Kap. 4 Objekt-Relationales Mapping

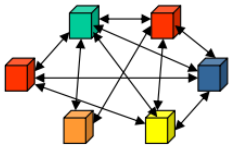
a) Schichtenarchitektur

Leitgedanke: starke Kohärenz

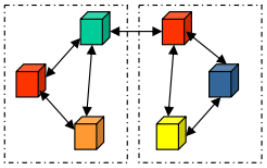
- Teilsystem ist für zusammenhängende, klar umrissene Aufgabe verantwortlich
- "Packe zusammen, was zusammen gehört"
- Einfachheit, Verständlichkeit, Redundanzfreiheit, bessere Teambildung

Leitgedanke: lose Kopplung

- schmale Schnittstellen zwischen Teilsystemen (kleine Menge von Methoden (API))
- Teilsysteme kennen nur wenige andere Teilsysteme



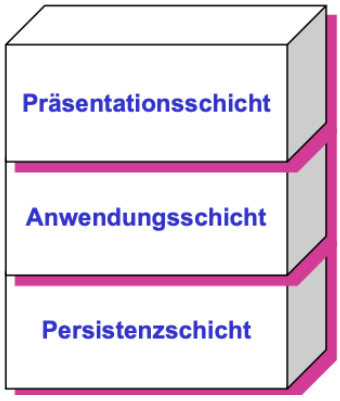
stark gekoppelte Teilsysteme



lose gekoppelte Teilsysteme

Drei-Schichten-Architektur

logische Strukturierung eines Software-Systems



Präsentationsschicht
- Präsentation fachlicher Daten
- Dialogkontrolle
  - weiß wenig von Applikationsschicht
  - implementiert keine fachlichen Abläufe

Anwendungsschicht
- Entitätsklassen für fachliche Daten
- Geschäftsprozess-Klassen für fachliche Abläufe
  - kennt keine Fenster
  - kennt keine DB-Tabellen

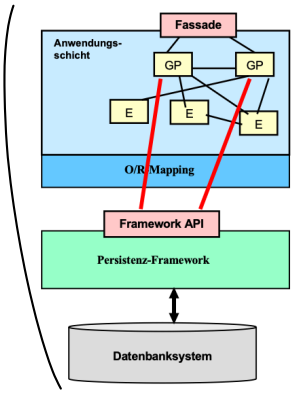
Persistenzschicht
- verwaltet fachliche Objekte in DB
  - kennt das Datenbank-Schema
  - enthält die SQL-Befehle bei RDBMS

Schnittstelle Anwendungs-/Persistenzschicht

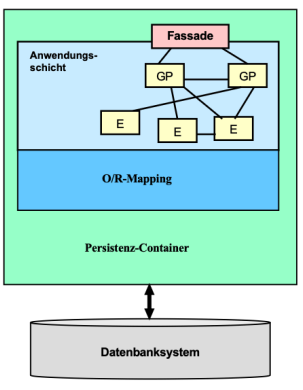
Anwendungsschicht (Applikationsschicht)
- weiß, welche Daten zusammengehören und in welcher Situation Zugriff auf DB notwendig ist
- soll keine technischen Details über DB-Anbindung enthalten

Persistenzschicht (Zugriffsschicht)
- Verwaltung fachlicher Objekte in DBMS
- O/R-Mapping (object to relational mapping)
  Abbildung von Objekten (bzw. Objektattributen) auf DB-Tabellen (bzw. Tabellenspalten)
- Persistenzdienste (Kernfunktionen)
  - Persistente Ablage von Laufzeitobjekten in DB - Erzeugung von Laufzeitobjekten aus DB
  - Suchfunktionalität
  - Transaktionen

Realisierungsalternativen von Persistenzkonzepten in Applikationsschicht



(a) Framework mit Persistenzdiensten



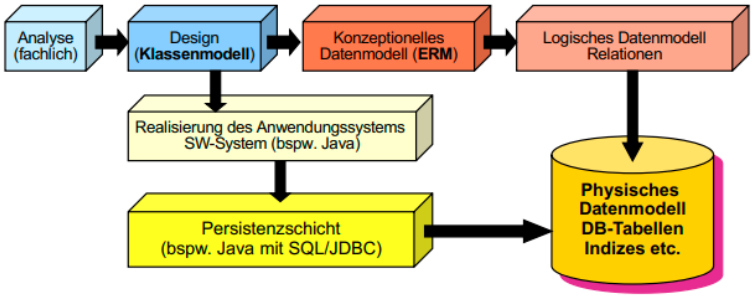
(b) Persistenz-Container, z.B. EJB

b) Konzept: Transformation eines Klassenmodells in Datenmodell

Transformation: Klassenmodell => Datenmodell

- Das Datenmodell ist immer eine Teilmenge des Klassenmodells, weil das Klassenmodell Methode besitzt, die nicht übernommen werden und Attribute, die nicht datentragend sind

Datenbank-Einbindung im objektorientierten Entwicklungsprozess



Prozess der Datenmodell-Erstellung

- Inkrementell: schrittweise erfolgend, sodass Änderungen vorherige Stufen Auswirkungen auf die nächsten Schritte hat

Klassenmodell ist Basis des Datenmodells

- hat datentragende Aspekte (Attribute)
- hat funktionale/ dynamische Aspekte (Methoden)

Inkrementelle Entwicklung auch bei Datenmodellierung

- in jeder inkrementellen Stufe wird Datenmodell der vorhergehenden Stufe erweitert

Statische Muster

- Transformationsschritte zur Überführung eines Klassenmodells in Datenmodell
- Attribute und Beziehungen finden sich im Datenmodell wieder

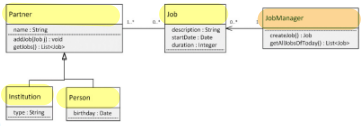
Transformation eines Klassenmodells in Datenmodell

- Schritt 1: Auswahl der persistenten Klassen (mit Attributen)
- Schritt 2: Hinzufügen eines Oid-Attributs (Schlüssel)
- Schritt 3: Abbildung einer Klasse auf eine Entität
- Schritt 4: Abbildung von Beziehungen (Bzh.)
- Schritt 5: Abbildung der Vererbungen
- Schritt 6: Überführung in physikalisches Modell

Transformationsschritte:

1.) Auswahl der persistenten Klassen

Beispiel Klassenmodell



Schritt 1: Auswahl der persistenten Klassen

Persistente Klassen (Entitätsklassen)

- enthalten (datentragende) Attribute, die dauerhaft abgespeichert werden

Transiente Klassen (process, session, control classes)

- GUI-Klassen, technische oder funktionale Klassen

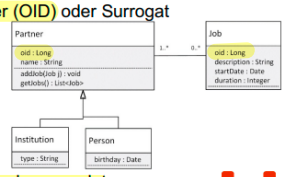
Beispiel

- alle Klassen außer JobManager sind datentragend

2.) Hinzufügen eines Oid-Attributs

Schritt 2: Hinzufügen eines Oid-Attributs

jede persistente Klasse muss ein Attribut Objekt-Identifier (OID) oder Surrogat (ggf. auch geerbt) besitzen



- zur Identifikation von Objekten (eindeutige Identität)
- zur Erzeugung von Objekten aus Datenbank
- Eigenschaften von Oids (künstliche Schlüssel)
  - **eindeutig**
  - **unveränderbar**, da i.d.R. als Primär-/Fremdschlüssel verwendet
- Damit: fachliche Attribute meist nicht als Oid geeignet
- Klassisch OID oft String (oder Long), ggf. auch UUID verwenden

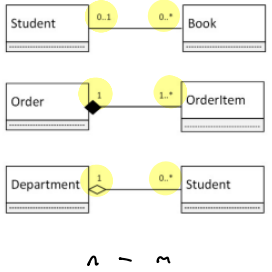
3.) Abbildung einer Klasse auf eine Entität

Schritt 3: Abbildung einer Klasse auf eine Entität

- jede persistente Klasse wird auf eigene Entität abgebildet
- jedes persistente Attribut wird auf Attribut der Entität abgebildet
- Oid-Attribut wird Schlüsselattribut
- mengenwertige Attribute (Arrays, Vektoren) werden auf eine eigene Entität abgebildet

4.) Abbildung von Beziehungen

Schritt 4: Abbildung von Beziehungen



- Assoziationen, Kompositionen, Aggregationen werden auf Beziehungen im logischen Datenmodell abgebildet
- Bei mindestens einer Maximalkardinalität von 1 ohne zusätzliche Entität möglich
- n:m-Beziehungen könnten durch eigene Beziehungsentität umgesetzt werden

5.) Überführung in relationales Modell

Schritt 5: Überführung in relationales Modell
(vgl. Transformationsprozess aus DBS 1!)

- jede Entität wird auf eigene Tabelle mit Schlüsselattribut als Primärschlüssel abgebildet
- jedes Attribut einer Entität wird auf Tabellenspalte abgebildet
  (mit "passendem" Datentyp:
  Zuordnung Java-Datentyp --> Datenbank-Datentyp)
- 1:N-Beziehungen (Master-Detail) werden auf Fremdschlüssel in Detail-Tabelle abgebildet
- M:N-Beziehungen (Many-Many) werden auf Beziehungstabelle abgebildet, deren Spalten die Primärschlüssel der beteiligten Tabellen enthalten (sofern nicht schon beim Übergang Klassenmodell → Entitätsmodell aufgelöst)

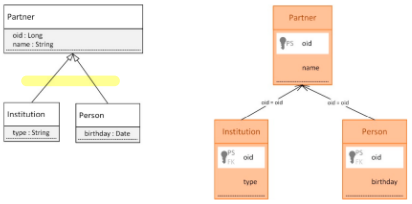
Abbildung von Vererbungshierarchien:

Vertikale Partitionierung

Institution & Person vererben oid von Partner

Option 1 (**Vertikale Partitionierung**)

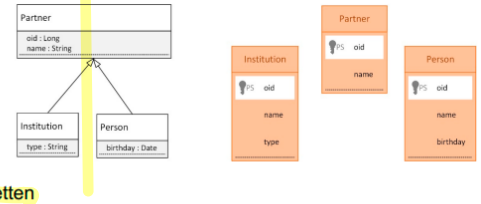
- Super- und Sub-Entitäten bilden jeweils eigene Tabellen
- Super-Entität wie sonst
- Tabellen für Sub-Typen enthalten Schlüsselattribute der Super-Entität (als eigene Schlüssel und Fremdschlüssel) und subtyp-spezifische Attribute



Horizontale Partitionierung

Option 2 (**Horizontale Partitionierung**)

- Super- und Subtypen erhalten Tabellen
- Alle Tabellen besitzen auch alle Attribute der Super-Typen
- Inhalt einer Tabelle bildet kompletten Zustand einer Entität ab
- Aber: globale Eindeutigkeit des PK schwierig



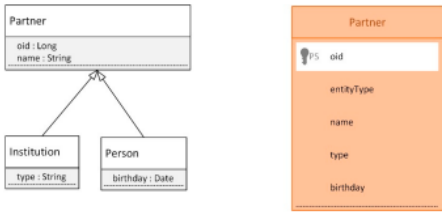
Partner:
- OID nicht eindeutig
- kein Zusammenhang

H

Universelle Relation

Option 3 (**Universelle Relation**):

- Eine Relation für die gesamte Vererbungshierarchie
- Alle Attribute aller Entitäten werden Attribute dieser Relation
- Zusätzlich eine Typ-Attribut zur Festlegung, welcher Entitätstyp diese Zeile ist
- Einfache Definition des PK



Vor / Nachteile der Abbildungsvarianten

Vertikale Partitionierung (eine Tabelle pro Klasse mit Bezug)

+ Struktur passt zum Klassenmodell
+ Konsistenzerhaltung einfach
- Um alle Attribute eines Objektes zu selektieren, werden (potenziell) viele Joins benötigt

Horizontale Partitionierung (eine Tabelle pro Klasse ohne Bezug)

+ Beim Zugriff auf Unterklasse wird nur eine Tabelle benötigt, d.h. gute Performance
- Abbildung von Beziehungen schwieriger (keine FK in der Datenbank möglich, da diverse Zielrelationen)
- Union nötig, um alle Objekte zu selektieren
- Update eines Attributes der Basisklasse für alle Instanzen muss über mehrere Tabellen laufen
- Globale Eindeutigkeit des PK schwierig sicherzustellen

Universelle Relation (eine Gesamttabelle)

+ Einfacher Zugriff auf alle Objekte / Attribute einer Hierarchie, dadurch gute Performance
+ Überlappende Vererbung redundanzfrei möglich (aber meist in Programmiersprache gar nicht möglich)
- Potentiell hohe Anzahl ungenutzter Attribute/NULL-Werte
- Not-Null Constraints in Unterklassen nicht möglich

- Wie selektieren Sie die Namen aller Partner in den 3 Optionen?
  - Vertikale Partitionierung (eine Tabelle pro Klasse mit Bezug)
  - Horizontale Partitionierung (eine Tabelle pro Klasse)
  - Universelle Relation (eine Gesamttabelle)
- Wie selektieren Sie alle Attribute einer Institution in den 3 Optionen?

1. Aufgabe
Vertikal: SELECT Name FROM Partner
Horizontal: SELECT Name FROM Partner UNION SELECT Name FROM Institution UNION SELECT Name FROM Person
Universal: SELECT Name FROM Partner
2. Aufgabe
Vertikal: SELECT * FROM Partner JOIN Institution ON Partner.oid = Institutoin.oid
Horizontal: SELECT * FROM Institution
Universal: SELECT oid, name, type FROM Partner WHERE entityType = "i"

c) OR-Mapping mit JPA: Einführung

Ziel des OR-Mappings:

- Abbildung von programmiersprachlichen Objekten auf relationale Datenbanken
- dabei möglichst große Werkzeugunterstützung
- möglichst transparente Abbildung, d.h., der Anwendungsprogrammierer soll möglichst wenig Persistenzcode programmieren müssen

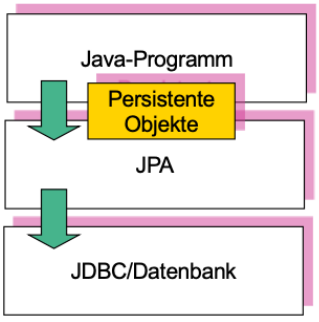
heute sind diverse Ansätze/Produkte verfügbar, z.B.

- JPA(Spezifikation mit mehreren Produkten)
  - Beispielimplementierungen OpenJPA, Hibernate
- ~~JDO(Spezifikation mit mehreren Produkten)~~

JPA: Java Persistence Architecture

Vollständig nicht möglich, weil die persistenten Klassen & Attribute vorgegeben werden müssen

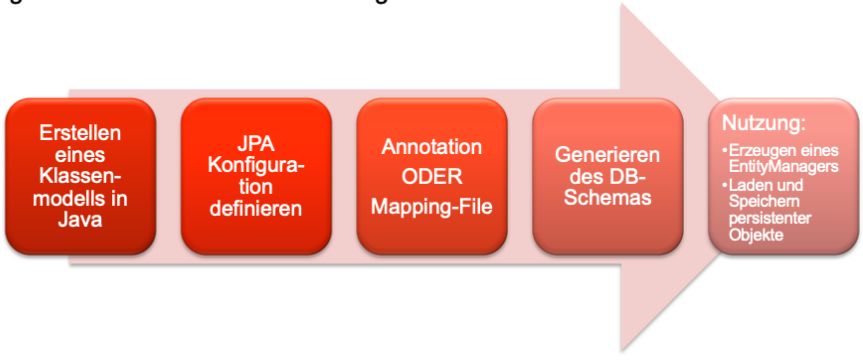
- Idee: Semi-Automatische Abbildung von Objekten auf relationale Tabellen
  - Vollständige Automatisierung nicht möglich (Warum?)
  - Einige Schritte nur mit Konfiguration möglich
- Damit kann jedes beliebige Java-Objekt persistiert werden, ohne dass Schnittstellen etc. definiert werden müssen
- Hibernate: Implementierung von JPA
- OpenJPA: Apache Implementierung der JPA
- EclipseLink: JPA-Implementierung der Eclipse Foundation
- Beide basieren auf Reflection API
  - Metadaten von Klassen werden zur Laufzeit untersucht
  - Annotationen steuern die konkrete Abbildung der Klassen auf die DB



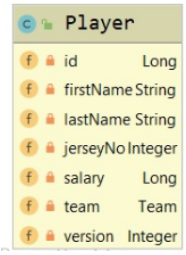
Wichtige Eigenschaften eines JPA-Providers

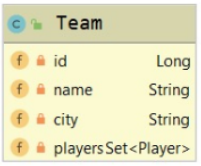
- JPA-Provider definiert das O/R-Mapping mit Unterstützung der Benutzerin durch Annotationen in den Klassen
- JPA-Provider generiert SQL-Statements für das Laden, Speichern, Löschen, Verändern von Objekten
- JPA-Provider kann mit beliebigen Technologien aus der Java / Application-Server-Welt kombiniert werden
- JPA-Provider besitzt eine eigene Query-Sprache
  - JPQL: Java Persistence Query Language (Standardisiert)
- API definiert in Package `javax.persistence`
- Können in Java EE Application Server verwendet werden

Vorgehensweise bei der Entwicklung mit JPA:



d) Hibernate als Realisierung





Beispiel:

- In dieser Beispielanwendung sollen Fußballteams und Spieler/innen verwaltet werden.
    - Zwischen Team und Spieler/in besteht eine 1:n-Beziehung.
- Dabei soll die eigentliche dynamische Zuordnung in Java-Klassen geschehen, jedoch sollen die Informationen in einer Datenbank abgelegt werden.
- Der Großteil des notwendigen Codes soll automatisch generiert werden.

Konfigurationsdatei: persistence.xml

- Property: Parameter

Legt Datenbankverbindung fest, plus weitere Optionen

```
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
                                 http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
             version="2.0">
    <persistence-unit name="team">
        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
        <properties>
            <property name="javax.persistence.jdbc.driver“
                      value="oracle.jdbc.driver.OracleDriver" />
            <property name="javax.persistence.jdbc.url“
                      value="jdbc:oracle:thin:@localhost:1521/orcl" />
            <property name="javax.persistence.jdbc.user" value="scott" />
            <property name="javax.persistence.jdbc.password" value="tiger" />
            <property name="hibernate.show_sql" value="true" />
            <property name="hibernate.hbm2ddl.auto" value="update" />
        </properties>
    </persistence-unit>
</persistence>
```

create

Gebräuchlicher: Mapping über Annotationen

- @GenerateValue: benutzt Sequenz
- Cascade: hier auch Referenzen persistieren

```
@Entity
@Table(schema = "team", name="hsteams")
public class Team {

        @Id
        @Column(name="team_id") @GeneratedValue
        private Long id;

        @Column(name="team_name", nullable=false)
        private String name;

        @Column(nullable=false)
        private String city;

        @OneToMany(mappedBy="team", cascade = CascadeType.PERSIST)
        private Set<Player> players = new HashSet<Player>();

        public Team() {
        }
…
}
```

Entity klasse
default schema

not NULL

Attribut-name in der anderen Klasse

1-n

JPA-Provider – Verwendung

Hibernate:

- Zur Verwendung müssen die JARs von Hibernate im Classpath liegen, außerdem muss der gewählte JDBC-Treiber (z. B. ojdbc8.jar für Oracle) im Classpath liegen.
- Einbindung z.B. über eine Maven-Abhängigkeit:

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>5.6.1.Final</version>
</dependency>
```

Laufzeit: Entity Manager und -Factory

- 1 em aus emf pro Transaktion empfohlen

Zur Speicherung von Objekten wird der **EntityManager** verwendet.

- Basis zur Benutzung von JPA / Hibernate
- Kern der Datenbankkommunikation
- Nutzbar zur Festlegung von Transaktionsgrenzen

Dazu muss zunächst eine **EntityManagerFactory** angelegt werden

- teure Operation
- deswegen nur eine **EntityManagerFactory je Datenbank/Persistence** Unit halten

```
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
...
EntityManagerFactory emf =
        Persistence.createEntityManagerFactory("moviedb");
```

aus der xml Datei?

Laufzeit: Abspeichern von Objekten

```
Team team = new Team();
team.setCity("Irgendwo");
team.setName("A-TEAM");

// add a player to the team.
Player player = new Player();
player.setFirstName("Quite");
player.setLastName("Expensive");
player.setJerseyNumber(1);
player.setAnnualSalary(4000000);
team.addPlayer(player);

// open an EntityManager and save the team
EntityManager em = emf.createEntityManager();
em.persist(team);
```

↳ Daten sofort in die Datenbank geschrieben?

Laufzeit: Laden von Objekten

- lazy loading: lädt nicht Referenzen, weil sonst zu viel geladen werden kann
- eager: lädt auch Referenzen
- Players wird zuerst mit dummy-Wert initialisiert. Man muss mit em navigieren, um Referenzen zu kriegen

```
...

Query q = em.createQuery(
     "SELECT t FROM de.hsh.entities.Team t " +
     "WHERE t.name = :name");
q.setParameter("name", teamName);

Team team = (Team) q.getSingleResult();

Set<Player> players = team.getPlayers();
em.close();

for (Player player : players)
     System.out.println(qplayer.getFormattedName());
// exception might be thrown here
```

ab hier kann man nicht mehr mit den Objekten arbeiten

NULL POINTER EXCEPTION?

q.team.setCity("");

e) Transaktionen und Objektzustände in JPA

JPA: Transaktionen

- Es müssen Transaktionen verwendet werden
- Aktionen innerhalb einer Transaktion werden entweder alle (`commit()`) oder gar nicht (`rollback()`) ausgeführt
- Dadurch Sicherstellung der Datenintegrität
- Eine Transaktion kann auch mehr als nur eine Aktion umschließen (application-level transaction)

```
private Long createAndStoreTeam(String name,
                                 String city) {
    EntityManager em = emf.createEntityManager();
    EntityTransaction tx = em.getTransaction();
    tx.begin();

    Team team = new Team();
    team.setCity(city);
    team.setName(name);
    em.persist(team);
    tx.commit();
    em.close();
    return team.getId();
}
```

Änderung von Objekten

- Wenn ein Objekt persistent ist, steht es unter der Kontrolle des `EntityManager`
- Das bedeutet: Jede Änderung wird "verfolgt" und automatisch persistiert
- Es wird keine Anweisung zum Speichern von Änderungen benötigt
- Spätestens, wenn der `EntityManager` geschlossen wird oder die Transaktion festgeschrieben wird
- Oder durch Aufruf der `flush()` -Methode

```
public static void modifyTeam() {
    EntityManager em = emf.createEntityManager();
    EntityTransaction tx = em.getTransaction();
    tx.begin();

    // Objekt aus der Datenbank laden
    Team bayern = (Team) em.createQuery(
        "SELECT t FROM Team t " +
        "WHERE t.name = 'FC Bayern'").getSingleResult();

    // Objekt verändern
    bayern.setName("FC Bayern München");

    // kein weiterer Aufruf zum Speichern notwendig!
    tx.commit();
    em.close();
}
```

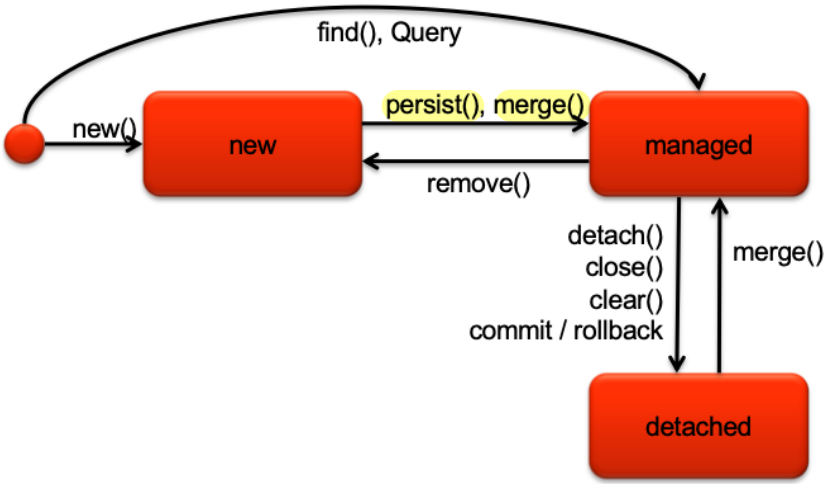
JPA: Detached Objekte

- Wenn der `EntityManager` geschlossen wird, kann die Applikation persistente Objekte weiterverwenden
- Die Objekte heißen dann "Detached"
- Sie können z.B. in einer GUI modifiziert werden
- Um die Änderungen zu speichern, müssen Sie wieder an einen `EntityManager` "Attached" werden

Daten die nicht mehr mit der Datenbank gekoppelt sind

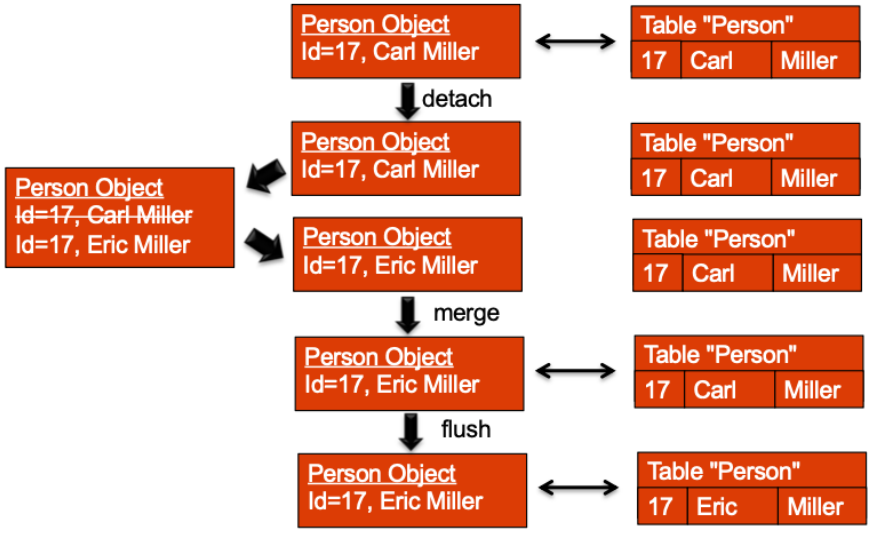
Lebenszyklus von Entities

- merge: gibt neues Objekt zurück, deshalb bei new besser persist() nutzten
- flush benutzten, wenn merge verwendet wird, um Änderungen zu speichern



Detached Objekte

- Hier lost update möglich, OR-Mapper nimmt immer RAM-Objekt als neustes und überschreibt dann im DBS
- Fehlerbehandlung: Transaktion um ganzen Block, aber zu lang sollten Transaktionen nicht sein, da auch nie Transaktion während GUI-Interaktionen vom Benutzter gemacht werden



Versionierung

Versionsprüfung bei Merge:

```
public Class Player {
…
    @Version
    @Column(name="VERSION")
    private Integer version;

    public Integer getVersion() {
        return version;
    }

    public void setVersion(Integer version) {
        this.version = version;
    }
…
}
```

für Detached Objekte? Vom mergen?

```
em = emf.createEntityManager();
em.getTransaction().begin();
Player p1 = (Player) em.createQuery(
    "SELECT p FROM Player p WHERE p.lastName = 'Götze'"
).getSingleResult();
em.getTransaction().commit();
```

1

holt spieler aus der DB

```
em = emf.createEntityManager();
em.getTransaction().begin();
Player player = (Player) em.createQuery(
    "SELECT p FROM Player p WHERE p.lastName = 'Götze'"
).getSingleResult();
player.setJerseyNumber(99);
em.getTransaction().commit();
```

2

holt auch und aktualisiert den

```
p1.setJerseyNumber(111);
em = emf.createEntityManager();
em.getTransaction().begin();
p1 = em.merge(p1);
em.getTransaction().commit();
```

1

javax.persistence.OptimisticLockException: Row was updated or deleted by another transaction (or unsaved-value mapping was incorrect)

↖ Versucht es auch =>

Oberste Klasse?

f) Abbildungsregeln

JPA: Mapping von Hierarchien

Deklaration in der Basisklasse:

```
@Entity
@Table(schema = "job", name = "partner")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
public class Partner { ...
```

Varianten:

- `TABLE_PER_CLASS` (horizontale Partitionierung)
- `JOINED` (Vertikale Partitionierung)
- `SINGLE_TABLE` (Universelle Relation)

Unterklasse:

```
@Entity
public class Person extends Partner {
```

Mapping von 1:N Beziehungen

Eine Klasse "besitzt" die Beziehung:

```
public class Player {
…
    @ManyToOne
    Team team;
…
}
```

Die andere Klasse referenziert darauf:

```
public class Team {
…
    @OneToMany(mappedBy="team", cascade = CascadeType.PERSIST)
    private Set<Player> players = new HashSet<Player>();
…
}
```

Mapping von N:M Beziehungen

Eine Klasse "besitzt" die Beziehung:

```
public class Job {
…
    @ManyToMany
    @JoinTable(name = "job_partners", schema = "job")
    private List<Partner> partners = new ArrayList<Partner>();
…
}
```

Die andere Klasse referenziert darauf:

```
public class Partner {
…
    @ManyToMany(mappedBy = "partners")
    private List<Job> jobs = new ArrayList<Job>();
…
}
```

Pflege von bi-direktionalen Beziehungen

Wenn beide Klassen die Beziehung "kennen":

- Beim Einfügen einer neuen Beziehung müssen beide Seiten gepflegt werden:

```
public class Partner {
…
    public void addJob(Job newjob) {
        this.jobs.add(newjob);
        if (!newjob.getPartners().contains(this)) {
            newjob.addPartner(this);
        }
    }
…
}
```

- Analoge Implementierung auf Seite Job

Uni-direktionale Beziehungen

Sollen Beziehungen nur in eine Richtung zugreifbar sein:

- Entsprechendes Attribut nur in einer Klasse definiert
- Dann auch nur Annotation auf dieser Seite
- Pflege der Beziehung erfolgt dennoch durch EntityManager
- Aber kein Zugriff auf Objekte, bei denen man als zugeordnetes Objekt vorkommt
- Für alle Arten von Beziehungen möglich (OneToOne, OneToMany, ManyToOne, ManyToMany)

JPA: Mapping von Beziehungen

Beziehungsarten

Beziehungen können 1:1, 1:n oder n:m sein

→ Annotation **`@OneToOne`**, **`@OneToMany`**, **`@ManyToOne`**, **`@ManyToMany`**

Beziehungen uni-direktional oder bi-direktional verwaltet

→ Verwendung von Annotation und Feld in einer oder beiden beteiligten Klassen

→ Gegenstück von **`@OneToMany`** ist **`@ManyToOne`**

Für One-Seiten werden Felder der entsprechenden Zielklasse genutzt

Für Many-Seiten der Beziehungen werden bestimmte Kollektionstypen dieser Zielklassen in Java verwendet, bspw. Set, List

Je nach Art der Beziehung können zusätzliche Parameter für die Annotation verpflichtend oder optional sein

JPA: Mapping von Beziehungen

Annotationsparameter

Beziehung ist optional oder nicht **`(optional = true/false)`**

→ Sorgt bspw. für outer oder inner join bei Abruf der Beziehungen

Parameter **`fetch`** gibt an, ob referenzierte Objekte automatisch mit geladen werden sollen

Parameter **`cascade`** definiert Verhalten bzgl. abhängiger Objekte, Details siehe **`CascadeType`** enum in JPA

→ Kann bspw. zur Abbildung von Komposition oder Aggregation genutzt werden

Parameter **`mappedBy`** gibt an, welches Java-Feld die inverse Beziehung bei bi-direktionalen Beziehungen enthält

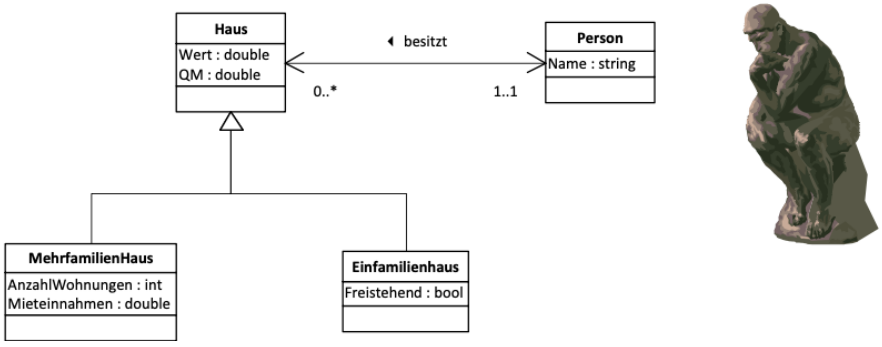
Zusätzlich können viele der genutzten Namen in der DB beeinflusst werden

JPA Bewertung

- JPA ermöglichen enge Bindung zwischen OOP und DBMS
  - weitgehende Vermeidung des Impedance Mismatch
- Bessere Performance für Anwendungen mit navigierendem Zugriff möglich
- Ermöglicht relativ schnell die Erzeugung einer einfachen Persistenz durch Generator-Funktionalität
- Anbindung (zumeist) nur an RDBMS möglich
- Nicht für alle Anfrageprobleme geeignet
  - Vor einer Entwurfsentscheidung sollte auf Hauptfunktionalitäten getestet werden
  - Bei größeren Datenmengen → Lasttests durchführen

OR-Mapper – Fragenblock

Schreiben Sie für das folgende Klassendiagramm die **minimal** nötigen Annotationen auf. Sie haben 5 Min Zeit.



g.) Anfragen auf dem Objekt modell in JPA

Anfragesprache JP-QL

- Komplexere Anfragen aus Performanzgründen direkt in DB ausführen
- Analog für größere Änderungs- oder Löschoperationen
- ABER: Abstraktion der Datenbank in Java-Anwendung soll erhalten bleiben
- Daher nötig: Anfragesprache ähnlich SQL, aber nicht auf dem Modell in der DB, sondern auf dem Entitätsmodell in Java
  → JP-QL
- Beispiel: Job und Partner; Abfrage der Beziehungen über Attribut von Job oder Partner, nicht über Kreuztabelle `JOB_PARTNER`!

Anfragesprache JP-QL

Besteht i.w. aus `SELECT, UPDATE, DELETE`
Jeweils vereinfachte Varianten der SQL-Kommandos

```
SELECT ... FROM ...
[WHERE ...]
[GROUP BY ... [HAVING ...]]
[ORDER BY ...]
```

```
DELETE FROM ... [WHERE ...]
```

```
UPDATE ... SET ... [WHERE ...]
```

Anfragesprache JP-QL

Im Projektionsteil des SELECT einfache skalare und aggregierende Funktionen möglich:

```
SELECT upper(j.description) FROM Job j
```

Im Selektionsteil Bedingungen auf Attributen möglich:

```
q = em.createQuery("SELECT j.description FROM Job j "+
          "WHERE j.description LIKE :name");
q.setParameter("name","My first job%");
```

Auch auf Beziehungsattributen mit Funktionen:

```
SELECT j.description FROM Job j WHERE size(j.partners) > 2
```

Anfragesprache JP-QL

Abfragen von Beziehungen erfolgt auf dem Entitätsmodell:

```
SELECT j.description, p.name FROM Partner p JOIN p.jobs j
WHERE j.description LIKE :name
```

Liefert Liste von Informationen zu Partner und zugehörigem Job, keine Verwendung der Kreuztabelle o.ä., stattdessen direkt auf den Entitäten
Auch andere Join-Varianten (bspw. outer join) möglich
Modell kann auch Vererbung nutzen:

```
SELECT j.description, p.birthDate FROM Person p JOIN p.jobs j
WHERE j.description LIKE :name
```

Liefert nur Personen und zugehörige Jobs

Anfragesprache JP-QL

Viele weitere nützliche Funktionalitäten hier nicht behandelt, oft ähnlich zu SQL:

- Sortierung
- Gruppierung (auch mit HAVING)
- Sub-Selects
- Statische Konstruktoren
- Details zu UPDATE und DELETE

In Hibernate: eigene, teilweise erweiterte, Variante HQL von JP-QL

Zusammenfassung

OR-Mapper: JPA und Beispielimplementierung Hibernate • Konzepte

- Mapping-Datei, Konfiguration
- Annotations-BasiertesMapping
- Beispiel
- Transaktionen und Versionierung
- Objekt-Lebenszyklus
- Mapping von Hierarchien und Beziehungen
- Abfrage von Entitäten mit JP-QL