstelle zur Ausführung von Abfragen
- immer wenn Primärschlüssel einer Entity nicht bekannt
- Suchkriterien zum Finden der Entities
- Query-Instanz wird mit Hilfe der aktuellen Session erzeugt
- String-Parameter muss eine gültige HQL-Abfrage
- oder mit createSQLQuery( ) ein SQL String definieren
  - Datenbankabhängig

```
session = HibernateUtil.getSessionFactory().openSession();
tx = session.beginTransaction();
// Neue Abfrage erzeugen:
Query q = session.createQuery("from User");
// Ausführung der Abfrage:
List<User> l = q.list();
// über die Ergebnisliste iterieren:
for ( User u : l) {
  System.out.println(u. getLastname() ());
}
tx.commit();
session.close();
```

Erzeugung einer Query-Instanz mit Hilfe der Hibernate Session

12.1.1 Methoden des Query Interface

Methode	Beschreibung
List list( )	Das Ergebnis wird als Liste zurückgegeben, das heisst, das Resultat der Abfrage befindet sich komplett im Speicher.
Object uniqueResult( )	Ist von vorneherein bekannt, dass eine Abfrage nur ein einziges Objekt zurück liefert, kann diese Methode verwendet werden, um den Umweg über die Liste zu sparen.
Iterator iterate( )	Ermöglicht die Iteration über die Ergebnismenge. Im Unterschied zu list() werden die Ergebnisse nicht komplett im Speicher gehalten, sondern es wird für jedes Element in der Ergebnismenge eine separate Abfrage durchgeführt. Die erste Abfrage liefert nur die Identifier der Entities. Dieses Verhalten führt dazu, dass die Verwendung von iterate() normalerweise langsamer ist als jene von list(). Sollten sich aber viele Entities der Ergebnismenge bereits im Speicher, also in der Session oder dem Second Level Cache befinden, ist iterate() schneller.
ScrollableResults scroll( )	Unterstützt der verwendete JDBC-Treiber scrollbare ResultSets kann diese Methode benutzt werden. Ihre Verwendung ähnelt jener von iterate() mit dem Unterschied, dass man sich in der Ergebnismenge vor- und rückwärts bewegen kann.

12.1.2 Ausführen von Abfragen

Query Interface liefert:

- einfach Instanzen der entsprechenden Wrapperklassen (Integer, Long, etc.)
- Object Array

skalierbare Abfrage

```
Query q = session.createQuery(
   "SELECT u.id, u.firstname, u.lastname FROM User u");
Iterator i = q.list().iterator();
while (i.hasNext()) {
   Object[] listelement = (Object[]) i.next();
   Long id = (Long) listelement[0];
   String first = (String) listelement[1];
   String last = (String) listelement[2];
   System.out.println(id + " " + first + " " + last);
}
```

SQL String durch createSQLQuery( )

```
Query query = session.createSQLQuery(
   "SELECT id, firstname, lastname FROM t_user");
Iterator i = query.list().iterator();
while (i.hasNext()) {
   Object[] listelement = (Object[]) i.next();
   BigInteger id = (BigInteger) listelement[0];
   String first = (String) listelement[1];
   String last = (String) listelement[2];
   System.out.println(id + " " + first + " " + last);
}
```

library:t_user

ColumnName	Type Name	Size	Scale	Nullable	AutoIncrement	Type
id	BIGINT	20	0	Not Null	false	-5
firstname	VARCHAR	255	0	Nullable	false	12
lastname	VARCHAR	255	0	Nullable	false	12

- Abhängigkeit von Datenbank
  - nicht migrationsfähig
  - TypeMapping (BIGINT) muss beachtet werden

Eingrenzung der Abfrage

- setMaxResults(int maxResults) grenzt die Ergebnissmenge ein
- setFirstResult(int firstResult) setzt einen Zeiger in der Ergebnismenge
  - Listenweise Ausgabe der Elemente

```
Query q = session
   .createQuery("SELECT u.id, u.firstname, u.lastname FROM User u")
     .setMaxResults(3).setFirstResult(1);
```

12.1.3 dynamische Abfragen

- benannte Parameter
- Wildcards ? und _

Verwendung von benannten Parametern

```
Query q = session
   .createQuery("from User user where user.firstname = :name");
q.setString("name", "my_First");
```

Verwendung von Wildcards

```
Query q = session
   .createQuery("from User user where user.firstname = ?");
q.setString(0, "my_First");
```

Im Gegensatz zu JDBC beginnt bei Hibernate der Index der Parameter immer bei Null !

12.1.4 namedQuery

- Definition per Metadaten
- als HQL oder SQL
  - Annotation @NamedQuery(...)

```
@Entity
@NamedQuery(name = "library.user.byName",
            query = "from User user where user.lasttname = ?")
@Table(name = "T_USER")
public class User {
```

  - XML Mapping

```
<hibernate-mapping package="library">
  <class name="User" table="T_USER">
    <id name="id">
      <generator class="native"/>
    </id>
    <property name="firstname" type="java.lang.String"
      update="true" insert="true" column="LASTNAME"
      length="32"/>
    <property name="lastname"/>
    <property name="email"/>
  </class>
  <query name="library.user.byName"> from User user where
    user.firstname = ? </query>
</hibernate-mapping>
```

Abfragen mit namedQueries

```
// Neue Abfrage erzeugen:
Query q = session.getNamedQuery("library.user.byName");
// Parameter setzen
q.setString(0, "my_Last");
// Ausführung der Abfrage:
List<User> l = q.list();
```

12.2 HQL – Hibernate Query Language

- Abfragesprache
- nicht case sensitive
- aber : Javaklassen und Properties entsprechend korrekt angeben
  - objektorientiert
    - Abfragen von Objekten über deren Vererbungsbeziehung
- "from user"
  - HQL : Abfrage der Entities vom Typ User oder einem Subtyp
  - SQL : Abfrage der Daten aus der Tabelle user

12.2.1 from

- Abfrage der Entity-Klasse

```
from library.User
```

```
from User              // Hibernate-Mappings Attribute
                          auto-import = true (DEFAULT)
```

```
from User  as u        // mit Alias
from User u
```

```
from User, Book        // kartesisches Produkt
```

12.2.2 where

- Einschränkung der Ergebnismenge

```
from User as user
where user.fristname = 'my_First"
```

```
from User
where firstname = 'my_First'
```

```
from User  as u        // mit Alias
from User u
```

```
from User, Book        // kartesisches Produkt
```

- Formulierung von Bedingungen
  - praktisch alle SQL bekannten Ausdrücke

Ausdruck	Beschreibung
and, or, not	logische Operatoren zur Verknüpfung von Ausdrücken
+, -, *, /	mathematische Operatoren
=, >=, <=, <>, !=, like	Vergleichsoperatoren
( )	Klammern zur Gruppierung von Ausdrücken
is null, is not null	Vergleiche auf Null
in, not in, between, is empty, is not empty, member of, not member of	Ausdrücke zum Umgang mit Mengen und Bereichen (Collections etc.)
case ... when ... then ... else ... end, case when ... then ... else ... end	Bedingungen mit Alternativen
||, concat(..., ...)	String Verknüpfungen
current_date(), current_time(), current_timestamp(), second(...), minute(...), hour(...), day(...), month(...), year(...)	Verwendung von Zeit und Datum
substring(), trim(), lower(), upper(), length(), locate()	Verwendung von Strings
abs(), sqrt(), bit_length(), mod()	sonstige mathematische Operatoren
str()	Umwandlung in einen String
cast(... as ...), extract(... from ...)	Casts zur Umwandlung in HQL-Datentypen
size(), minelement(), maxelement(), minindex(), maxindex()	sonstige Ausdrücke zum Umgang mit Collections
sign(), rtrim(), sin()	sonstige SQL-Funktionen

Beispiele:

```
from User user
where user.books.size > 3        // mit mehr als drei Bücher
```

```
from User user
where user.firstname between 'A' and 'M'
```

```
from User  user
where user.firstname in ('Markus', 'Martin')
```

```
from Book book
where book.name like '%Hibernate%'
```

12.2.3 order by

- Sortierung der Ergebnismenge
  - nur über die aktuelle Klasse (Datenbank Tabelle)

```
from User
order by lastname
```

```
from User
order by lastname asc, firstname desc
```

12.2.4 Verbundoperationen

- liefern Kombinationen der beteiligten Entities
  - inner join / join
    - 1:1-Zuordnung. Elemente ohne Zuordnung sind ausgeschlossen
  - left outer join / left join
  - 1:1-Zuordnung. inklusieve Elemente ohne Zuordnung der linken Tabelle
  - right outer join / right join
    - inklusieve Elemente ohne Zuordnung der rechten Tabelle
  - full join
    - kartesisches Produkt

```
select user, book                    // alle User mit Books
from User user
inner join user.books book
```

```
select user, book
from User user
inner join user.books book
with book.name like '%Hibernate%'
```

```
from Publisher publisher             // mit fetch Strategie
left join fetch publisher.books
```

Fetch-Joins

- erlaubt Abfragen von Beziehungen und Collections inklusive der 'Parent Objects'
- überschreibt outer-join und lazy Deklaration für Beziehungen und Collectionen
- Einschränkungen von Fetch-Joins
  - Ergebinsmenge kann nicht iteriert werden
  - with - Konstrukt nicht geeignet
  - setMaxResults() / setFirstResult() kann nicht verwendet werden
    - Duplikate der Parents sind möglich

12.2.5 select

- definiert Ergebnismenge der Objekte und Attribute

```
select book                          // Instanzen der from Entities
from User user                       // also Book-Instanzen
inner join user.books book
```

```
select user, book                    // Liste von Object[]
from User user                       // Object[0] ist User Object
inner join user.books book           // Object[1] ist Book Object
```

```
select book, book.titel              // Liste von Object[]
from User user                       // Object[0] ist User Object
inner join user.books book           // Object[1] ist String Object
```

```
select new list(user, book)          // List als Ergebnistyp
from User user
inner join user.books book
```

```
select new UserBooks(user, book)  // typisiertes Java Object
from User user                    // UserBooks als Ergebins
inner join user.books book        // (selbst definierte Javaklasse)
```

12.2.6 Aggregat-Funktionen

- zur Konsolidierung oder Verdichtung der Ergebnismenge
- Distributive Funktionen
  - sum( ), min( ), max( ), count(*), count( ), count(distinct ...), count(all ...)
- Algebraische Funktionen
  - avg( )

```
select count(*)                   // Anzahl aller User
from User
```

```
select count(distinct user.firstname)
from User  user                   // Anzahl aller eindeutiger Namen
```

12.2.7 group by

- Gruppieren von Aggregat-Funktionen

```
select publisher.name, count(book)    // Liste aller Herausgeber
from Book book                        // mit Anzahl Bücher
inner join book.publisher publisher
group by publisher.name
from User
```

12.2.8 Polymorphe Abfragen

- SQL kennt das Konzept der Vererbung nicht
- HQL unterstüzt polymorphe Abfragen
  - liefert Ergebnismenge von Entity-Instanzen einer gemeinsamen Vererbungshierarchie

```
from Book                         // liefert auch
                                  // Instanzen von Subklassen
```

```
from java.lang.Object             // alle pesistenten Objekte
```

12.2.9 Subqueries

- falls die darunterliegende Datenbank Subqueries verwendet

```
from Book bigbook                    // Bücker die mehr Seiten als
where bigbook.size >                 // das Durchschnittsbuch haben
   ( select avg(book.size)
     from Book book )
```

```
from Book book                       // Wertemenge > 1
where not (book.title, book.isbn) in
   ( select book.title, book.isbn
     from Paperback book )
```

12.2.10 Criteria API

- einfache Art komplexe Suchmasken zu implementieren
- Alternative für komplett objektorientierten Ansatz
  - Klasse als Parameter
  - z.B. zum Finden von Entities ohne PK Information

```
Criteria criteria = session.createCriteria(User.class);
List users = criteria.list();
```

Methoden der Crteria API

- list( )
  - Ergebnisse als Liste
- uniqueResult( )
  - einzelnes Objekt
- scroll( )
  - ein ScrollableResults Objekt zum vorwärts und rückwärts bewegen

```
Criteria criteria = session.createCriteria(User.class);
ScrollableResults users = criteria.scroll();
while (users.next()) {
  User user = (User) users.get(0);
  System.out.println(user.toString());
}
```

- seitenweise Abfrage

```
Criteria criteria = session.createCriteria(User.class);
criteria.setMaxResults(10);
List users = criteria.list();
for (int counter = 0; users.size() != 0; counter += 3) {
   System.out.println(users.size() + " users found");
   criteria.setFirstResult(counter);
   users = criteria.list();
}
```

Restrictions

- Einschränkungen der Ergebnismenge

```
Criteria criteria = session.createCriteria(User.class);
criteria.add(Restrictions.eq("firstname", "Hans"));
List<User> users = criteria.list();
for (User u : users) {
   System.out.println(u.toString());
}
```

Methoden zur Erzeugung von Criterions

Methoden	Beschreibung
idEq(Object id)	Vergleicht den Primärschlüssel einer Entity
eq(String param, Object obj)	Vergleich auf Gleichheit des Attributs param mit obj
ne(String param, Object obj)	Vergleich auf Ungleichheit
like(String param, Object obj)	Entspricht der HQL-Bedingung like
ilike(String param, Object obj)	Entspricht der HQL-Bedingung like, wobei die Gross/Kleinschreibung ignoriert wird
gt(String param, Object obj)	Grösser als
lt(String param, Object obj)	Kleiner als
le(String param, Object obj)	Kleiner oder gleich
ge(String param, Object obj)	Grösser oder gleich
between(String param, Object o1, Object o2)	Im Wertebereich zwischen o1 und o2
in(String param, Object[] objs)	In objs enthalten
in(String param, Collection objs)	In objs enthalten
isNull(String param)	Vergleich auf null

Methoden	Beschreibung
eqProperty(String param1, String param2)	Vergleicht zwei Attribute einer Entity auf Gleichheit
neProperty(String param1, String param2)	Vergleicht zwei Attribute einer Entity auf Ungleichheit
ltProperty(String param1, String param2)	Vergleicht zwei Attribute einer Entity auf "Kleiner als"
leProperty(String param1, String param2)	Vergleicht zwei Attribute einer Entity auf "Kleiner oder Gleich"
gtProperty(String param1, String param2)	Vergleicht zwei Attribute einer Entity auf "Grösser als"
geProperty(String param1, String param2)	Vergleicht zwei Attribute einer Entity auf "Grösser oder Gleich"
isNotNull(String param)	Vergleicht auf nicht null
and(Criterion c1, Criterion c2)	Verknüpft zwei Criterions mit einem logischen "und"
or(Criterion c1, Criterion c2)	Verknüpft zwei Criterions mit einem logischen "oder"
not(Criterion)	Negiert die angegebene Bedingung
allEq(Map map)	Vergleicht die in der Map angegebenen Attributname/Werte-Paare auf Gleichheit
isEmpty(String param)	Prüft, ob das Collection-Attribut param keine Elemente enthält
isNotEmpty(String param)	Prüft, ob das Collection-Attribut param Elemente enthält
sizeEq(String param, int n)	Prüft, ob das Collection-Attribut param genau n Elemente enthält
sizeNe(String param, int n)	Prüft, ob das Collection-Attribut param ungleich n Elemente enthält
sizeGt(String param, int n)	Prüft, ob das Collection-Attribut param mehr als n Elemente enthält
sizeLt(String param, int n)	Prüft, ob das Collection-Attribut param weniger als n Elemente enthält
sizeGe(String param, int n)	Prüft, ob das Collection-Attribut param gleich oder mehr als n Elemente enthält

Methoden	Beschreibung
sizeLe(String param, int n)	Prüft, ob das Collection-Attribut param weniger oder gleich n Elemente enthält

sqlRestriction(String sql) definiert Bedingungen gemäss SQL-Syntax

- {alias} ist Platzhalter für Tabelle

```
Criteria criteria = session.createCriteria(User.class);
criteria.add(Restrictions
   .sqlRestriction("{alias}.email like 'e_'"));
```

- Alternative

```
Criteria criteria = session.createCriteria(User.class);
criteria.add(Property.forName("email").like("%e%"));
```

Sortierung

- addOrder(Order order) erlaubt beliebig viele Sortierkriterien

```
Criteria criteria = session.createCriteria(User.class);
criteria.add(Property.forName("email").like("%e%"));
criteria.addOrder(Order.desc("lastname"));
criteria.addOrder(Order.asc("firstname"));
```

Assoziationen

- zwei Methoden erlauben Assoziationen mit Criteria zwischen Entities

Methoden	Beschreibung
Criteria createCriteria(String associationPath)	Es wird eine neue Criteria-Instanz erzeugt, welche sich bereits auf die Elemente der durch associationPath benannten Assoziation bezieht. Mit dieser neuen Criteria-Instanz können weitere Bedingungen angegeben werden, die sich auf die Elemente der Assoziation beziehen.
Criteria createAlias(String associationPath, String alias)	Es wird keine neue Instanz von Criteria erzeugt, sondern die bestehende Instanz bezieht sich nun auch auf die Elemente der Assoziation. Die durch einen Join hinzugefügten Entities können über alias für weitere Bedingungen der Abfrage verwendet werden

Abfragen mit org.hibernate.criterion.Example

- Formulierung von Abfragen anhand von Beispielinstanzen

```
Criteria criteria = session.createCriteria(User.class);
User exampleUser = new User("f5", null, new Address(null,null));
criteria.add(Example.create(exampleUser));
List<User> users = criteria.list();
```

DetachedCriteria

- erzeugt Criteria ausserhalb einer Hibernate Session
- Clientapplikation definiert serverseitige Abfrage
- getExecutableCriteria(Session session) bindet DetachedCriteria-Instanz an Session

```
DetachedCriteria detachedCriteria = DetachedCriteria
   .forClass(User.class);
detachedCriteria.add(Restrictions.isNotNull("firstname"));
Session session = HibernateUtil.getSessionFactory().openSession();
Transaction tx = session.beginTransaction();
Criteria c = detachedCriteria.getExecutableCriteria(session);
List<User> users = c.list();
```

12.2.11 Native SQL Abfrage

- Methode createSQLQuery(String queryString) zur Nutzung spezifischer Datenbank Funktionalitäten
- koppelt Datenbank an Applikation

```
SQLQuery sqlQuery = session.createSQLQuery("select * from t_user");
sqlQuery.addEntity(User.class);
List<User> users = sqlQuery.list();
```

- mit Platzhalter für Aliase der Tabellenspalten

```
SQLQuery sqlQuery = session
   .createSQLQuery("select {user.*} from t_user user");
sqlQuery.addEntity("user", User.class);
```

- addJoin(...) fügt Assoziationen der SQL-Abfrage zu

```
SQLQuery sqlQuery = session
   .createSQLQuery("select {user.*}, {book.*}"
```

```
                + "from t_user user, t_book book"
                + "where user.book_id = book.id");
sqlQuery.addEntity("user", User.class);
sqlQuery.addJoin("book", "user.books");
```

- skalare Ergebnisse

```
SQLQuery sqlQuery = session
   .createSQLQuery("select count(*) as count from t_user");
sqlQuery.addScalar("count", Hibernate.INTEGER);
Integer count = (Integer) sqlQuery.uniqueResult();
```

12.2.12 Hibernate Filter

- Basis ähnlich wie where-Bedingung
- wiederverwendbar
- parametrierbar

Definition eines Filters

```
@Entity
@Table(name = "T_USER")
@FilterDef( name = "findName",
            parameters = {
  @ParamDef(name = "last", type = "String") })
@Filters( {
  @Filter(name = "findName", condition = ":last = lastname") })
public class User {
```

- Anwendung

```
Filter nameFilter = session.enableFilter("findName");
nameFilter.setParameter("last", "13");
Query query = session.createQuery("from User");
List<User> users = query.list();
```

13 Hibernate Types

- Typ Mapping Java : SQL
  - Attribute
  - Komponenten

13.1 Hibernate Mapping Types

- Hibernate Mapping-Typen werden klein geschrieben

Hibernate-Mapping-Typ	Java-Typ	Standard-SQL-Datentyp
string	java.lang.String	VARCHAR
short	short oder java.lang.Short	SMALLINT
integer	int oder java.lang.Integer	INTEGER
long	long oder java.lang.Long	BIGINT
float	float oder java.lang.Float	FLOAT
double	double oder java.lang.Double	DOUBLE
big_decimal	java.math.BigDecimal	NUMERIC
big_integer	java.math.BigInteger	NUMERIC
character	java.lang.String	CHAR(1)
byte	byte oder java.lang.Byte	TINYINT
boolean	boolean oder java.lang.Boolean	BIT
yes_no	boolean oder java.lang.Boolean	CHAR(1) ('Y' oder 'N')
true_false	boolean oder java.lang.Boolean	CHAR(1) ('T' oder 'F')
date	java.util.Date oder java.sql.Date	DATE
time	java.util.Date oder java.sql.Time	TIME
timestamp	java.util.Date oder java.sql.Timestamp	TIMESTAMP
calendar	java.util.Calendar	TIMESTAMP
calendar_date	java.util.Calendar	DATE

Hibernate-Mapping-Typ	Java-Typ	Standard-SQL-Datentyp
binary	byte[]	VARBINARY oder BLOB
text	java.lang.String	CLOB
serializable	java.io.Serializable (beliebige Klasse welche das Interface implementiert)	VARBINARY oder BLOB
clob	java.sql.Clob	CLOB
blob	java.sql.Blob	BLOB
class	java.lang.Class	VARCHAR
locale	java.util.Locale	VARCHAR
timezone	java.util.TimeZone	VARCHAR
currency	java.util.Currency	VARCHAR

Die Standard Hibernate Mapping-Typen

- Schemagenerierung durch Übersetzten in SQL-Standard-Typen mit Hilfe des SQL-Dialekt-Mechanismus
- spezielle Mapping-Typen für unveränderlich Objekte wie
  - date, time, timestamp, calendar, calendar_date, serializable und binary

durch

  - imm_date, imm_time, etc.

- alle Typen als Attribute von Entities verwendbar
  - Ausnahme : binary, blob und clob
- alle Typen unterstützen Null-Semantik
  - entsprechenden Attribute kann auf Wert null gesetzt werden
- alle Typen als entsprechende Konstanten unter org.hibernate.Hibernate definiert
  - z. B. org.hibernate.Hibernate.INTEGER

13.2 Benutzerdefinierte Mapping Types

- custom value types
  - wenn Werte-Typen (keine vollwertigen Entities) an mehreren Stellen auftreten
  - Alternative zum Default- Mapping (z.B. Integer Wert als VARCHAR)
  - Mapping-Typen aus mehreren Feldern
    - belegen mehrere Spalten in der Datenbank

org.hibernate.UserType

- Email als separate Tabelle

```
package library;
import java.io.Serializable;
public class EMailAddress implements Serializable {
  private static final long serialVersionUID = 42L;
  private String userName;
  private String domainName;
  public EMailAddress(String domainName, String userName) {
    this.domainName = domainName;
    this.userName = userName;
  }
  public String getDomainName() {
    return domainName;
  }
  public void setDomainName(String domainName) {
    this.domainName = domainName;
  }
  public String getUserName() {
    return userName;
  }
  public void setUserName(String userName) {
    this.userName = userName;
  }
}
```

Klasse EmailAddress

- Klasse EmailAddressUserType implementiert UserType Interface
  - ermöglicht Einbindung der Klasse EmailAddress als benutzerdefinierter Mapping-Typ

```
package library;
import java.io.Serializable;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Types;
import org.hibernate.Hibernate;
import org.hibernate.HibernateException;
import org.hibernate.usertype.UserType;
public class EmailAddressUserType implements UserType {
  private static final int[] TYPES = { Types.VARCHAR,
                                       Types.VARCHAR };
  public int[] sqlTypes() {
    return TYPES;
  }
  public Class returnedClass() {
    return EMailAddress.class;
  }
  public boolean equals(Object x, Object y) {
    if (x==y) return true;
    if (x==null || y==null) return false;
    return    ( (EMailAddress) x ).getDomainName().equals(
              ( (EMailAddress) y ).getDomainName() )
          &&  ( (EMailAddress) x ).getUserName().equals(
              ( (EMailAddress) y ).getUserName() );
```

```
    }
    public int hashCode(Object x) throws HibernateException {
       EMailAddress a = (EMailAddress) x;
       return a.getDomainName().hashCode() +
              31 * a.getUserName().hashCode();
    }
    public Object assemble(Serializable cached, Object owner) {
       return deepCopy(cached);
    }
    public Object deepCopy(Object x) {
       if (x==null) return null;
       EMailAddress input = (EMailAddress) x;
       EMailAddress copy = new EMailAddress
          (input.getDomainName(), input.getUserName());
       return copy;
    }
    public Serializable disassemble(Object value) {
       return (Serializable) deepCopy(value);
    }
    public boolean isMutable() {
       return true;
    }
    public Object nullSafeGet(ResultSet rs, String[] names,
       Object owner) throws HibernateException, SQLException {

       String domainName = (String)
       Hibernate.STRING.nullSafeGet(rs, names[0]);

       String userName = (String)
          Hibernate.STRING.nullSafeGet(rs, names[1]);

       return ( domainName==null && userName==null ) ?
          null : new EMailAddress( domainName, userName );
    }
    public void nullSafeSet(PreparedStatement st, Object value,
       int index) throws HibernateException, SQLException {

       String domainName = (value==null) ?
          null : ((EMailAddress)value).getDomainName();

       String userName = (value==null) ?
          null : ((EMailAddress)value).getUserName();

       Hibernate.STRING.nullSafeSet(st, domainName, index);
       Hibernate.STRING.nullSafeSet(st, userName, index+1);
    }
    public Object replace(Object original, Object target,
       Object owner) throws HibernateException {
       return original;
    }
}
```

Die Klasse EmailAddressUserType

- benutzerdefinierter Mapping-Typ implementiert folgende Methoden (Interfaces UserType)

Methode	Beschreibung
`public int[] sqlTypes()`	Liefert die SQL-Datentypen, die von diesem Mapping-Typ verwendet werden. Die möglichen SQL-Datentypen sind in java.sql.Types definiert. Hibernate verwendet diese Information u. a. für die Generierung des Datenbankschemas. Es wird ein Array zurückgegeben, da sich ein Mapping auch auf mehrere Spalten beziehen kann.
`public int[] sqlTypes()`	Liefert die SQL-Datentypen, die von diesem Mapping-Typ verwendet werden. Die möglichen SQL-Datentypen sind in java.sql.Types definiert. Hibernate verwendet diese Information u. a. für die Generierung des Datenbankschemas. Es wird ein Array zurückgegeben, da sich ein Mapping auch auf mehrere Spalten beziehen kann.
`public Class returnedClass()`	Liefert die Klasse, die durch diesen Mapping-Typ gemappt wird. Das Objekt, das von nullSafeGet(...) zurückgeliefert wird, muss eine Instanz dieser Klasse sein.
`public boolean equals(Object x, Object y)`	Vergleicht die beiden Objekte auf Gleichheit der persistenten Attribute.
`public int hashCode(Object x)`	Liefert einen Hashcode, der konsistent mit dem Ergebnis von equals(...) ist.
`public Object assemble(Serializable cached, Object owner)`	Diese Methode wird aufgerufen, um ein Objekt wiederherzustellen, das aus dem Cache geladen wurde. Handelt es sich um ein Objekt, dessen Attribute veränderbar sind, sollte eine Kopie des Objekts zurückgegeben werden. Die Entity, welche das aktuelle Objekt referenziert, wird als Parameter owner übergeben.
`public Serializable disassemble(Object value)`	Wird aufgerufen, bevor ein Objekt in den Cache verschoben wird. Bei Objekten, deren Attribute veränderbar sind, sollte eine Kopie zurückgegeben werden.
`public Object deepCopy(Object x)`	Liefert eine Kopie des übergebenen Objekts. Eventuell referenzierte Entities oder Collection dürfen nicht kopiert werden. Bei unveränderlichen Klassen muss keine Kopie erzeugt werden.
`public boolean isMutable()`	Gibt an, ob die Klasse, auf die sich dieser UserType bezieht, veränderbar ist. Hibernate kann einige Performanceoptimierungen durchführen, für den Fall, dass es sich um unveränderliche Objekte handelt.

```
public Object nullSafeGet(ResultSet rs, String[] names, Object owner)
```

Diese Methode liest die Ergebnisse einer Query aus dem JDBC ResultSet und erzeugt daraus ein Objekt der Klasse, die gemappt werden soll. Die Methode muss mit null-Werten umgehen können. Die Entity, welche das aktuelle Objekt referenziert, wird als Parameter owner übergeben.

```
public void nullSafeSet(PreparedStatement st, Object value, int index)
```

Diese Methode liest die Attribute des Objekts value und trägt diese in das JDBC PreparedStatement ein. Das Einfügen der Attribute muss mit dem Index index beginnen.

- Entity User mit benutzerdefinierter EmailAddress Klasse:

```
@Entity
@Table(name = "T_USER")
public class User {
   private Long id;
   private Address address;
   private String firstname;
   private String lastname;
   private EMailAddress emailAddress;
...
   @Type(type = "library.EmailAddressUserType")
   @Columns(columns = { @Column(name = "domainname"),
                        @Column(name = "username") })
   public EMailAddress getEmailAddress() {
      return emailAddress;
   }
   public void setEmailAddresses(EMailAddress emailAddress) {
      this.emailAddress = emailAddress;
   }
...
```

Änderungen in Klasse User

- String-Attribut emailAddress durch Instanz der Klasse EMailAddress ersetzt
- Annotation @Type definiert die Mapping-Klasse angegeben
- Annotaion @Columns und @Column definieren die Namen der Tabellenspalten
- Mapping Types können nicht in Querries verwendet werden (als Abfragekriterium)
- Alternativen :
  - CompositeUserType Interface erlaubt komplexe Typ Mappings
  - ParameterizedType Interface erlaubt Parameter über Metadaten zu setzen
  - EnhancedUserType Interface erlaubt die Umwandlung von User Types in XML
  - UserCollectionType Interface ermöglicht die Implementierung einer benutzerdefinierten PersistentCollections
  - UserVersionType Interface ermöglicht User Type als Versions-Attribut einzusetzen

14 Fetching und Caching

- Datenbankzugriffe resultieren in Performance-Probleme
  - geeignete Fetching-Strategie wählen
  - EJB QLs oder HQLs Bulk-Update/-Delete verwenden
  - Second Level Cache verwenden

14.1 Fetching-Strategien

JPA Spezifikation definiert zwei Fetching-Strategien:

- Lazy Loading
  - Daten erst laden wenn ein Zugriff auf das Attribut oder das Objekt erfolgt
  - Persistence Provider ist nicht verpflichtet Lazy Loading zu unterstützen
  - Umfang einer Umsetzung von Lazy Loading ungenau definiert
- Eager Loading
  - Daten sofort vollständig laden

Fetching-Strateg bei Attributen

```
@Basic(fetch = FetchType.LAZY)
private String email;
```

Fetching-Strateg bei Beziehungen

```
@OneToMany(fetch = FetchType.EAGER)
private Set<Book> books = new HashSet<Book>();
```

- Hibernate unterstützt Lazy Loading
- Entity muss vom EntityManager bzw. der Session verwaltet werden
- Zugriff auf nicht geladenes Attribut einer detached Entity wirft Exception

Fetching-Strategie

Annotation	Defaultwert
@Basic(fetch = FetchType.EAGER/LAZY)	FetchType.EAGER
@OneToOne(fetch = FetchType.EAGER/LAZY)	FetchType.EAGER
@ManyToOne(fetch = FetchType.EAGER/LAZY)	FetchType.EAGER
@OneToMany(fetch = FetchType.EAGER/LAZY)	FetchType.LAZY

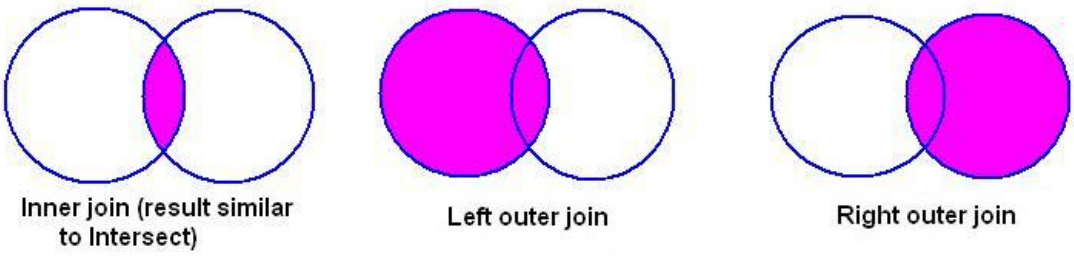
Annotation	Defaultwert
@ManyToMany(fetch = FetchType.EAGER/LAZY)	FetchType.LAZY

Fetching-Strategien

EJB QL Fetch Joins

- Spezifikation definiert für EJB QL-Abfragen Fetch Joins
- Beziehungen zwischen Entities bei Ausführung der Abfrage laden
- als Inner Join implementiert
- Loading Strategie (Eager / Lazy) anwendbar
- Hibernate bietet zusätzliche Optionen:
  - Batch-Fetching
  - Subselect-Fetching

Join Strategien



Fetch Join Syntax

```
from [left [outer]] / [inner] join fetch association
```

- Fetch Join verwendet Fetch Strategien
  - Collections von Entities werden bei Bedarf der Ergebnismenge hinzugefügt
  - effizientes Laden von Entities und deren Abhängigkeiten

14.2 N + 1 Problematik

- N + 1-Problem muss einen zusätzlichen Datensatz laden

Beispiel

- Abfrage ladet Bücher verschiedener Verlage
- Iteration über die Bücher

```
List<Book> books = session.createQuery("from Book where ...");
for(Book book : books) {
   String publisherName = book.getPublisher().getName();
   ...
}
```

- Iteration greift auf den jeweiligen Verlag zu
- bei LAZY Loading entsteht N + 1-Problem
  - zusätlicher Datenbankzugriff

```
select * from BOOK_TABLE where ...                          N + 1
select * from PUBLISHER_TABLE where id = 1                    1
select * from PUBLISHER_TABLE where id = 2                    2
select * from PUBLISHER_TABLE where id = 3                    3
select * from PUBLISHER_TABLE where id = 4                    4
select * from PUBLISHER_TABLE where id = 5                    5
...                                                           N
```

- Hibernates Defaultverhalten
  - als Select-Fetching bezeichnet
- drei weitere Strategien um N + 1-Problematik zu optimieren
  - Batch-Fetching, definiert die Bach Grösse der Datenmenge
  - Join-Fetching,
  - Subselect-Fetching

- Strategien erlauben Definition für eine Beziehung als auch dynamisch in HQL
  - je nach Anwendungsfall kann entsprechende Strategie gewählt und so das Defaultverhalten überschrieben werden

14.2.1 Batch-Fetching

- definiert Batchgrösse der im Vorraus zu landenden assoziierten Objekte

Definition der Abfrage

```
from Book b batch fetch size = 5 b.publisher
```

- resultiert in folgendem SQL

```
select * from BOOK_TABLE where ...
select * from PUBLISHER_TABLE where id in (1, 2, 3, 4, 5)
```

- reduziert N + 1-Problem zu N/Batchgrösse + 1-Problem (Abfragen)

Annotation

```
@ManyToOne
@org.hibernate.annotations.BatchSize(size = 5)
public Publisher getPublisher() {
   return publisher;
}
```

14.2.2 Join-Fetching

- lädt Abhängigkeiten mit einem Join
  - geeignet wenn alle Beziehungsinformationen gebraucht werden
  - geeignet für kleine Ergebnismengen

Definition der Abfrage

```
from Book b join fetch b.publisher
```

- resultiert in folgendem SQL

```
select * from BOOK_TABLE b
left join PUBLISHER_TABLE p on p.id = b.publisher_id
where ...
```

- Joins auf mehrere Collections können grosse Ergebnismengen generieren
  - schlechte Performance

Annotation

```
@org.hibernate.annotations.LazyToOne(
   org.hibernate.annotations.LazyToOneOption.FALSE)
@org.hibernate.annotations.Fetch(
   org.hibernate.annotations.FetchMode.JOIN)
```

14.2.3 Subselect-Fetching

- performante Strategie beim Laden von Collections
- jedoch ein Zugriff mehr

Definition der Abfrage

```
from Book b subselect fetch b.publisher
```

- resultiert in folgendem SQL

```
select * from BOOK_TABLE where ...
select * from PUBLISHER_TABLE where id in
     (select customer_id from BOOK_TABLE where ...)
```

- erlaubt dynamische Abfragen

Annotation

```
@OneToMany(mappedBy = "publisher")
@org.hibernate.annotations.Fetch(
   org.hibernate.annotations.FetchMode.SUBSELECT)
public Set<Book> getBooks() {
   return books;
}
```

14.3 Hibernate Query Cache

- sinnvoll für wiederholende identische Abfragen
- hinterlegt nur die IDs (Primärschlüssel) in der Ergebnisliste
- Second Level Cache
  - bessere Performance
  - stellt Entities im Query Cache bereit

Query Cache durch die Hibernate Konfiguration

```
<property name="hibernate.cache.use_query_cache">true</property>
```

- Abfragen mit setCacheable(true) aufrufen damit diese gecached werden

Query Cache Anwendung

```
List<Book> books = session.createQuery(
   "from Book where autor := autor").setString(
   "author", "Gavin King").setCacheable(true).list();
```

14.4 Second Level Cache

- Caches optimieren Datenbankanwendungen
- Cache hält bereits geladene Daten
- Datenbankzugriffe nur nötig falls gesuchte Daten im Cache nicht vorhanden sind
- Hibernate nutzt First Level Cache
  - in der Hibernate Session enthalten
  - Lebensdauer beschränkt sich auf die der Session
  - Idee : möglichst wenig SQL pro Transaktion
- optional Second Level Cache
  - Entities stehen der gesamten Anwendung zur Verfügung

14.4.1 Strategien und Konfiguration

- vier verschiedene Caching-Strategien stehen zur Verfügung
- auf Entity-Ebene oder für Collections

Methoden	Beschreibung
Read-Only	Das ist die einfachste und zugleich performanteste Strategie. Sie kann eingesetzt werden, wenn ausschliesslich lesend auf die Datenbank zugegriffen wird.
Read-Write	Bei der Strategie read-write dürfen Daten auch geändert werden. Bitte beachten Sie, dass diese Strategie nicht mit dem Isolation Level Serializable verwendbar ist und dass die Session am Transaktionsende mit session.close() oder session.disconnect() beendet werden muss.
Nonstrict Read-Write	Mit nonstrict read-write kann nicht garantiert werden, dass zwei Transaktionen nicht gleichzeitig denselben Datensatz ändern. Daher ist die Strategie geeignet, wenn Daten nur selten geändert werden und es sehr unwahrscheinlich ist, dass zwei Transaktionen dieselben Daten ändern. Wie bei read-write muss die Session am Transaktionsende geschlossen werden.
Transactional	Kann nur in einer JTA-Umgebung verwendet werden und entspricht dem Isolation Level Repeatable Read.

Annotation @Cache

```
@Entity
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
@Table(name = "T_USER")
public class User {
```

hibernate.cfg.xml

- Klasse des Cache Providers angeben

```
<property name="hibernate.cache.provider_class">
  org.hibernate.cache.OSCacheProvider
</property>
```

14.4.2 Second Level Cache Provider

- Cache Provider und Caching-Strategien

Cache Provider	Read-Only	Read-Write	Nonstrict Read-Write	Transactional
EHCache http://ehcache.sourceforge.net/	✓	✓	✓	✗
OSCache http://www.opensymphony.com/oscache/	✓	✓	✓	✗
SwarmCache http://swarmcache.sourceforge.net/	✓	✓	✗	✗
JBoss Cache http://www.jboss.org/products/jbosscache	✓	✗	✗	✓

Von Cache Providern unterstützte Strategien

14.5 Massen-Update und -Delete

- mehrere Einträge einer Tabelle mit einem Datenbankzugriff zu bearbeiten
- ändert auch den Objektzustand

Massen-Update:

```
update Book b set b.author = 'newname'
where b.author = 'oldname'
```

15 Anhang

15.1 Hibernate Mapping Types

In Hibernate sind bereits Mapping Types für eine Vielzahl von Javatypen (primitive wie auch Klassen) enthalten. Die Hibernate Mapping-Typen werden stets klein geschrieben.

Hibernate-Mapping-Typ	Java-Typ	Standard-SQL-Datentyp
string	java.lang.String	VARCHAR
short	short oder java.lang.Short	SMALLINT
integer	int oder java.lang.Integer	INTEGER
long	long oder java.lang.Long	BIGINT
float	float oder java.lang.Float	FLOAT
double	double oder java.lang.Double	DOUBLE
big_decimal	java.math.BigDecimal	NUMERIC
big_integer	java.math.BigInteger	NUMERIC
character	java.lang.String	CHAR(1)
byte	byte oder java.lang.Byte	TINYINT
boolean	boolean oder java.lang.Boolean	BIT
yes_no	boolean oder java.lang.Boolean	CHAR(1) ('Y' oder 'N')
true_false	boolean oder java.lang.Boolean	CHAR(1) ('T' oder 'F')
date	java.util.Date oder java.sql.Date	DATE
time	java.util.Date oder java.sql.Time	TIME
timestamp	java.util.Date oder java.sql.Timestamp	TIMESTAMP
calendar	java.util.Calendar	TIMESTAMP
calendar_date	java.util.Calendar	DATE
binary	byte[]	VARBINARY oder BLOB
text	java.lang.String	CLOB
serializable	java.io.Serializable (beliebige Klasse welche das Interface implementiert)	VARBINARY oder BLOB
clob	java.sql.Clob	CLOB
blob	java.sql.Blob	BLOB
class	java.lang.Class	VARCHAR

Hibernate-Mapping-Typ	Java-Typ	Standard-SQL-Datentyp
locale	java.util.Locale	VARCHAR
timezone	java.util.TimeZone	VARCHAR
currency	java.util.Currency	VARCHAR

Die Standard Hibernate Mapping-Typen

15.2 Hibernate SQL Dialects (`hibernate.dialect`)

RDBMS	Dialect
DB2	`org.hibernate.dialect.DB2Dialect`
DB2 AS/400	`org.hibernate.dialect.DB2400Dialect`
DB2 OS390	`org.hibernate.dialect.DB2390Dialect`
PostgreSQL	`org.hibernate.dialect.PostgreSQLDialect`
MySQL	`org.hibernate.dialect.MySQLDialect`
MySQL with InnoDB	`org.hibernate.dialect.MySQLInnoDBDialect`
MySQL with MyISAM	`org.hibernate.dialect.MySQLMyISAMDialect`
Oracle (any version)	`org.hibernate.dialect.OracleDialect`
Oracle 9i/10g	`org.hibernate.dialect.Oracle9Dialect`
Sybase	`org.hibernate.dialect.SybaseDialect`
Sybase Anywhere	`org.hibernate.dialect.SybaseAnywhereDialect`
Microsoft SQL Server	`org.hibernate.dialect.SQLServerDialect`
SAP DB	`org.hibernate.dialect.SAPDBDialect`
Informix	`org.hibernate.dialect.InformixDialect`
HypersonicSQL	`org.hibernate.dialect.HSQLDialect`
Ingres	`org.hibernate.dialect.IngresDialect`
Progress	`org.hibernate.dialect.ProgressDialect`
Mckoi SQL	`org.hibernate.dialect.MckoiDialect`
Interbase	`org.hibernate.dialect.InterbaseDialect`
Pointbase	`org.hibernate.dialect.PointbaseDialect`

RDBMS	Dialect
FrontBase	`org.hibernate.dialect.FrontbaseDialect`
Firebird	`org.hibernate.dialect.FirebirdDialect`

15.3 Package org.hibernate

Interface Summary

Criteria	Criteria is a simplified API for retrieving entities by composing Criterion objects.
Filter	Type definition of Filter.
Interceptor	Allows user code to inspect and/or change property values.
Query	An object-oriented representation of a Hibernate query.
ScrollableResults	A result iterator that allows moving around within the results by arbitrary increments.
Session	The main runtime interface between a Java application and Hibernate.
SessionFactory	Creates Sessions.
SQLQuery	Allows the user to declare the types and select list injection points of all entities returned by the query.
StatelessSession	A command-oriented API for performing bulk operations against a database. A stateless session does not implement a first-level cache nor interact with any second-level cache, nor does it implement transactional write-behind or automatic dirty checking, nor do operations cascade to associated instances.
Transaction	Allows the application to define units of work, while maintaining abstraction from the underlying transaction implementation.

Class Summary

CacheMode	Controls how the session interacts with the second-level cache and query cache.
ConnectionReleaseMode	Defines the various policies by which Hibernate might release its underlying JDBC connection.
EmptyInterceptor	An interceptor that does nothing.
EntityMode	Defines the representation modes available for entities.
FetchMode	Represents an association fetching strategy.
FlushMode	Represents a flushing strategy.

Class Summary

Hibernate	Provides access to the full range of Hibernate built-in types.
LockMode	Instances represent a lock mode for a row of a relational database table.
ReplicationMode	Represents a replication strategy.
ScrollMode	Specifies the type of JDBC scrollable result set to use underneath a `ScrollableResults`

Exception Summary

AssertionFailure	Indicates failure of an assertion: a possible bug in Hibernate.
CallbackException	Should be thrown by persistent objects from `Lifecycle` or `Interceptor` callbacks.
DuplicateMappingException	Raised whenever a duplicate for a certain type occurs.
HibernateException	Any exception that occurs inside the persistence layer or JDBC driver.
InstantiationException	Thrown if Hibernate can't instantiate an entity or component class at runtime.
InvalidMappingException	Thrown when a mapping is found to be invalid.
JDBCException	Wraps an `SQLException`.
LazyInitializationException	Indicates access to unfetched data outside of a session context.
MappingException	An exception that usually occurs at configuration time, rather than runtime, as a result of something screwy in the O-R mappings.
MappingNotFoundException	Thrown when a resource for a mapping could not be found.
NonUniqueObjectException	This exception is thrown when an operation would break session-scoped identity.
NonUniqueResultException	Thrown when the application calls `Query.uniqueResult()` and the query returned more than one result.
ObjectDeletedException	Thrown when the user tries to do something illegal with a deleted object.
ObjectNotFoundException	Thrown when `Session.load()` fails to select a row with the given primary key (identifier value).

PersistentObjectException	Thrown when the user passes a persistent instance to a `Session` method that expects a transient instance.
PropertyAccessException	A problem occurred accessing a property of an instance of a persistent class by reflection, or via CGLIB.
PropertyNotFoundException	Indicates that an expected getter or setter method could not be found on a class.
PropertyValueException	Thrown when the (illegal) value of a property can not be persisted.
QueryException	A problem occurred translating a Hibernate query to SQL due to invalid query syntax, etc.
QueryParameterException	Parameter invalid or not found in the query
SessionException	Thrown when the user calls a method of a Session that is in an inappropropriate state for the given call (for example, the the session is closed or disconnected).
StaleObjectStateException	A `StaleStateException` that carries information about a particular entity instance that was the source of the failure.
StaleStateException	Thrown when a version number or timestamp check failed, indicating that the `Session` contained stale data (when using long transactions with versioning).
TransactionException	Indicates that a transaction could not be begun, committed or rolled back.
TransientObjectException	Thrown when the user passes a transient instance to a `Session` method that expects a persistent instance.
TypeMismatchException	Used when a user provided type does not match the expected one
UnresolvableObjectException	Thrown when Hibernate could not resolve an object by id, especially when loading an association.
WrongClassException	Thrown when `Session.load()` selects a row with the given primary key (identifier value) but the row's discriminator value specifies a subclass that is not assignable to the class requested by the user.

http://www.hibernate.org/hib_docs/v3/api/org/hibernate/package-summary.html

15.4 Literaturverzeichnis

- Bauer, King: Java Persistence with Hibernate. Manning
- Härder, Theo ; Rahm, Erhard: Datenbanksysteme – Konzepte und Techniken der Implementierung. Springer
- Elliott, James: Hibernate: A Developer's Notebook. O'Reilly
- Feddema, Helen: DAO Object Models. O'Reilly
- Beeger, Haase, Roock, Santiz: Hibernate : Persistent in Java-Systemen mit Hibernate: Dpunkt
- Kehle, Robert: Hibernate und das Java Persistence API. Sotware und Support
- Minter, Linwood: Beginning Hibernate: From Novice to Professional. Apress

15.5 Onlineverzeichnis

- Hibernate Reference Documentation
  http://www.hibernate.org
- Enterprise JavaBeans Spezifikation - Java Persistence API, Version 3.0
  http://www.jcp.org/en/jsr/detail?id=220
- Fussel, Mark L.: Foundations of Object-Relational Mapping
  http://www.chimu.com/publications/objectRelational