
Software Engineering

Prof. Adrian A. Müller, PMP

Fachbereich Informatik und Mikrosystemtechnik
Fachhochschule Kaiserslautern, Standort Zweibrücken

Inhalte

Teil 1

- Möglichkeiten der Datenhaltung
 - Objektorientiert
 - „Flach“
 - Relational
- Der „Object-Relational (Impedance) Mismatch“
- Schnittstellen zu relationalen Datenbank-Systemen
 - Anfragen
- OO-Relationales Mapping
 - Vererbung
 - Polymorphismus
 - Assoziationen, Aggregationen und Kompositionen

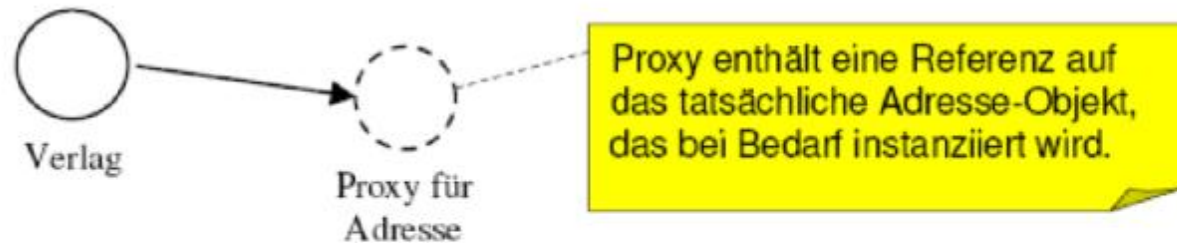
Teil2

- Integration relationaler Datenbanken
- Persistenzschicht
- Beispiel:
 - Java Persistence API (JPA)
- Zusammenfassung

- Hinweis
 - Dieses Modul baut auf der Vorlesung „Systemanalyse“ (Allweyer, Steffens) auf
 - Weitere Quellen:
 - Heide Balzert, WS ,05
 - Alexander Kunkel, JAP 2.0

Integration relationaler Datenbanken

- Dereferenzierung von Beziehungen (Navigation)
- Verwendung von Proxys oder *Lazy Instanciation*

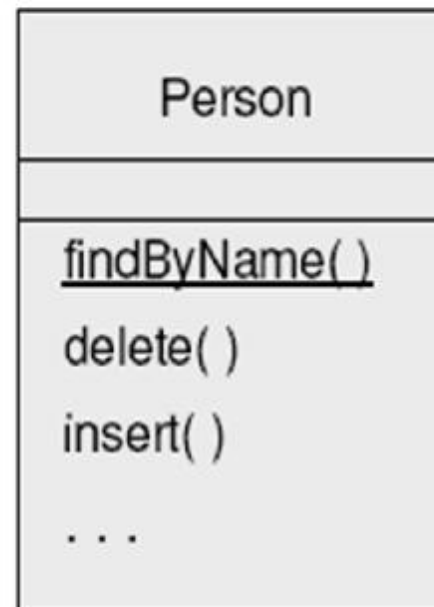


```
class Verlag {  
    private Adresse adr = null;  
    Adresse getAdresse()  
    {  
        if( this.adr == null )  
        {  
            // instanziiere Adresse  
        }  
        return this.adr;  
    }  
}
```

Erst beim Zugriff wird die Adresse instanziiert.

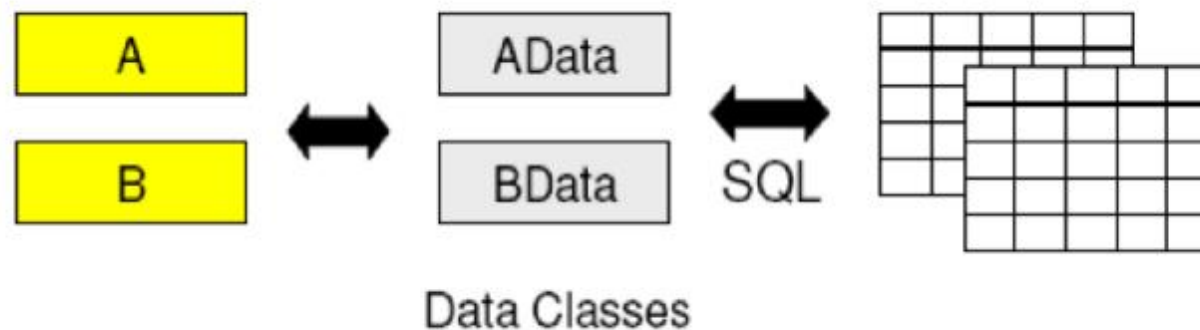
Integrationslösungen (1)

- Einfache Lösung 1: Direct Data Access
 - Hardcoded SQL in Modellobjekten



Integrationslösungen (2)

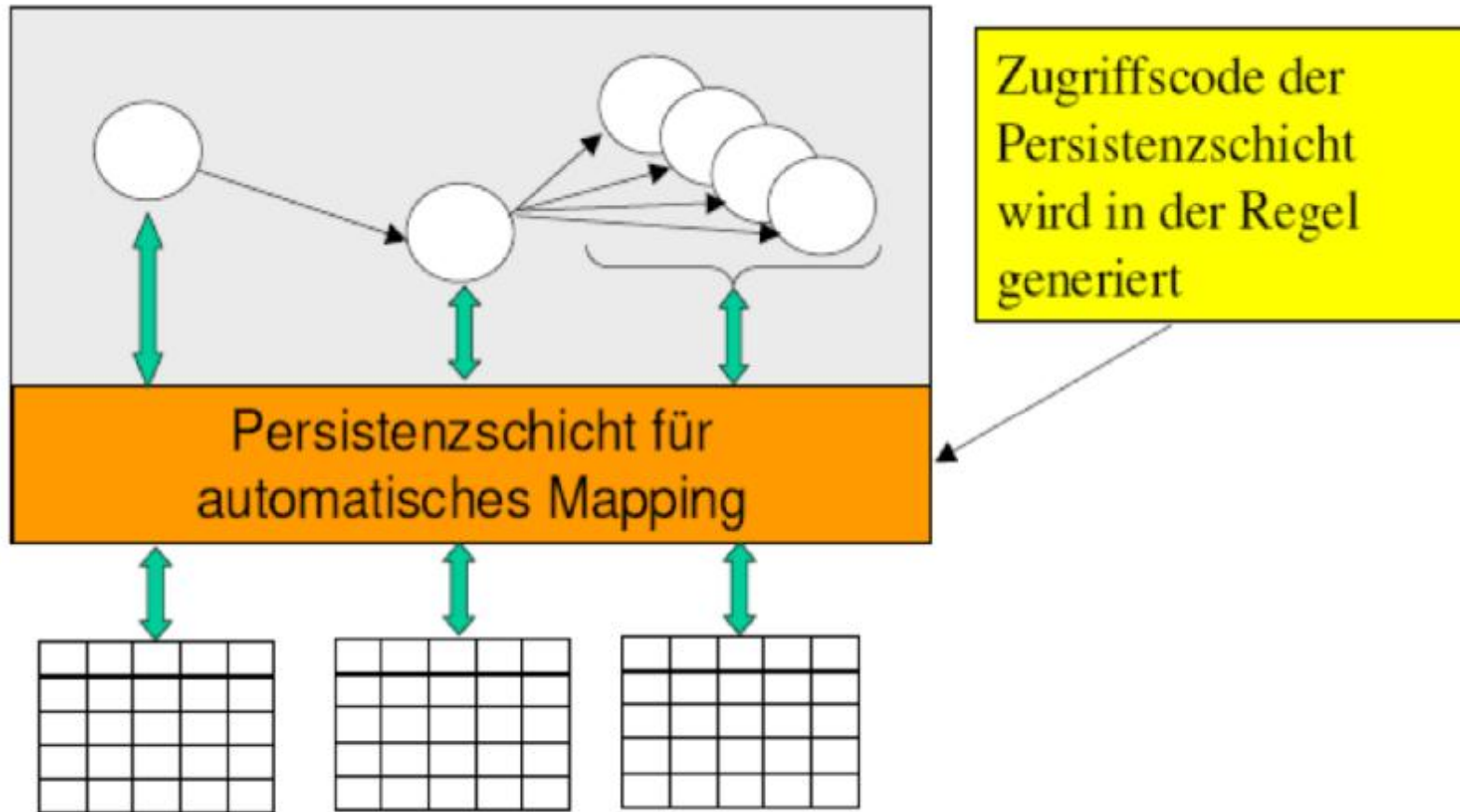
- Einfache Lösung 2: Data Access Layer
 - Zu Modellobjekten korrespondierende Datenklassen



Integrationslösungen (3): O/R Mapping

- Separate Schicht kapselt Datenbankzugriffe (Persistenzschicht)
 - Kapseln der Datenbankfunktionalität ermöglicht transparenten Objektzugriff.
 - Anwendungsentwickler benötigen kein Wissen über die Datenbank und das verwendete Schema.
 - Änderungen am Schema der Datenbank werden durch die Persistenzschicht abgeschirmt.

Persistenzschicht



Anforderungen an die Persistenzschicht

- Support verschiedener Datenquellen (RDB, OODB, CICS, IMS-DC, etc.)
- Transaktionsunterstützung und Locking
- Zugriffsschutz auf die Datenbank
- Caching
- Proxys (fetch-policies)
- Objektidentitäten
- Fehlerbehandlung bei fehlerhaften Datenbank-Operationen
- Query Language (z.B. SQL3)
- etc.

Einfache Persistenzschicht

1. Abbildung auf Tabellen
2. Database Broker-Muster
3. Materialisierung von Objekten
4. Optimierung der Materialisierung mittels Cache
5. Virtual Proxy Muster
6. Materialisierung von Objektstrukturen
7. Transaktionen

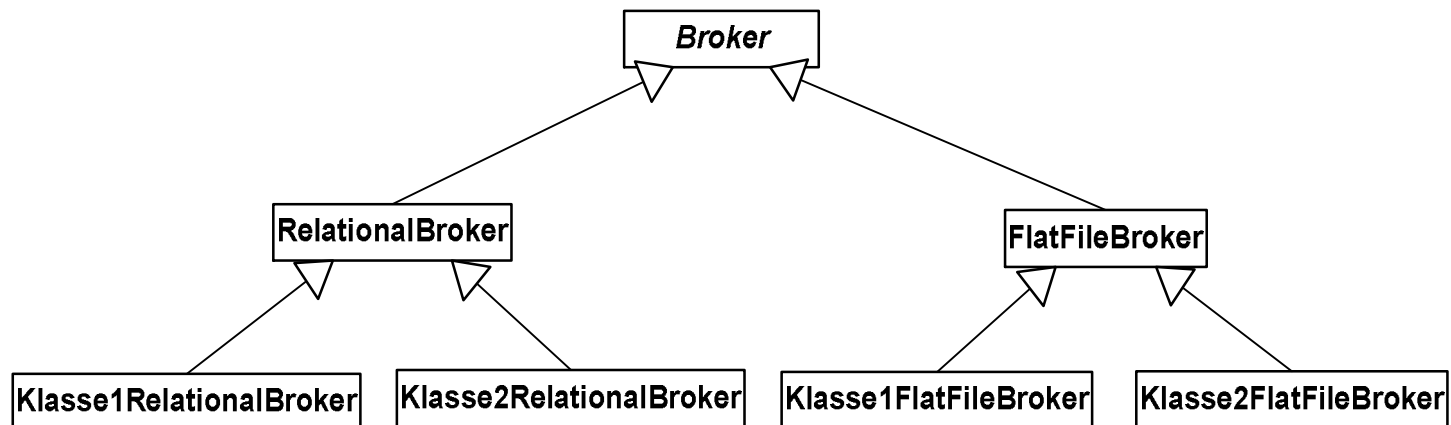
Quelle: Balzert, H.: Lehrbuch der Objektmodellierung

1. Abbildung auf Tabellen

- Einfache Klasse wird auf eine Tabelle abgebildet, wobei die Objekt-ID (OID) hinzugefügt wird
- Klassenattribute werden in einer eigenen Tabelle gespeichert
- 1:*-Assoziationen: Eintrag der entsprechenden OIDs als Fremdschlüssel in die Tabelle am „many“-Ende der Assoziation
- *:*-Assoziationen: Nutzung einer eigenen Tabelle
- Vererbung – mehrere Möglichkeiten
 - Eine Tabelle für alle Objekte der gesamten Vererbungsstruktur
 - Eigene Tabelle für jede konkrete Klasse
 - Eigene Tabelle für jede Klasse

2. Database Broker-Muster

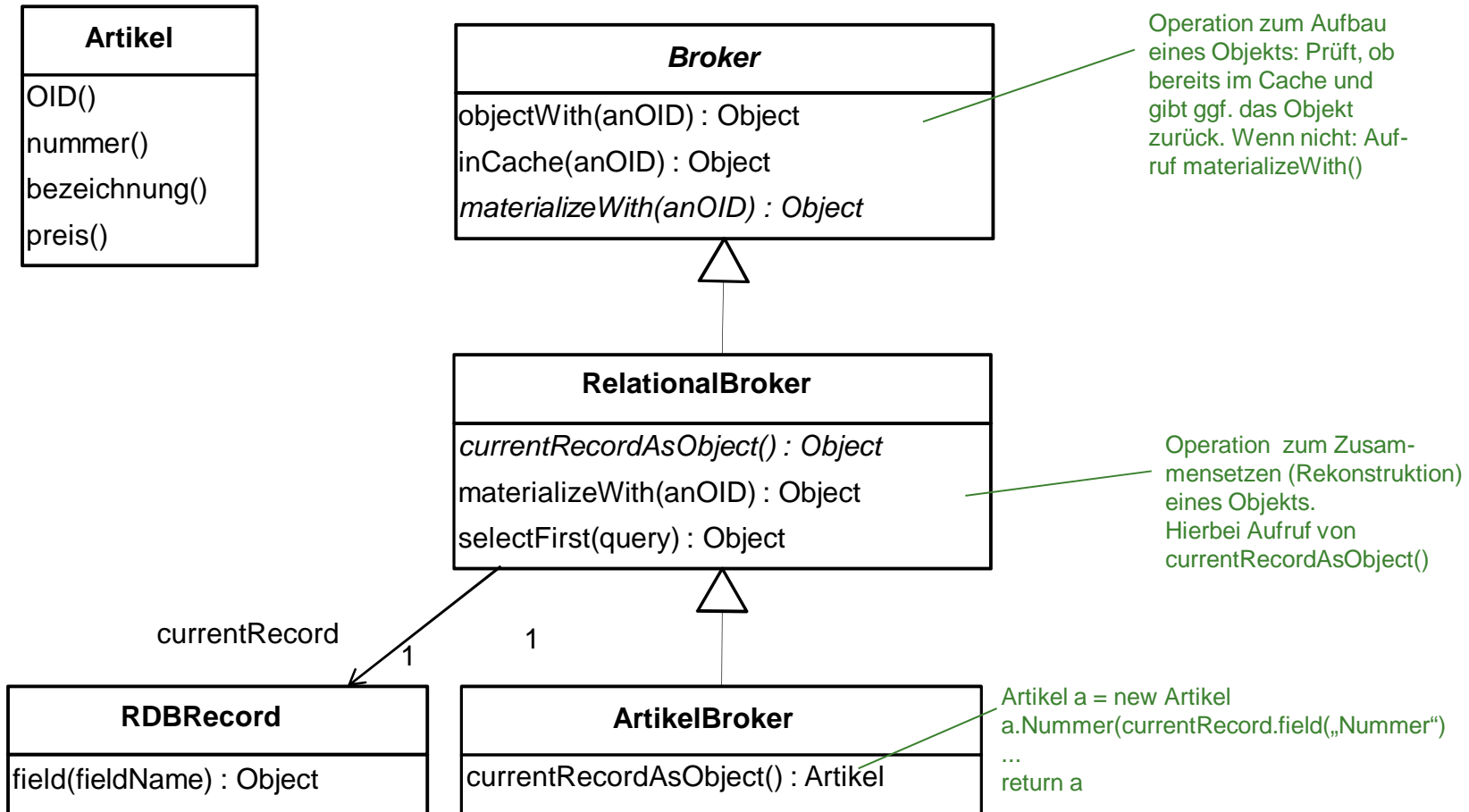
- Nutzung einer Klasse „Broker“
 - Zerlegen der Objekte in Datensätze (De-Materialisierung)
 - Wiedergewinnen der Datensätze (Materialisierung)
 - Cache-Verwaltung
- Jede persistente Klasse kann einen eigenen Broker haben



3. Materialisierung von Objekten

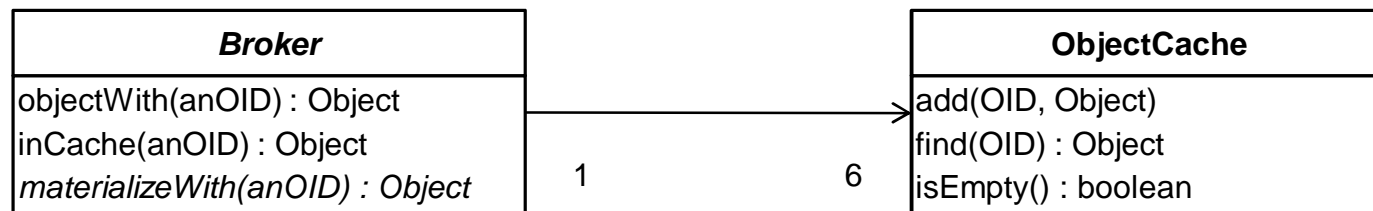
- Nutzung des Schablonenmethoden-Musters zum Aufbau eines Persistence-Frameworks
 - Siehe folgende Folie
- Dies ist ein Beispiel für ein typisches Framework
 - Abstrakte Oberklassen verwenden Schablonen-Methoden
 - Die Software-Entwickler fügen Unterklassen hinzu
 - In den Unterklassen werden elementare Operationen definiert, um die geerbten Schablonenmethoden zu vervollständigen

Materialisierung von Objekten



4. Optimierung d. Materialisierung mittels Cache

- Materialisierung von Objekten ist relativ aufwändig
- Zur Verbesserung der Performance werden materialisierte Objekte in einem Cache-Speicher gehalten
- Falls jede Anwendungsklasse einen eigenen Broker besitzt, gibt es auch für jede Klasse einen eigenen Cache
- Es kann bis zu 6 Caches pro Broker geben, wobei die Caches dann jeweils einen Transaktionszustand realisieren (s.u.)



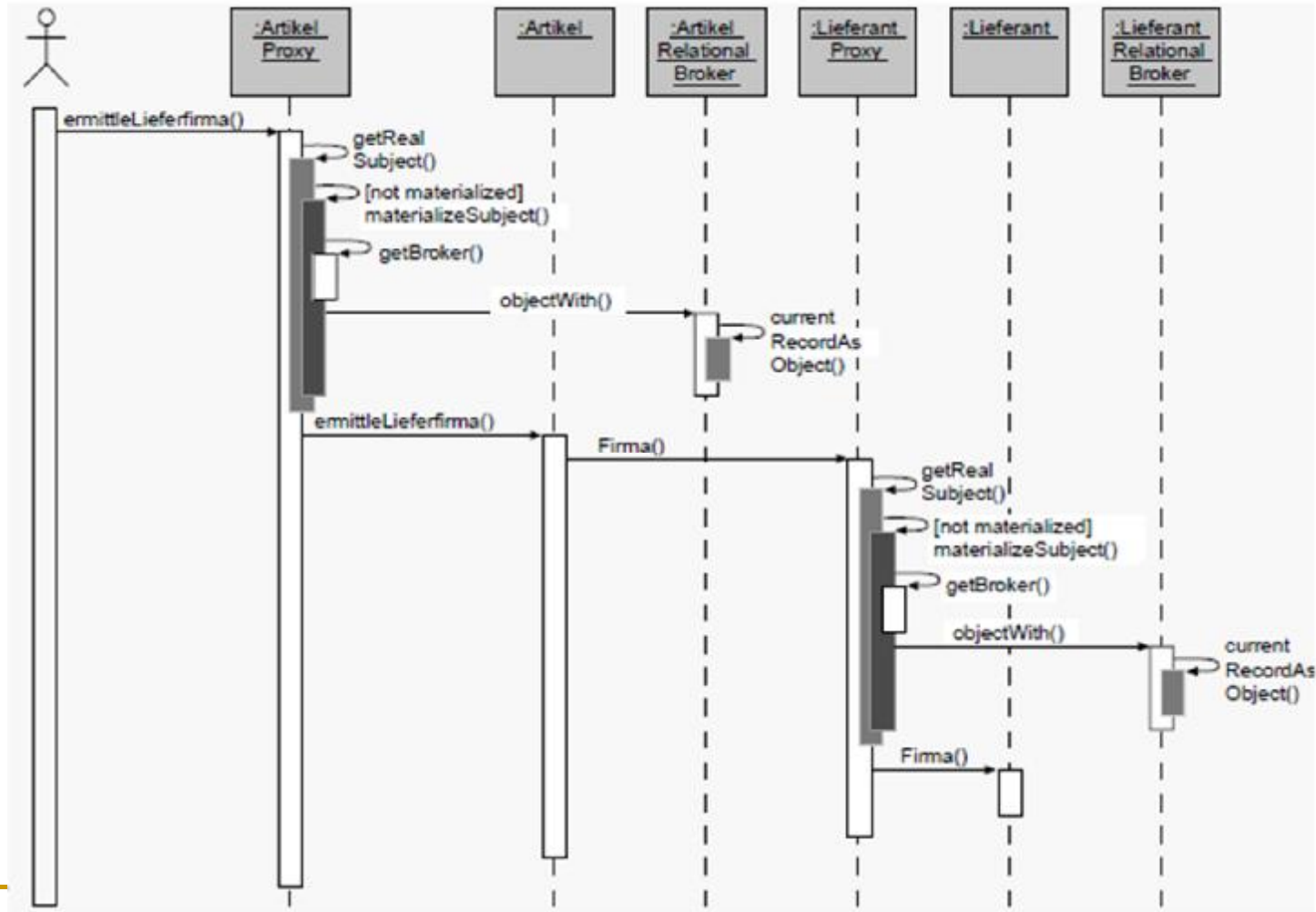
5. Virtual Proxy-Muster

- U. U. kann es sinnvoll sein, ein Objekt erst dann zu materialisieren, wenn es benötigt wird
- Einsatz des Proxy-Musters
 - Proxy-Objekt kennt nur die OID und die Signaturen des eigentlichen Objekts
 - Wird eine Botschaft, an das Proxy-Objekt gesandt, stößt dieses die Materialisierung des Objektes an und leitet die Botschaft an dieses weiter

6. Materialisierung von Objekt-Strukturen

- Es sollen nicht nur einzelne Objekte materialisiert werden, sondern gesamte Strukturen (z. B. ein Lieferant mit allen Artikeln, die er liefert)
- Wird die Gesamtstruktur materialisiert, so ist dies sehr aufwändig
- Lösung:
 - Materialisierung on-demand
 - Erst wenn zu dem assoziierten Objekt eine Botschaft geschickt wird, wird dieses durch sein Proxy materialisiert

Materialisierung von Objekt-Strukturen: Beispiel (Quelle: H. Balzert)



7. Transaktionen

Transaktionszustand	Bedeutung	Aktion bei commit	Aktion bei roll-back
new clean	neu erzeugte, nicht veränderte Objekte	Objekt in DB einfügen und in old clean Cache verschieben	Objekt im Cache löschen
old clean	alte materialisierte, nicht veränderte Objekte	keine Aktion	keine Aktion
new dirty	neu erzeugte, veränderte Objekte	Objekt in DB einfügen und in old clean Cache verschieben	Objekt im Cache löschen
old dirty	alte materialisierte, veränderte Objekte	Objekt in DB aktualisieren und in old clean Cache verschieben	Objekt im Cache löschen
new delete	neu erzeugte Objekte, die gelöscht werden	Objekt aus Cache löschen	Objekt im Cache löschen
old delete	alte materialisierte Objekte, die gelöscht werden	Objekt aus DB entfernen und Cache löschen	Objekt im Cache löschen

Beispiel: Java Persistence API (JPA)

- Spezifikation für die Persistierung von Java-Objekten in relationale Datenbanken
- Ursprünglich aus Java EE (EJB), es handelt sich aber um eine leichtgewichtige Lösung, auch für Java SE-Anwendungen
- Verschiedene Implementierungen (JPA-Provider)
 - z. B. Hibernate, EclipseLink (JAP 2.0), TopLink
- Objekt-relacionales Mapping von POJOs (Plain Old Java Objects) mittels Meta-Daten
 - als Annotations im Java Code, *oder*
 - in separater XML-Datei
- Weitere Elemente
 - Abfragesprache
 - Implementierungen ermöglichen zumeist auch die Erzeugung des DB-Schemas

JPA: Entity-Klasse mit Annotation

Markieren der Klasse als „Entity“. Ohne weitere Angaben ist der Name der Tabelle der selbe wie der der Klasse.

Attribut „Name“ als Primärschlüssel.

Attribute werden ohne weitere Angaben automatisch in gleichnamige Tabellenspalten gespeichert.

```
@Entity
public class Person {
    @Id
    private String name;


    public String getName() {
        return name;
    }

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

    private String vorname;

    public String getVorname() {
        return vorname;
    }

    public void setVorname(String vorname) {
        this.vorname = vorname;
    }
}
```



PERSON
NAME : VARCHAR(255)
VORNAME : VARCHAR(255)

JPA: Insert

Person

```
@Entity
public class Person {
    @Id
    private String name;

    public String getName() {
        return name;
    }

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

    private String vorname;

    public String getVorname() {
        return vorname;
    }

    public void setVorname(String vorname) {
        this.vorname = vorname;
    }
}
```

Insert

```
EntityManager em = JpaUtil.getEntityManager();

Person person = new Person();
person.setName("Duck");
person.setVorname("Donald");

em.getTransaction().begin();
em.getTransaction().persist(person);
em.getTransaction().commit();
```

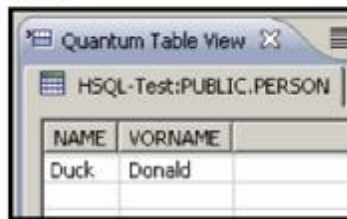
Entity-Objekt erzeugen

Objekt speichern

Hibernate-Log

```
insert into Person (vorname, name) values (?, ?)
```

DB



NAME	VORNAME
Duck	Donald

JPA: Update

Person

```
@Entity
public class Person {
    @Id
    private String name;

    public String getName() {
        return name;
    }

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

    private String vorname;

    public String getVorname() {
        return vorname;
    }

    public void setVorname(String vorname) {
        this.vorname = vorname;
    }
}
```

Update

```
EntityManager em = JpaUtil.getEntityManager();
Person person = em.find(Person.class, "Duck");
person.setVorname("Dagobert");
em.getTransaction().begin();
em.getTransaction().commit();
```

Objekt anhand
Primärschlüssel laden

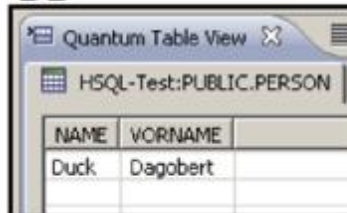
Setze neuen Vornamen

Entity-Objekt speichern,
em.persist(...) ist unnötig

Hibernate-Log

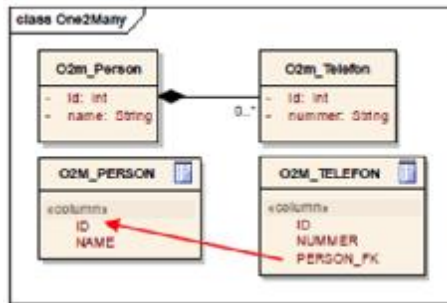
```
select person0_name ... where person0_name=?
update Person set vorname=? where name=?
```

DB



NAME	VORNAME
Duck	Dagobert

JPA: Beziehungen unidirektional, ohne zusätzliche Mapping Tabelle



```

Person
@OneToMany(cascade = CascadeType.ALL)
@JoinColumn(name = "person_fk")
public List<O2m_Telefon> getTelefonNummern() {
    if (telefonNummern == null) {
        telefonNummern = new ArrayList<O2m_Telefon> ();
    }
    return telefonNummern;
}
    
```

Hier angeben, dass es sich um eine 1:n-Beziehung handelt.

```

Telefon
@Id
@GeneratedValue
public Long getId() {
    return id;
}
    
```

Testcode - Insert

```

EntityManager em = JpaUtil.getEntityManager();

O2m_Person person = new O2m_Person();
person.setVorname("Daisy");
person.setName("Duck");

O2m_Telefon telefon = new O2m_Telefon();
telefon.setNummer("0123-23456");
telefon.setBemerkung("privat");
person.getTelefonNummern().add(telefon);

telefon = new O2m_Telefon();
telefon.setNummer("0177-23456");
telefon.setBemerkung("mobil");
person.getTelefonNummern().add(telefon);

em.getTransaction().begin();
em.persist(person);
em.getTransaction().commit();
    
```

Spalte für den Fremdschlüssel. Ohne `@JoinColumn` wird automatisch eine separate Tabelle für die Beziehung benutzt.

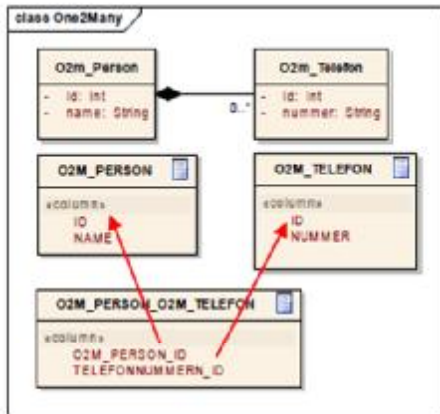
Hibernate-Log HSQLDB

```

insert into O2m_Person (id, name, ...)
call identity()
insert into O2m_Telefon (bemerkung, ...)
insert into O2m_Telefon (bemerkung, ...)
update O2m_Telefon set person_fk=? where nummer=?
update O2m_Telefon set person_fk=? where nummer=?
    
```

Quelle: Alexander Kunkel, <http://www.kunkelgmbh.de/jpa/JPA%20mit%20Hibernate.pdf>

JPA: Beziehungen unidirektional, mit zusätzlicher Mapping Tabelle



Person

```

@OneToMany(cascade = CascadeType.ALL)
public List<O2m_Telefon> getTelefonNummern() {
    if (telefonNummern == null) {
        telefonNummern = new ArrayList<O2m_Telefon>();
    }
    return telefonNummern;
}
    
```

Hier angeben, dass es sich um eine 1:n-Beziehung handelt.

Telefon

```

@Id
@GeneratedValue
public Long getId() {
    return id;
}
    
```

Testcode - Insert

```

EntityManager em = JpaUtil.getEntityManager();

O2m_Person person = new O2m_Person();
person.setVorname("Daisy");
person.setName("Duck");

O2m_Telefon telefon = new O2m_Telefon();
telefon.setNummer("0123-23456");
telefon.setBemerkung("privat");
person.getTelefonNummern().add(telefon);

telefon = new O2m_Telefon();
telefon.setNummer("0177-23456");
telefon.setBemerkung("mobil");
person.getTelefonNummern().add(telefon);

em.getTransaction().begin();
em.persist(person);
em.getTransaction().commit();
    
```

Hibernate-Log

```

insert into O2m_Person (id, name, vorname) values ...
call identity()
insert into O2m_Telefon (id, nummer, bemerkung) ...
call identity()
insert into O2m_Telefon (id, nummer, bemerkung) ...
call identity()
insert into O2m_Person_O2m_Telefon (O2m_Person_id, ...
insert into O2m_Person_O2m_Telefon (O2m_Person_id, ...
    
```

Quelle: Alexander Kunkel,
<http://www.kunkelgmbh.de/jpa/JPA%20mit%20Hibernate.pdf>

Zusammenfassung:

Vor- und Nachteile des O/R Mappings

- Clean OO design. Hiding the relational model specifics lets the object model be more cleanly analyzed and applied.
 - Productivity. Simpler code as the object model is free from persistence constraints. Developers can navigate object hierarchies, etc.
 - Separation of concerns and specialization. Let the DB people worry about DB structure and the Object people worry about their OO models.
 - Time savings. The O/R mapping layer saves you from writing the code to persist and retrieve objects. O/R mapping tool vendors claim 20-30% reduction in the code that needs to be written. Writing less code also means less testing.
 - "Needless Repetition" deadly sin (a.k.a. DRY--"Don't Repeat Yourself").
 - Writing these mapping files is a daunting task
 - Query performance
 - Limited query capabilities OR
 - Performance problems
- Zumindest Caches, Lazy Initialization sind erforderlich