Kap. 4 Objekt-Relationales Mapping

Drei-Schichten-Architektur

Präsentationschicht

- Präsentiert fachliche Daten (Nutzer eingaben aufnehmen)

Anwendungsschicht (Applikationsschicht)

- soll keine technischen Details über DB-Anbindung enthalten

Persistenzschicht (Zugriffsschicht)

- Verwaltung fachlicher Objekte in DBMS

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

b) Transformation

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

Transformation eines Klassenmodells in Datenmodell

- **Schritt 1: Auswahl der persistenten Klassen**
  - Enthalten Attribute
- **Schritt 2: Hinzufügen eines Oid-Attributs**
  - Primär-Schlüssel
- **Schritt 3: Abbildung einer Klasse auf eine Entität**
- **Schritt 4: Abbildung von Beziehungen**
- **Schritt 5: Abbildung der Vererbungen**
- **Schritt 6: Überführung in physikalisches Modell**

Abbildung von Vererbungshierarchien:

Vertikale Partitionierung (JOINED)

- Super- und Sub-Entitäten bilden jeweils eigene Klassen

VT	NT
- + Struktur passt zum Klassenmodell - + Konsistenzerhaltung einfach	- - Um alle Attribute eines Objektes zu selektieren, werden viele JOINS benötigt

Horizontale Partitionierung (TABLE PER CLASS)

- Inhalt einer Tabelle bildet kompletten Zustand eine Entität ab!

VT	NT
- + Beim Zugriff auf Unterklasse wird nur eine Tabelle benötigt, d.h. gute Performance	- UNION nötig - Abbildung von Bzh. Schwieriger - Keine eindeutigen PK

Universelle Relation (SINGLE TABLE)

- Alle Attribute aller Entitäten werden Attribute EINER Relation

VT	NT
- + Einfacher Zugriff auf alle Objekte / Attribute, dadurch gute Performance	- Viele NULL-Werte - NOT NULL-Constrains nicht möglich

mehrfach vererbung ist moeglich

```
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
```

default?

c) OR-Mapping mit JPA: Einführung

- Abbildung von programmiersprachlichen Objekten auf relationale Datenbanken
- Vollständig nicht möglich, weil die persistenten Klassen & Attribute vorgegeben werden müssen

Hibernate

- `@Table(name = "")`
- `@Entity`
- `@Column`
- `@Id` `@GenerateValue:` benutzt Sequenz
- `@OneToOne`
- `@OneToMany(mappedBy="asdasd")` - `@ManyToOne`
- `@ManyToMany`
- `@Inheritance(strategy=InheritanceType....)`
  - InheritanceType
    - TABLE_PER_CLASS, JOINED, SINGLE_TABLE
    - (Horizontal) (Vertikal) (Universell)

```java
@Entity
@Table(name = "kunde")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
public class Kunde{
    @Column @Id @GeneratedValue
    private int id;

    @Column(nullable = false)
    private String name;

    @OneToMany(mappedBy = "kunde", cascade =
CascadeType.MERGE)
    private List<Projekt> hatBeauftragt = new
ArrayList<Projekt>();

    @ManyToMany(cascade = CascadeType.MERGE)
    @JoinTable(name = "movieGenrehib")
    private Set<Genre> genres = new
HashSet<Genre>();

}
```

```java
@Entity
@Table(name = "Geschäftskunde")
public class Geschäftskunde extends Kunde{
    @Column @Id @GeneratedValue
    private int id;

    @Column
    private String rechtsform;
}
```

```java
@Entity
@Table(name = "projekt")
public class Projekt{
    @Column @Id @GeneratedValue
    private int id;

    @Column
    private String name;

    @ManyToOne
    private Kunde kunde;
}
```

Mapping von 1:N Beziehungen

Eine Klasse "besitzt" die Beziehung:

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

Die andere Klasse referenziert darauf:

```java
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:

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

Die andere Klasse referenziert darauf:

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

Laufzeit: Entity Manager und -Factory

Zur Speicherung von Objekten wird der **EntityManager** verwendet.
Dazu muss zunächst eine **EntityManagerFactory** angelegt werden

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

Aktionen innerhalb einer Transaktion werden entweder alle (`commit()`) oder (`rollback()`) ausgeführt
→ Dadurch Sicherstellung der Datenintegrität

```
public List<String> getIWAS() throws Exception {
    EntityManager em =
EMConnection.getEntitymManager().createEntityManager();
    EntityTransaction tx = em.getTransaction();
    ...
    try {
        tx.begin();
        ...
        Collections.sort(GenreList);
        tx.commit();
    } finally {
        if (tx.isActive()) {
            tx.rollback();
        }
        em.close();
    }
}
```

JPA: Detached Objekte

- Wenn der `EntityManager` geschlossen wird, kann die Applikation 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

Lebenszyklus von Entities

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

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