Hochschule **Rosenheim**
University of Applied Sciences

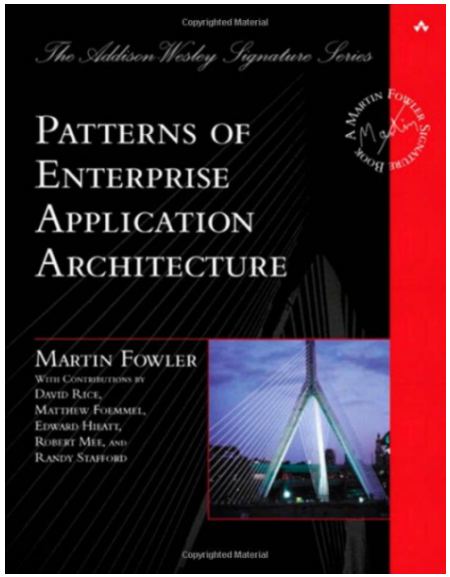
Kapitel 9 – Anwendungsprogrammierung und JPA

Vorlesung Datenbanken

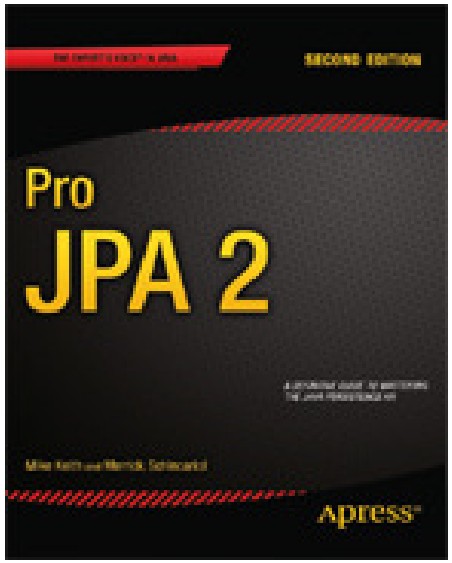
Prof. Dr. Kai Höfig

Weiterführende Literatur

- Martin Fowler: Patterns of Enterprise Application Architecture (als ebook im OPAC)



- Mike Keith; Merrick Schincariol: Pro JPA 2, Second Edition (als ebook im OPAC)



- Michael Inden Persistenzlösungen und REST-Services

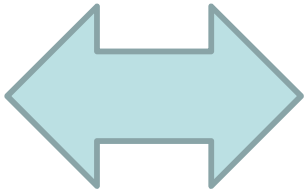
Client-Server Modell und wie wir bisher gearbeitet haben

- Prinzip: Client nimmt Dienste eines Servers in Anspruch

- Das MSSQLDBMS war bisher unser Client.
- In diesem Kapitel schauen wir uns an, wie Applikationen auf eine Datenbank zugreifen und lernen dies am Beispiel Java JPA kennen.

Ziel heute: Persistierung von Objekten

```java
public class Auto {
    private String hersteller;
    private String name;
    private int preis;

    public Auto(String hersteller, String name){
        this.hersteller=hersteller;
        this.name=name;
        this.preis=0;
    }

    public void setPreis(int preis){
        this.preis=preis;
    }
}
```



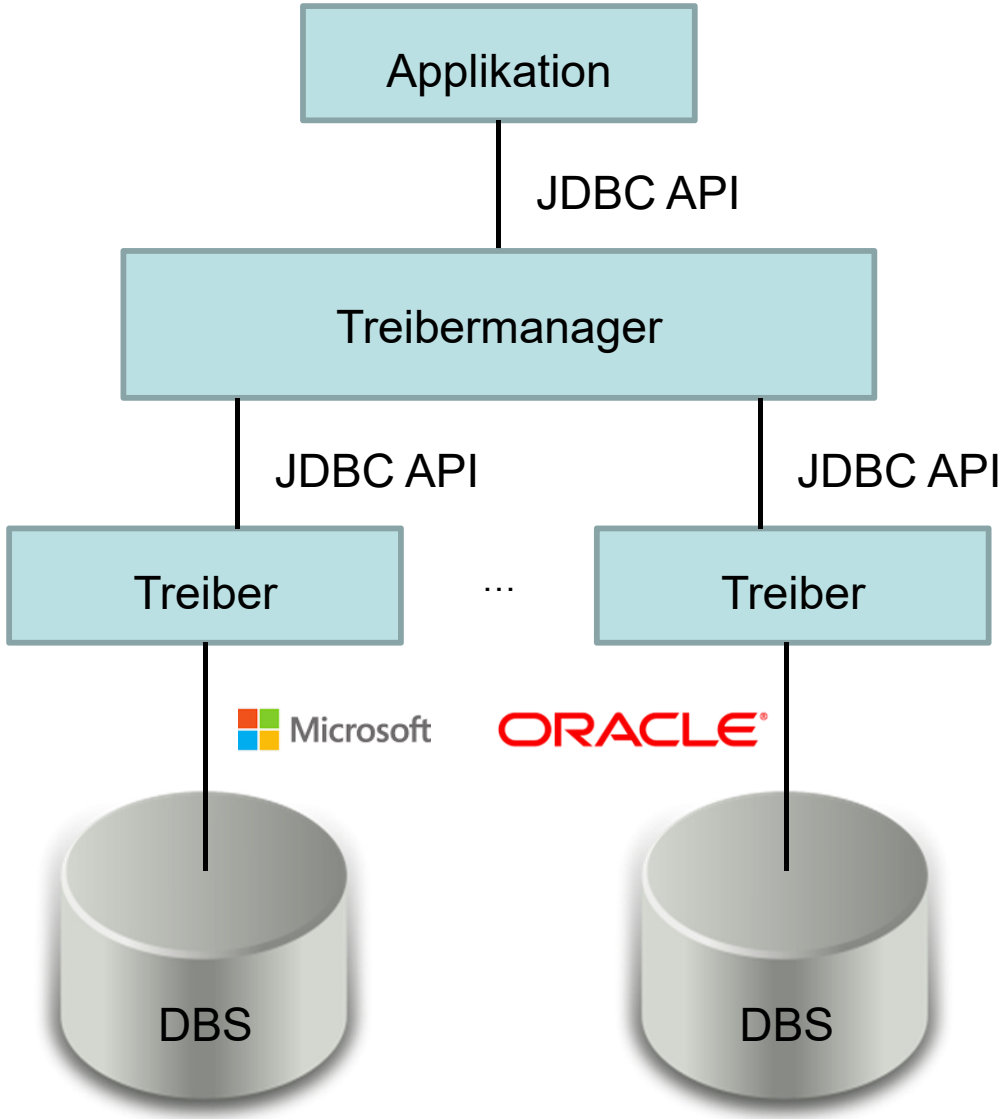
Ergebnisse | Meldungen

	name	hersteller	preis
1	Polo	VW	29000
2	1er	BMW	36500

- Wir wollen einmal erzeugte Objekte nun speichern (**persistieren**) und später wieder aus der Datenbank laden.
- Dazu müssen sie **serialisiert** (speicherbar gemacht) werden und dann deserialisiert (geladen) werden.
- Objekte und Tupel sind unterschiedliche Modelle. Man spricht daher bei der Persistierung auch vom **impedance mismatch**

JDBC und Persistenz im Überblick

- JDBC
  - Datenbankzugriffsschnittstelle für Java
  - Abstrakte, datenbankneutrale Schnittstelle: Zugriff auf verschiedene Datenbanksysteme über systemspezifische Treiber möglich
  - Low-Level-API: direkte Nutzung von SQL
- **Persistierung** mit JDBC Bordmitteln ist eine manuell zu implementierende Aufgabe, langatmig und fehleranfällig.
- JDBC ist ein weit verbreiteter Standard und ermöglicht es relativ einfach eine Datenbank abzufragen. Für Persistierung gibt es besseres.



Java.sql

- Java-Package `java.sql`
  - `DriverManager`: Einstiegspunkt, Laden von Treibern

```
Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
```

  - `Connection`: Datenbankverbindung

```
Connection conn = DriverManager.getConnection(connectionString,username,password);
```

  - `Statement`: Ausführung von Anweisungen über eine Verbindung

```
conn.createStatement().execute("INSERT INTO Person (name) VALUES ('Maria')");
```

  - `ResultSet`: verwaltet Ergebnisse einer Anfrage, Zugriff auf einzelne Spalten

```
ResultSet result = conn.createStatement().executeQuery("SELECT name FROM Person");
```

`Java.sql.ResultSet` im Überblick

- `boolean next()`
  Moves the cursor forward one row from its current position. Can be used to iterate through result set using a while loop.

- `boolean first()`
  Moves the cursor to the first row in this ResultSet object.

- `double getString(int columnIndex)`
  Retrieves the value of the designated column in the current row of this ResultSet object as a String in the Java programming language. (**many more datatypes available**)

- `void close()`
  Releases this ResultSet object's database and JDBC resources immediately instead of waiting for this to happen when it is automatically closed.

```
ResultSet allPersons = conn.createStatement().executeQuery("SELECT name FROM person");
while(allPersons.next()){
    System.out.println(allPersons.getString("name"));
}
allPersons.close();
```

https://docs.oracle.com/javase/7/docs/api/java/sql/ResultSet.html

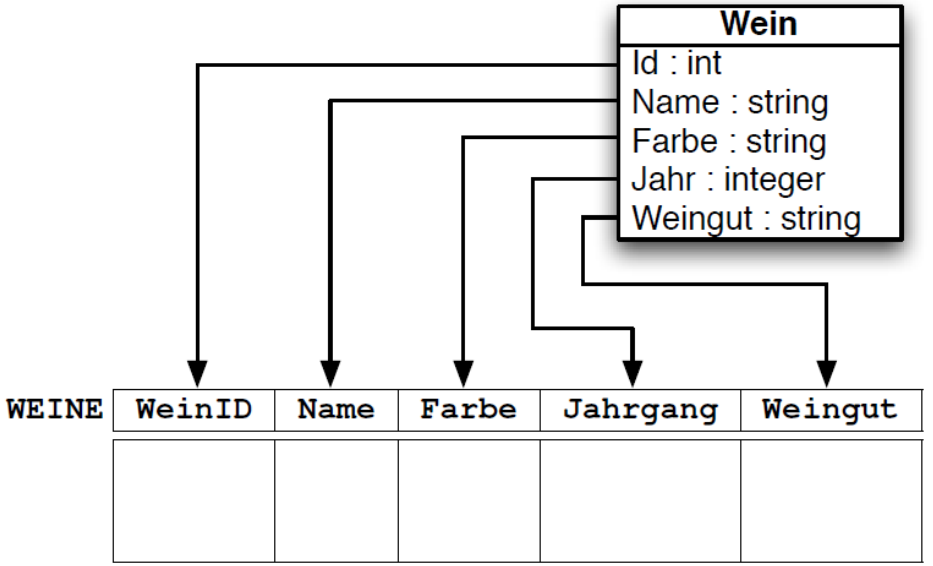
Embedded SQL

- Alternative Schreibweise zu den Klassen in `Java.sql` ist embedded sql.
- Alle Anweisungen werden dann in reguläre JDBC Transaktionen übersetzt.
- Beispiel:

```
String name;
String anbaugebiet = "Toskana";
String region = "Italien";
#sql { SELECT weingut INTO :name FROM erzeuger WHERE anbaugebiet = :anbaugebiet AND region = :region };
#sql iter = { SELECT name, farbe, jahrgang FROM weine };
while (iter.next ()) {
    System.out.println(iter.name()+":"+iter.farbe()+" "+iter.jahr());
}
```

Die Java Persistence API (JPA)

- JPA bietet gegenüber der reinen Verwendung von JDBC eine bessere Abstraktion.
- Gute Integration in das Java Collection Framework
- Hoher Grad der Automatisierung zur Persistenz. Serialisierung und Deserialisierung über Tags.
- 3 Kernelemente
  1. Annotation der zu serialisierenden Klassen.
  2. (minimale) Konfiguration in der Datei `persistence.xml`
  3. Persistierung durch Verwendung von `EntityManager`



Beispiel annotierte Klasse

- `@Entity` markiert eine Klasse als zur Persistierung vorgesehen. Instanzen einer als Entity annotierten Klasse werden zu Tupeln in der Datenbank.
- `@Table` legt eine bestimmte Tabelle zur Persistierung fest (optional)
- `@Id` legt das oder die Schlüsselattribute fest.
- `@GeneratedValue` makriert das Attribut als generierten Schlüssel

```java
@Entity
@Table(name = "Person")
public class Person implements Serializable
{
    @Id
    @GeneratedValue
    //@GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "Vorname")
    private String firstName;

    @Column(name = "Name")
    private String lastName;
```

- `@Column` legt ein Attribut als Spalte fest. (Name optional)
- Vollständige Liste unter https://www.objectdb.com/api/java/jpa/annotations

Annotationen für Relationships

- `@ManyToMany`
  Für die Darstellung dieser Relationship wird automatisch eine neue Tabelle angelegt. Zusätzliche Attribute lassen sich darüber nur speichern, wenn die Realtionship manuell aufgeteilt wird.

- `@ManyToOne`
  Die Klasse in der die Annoation steht, referenziert ein Element einer anderen Entität

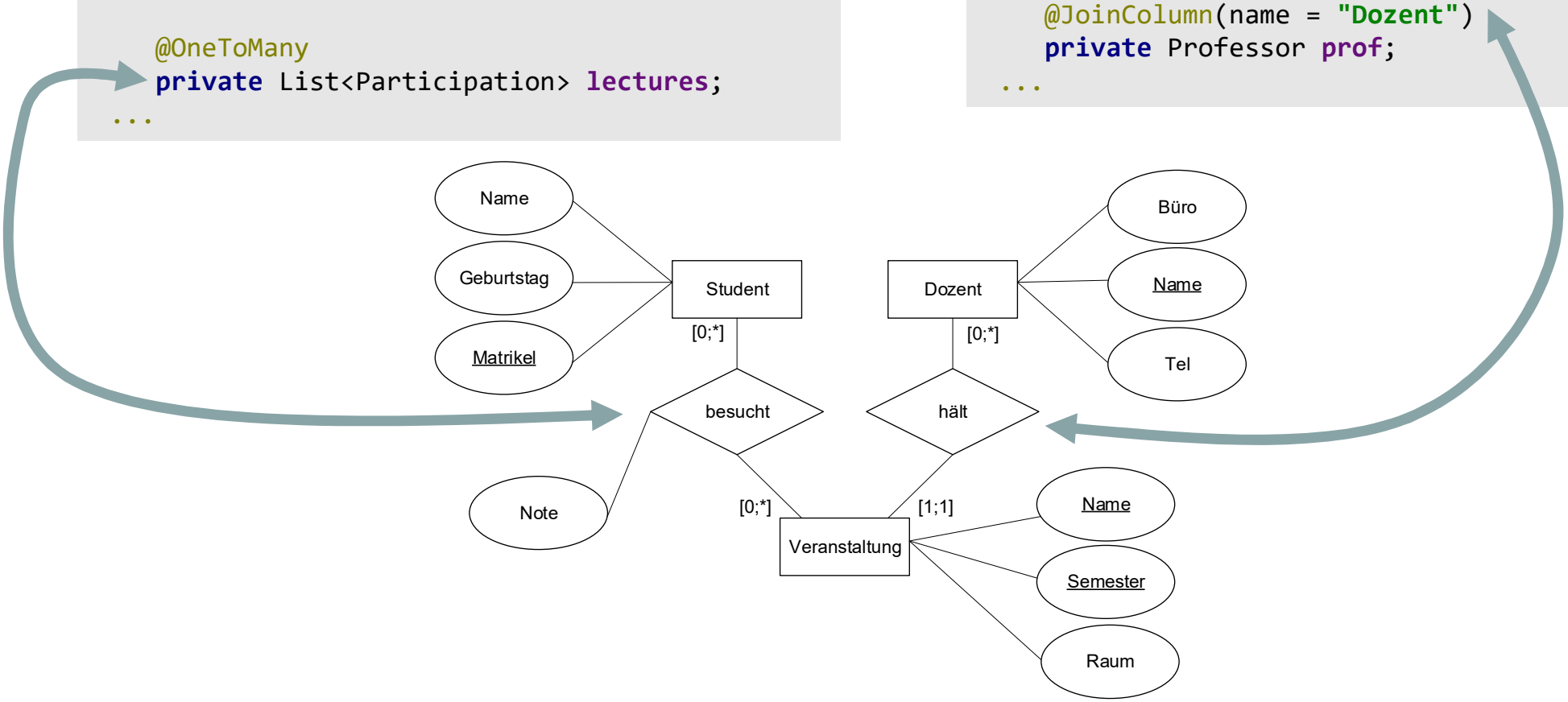
- `@OneToMany`
  Die Klasse in der die Annotation steht, referenziert mehrere Elemente einer anderen Entität. Meißtens ist das ein Listentyp des Java Collection Frameworks.

- `@OneToOne`
  Die Klasse in der die Annotation steht, referenziert ein Element einer anderen Entität. Das Attribut wird zum Schlüssel.

Annotation von Relationships Beispiel

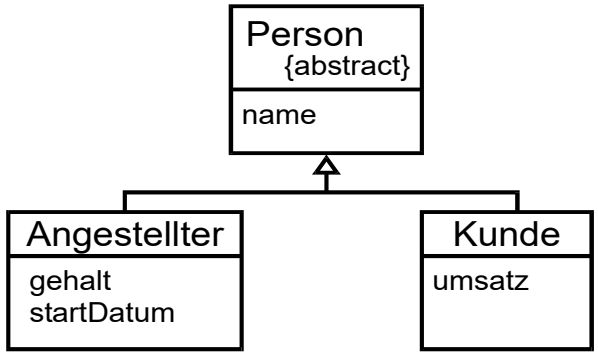
```java
@Entity
@Table(name = "Student")
public class Student  implements Serializable {

    ...

    @OneToMany
    private List<Participation> lectures;
 ...
```

```java
@Entity
@Table(name = "Veranstaltung")
public class Lecture implements Serializable {

    @ManyToOne
    @JoinColumn(name = "Dozent")
    private Professor prof;
 ...
```



Durch die Annotation der Schlüssel mit `@Id` der referenzierten Klassen, ist die Angabe von `@JoinColumn` meist überflüssig, es sei den dass (sowie in diesem Beispiel) der Name der Attribute manuell festgelegt wurde.

Abbildung von Vererbung



default

Null-Value Style (`single_table`)

Person					
OID	name	gehalt	startDatum	umsatz	objTyp

`table_per_class`

Kunde		
OID	name	umsatz

Angestellter			
OID	name	gehalt	startDatum

ER Style (`joined`)

Person		
OID	name	objTyp

Kunde	
OID (FS)	umsatz

Angestellter		
OID (FS)	gehalt	startDatum

```java
@Entity
@Inheritance(strategy= InheritanceType.JOINED)
public class Employee extends Person implements Serializable {
    @Column
    private String company;

    public Employee(String firstName, String lastName, String company) {
        super(firstName, lastName);
        this.company = company;
    }
}
```

Konfiguration mittels `persistence.xml`

- In der Datei persistence.xml wird eine minimale Konfiguration für das JPA Framework vorgenommen.
  1. Verbindungseinstellungen zur Datenbank (URL, Benutzername, Passwort, Treiber).
  2. Welche Klassen persistiert werden sollen (alle oder bestimmte).
  3. Wie mit Attribut- und Klassenänderungen umgegangen werden soll.

     `none`
     No schema creation or deletion will take place.

     `create`
     The provider will create the database artifacts on application deployment. The artifacts will remain unchanged after application redeployment.

     `drop-and-create`
     Any artifacts in the database will be deleted, and the provider will create the database artifacts on deployment.

     `drop`
     Any artifacts in the database will be deleted on application deployment.

Zugriff auf die Datenbank über EntityManager

- EntityManager
  - Ist der Kontext, der Entitäten/Objekte verwaltet
  - Stellt Verbindung zur Datenbank bereit
  - Liefert Umgebung für Transaktionen

- Arbeiten mit dem EntityManager

```
EntityManagerFactory emf =
   Persistence.createEntityManagerFactory("WeinVerwaltung");
   // vgl. persistence.xml, siehe später

EntityManager em = emf.createEntityManager();

// … Mach was spannendes

em.close();
emf.close();
```

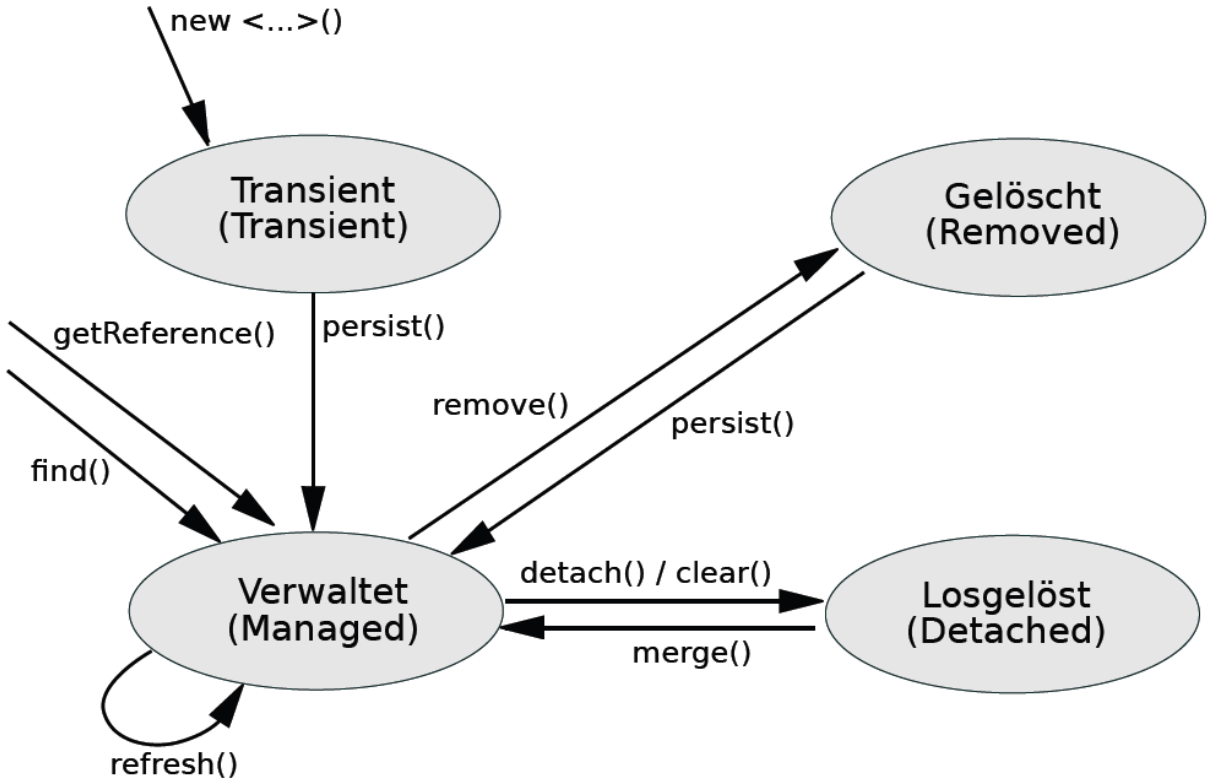
Zugriff auf die Datenbank über EntityManager

- Daten anlegen / ändern / löschen

```
EntityManager em = …;
em.getTransaction().begin();

Erzeuger mueller = em.find( Erzeuger.class, "Müller");

Wein w = new Wein (4714, " Dom Perinjong", WeinFarbe.ROSE, 2013, mueller);
em.persist(w);

Wein w2 = em. find ( Wein.class, 4713); // früher gespeichert
System.out.println("Gefunden:" + w2);

em.remove(w2);

em.getTransaction().commit ();
```

- Sobald Entity von EntityManager verwaltet wird, werden alle Änderungen automatisch gespeichert
- Anmelden (persist) und Entfernen (remove) bzw. Abmelden (detach) explizit erforderlich

Lebenszyklus einer Entität



- = Zustandsmodell der Entitäten in Bezug auf EntityManager

Quelle Grafik: Java Persistencs API 2.0, Hanser

Queries in JPA

JPA bietet sehr viele Optionen für Queries

- „reines, low-level“ SQL
- JPA QL - „objektorientiertes, Java-standardisiertes SQL“
  - Beispiel:

```
TypedQuery<Wein> q
  = em.createQuery("select w from Wein w where w.jahrgang > :aJahrgang",
Wein.class);
q.setParameter("aJahrgang", 2000);

List<Wein> neuerAls2000 = q.getResultList();

for(Wein fromDB: neuerAls2000) {
   System.out.println("Neuer als 2000:" + fromDB);
}
```

- Query by Criteria (ähnlich Query by Example)
  - Programmatischer Aufbau des Queries