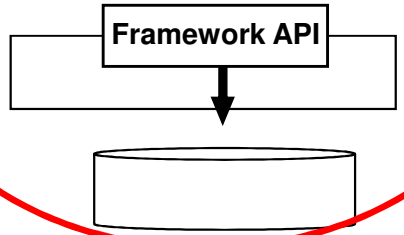
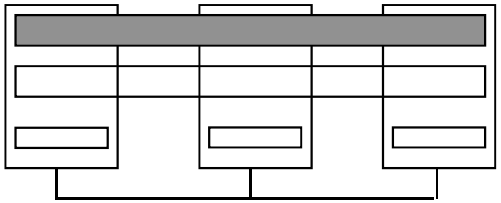
Software Engineering III – Wo sind wir?

Softwarearchitekturen

Softwarearchitekturen: Begriffe & Fallbeispiel Persistenz-Framework

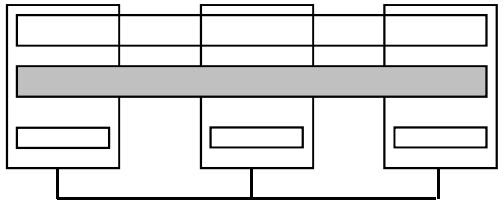


Verteilte Softwarearchitekturen



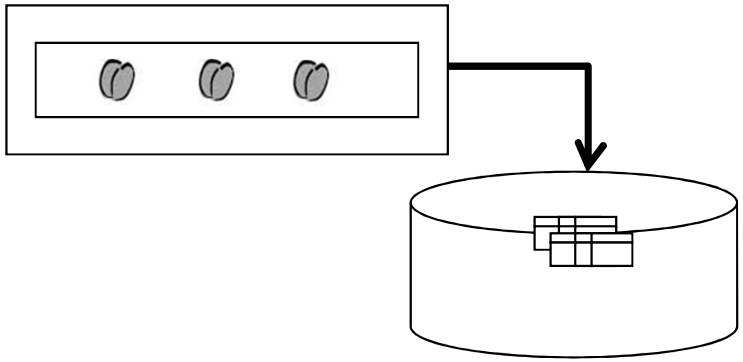
Middleware/Verteilung

Sockets, RMI, MoM/JMS Web Services



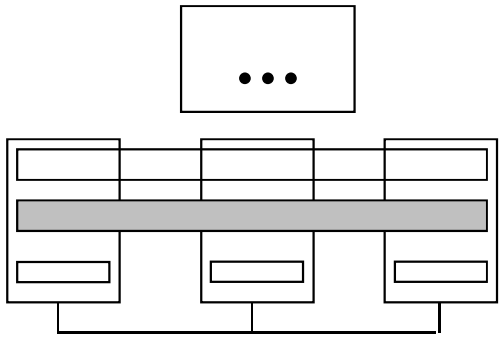
Enterprise Java

Java EE (EJB, …)



Verteilung – weitere Konzepte

Weitere Middleware



Gliederung

1. Softwarearchitektur

2. Praktische Fallstudie: Softwarearchitektur für ein Persistenz-Framework

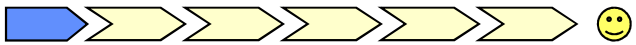
   2.1 Entwurfsziele der Persistenzschicht

   2.2 Anwendungsentwicklung mit dem Framework

   2.3 Framework-Architektur

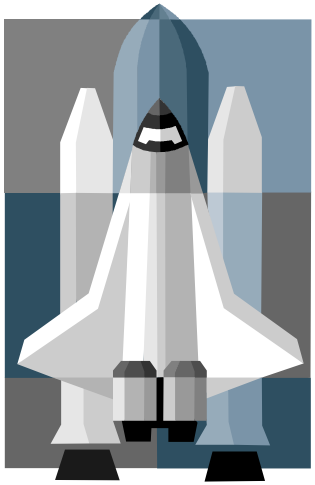
   2.4 Framework-Implementierung (Ausschnitte)

   2.5 Bewertung des Frameworks



Frage

- **Welche Persistenz-Frameworks habe ich schon genutzt?**



  - **JDBC**
  - **Hibernate / JPA**
  - **Talend ESB/SOA DB-Adapter**
  - **(Serialisierung von Objekten)**

Zentrale Aufgaben der Persistenzschicht

- **Verwaltung der fachlichen Objekte** der Applikationsschicht **in DBMS**
- bei relationalem DBMS, **O/R-Mapping** (object to relational mapping)
  - Abbildung von Objekt-Attributen auf Tabellenspalten

Dienste aus Sicht der Applikationsschicht

1. Erzeugung von Laufzeitobjekten aus in DB befindlichen Daten

(a) Objektidentität,
(b) Laden von Objekten,
(d) Beziehungen zwischen Objekten

2. Speicherung von Laufzeitobjekten in DB;

(c) Abspeichern von Objekten,
(e) Transaktionen

(a) Objektidentität

- **Objekte müssen eindeutig identifiziert werden können**

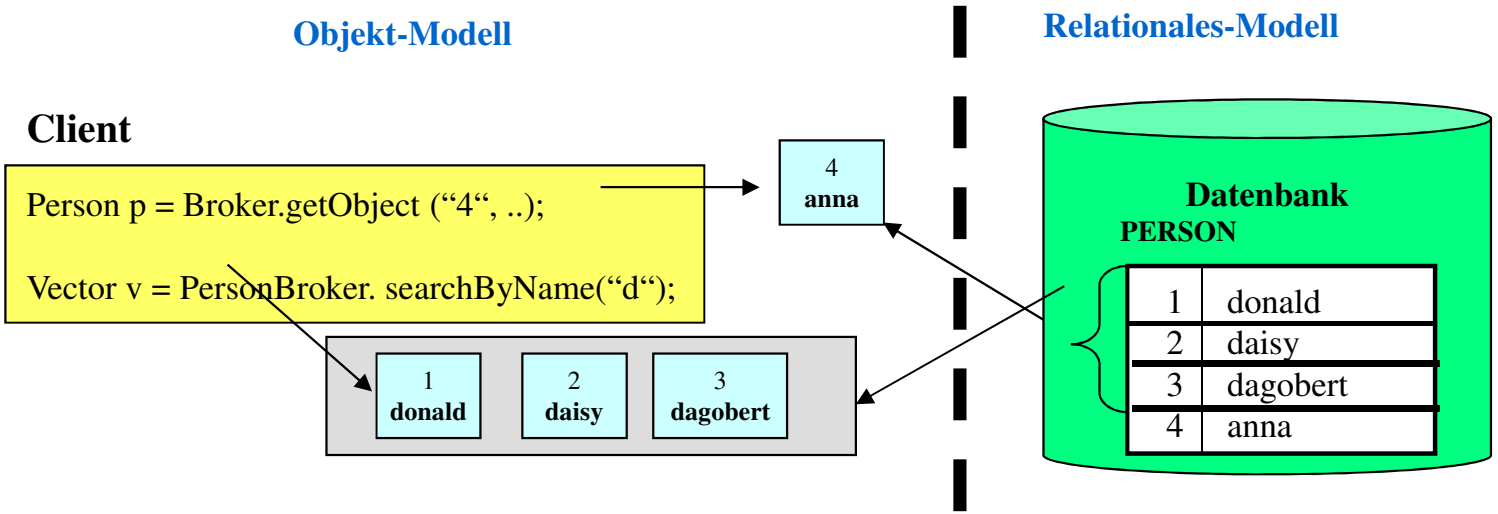
- **Oid**
  - ist systemgenerierter, globaler und eindeutiger Bezeichner
  - dient auch als Primärschlüssel, darf deshalb nicht verändert werden
  - Attribut des Laufzeitobjekts; muss bei Objekterzeugung erzeugt werden

- **Anforderung**
  - dabei soll transparent bleiben, ob Objekt bereits im Laufzeitsystem existiert oder aus DB geladen werden muss

(b) Laden von Objekten

- Varianten für **Suchanfragen**
  - über **Oid**, z.B. `Person p = (Person) Broker.getObject("4711");`
  - über Suchkriterien
    - über **Attributwerte:** `Vector result = PersonDBBroker.findEntries("Sch*");`
    - über **Beziehungen**: `Vector adresses = AdressDBBroker.findByPerson(tom);`

- **Persistenzschicht erzeugt Objekte** mittels Konstruktoren oder **Fabrikmethoden** und füllt diese mit Daten aus DB (inkl. Typkonvertierung)



(c) Abspeichern von Objekten (1 / 2)

- Fallunterscheidung
  - **Insert** – Einfügen eines neuen Objektes
    sobald neues Objekt in Anwendungsschicht erzeugt, müssen seine Daten in DB eingefügt werden
  - **Update** – Ändern eines Objektes
    Änderung in Anwendungsschicht in DB nachziehen
  - Delete – Löschen eines Objektes
- **Synchronisationszustände** beschreiben Verhältnis zwischen Laufzeitobjekt und Datenbankinhalt

SYNCHRONISATIONS-ZUSTÄNDE		
Zustand	**Definition**	**DB-Aktion**
New	das Laufzeitobjekt befindet sich noch nicht in der Datenbank	**INSERT**
Clean	Zustand von Laufzeitobjekt und Datenbank sind identisch	**---**
Dirty	Zustand von Laufzeitobjekt und Datenbank sind unterschiedlich	**UPDATE**
Delete	das Laufzeitobjekt soll in der Datenbank gelöscht werden	**DELETE**

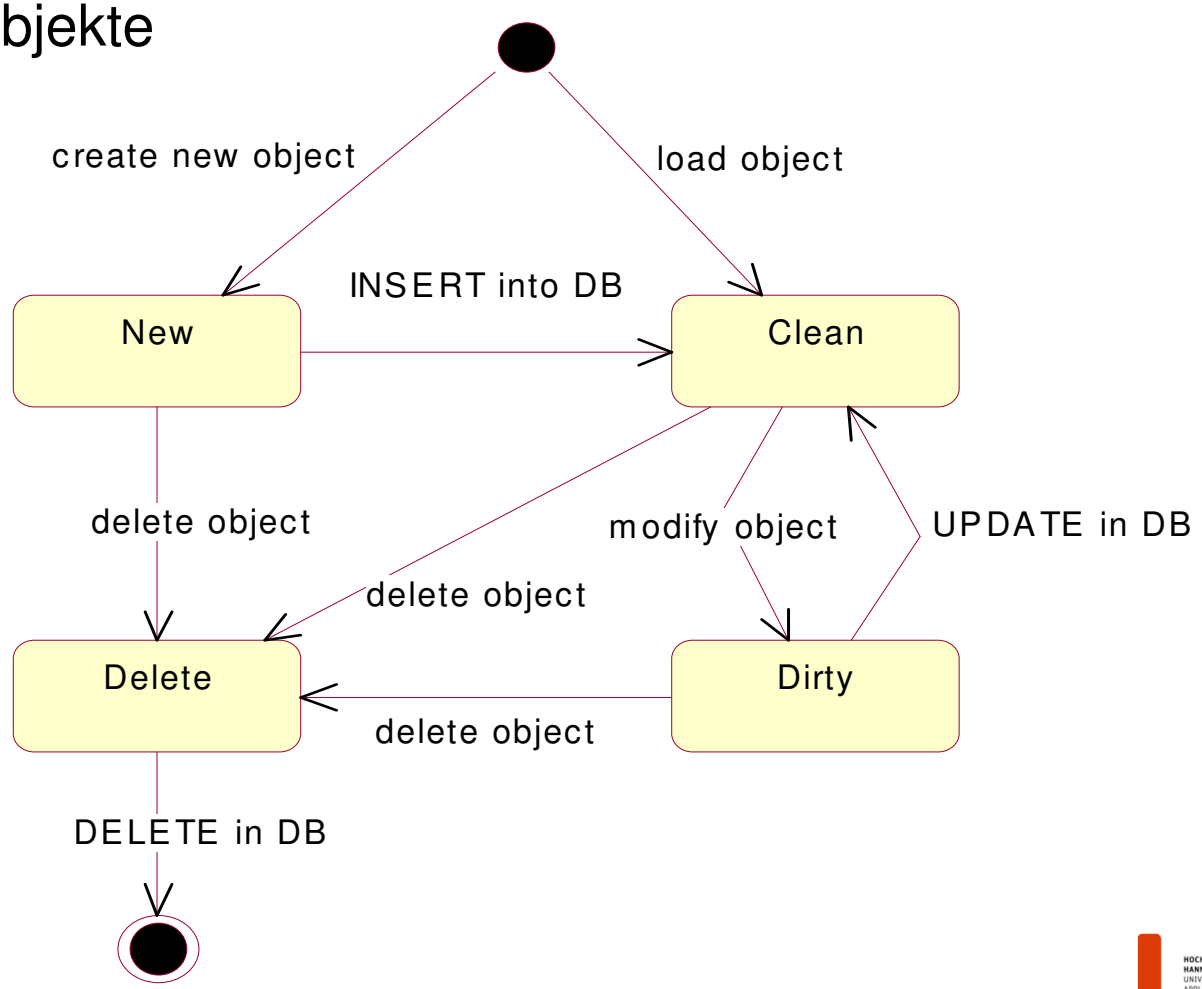
Abspeichern von Objekten (2 / 2)

- **Lebenszyklus** persistenter Objekte
  - Objekte können entweder

    neu erzeugt (Zustand new)

    oder

    aus DB geladen werden (Zustand clean)



(d) Verwaltung von Beziehungen

- Persistenzschicht sorgt für Abbildung:
  Objektreferenzen ⇔ Fremdschlüssel in DB

- **Laden von in Beziehung stehenden Objekten**
  - normale get()-Methoden
  - Loading on Demand: erst bei Aufruf von get() werden Objekte geladen

- **Ändern von Beziehungen**
  - normale set()-Methoden (bzw. add() bei mehrwertigen Beziehungen)
  - ggf. Objekte vorher laden

```
public class Person {
  private Vector mAddresses;
  public Vector getAddresses () {
    if (mAddresses == null)
        mAddresses = this.loadAddresses();
    return mAddresses;
  }
  public void addAddress(Address address){
    getAddresses();
    mAddress.add(address);
} ...
```

(e) Transaktionen

- Transaktionen können nur **in Applikationsschicht festgelegt** werden
  - komplexe Geschäftsprozesse ändern mehr als ein Objekt
  - wissen welche Daten zusammengehören und wann DB-Änderung notwendig ist

- Persistenzschicht muss folgenden **Ablauf** garantieren
  - für alle zur Transaktion zugehörigen Objekte wird DB-Befehl durchgeführt
  - falls alle Befehle erfolgreich sind, so wird die Transaktion mit **commit** abgeschlossen, andernfalls wird ein **rollback** ausgeführt
  - Fehlschlagen der Transaktion löst Fehlermeldung aus

Anforderungen an die Architektur (1)

1. Wartbarkeit und Erweiterbarkeit

- generell: Änderungen sollen nur lokale Wirkung haben

- **Redundanzfreiheit:**
  - Abbildung Objekte auf DB-Strukturen (O/R-Mapping) nur **einmal** spezifiziert
  - Ablauf eines DB-Zugriffs nur **einmal** implementiert
    - DB-Verbindung herstellen
    - Daten auslesen und in Laufzeitobjekt transformieren
    - Durchführen einer Transaktion (ggfls. rollback)
    - beteiligte Ressourcen freigeben
    - Verwaltung von Connections, Statements, Nachschauen im Cache, ...

Anforderungen an die Architektur (2)

2. Entkoppelung der Schichten

- **Applikationsschicht enthält nur fachliche Klassen**
  - Trennung zwischen fachlichen Klassen und DB-Informationen
  - Änderungen des DB-Schemas haben keinen Einfluss auf Applikationsschicht
  - kein Code für technische Realisierung des DB-Zugriffs,
    z.B. keine JDBC-Klassen, **kein SQL-Code**

- **Persistenz-Framework enthält ausschließlich technische Klassen**
  - kompletter technischer Code für DB-Zugriffe,
    z.B. gesamter SQL-Code
  - Code für den DB-Zugriff redundanzfrei an nur einer Stelle

Übersicht: Framework vs. Bibliothek

A. Funktions- oder Klassenbibliotheken

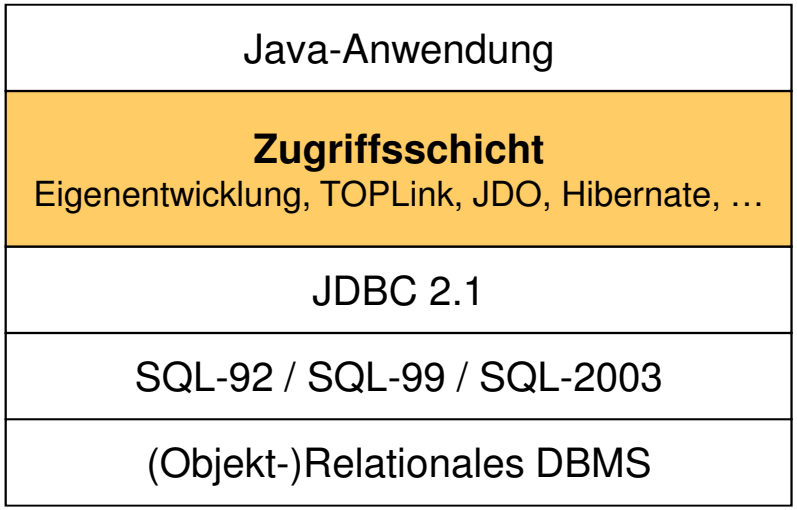
- Idee: Menge von Funktionen oder Methoden für spezielle Aufgaben
- erweitern Programmiersprachen um mächtige Befehle
- keine (vorgegebene) Unterstützung zu ihrer Verwendung
  z.B. Reihenfolge der Aufrufe, Struktur der Anwendung

B. Frameworks

- Idee: Gerüst für Standardabläufe
  - Menge von Komponenten (i.d.R. Klassen) mit definierte Kooperationsverhalten
- Komponenten (Hot Spots) mit Einschubmethoden können hinzugefügt bzw. ausgetauscht werden
  - nutzt Schablonenmethoden-Muster (template pattern)
- Einschubmethoden werden durch Framework aufgerufen
  - *"Don't call the framework, the framework calls you"*

Schritte des Anwendungsentwicklers

1. **Framework-API** in Applikationsschicht verwenden
   - Daten aus DB zu lesen
   - Daten ab in DB speichern (inkl. Transaktionen)
2. **Entitätsklassen** der Anwendungsschicht anpassen
3. **Mapping** zwischen fachlichen Klassen und DB-Tabellen definieren
4. komplexer **Suchmethoden** implementieren



→ **Details werden im Dokument auf Server beschrieben!!**

Demo

Kleine Anwendungsdemo: Persistenz-Framework

1. Framework-API verwenden (1)

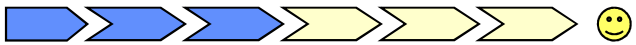
- **Laden von Objekten**
  - Suche nach Primärschlüssel
    - 2. Parameter ist die Klasse des Objekts
  - über Suchkriterien
    - nach bestimmten Attributen oder in Beziehung stehenden Objekten
    - für die Klasse `MyClass (z.B. Person)` bietet `MyClassDbBroker` (z.B. PersonDbBroker) die entsprechende Suchmethoden
  - der Zugriff auf DB kann Exception werfen!

```
try{
        Person tom = (Person) Broker.getObject("1", Person.class);
        Vector pers = PersonDbBroker.exemplar().findByName("Tom");
        Vector ads = AdresseDbBroker.exemplar().findByOwner(tom);
}catch (Exception e) {...}
```

1. Framework-API verwenden (2)

- **Speichern von Objekten**
  - separate Klasse `Transaction` zum Setzen der Transaktionsklammer

```
try{
    Adresse a = new Adresse("...");
    Person p = PersonDbBroker.findByName("Meyer");

    ...        // verändern der Objekte

    Transaction t1 = new Transaction();
    t1.addObject(p);
    t1.addObject(a);
    t1.transactionCommit();
}catch (Exception e) {...}
```



1. Framework-API verwenden (3)

- komplettes Beispiel

```
try{
   // Java-Objekt aus der DB holen
   Person p1 = (Person) Broker.getObject("1", Person.class);
   p1.setName("Daisy", "Duck");
   // (neues) Java-Laufzeit-Objekt erzeugen
   Person p2 = new Person();        //setzt automatisch die Oid
   p2.setName("Donald", "Duck");
   // Objekte innerhalb einer Transaktion in die DB schreiben
   Transaction t1 = new Transaction();
   t1.addObject(p1); t1.addObject(p2);
   t1.transactionCommit();
   // viele Objekte aus der DB erzeugen
   Vector v = PersonDbBroker.exemplar().searchPerson("Duc*");
   for (int i=0; i < v.size(); i++){
      System.out.println(  (Person)v.get(i)  );
   }
}catch (Exception e) {e.printStackTrace();}
```

2. Entitätsklassen anpassen

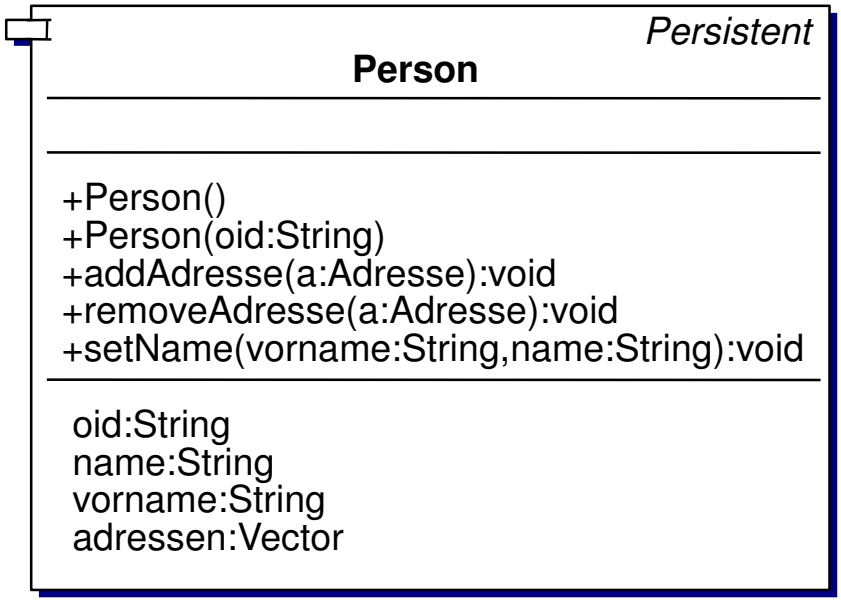
- jede persistente Klasse muss `Persistent` implementieren
- Default-Konstruktor **registriert neues Laufzeitobjekt** im Framework

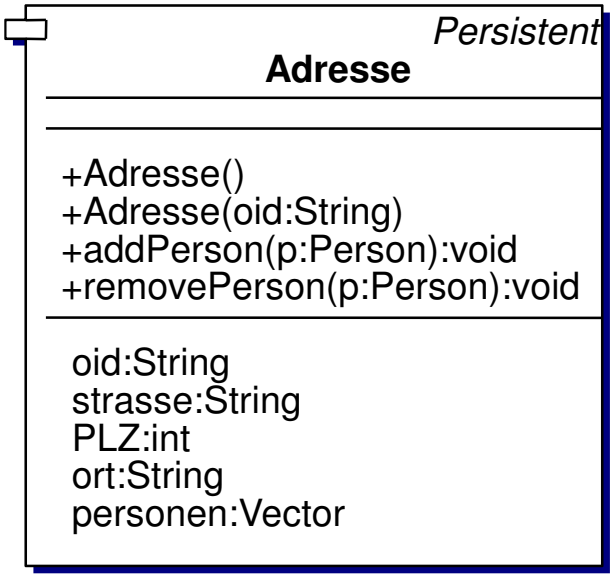
```
import de.fhhannover.inform.persistence.*;

public class Person implements Persistent {
   protected String oid ; // Oid als String

   public void setOid(String oid)  {this.oid = oid; }
   public String getOid() { return this.oid; }

   // zwei Konstruktoren:
   public Person() {Broker.register(this);
   }

   public Person(String oid) {this.oid = oid;
   }
.... (Datenattribute & set- / get-Methoden)
```

3. + 4. (ObjectRelational) Mapping und Suchfunktionen

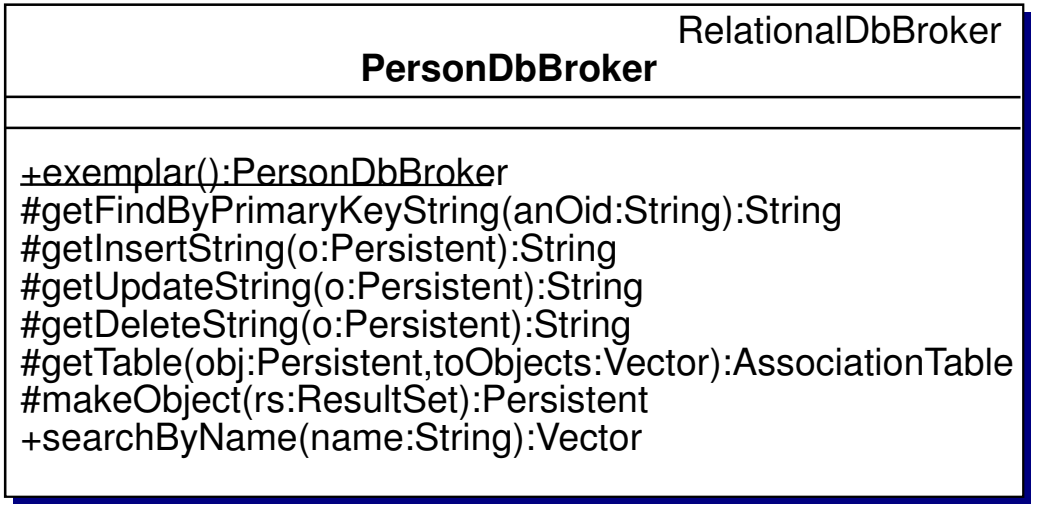
```
public class PersonDbBroker extends RelationalDbBroker {
   protected String getInsertString (Persistent o){
        Person p = (Person) o;
        return "insert into Person(pers_id, name, vorname, geburtstag)"
              + " values("" + p.getOid()  + "' , '" + p.getName() + "', '“
               + p.getVorname() + "', {d '" + p.getGeburtstag()  + "'} )";
   }
 // . . .   Weitere Mapping-Methoden (update, . . .)
   protected Persistent makeObject(ResultSet rs) throws SQLException {
        Person person = new Person( rs.getString("pers_id") );
        person.setVorname( rs.getString("vorname") );
        person.setName( rs.getString("name") );
        person.setGeburtstag(rs.getDate("geburtstag"));
        return person;
   }
   //Suchmethoden
   protected String getFindByPrimaryKeyString(String anOid) {
        return  "select * from Person where pers_id ='" +anOid +"'";
   }
   public Vector findByName(String name) throws SQLException {
        String sql = "select * from Person where name like '“ + name + "%'";
        return getObjectsFromSelect(sql);
   }
```

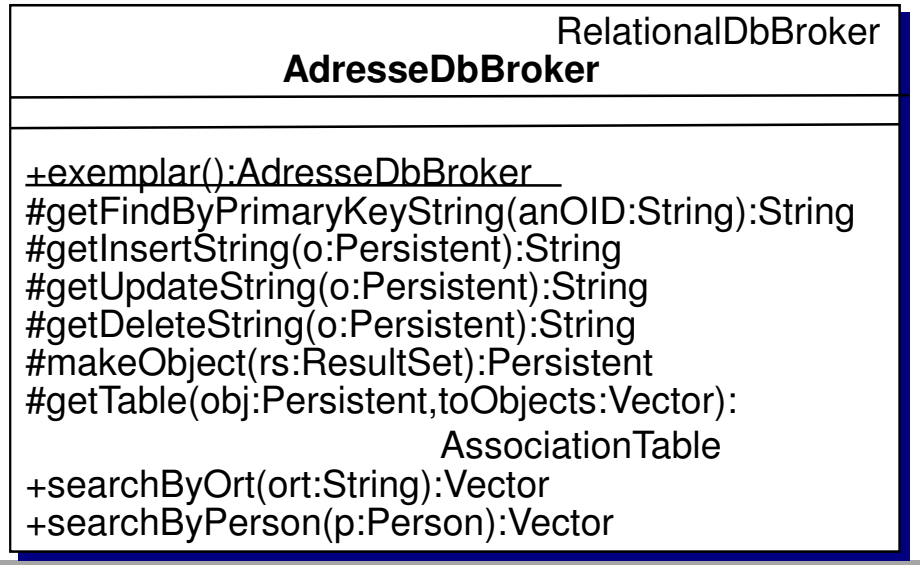
Überblick – vom Anwendungswickler zu implementierende Klassen





Applikationsschicht





Persistenzschicht

Nutzung Persistenz-Framework – Fragenblock

Diskutieren Sie 3-4 Minuten die folgende Frage mit Ihrer Nachbarin / Ihrem Nachbarn:

- **Welche Schritte muss ein/e Anwendungsentwickler/in bei der Arbeit mit dem Persistenz-Framework durchführen?**

Gliederung

1. Softwarearchitektur

2. Praktische Fallstudie: Softwarearchitektur für ein Persistenz-Framework

   2.1 Entwurfsziele der Persistenzschicht

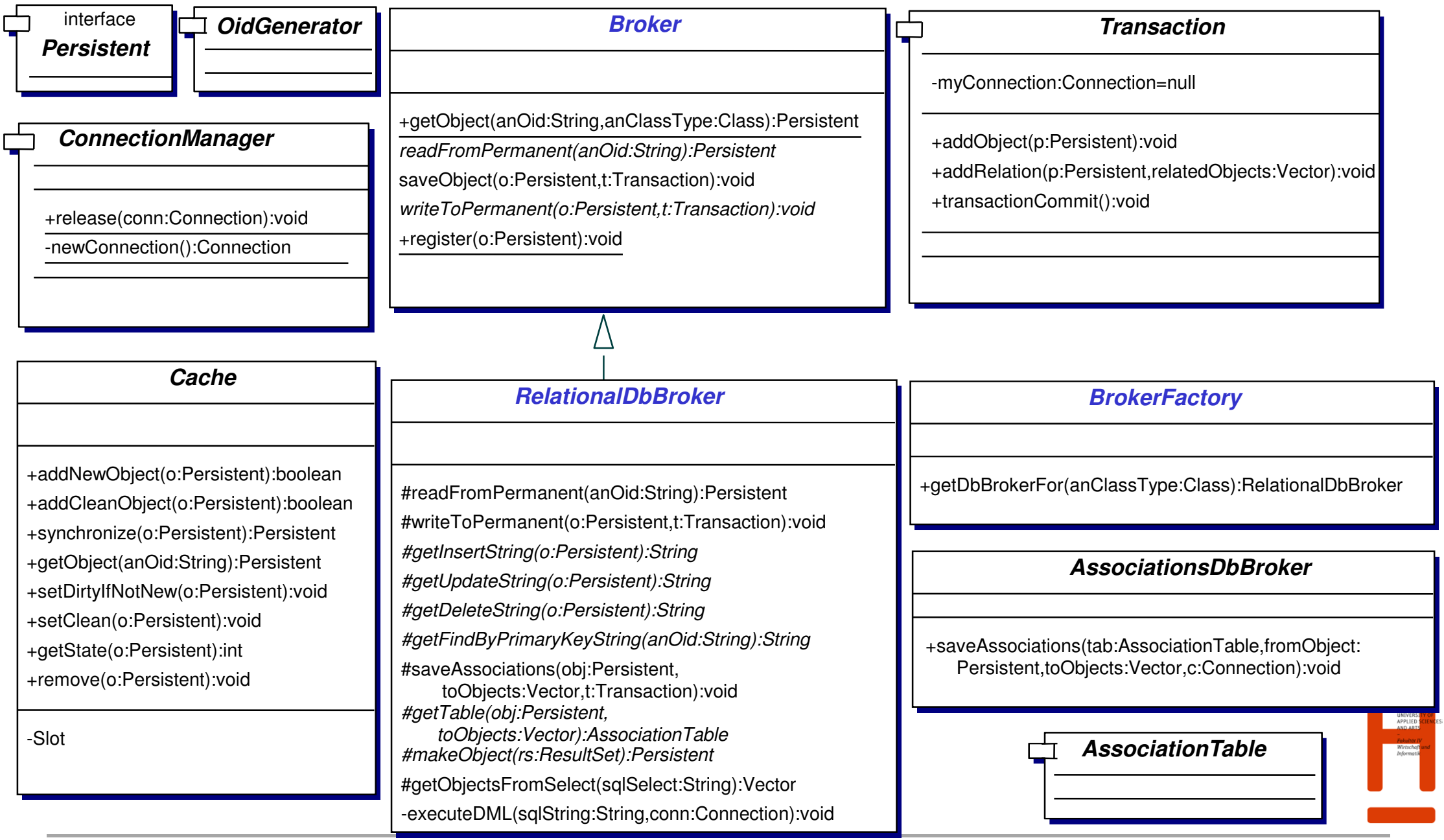
   2.2 Anwendungsentwicklung mit dem Framework

   2.3 Framework-Architektur - intern

   2.4 Framework-Implementierung (Ausschnitte)

   2.5 Bewertung des Frameworks

2.3 Framework-Architektur intern – Überblick



A. Schnittstelle aller persistenten Klassen

- alle persistenten Klassen müssen das Interface `Persistent` implementieren
- Framework setzt voraus, dass alle Objekt eine Property `oid` vom Typ String haben

```
public interface Persistent  {
    public String getOid();
      public void setOid(String oid);
  }
```

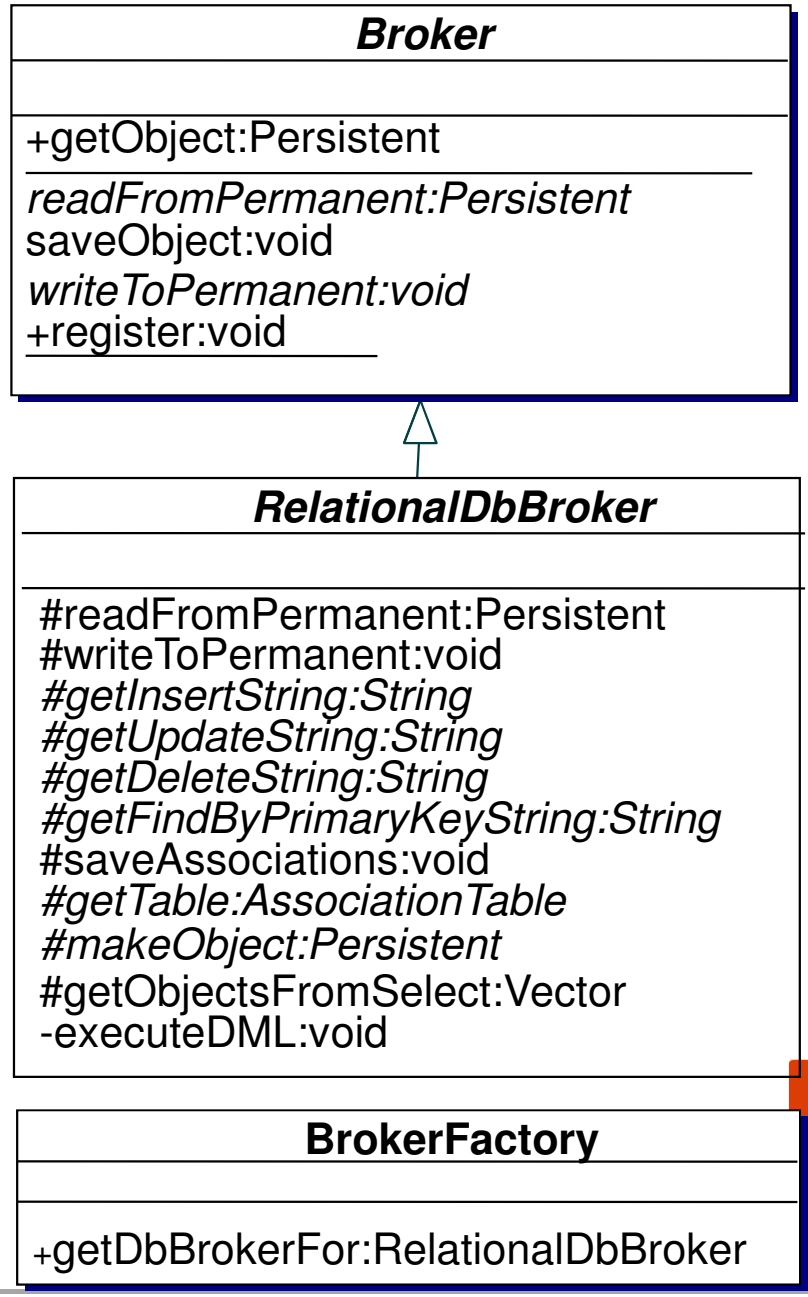
B. Verwaltung des Objekt-Lebenszyklus

Cache
+addNewObject:boolean +addCleanObject:boolean +getObject:Persistent +setDirtyIfNotNew:void +setClean:void +setDelete:void +getState:int +getState:int +remove:void
-Slot

- jedes Objekt unterliegt einem definierten Lebenszyklus
- Synchronisationszustand beschreibt Verhältnis zwischen Objekt und RDBMS (**CLEAN, NEW, DIRTY, DELETE**)
- Klasse `Cache`
  - verwaltet (in Hash-Tabelle) die Synchronisations - zustände aller Objekte
    - Methoden zum Einfügen von Objekten
    - Methoden zum Setzen und Lesen der Zustände
  - zentraler Zugriffspunkt für alle aus RDBMS geladenen Objekte
    - erst im Cache prüfen, ob Objekt bereits geladen wurde

C. Laden und Speichern von Objekten

- Kernaufgabe des Frameworks
- **`Broker`**
  - Schnittstelle für Standardabläufe zum **Laden**/**Speichern** eines einzelnen Objekts
- **`RelationalDbBroker`**
  - Implementierung der Standardabläufe zum Laden/Speichern eines einzelnen Objekts
- **`PersonDbBroker`**
  - Implementierung des O/R-Mapping für fachliche Klasse
    - SQL-Befehle
    - Erzeugung fachlicher Objekte aus DB
- **`BrokerFactory`**
  - Fabrikmethode, um XyDbBroker zu erzeugen



D. Transaktionsverarbeitung und Connection-Pooling

- **Transaction**
  - Klasse zur Spezifikation von Transaktionen
    addObject(..), addRelation(..)
  - Durchführung einer Transaktion:
    transactionCommit(..)

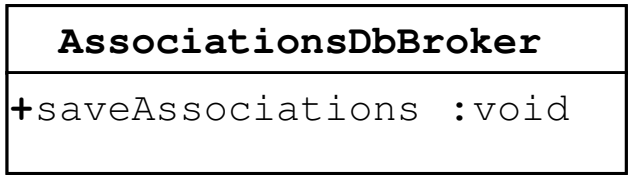
Transaction
-transactionObjects :Vector
+addObject :void +addRelation :void +transactionCommit :void

- **Connection-Manager**
  - verwaltet Datenbank-Verbindungen
  - nicht mehr benötigte Connection-Objekte werden nicht freigegeben, sondern in einem Pool für späteren Gebrauch vorgehalten

ConnectionManager
-connectionPool :Vector
-newConnection :Connection +release :void

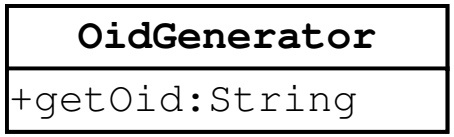
E. Persistenz von Relationen

- `AssociationDbBroker`
  - kann m:n-Beziehungen in eigenen Beziehungstabllen verwalten
- `AssociationsTable`
  - Hilfsklasse, die zu einer (m:n)-Beziehungstabelle gehört



F. Erzeugung von Oids

- `OidGenerator`
  - generiert Oids



Gliederung

1. Softwarearchitektur

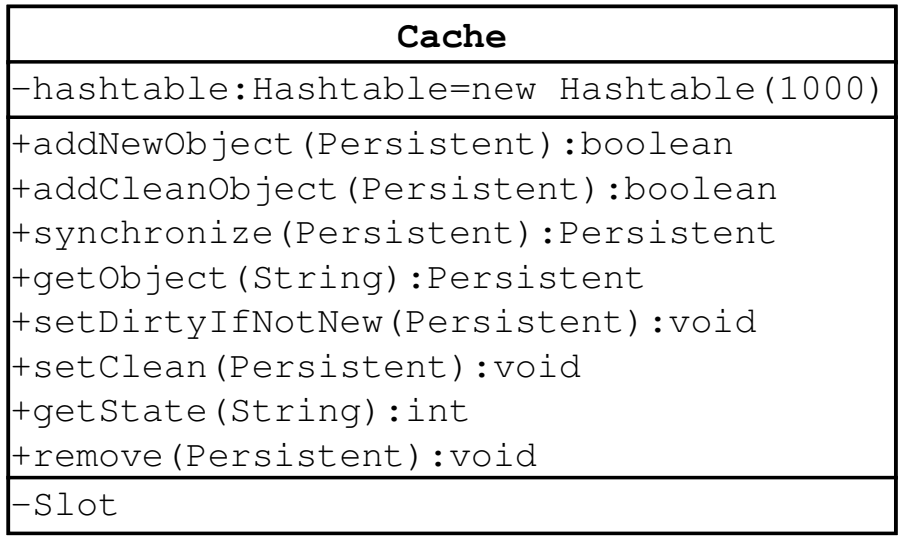
2. Praktische Fallstudie: Softwarearchitektur für ein Persistenz-Framework
   - 2.1 Entwurfsziele der Persistenzschicht
   - 2.2 Anwendungsentwicklung mit dem Framework
   - 2.3 Framework-Architektur
   - 2.4 Framework-Implementierung (Ausschnitte)
   - 2.5 Bewertung des Frameworks

2.4.1 Persistence-Interface

- legt Schnittstelle für Zugriff auf OID vom Typ String fest (`getOid(), setOid()`)

2.4.2 Verwaltung des Objekt-Lebenszyklus

- innerere Klasse **`Slot`** verwaltet für jedes Objekt folgende (Meta-)Informationen
  - oid
  - Referenz auf das Laufzeit-Objekt
  - Zustand des Objektes (CLEAN, NEW, DIRTY, DELETE)
- Klasse **`Cache`** verwaltet mit Hilfe einer Hash-Tabelle alle **`Slot`**-Objekte
  - Methoden zum Hinzufügen/Entfernen von Objekten
  - Methoden zum Setzen/Lesen des Slot-Zustandes



Cache-Verwaltung

- verschiedene Implementierungsvarianten
  - Cache für alle Klassen ⇔ jede Klasse hat eigenen Cache
  - lokaler Cache auf jedem Client ⇔ globaler Cache für alle Clients auf Server

⇒ **Diskussion der Vor- und Nachteile**

- Garbage Collection des Cache
  - Problem: auch fachlich nicht benötigte Objekte werden im Cache referenziert, d.h. die normale Garbage Collection greift nicht
  - Cache kann sehr groß werden
  - eigene Garbage Collection hier nicht implementiert

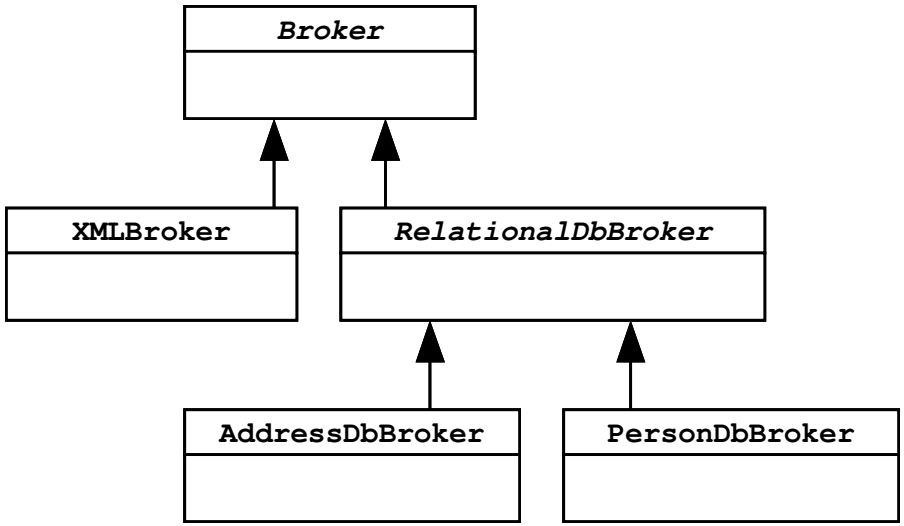
Cache-Varianten - Diskussion der Vor- und Nachteile

Diskutieren Sie mit Ihren Nachbarn

- Cache pro Klasse (vs. gemeinsamer Cache für alle Klassen)
  - +
  - -
- Gemeinsamer Cache auf Server
  - +
  - -
- Mehrere verteilte Caches
  - +
  - -

Broker

- abstrakte Klasse mit 2 Methoden zum **Laden** und **Speichern** von Objekten in DB
  - `getObject(...): Persistent`
  - `saveObject(...)`
- Methode zum Registrieren neuer Objekte
  - `register (Persistent p)`
- unterstützt Cache-Handling
- Delegation an Unterklassen
  - keine Kenntnis, wo die Objekte abgelegt sind (RDBMS, XML etc.)



RelationalDbBroker (oder XMLBroker oder ...)

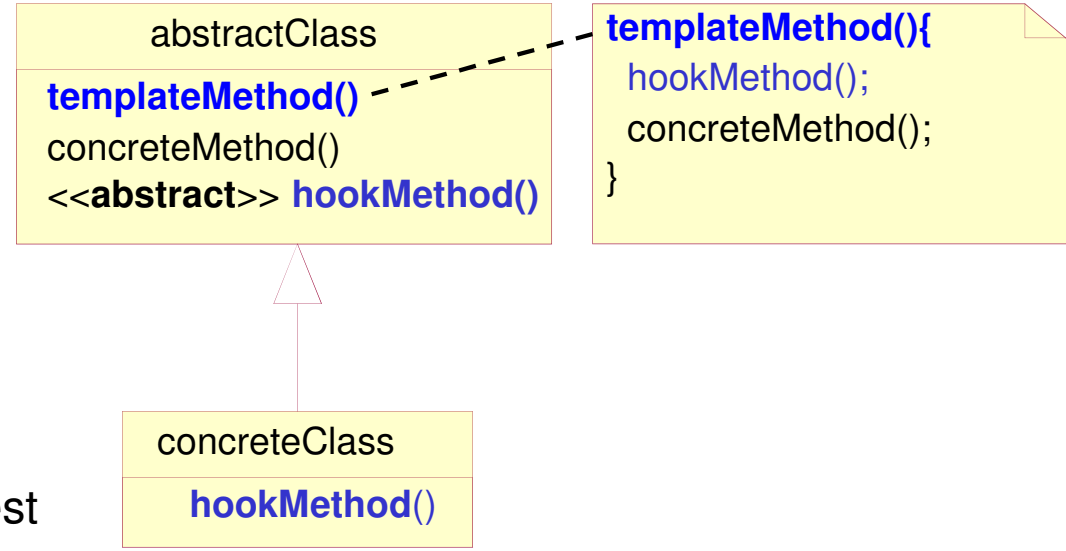
- kennt die eingesetzte Technologie (RDBMS,...)
- implementiert die Standardabläufe
- verwaltet Connections, Statements,...

PersonDbBroker (oder ...)

- enthält die SQL-Befehle der fachlichen Klasse
- nur hier Wissen über das DB-Schema

Schablonen-Muster (Template Method)

- das Basis-Pattern für Frameworks
  - zur Festlegung von Standardabläufen
  - Alternative: Strategie-Pattern



- **abstrakte Oberklasse**
  - implementiert **Template-Methoden**
    - legen Abläufe für alle Unterklassen fest
    - rufen **Hook-Methoden** auf
    - Hook-Methode sind abstrakt u. erst in Unterklasse implementiert
    - bleiben unverändert: **FrozenSpots**

- **konkrete Unterklassen**
  - erbt in Template-Methoden definierte Abläufe
    - Hook-Methoden implementieren Unterklassen-spezifisches Verhalten (**HotSpots**)

Verantwortung der Klasse Broker

Broker
+getObject(String, Class):Persistent +register(Persistent):void saveObject(Persistent, Transaction):void *readFromPermanent(String):Persistent* *writeToPermanent(Persistent, Transaction):void*

- bietet Schnittstelle zum **Laden und Speichern eines Objekts** aus/in Datenbank
  - `getObject(String anOid): Persistent`
  - `saveObject(Persistent obj)`
- unterstützt Cache-Handling, u.a. eine Methode zum Registrieren neuer Objekte
  - `register(Persistent obj)`
- Delegation spezieller Aktionen (SQL-Zugriff) an Unterklassen per Hook-Methoden:
  - `writeToPermanent()` und `readFromPermanent()`
- hat kein Wissen darüber, wie Persistenz erreicht wird (**technologie-unabhängig**)

Arbeitsweise der Klasse Broker (1)

- holt Objekt mit Parameter `anOid` (falls dort vorhanden) aus Cache
- sonst: aus der DB holen
  - Erzeugung des für Klasse passenden DbBroker-Objektes (bspw. `PersonDbBroker`)
  - Aufruf der Hook-Methode `readFromPermanent(anOid)`
  - Einfügen in Cache mit Zustand CLEAN

```
public abstract class Broker {
       public static Persistent getObject(String anOid, Class anClassType)
                                                    throws Exception{
        Cache myCache = Cache.exemplar();
        Persistent obj = myCache.getObject(anOid);
        if (obj != null){return obj;}  // Object ist bereits im Cache
        else {
           Broker broker =

                     BrokerFactory.exemplar().getDbBrokerFor(anClassType);
          obj = broker.readFromPermanent(anOid);  // Hook-Methode
          if (obj != null){  myCache.addCleanObject(obj);}
          return obj;
        }
}   }
```

Arbeitsweise der Klasse Broker (2)

- **delegiert** den Aufruf **an** in **Unterklasse** implementierte Hook-Methode, dort erfolgt der eigentliche Datenbank-Durchgriff per JDBC

```
void saveObject(Persistent o, Transaction t) throws Exception{
     this.writeToPermanent(o, t);
}
```

- abstrakte Hook-Methoden in Broker

```
abstract Persistent readFromPermanent(String anOid)
        throws Exception;
abstract void writeToPermanent(Persistent o,Transaction t)
throws Exception;
```

- erzeugt eine oid und setzt diese für das übergebene Objekt
- registriert darüber hinaus das Objekt im Cache
- Aufruf in jedem Default-Konstruktor: `Person(){Broker.register(this);}`

```
static public void register(Persistent o){
                o.setOid( OidGenerator.getOid() );
                Cache.exemplar().addNewObject(o);
}
```

Broker-Architektur – RelationalDbBroker

RelationalDbBroker
#readFromPermanent(String):Persistent #writeToPermanent(Persistent, Transaction):void #getObjectsFromSelect(String):Vector -executeDML(String, Connection):void *#getFindByPrimaryKeyString(String):String* *#getInsertString(Persistent):String* *#getUpdateString(Persistent):String* *#getDeleteString(Persistent):String* *#makeObject(ResultSet):Persistent*

- realisiert technischen Zugriff auf RDBMS
  - Erzeugung eines **JDBC-Connection-Objekts** und Erzeugung eines **Statement-Objekts**
  - **Ausführen** eines **SQL**-Befehls, der vom XYDbBroker abgefragt wird
  - ggf. **Erzeugung** eines **Java-Objekts**
  - Behandlung von Exceptions und Freigabe von Ressourcen
- kein anwendungsspezifisches Wissen (über DB-Schema)

Struktur der Klasse RelationalDbBroker

- **implementierte Template-Methoden**
  - `readFromPermanent(oid): Persistent`
  - `writeToPermanent(Persistent, Transaction)`
  - `getObjectsFromSelect(sql): Vector` erzeugt Java-Objekte

- **verwendete Hook-Methoden**
  - für SQL-Strings `getInsertString(),...`
  - um aus ResultSet Objekt zu erzeugen: `makeObject(Resultset rs)`

- **Helper-Funktionen**
  - `saveAssociation()` behandelt Many-Many-Beziehungen
  - `executeDML(String sql, Connection c)` führt DML-Befehl aus

2.4.3 Broker-Architektur – RelationalDbBroker

```
abstract class RelationalDbBroker extends Broker{
    protected Persistent readFromPermanent (String anOid)
                                              throws SQLException {
        // Hook-Methode liefert Select-String
        String selectString = getFindByPrimaryKeyString(anOid);

        // Aufruf der Helper-Methode
        return (Persistent)
                getObjectsFromSelect(selectString).firstElement();
    }
    ...
```


2.4.3 Broker-Architektur – RelationalDbBroker

```
protected Vector getObjectsFromSelect (String sqlSelect)
                                              throws SQLException{
      ...
      try{
         conn = ConnectionManager.getConnection();
          stmt = conn.createStatement();
          rs = stmt.executeQuery(sqlSelect);
          while(rs.next()){
              p = makeObject(rs); // Hook-Methode
             //gleiche mit Cache ab
              p = (Persistent) Cache.exemplar().synchronize(p);
              result.add(p);
          }
      }catch (SQLException e){throw e;       // hochwerfen
      }finally{
         if (stmt != null) stmt.close();
         if (conn != null) ConnectionManager.release(conn);
      }
      return result;
   }
```

- Hook-Methode: macht aus einem Datensatz des ResultSets ein Java-Objekt !

```
abstract protected Persistent makeObject(ResultSet rs)throws SQLException;
```

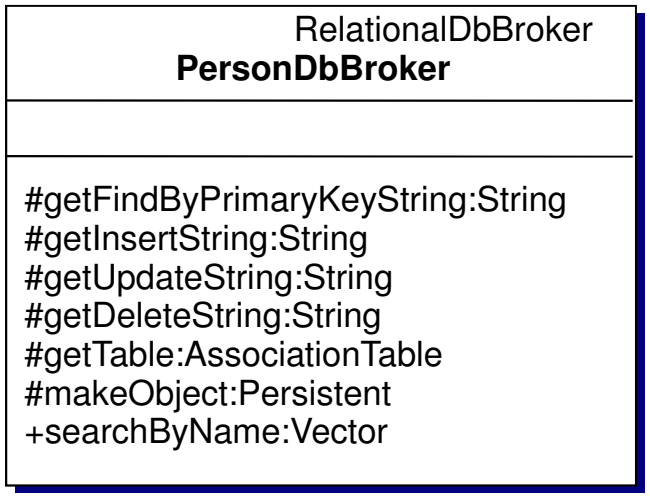
```
protected void writeToPermanent(Persistent o, Transaction t)
                                                throws SQLException {

      // besorge Connection der Transaction
      Connection conn = t.getConnection();
      String sqlString = null;
      ...
      if(myCache.getState(o) == Cache.NEW){
          // Hook-Methode
          sqlString = this.getInsertString(o);
       }else if(myCache.getState(o) == Cache.DIRTY){
          //Hook-Methode
          sqlString = this.getUpdateString(o);
       }
       this.executeDML(sqlString, conn);//  Helper-Methode
}
```

2.4.3 Broker-Architektur – RelationalDbBroker

```
private void executeDML(String sqlString, Connection conn)
    throws SQLException {
        Statement stmt = null;
        try{
            stmt = conn.createStatement();
            stmt.executeUpdate(sqlString);
        } catch (SQLException e){
            throw e;    // hochwerfen
        }finally{
            if (stmt != null) stmt.close();
        }
}
```

Broker-Architektur – PersonDbBroker

- ist **Singleton** und genügt der **Namenskonvention**
  (Konvention: “meinName“DbBroker – hier also: PersonDbBroker)
- kennt die Tabelle für die **fachliche Klasse**
  (inkl. ihrer Beziehungen),
  d.h. hier findet das **O/R-Mapping** statt



- implementiert die **Hook-Methoden** für das Framework
  - liefert die benötigten SQL-Strings `getInsertString(Persistent p),...`
  - erzeugt aus ResultSet ein Java-Objekt `makeObject(Resultset rs)`

- implementiert die speziellen **Such-Methoden**
  - nutzt dabei die Helper-Methode
    `getObjectsFromSelect(sql): Vector` der Oberklasse `RelationalDbBroker`

2.4.3 Broker-Architektur – PersonDbBroker

```
class PersonDbBroker extends RelationalDbBroker{
  // Implementierung der Hook-Methoden
      protected String getFindByPrimaryKeyString(String anOid) {
        return  "select * from Person where pers_id = '" + anOid + "'";
      }
      protected String getInsertString (Persistent o){
         Person p = (Person) o;
         return "insert into Person (pers_id, name, vorname) values("
              + "'" + p.getOid()+ "' , '" + p.getName()+ "', '"
              + p.getVorname())";
      }
      protected String getUpdateString (Persistent o) { //analog }
      protected String getDeleteString (Persistent o) { //analog }

      protected Persistent makeObject(ResultSet rs) throws SQLException{
                // Konstrukor benutzen, der keine oid erzeugt!
                Person person = new Person( rs.getString("pers_id") );
                person.setVorname( rs.getString("vorname") );
                person.setName( rs.getString("name") );
                return person;
    } }
```

2.4.3 Broker-Architektur – PersonDbBroker

```
// implementieren speziellerer Suchmethoden
public Vector searchByName(String name) throws SQLException {
  String select = "select * from Person where name like  '"
       + name + "%'";

  // benutzt Helper-Methode aus RelationalDbBroker
  return getObjectsFromSelect(select);
}
```

BrokerFactory
+getDbBrokerFor(Class):RelationalDbBroker

BrokerFactory

- erzeugt zu Klassenobjekt den zugehörigen Broker (hier: **mit Reflection**-Mechanismus)
- ist Singleton

```
import java.lang.reflect.*;
public class BrokerFactory {
....  // Singleton
public RelationalBroker getDbBrokerFor(Class aClassType)
   throws Exception{
   // erzeugt Klassenobjekt vom Typ xyDbBroker
   Class brokerClass = Class.forName(aClassType.getName() +
                "DbBroker");
   Method exemplar = brokerClass.getMethod("exemplar",null);
   return (RelationalDbBroker)exemplar.invoke(null,null);
}
```

- **`Class.forName(String name):`** erzeugt ein Class-Objekt der Klasse mit Meta-Information
- **`getMethod("exemplar",null):`** erzeugt Methode exemplar ohne Parameter (Class[ ] == null)
- **`exemplar.invoke(null,null):`** ruft Methode auf für Objekt null (statische Methode), keine Parameter (Class[ ] == null)

Persistenz-Framework intern - Fragenblock

- **Diskutieren Sie 5 Minuten die folgenden Fragestellungen mit Ihrer Nachbarin / Ihrem Nachbarn:**

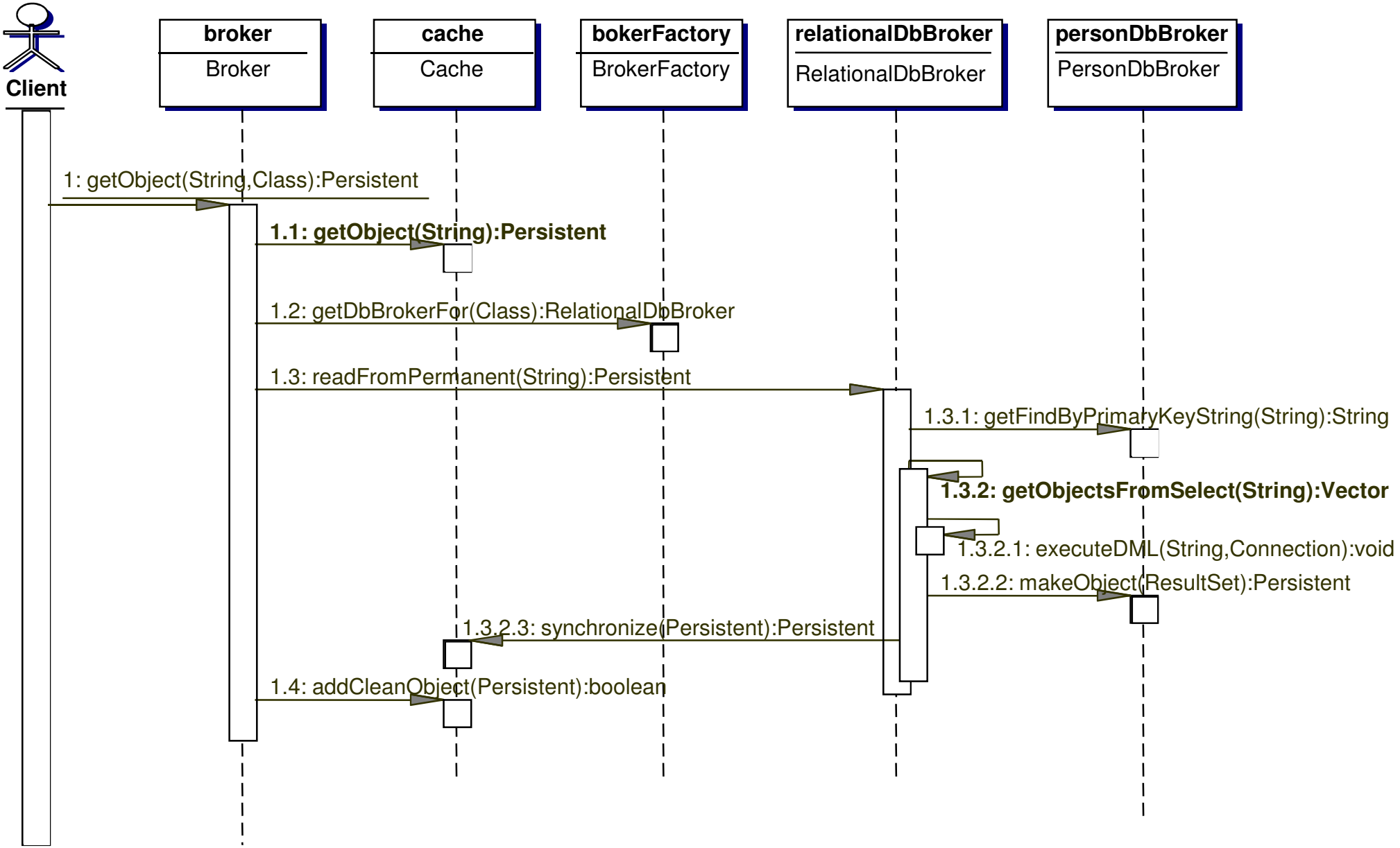
- **Wozu dient das „Template Pattern“ und wie funktioniert es?**

- **Was ist und wie funktioniert**

  die Broker-Hierarchie im Persistenz-Framework?

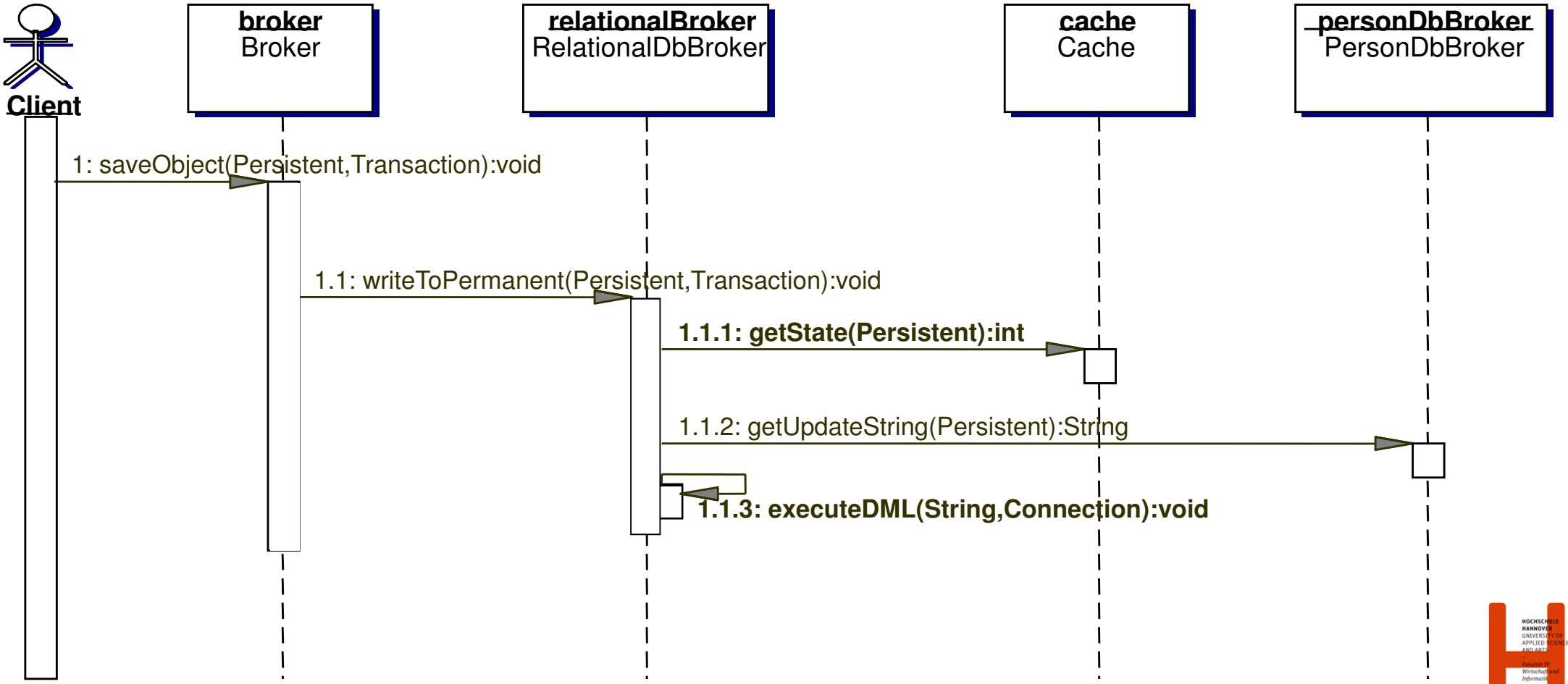
2.4.3 Broker-Architektur – Abläufe: „Spezielles Objekt suchen“

```
Person p1 = (Person) Broker.getObject("4711", Person.class);
```



2.4.3 Broker-Architektur – Abläufe: „Objekt sichern“

```
Broker.saveObject("4711", Person.class);
```



Oid-Generator

OidGenerator
+getOid:String

- nutzt entweder Oracle-Sequence oder
- UID aus dem Package java.rmi.server erzeugt

```
public class OidGenerator {
    static public String getOid() {
          if (useSequence)
              return getOidBySequence();
          java.rmi.server.UID uid = new java.rmi.server.UID();
          return uid.toString();
      }

    private static String getOidBySequence(){
        ...
       ResultSet rs = stmt.executeQuery(
             "select  myId.nextval from dual");
      rs.next();
      return rs.getString("nextval");
       ...
}
```

Connection Pooling

ConnectionManager
-connectionPool :Vector
-newConnection :Connection +release :void

- Erzeugen einerDatenbank-Connections ist zeitaufwendig (Auffinden des Servers, Authentifizierung, ...)
- Idee
  - eine einmal geöffnete DB-Connection wird nicht per `close()` geschlossen, sondern in einem **Pool** (Vector) aufbewahrt
  - **`getConnection(): Connection`**
    - holt (falls möglich) eine Connection aus dem Pool
    - ggf. wird eine neue Connection erzeugt
  - **`releaseConnection(Connection conn): void`**
    - gibt eine Connection frei, d.h. nimmt sie in den Pool auf
    - sie kann dann für andere Transaktionen genutzt werden

2.4.5 Connection-Pooling

```
public class ConnectionManager {
   private static ....;
   // dient dem Connection-Pooling:
   private static Vector connectionPool = new Vector();
   static {  // die Initialisierung findet nur einmal statt
        b = ResourceBundle.getBundle("connect");
        uid    = b.getString("uid");     pwd    = b.getString("pwd");
        driver = b.getString("driver");  dburl  = b.getString("dburl");
        try{  Class driverClass = Class.forName(driver);     }
        catch (Exception e){ e.printStckTrace(); }
   }
   static public Connection getConnection() throws SQLException  {
     if( connectionPool.isEmpty() ){
           connectionPool.addElement( newConnection() );}
     return (Connection) connectionPool.remove(0);
   }
   public static void release(Connection conn){
      connectionPool.add(conn);
   }
   static private Connection newConnection() throws SQLException {
      Connection conn = DriverManager.getConnection (dburl, uid, pwd);
      conn.setAutoCommit(false);
      return conn;
}  }
```

Transaktionen

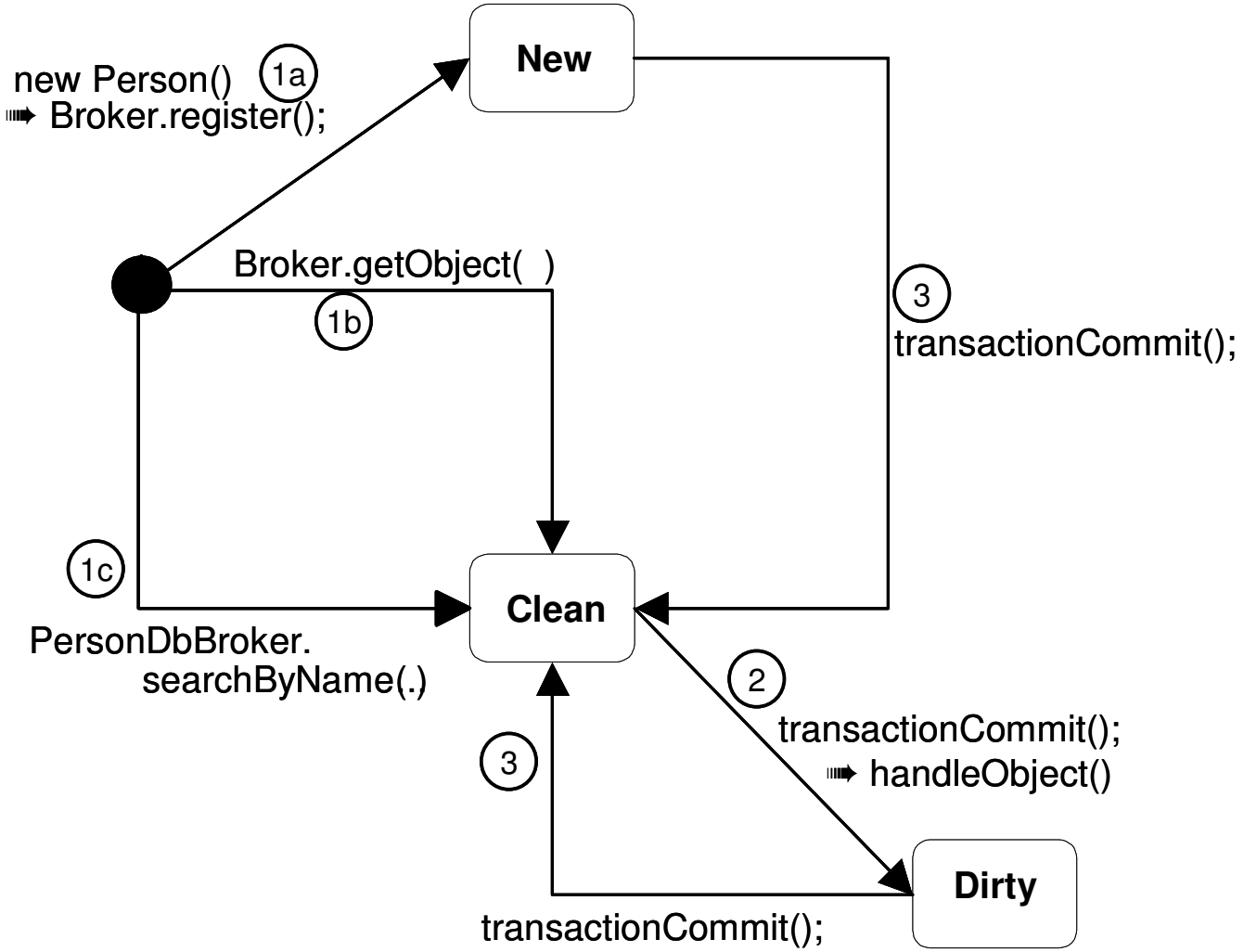
- werden immer im Kontext von fachlichen Geschäftsprozess-Methoden festgelegt!
  - nur dort ist bekannt, welche Objekte **atomar** abzuspeichern sind

```
void geheZurKasse(Warenkorb w, Kunde k) {
        k.erhoeheSchulden( w.getPreis() );
        Rechnung r = new Rechnung(w, k);
        k.addRechnung(r);
        Transaction t = newTransaction();
        t.addObject(k); t.addObject(r);
        t.addRelation(k, k.getRechnungen() );
        t.transactionCommit();

}
```

- Klasse `Transaction` realisiert Transaktionsverarbeitung
  - legtTransaktions-Kontext (`addObject(), addRelation()`) fest (Reihenfolge wichtig!!)
  - `transactionCommit( )`
    - führt für jedes Objekt der TransaktionDB-Aktion durch
    - falls alle DB-Aktionen erfolgreich ⇒ commit (sonst rollback)
    - Cache-Aktualisierung

Details → Anhang

Verwaltung der Cache-Zustände – Synchronisation mit der DB

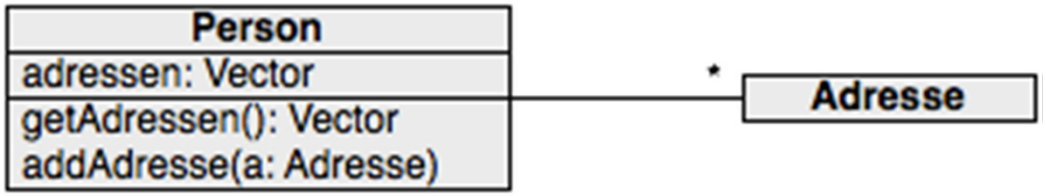


- **Sperrkonzepte**
  - **Problem:** zwei Clients laden das selbe Objekt aus der DB
    - und ändern es inkonsistent
    - das ist problemlos in vielen Anwendungen (bspw. Adressverwaltung)
    - aber nicht in allen (bspw. Flugbuchung)
  - **Lösung: optimistisches Sperren**
    - ist nicht im Framework implementiert
    - Konflikterkennung über Zeitstempel!

    1. jedes persistente Objekt obj. enthält OID (in Objekt und DB-Tabelle)
    2. beim Laden eines Objekts wird timestamp gelesen
    3. update set ... timestamp=timestamp + 1
       where ... AND timestamp = obj.timestamp
    4. wird kein Datensatz geändert, gab es Änderung durch anderen Client

Fachliche Methoden zur Verwaltung von Beziehungen

- Beziehungen durch Objektreferenzen bzw. Vektoren mit Objektreferenzen abgebildet
  - normale get- und set-Methoden für Beziehungen
  - **loading-on-demand**-Mechanismus




```
public Vector getAdressen() throws Exception{
   if (this.adressen.size() == 0)
      adressen = AdresseDbBroker.exemplar().searchByPerson(this);
   return this.adressen;
}
```

```
public void addAdresse(Adresse a){
       this.getAdressen();
       if (! this.getAdressen().contains(a) ) adressen.add(a);
       a.addPerson(this);
}
...
```

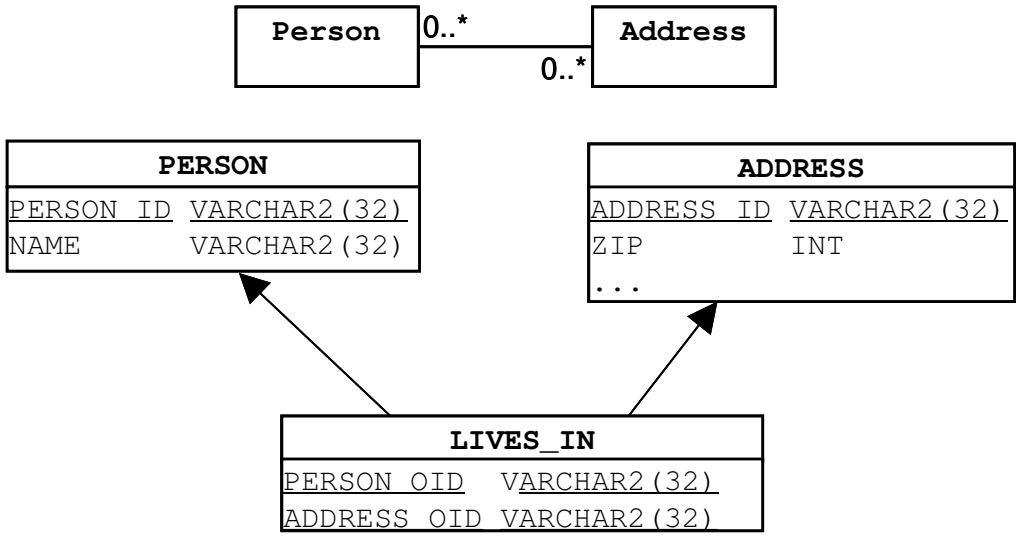
1:N-Beziehungen

- werden in DB per **Fremdschlüssel** in der Detail-Tabelle realisiert
- sind mit bisher vorgestellten Mitteln des Frameworks kein Problem
- in den Insert- und Update-Befehlen wird auch der Fremdschlüssel gesetzt
  - steht durch book.getOwner().getOid() zur Verfügung

```
protected String getUpdateString(Persistent o) {
    Book book = (Book) o;
     return "UPDATE BOOK set " +
            "title='"   + book.getTitle() +
            "',person_id='" + ((Persistent)book.getOwner()).getOid() +
            "' where book_id='" + ( (Persistent)book).getOid() + "'";
  }
```

M:N-Beziehungen

- werden in Datenbank über eigene Beziehungstabelle realisiert
- Änderung von N:M-Beziehungen nur im Rahmen von Transaktionen




```
Person p = (Person) Broker.getObject("1", Person.class);
Address a = (Address) Broker.getObject("123", Address.class);
p.addAddress(a);
Transaction t1 = new Transaction();
t1.addRelation(p.getAddresses());
t1.transactionCommit();
```

M:N Beziehungen - Implementierungsdetails → Anhang

Persistenz-Framework – Bewertung & Abschließende Bemerkungen (1)

- Beispiel für umfangreiche (lokale) Software-Architektur

- generelle Ziele erreicht
  - mächtiges API ⇒ geringer Aufwand für Entwicklung der Persistenzschicht
  - Entkopplung der fachlichen Klassen von DB-Zugriffscode
  - relativ leicht zu verstehen
  - (nur Anhang: Weitere Details – u.a. Behandlung von n:m-Beziehungen)

- Problemkreise nicht immer optimal gelöst
  - Fehlende Garbage-Collection für Cache
  - Fehlende Unterstützung von Vererbung
  - Fehlende Security-Unterstützung
  - (Anhang: Nur recht einfaches Transaktionskonzept)
  - . . .

Persistenz-Framework – Bewertung & Abschließende Bemerkungen (2)

- generelle Nachteile
  - keine Objektverteilung
  - proprietäre Schnittstelle

- Aufwand!!! Entwicklung einer Persistenzschicht erfordert mehrere Personenjahre
  - liegt oft in selben Größenordnung wie Anwendungsentwicklung
  - es ist meist sinnvoller Standardbibliotheken einzusetzen!

- Beispiele für andere Optionen (→ s.a. Datenbanksysteme 2)
  - Produkte / Frameworks: Oracle/TopLink, Hibernate, MS Linq, …
  - Standardschnittstellen: Java Persistence API, Java Data Objects, …
  - …

Literatur „Software-Architektur“

- Verwendete Quellen
  - [DunHol03]: J. Dunkel, A. Holitschke: Softwarearchitektur für die Praxis, Springer, 2003

- Weitere Quellen
  - [BrWh96]: Brown, K., Whitenack, B.: Crossing Chasms: The Static Patterns. In J. Coplien et al. Pattern Langage of Program Design, Addison Wesley, 1996
  - [HaTW95]: Hahn, W., Toeniessen, F., Wittkowski, A.: Eine objektorientierte Zugriffsschicht zu relationalen Datenbanken, Informatik-Spektrum 18, 143-151, Springer 1995
  - [Wanner95]: Waner, G.: Objektorientierte Anwendungsentwicklung mit relationalen Datenbanken, HMD, Heft 183, Hüthig, 101-119, 1995


HOCHSCHULE
HANNOVER
UNIVERSITY OF
APPLIED SCIENCES
AND ARTS
Fakultät IV
Wirtschaft und
Informatik

Anhang

Weitere Code-Fragmente

Ablaufdiagramm

Transaktionen „intern“

Beziehungen

2.4.2 CACHE-CODE Objekt-Lebenszyklus

```
public class Cache {
   private static Cache instance = null;
   private static Hashtable hashtable = new Hashtable();

   public static final int CLEAN = 1;
   public static final int DIRTY = 2;
   public static final int NEW   = 3;

   protected Cache(){  }  // verhindert direkte Erzeugung
   static Cache exemplar() {
      if (instance == null){  instance = new Cache();}
      return instance;
   }

      private class Slot{        // innere Klasse
         private int state;
         private String oid;
         private Persistent reference;
         Slot (String oid, Persistent o, int state){
             this.state = state; this.oid = oid; this.reference = o;
         }
         ...
```

2.4.2 CACHE-CODE Objekt-Lebenszyklus

- **Methoden zum Hinzufügen/ Entfernen von Objekten**

```
public boolean addNewObject(Persistent o){
        Slot slot = new Slot(o.getOid(), o, NEW);
        hashtable.put(o.getOid(), slot);
        return true;
}
public void remove(Persistent o){
        Slot slot = (Slot)hashtable.get(o.getOid() );
        if (slot != null) hashtable.remove(o.getOid() );
}
```

- **Methoden zum Setzen/ Lesen des Slot-Zustandes**

```
public void setClean(Persistent o){
        Slot slot = (Slot)hashtable.get(o.getOid() );
        if(slot != null) slot.state = CLEAN;
}
public int getState(Persistent o){
        Slot slot = (Slot)hashtable.get( o.getOid() );
        if (slot == null){    return 0;}
        else{  return slot.state;}
}
```

- **Methoden, um ein Objekt aus dem Cache zu holen**
  - synchronisiert ein aus der DB zu lesendes Objekt

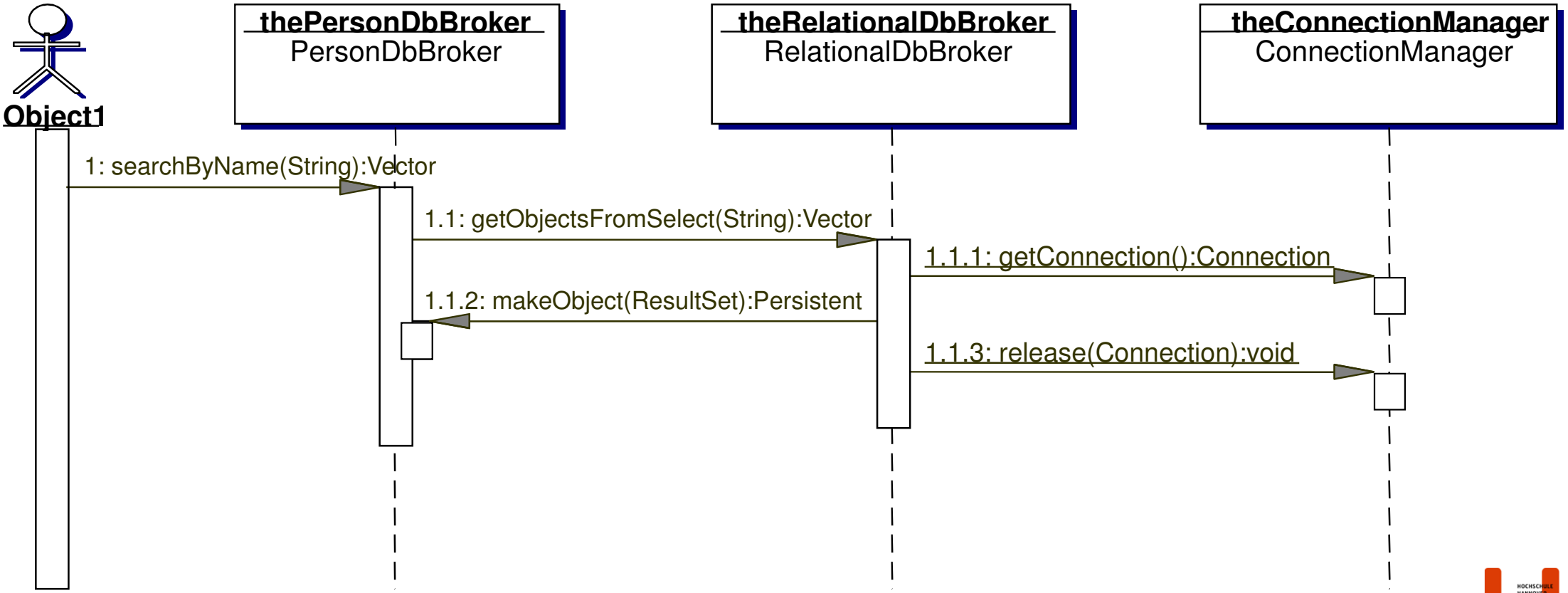
    ⇒ ein neues Objekt wird in Cache eingefügt oder

    ⇒ ein im Cache vorhandenes Objekt wird zurückgegeben

```
public Persistent synchronize(Persistent o){
       // Methode getObject() gibt Object zurück,
       //falls im Cache vorhanden
       Persistent tmp= getObject( o.getOid() );
       if(tmp != null)
          return tmp;
       addCleanObject(o);
       return o;
}
```

2.4.3 Broker-Architektur – Abläufe

```
Vector v = PersonDbBroker.exemplar().searchByName("Du");
```



Class Transaktion

- besitzt eigene Connection
- verwaltet alle zugehörigen Objekt in einem Vektor

Transaction
-transactionObjects:Vector=null
+getConnection():Connection +setConnection(Connection):void +addObject(Persistent):void +addRelation(Persistent, Vector):void +deleteObject(Persistent):void +transactionCommit():void -saveAllObjects():void -handleObject(Persistent):void -handleRelation(Relation):void

```
public class Transaction {
    private Vector transactionObjects = new Vector();
    private Cache cache = Cache.exemplar();;
    private Connection myConnection = null;

    public Transaction() throws SQLException{
        myConnection = ConnectionManager.getConnection();
}

   public void addObject(Persistent p){
      transactionObjects.addElement(p);
    }
```

Durchführung einer einer Transaktion

```
public void transactionCommit() throws Exception{
      try{
          saveAllObjects();
          myConnection.commit();    // Commit auf JDBC-Connection
          cleanCache();             // alle Objekte sind clean
      }
      catch(SQLException e1){      // Fehler bei Transaktion
          try{
            myConnection.rollback();
            throw e1;                // Exception hochreichen
          }
          catch(SQLException e)      // Fehler bei  rollback zur Db
             throw e2;
          }
      }
      finally{
        ConnectionManager.release(this.myConnection);
      }
   }
```

DB-Befehle gegen DB absetzen

`private saveAllObjects()`

- iteriert durch alle Objekte der Transaktion
- unterscheidet zwischen `Persistent` und `Relation`-Objekten und delegiert zur Weiterbehandlung

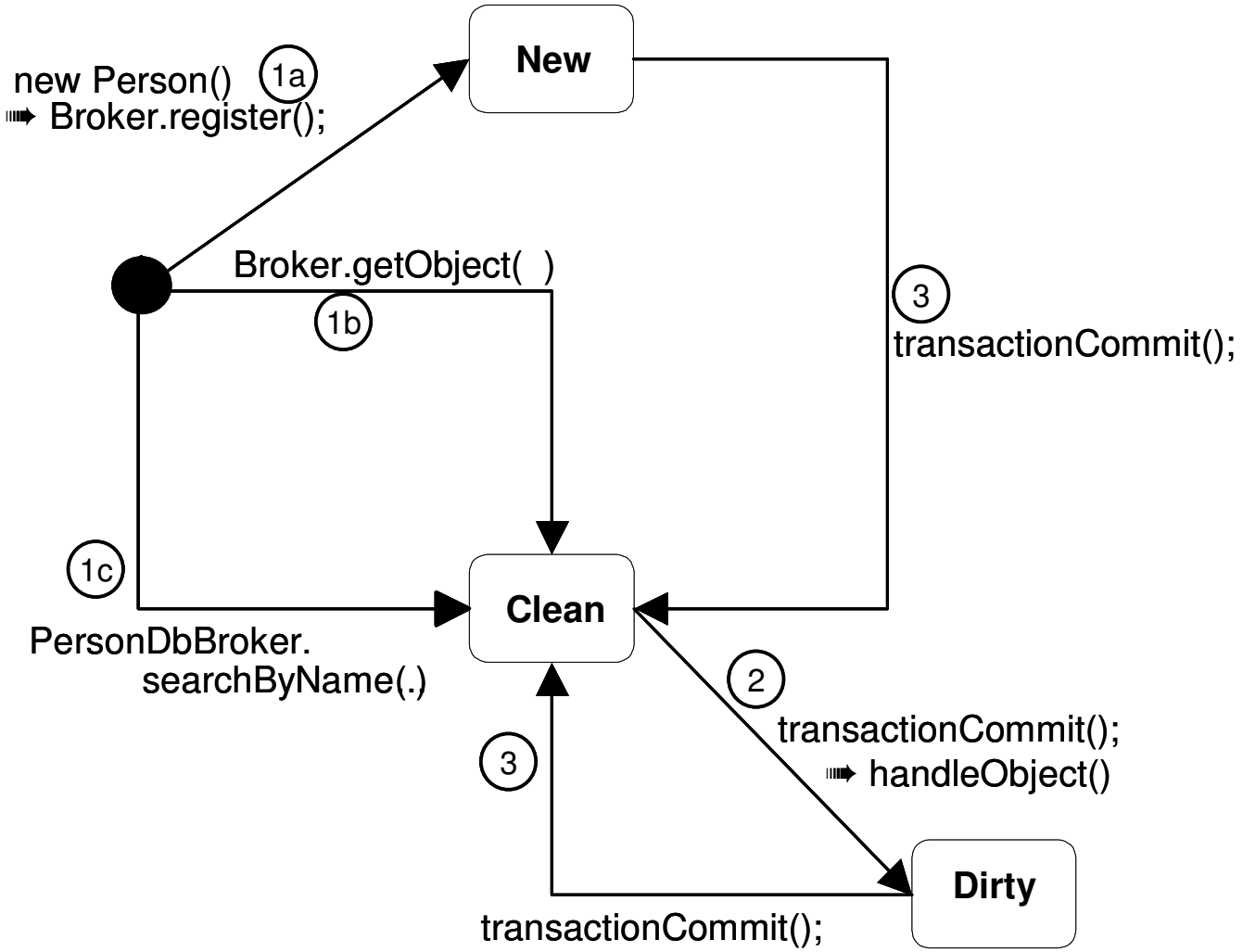
```
private void saveAllObjects() throws Exception {
   Object o;
   Enumeration objects= transactionObjects.elements();
   while (objects.hasMoreElements() ) {
      o = objects.nextElement();
      if (o instanceof Persistent)
                  handleObject((Persistent) o);
      if (o instanceof Relation)
          handleRelation((Relation) o );
   }
}
```

Hilfsmethode saveAllObjects()

- holt den passenden Broker und nutzt dessen `saveObject`-Methode
- jedes nicht-neue Objekt wird auf DIRTY gesetzt
  (befreit den Anwendugsentwickler, dies in fachlichen Methoden zu tun)

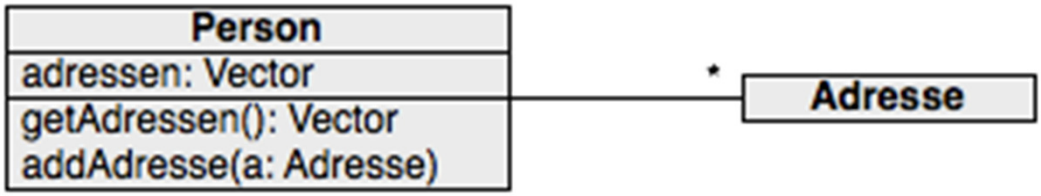
```
private void handleObject(Persistent p) throws Exception {
        cache.setDirtyIfNotNew(p);
        Broker b =
           BrokerFactory.exemplar().getDbBrokerFor(p.getClass());
        b.saveObject(p, this);
}
```

Verwaltung der Cache-Zustände – Synchronisation mit der DB



Fachliche Methoden zur Verwaltung von Beziehungen

- Beziehungen durch Objektreferenzen bzw. Vektoren mit Objektreferenzen abgebildet
  - normale get- und set-Methoden für Beziehungen
  - **loading-on-demand**-Mechanismus




```
public Vector getAdressen() throws Exception{
   if (this.adressen.size() == 0)
      adressen = AdresseDbBroker.exemplar().searchByPerson(this);
   return this.adressen;
}
```

```
public void addAdresse(Adresse a){
       this.getAdressen();
       if (! this.getAdressen().contains(a) ) adressen.add(a);
       a.addPerson(this);
}
...
```

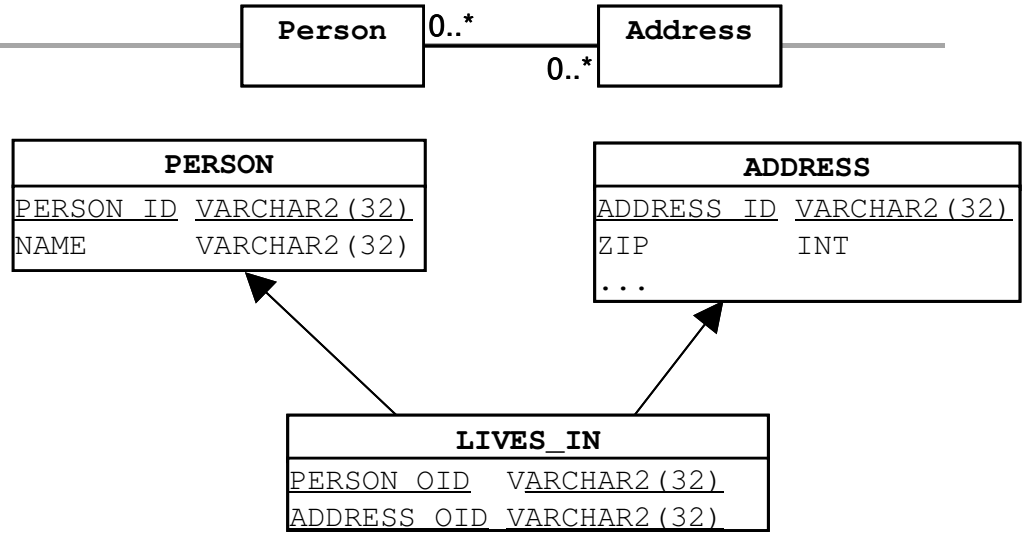
1:N-Beziehungen

- werden in DB per **Fremdschlüssel** in der Detail-Tabelle realisiert
- sind mit bisher vorgestellten Mitteln des Frameworks kein Problem
- in den Insert- und Update-Befehlen wird auch der Fremdschlüssel gesetzt
  - steht durch book.getOwner().getOid() zur Verfügung

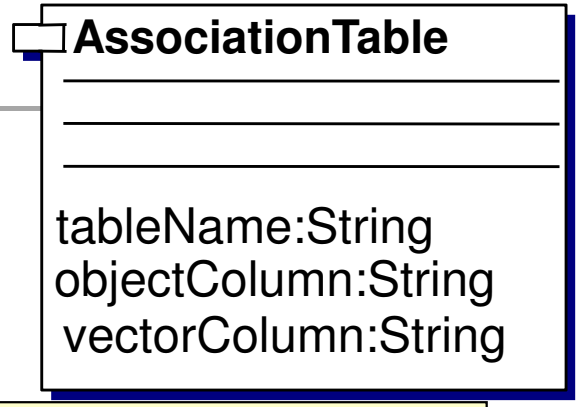
```
protected String getUpdateString(Persistent o) {
    Book book = (Book) o;
     return "UPDATE BOOK set " +
            "title='"   + book.getTitle() +
            "',person_id='" + ((Persistent)book.getOwner()).getOid() +
            "' where book_id='" + ( (Persistent)book).getOid() + "'";
  }
```

M:N-Beziehungen

- werden in Datenbank über eigene Beziehungstabelle realisiert
- Änderung vonN:M-Beziehungen erst im Rahmen von Transaktionen




```
Person p = (Person) Broker.getObject("1", Person.class);
Address a = (Address) Broker.getObject("123", Address.class);
p.addAddress(a);
Transaction t1 = new Transaction();
t1.addRelation(p.getAddresses());
t1.transactionCommit();
```



O/R-Mapping für Beziehungen

- eigener Mechanismus im Framework
- welche Tabelle und welche Spalten sind zuständig?

```
class PersonDbBroker extends RelationalDbBroker{

   protected AssociationTable getTable
      (Persistent obj, Vector toObjects){
      Person p = (Person)obj;
    // es geht um den Vector getAdressen
      if (toObjects == p.getAdressen() )
         return new AssociationTable("wohnt_in","pers_id","adr_id");

      if (toObjects == p.getAutos() )
         return new AssociationTable("fährt_in","pers_id","car_id");
      ...
      return null;
   }
}
```

Relationen in Transaktionen (1)

- Ausgangspunkt: `t.addRelation()` in `Transaction`

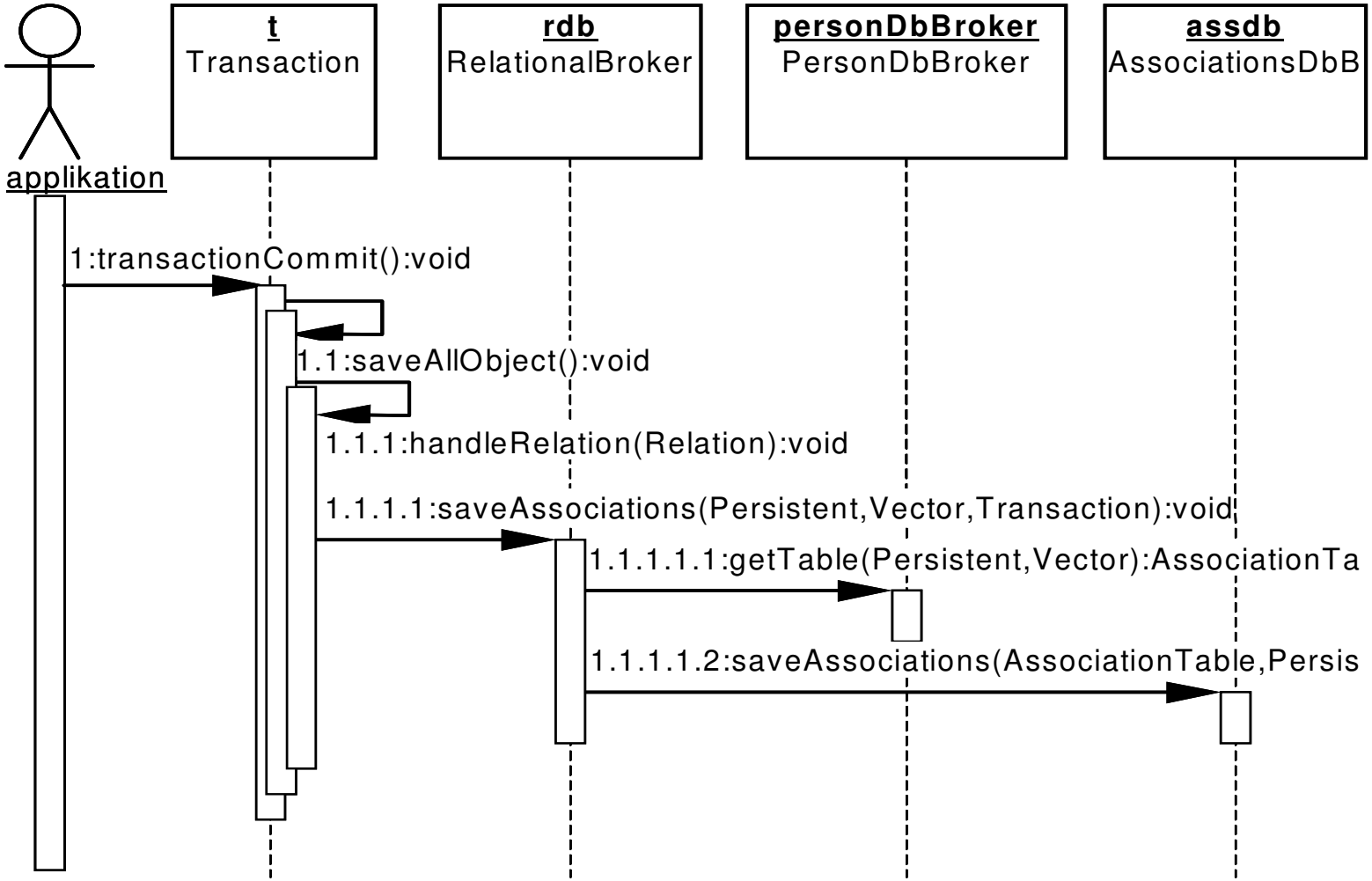
```
public void addRelation(Persistent p, Vector relatedObjects){
      transactionObjects.add( new Relation(p, relatedObjects) );
}
```

- Datencontainer für N:M-Beziehung: Klasse `Relation`
  - (innere) Hilfsklasse von `Transaction`
  - merkt sich Persistent-Objekt `p` und einen Vector der in Beziehung stehenden Objekte

```
private class Relation {
   private Persistent object;
   private Vector relatedObjects;
   Relation (Persistent object, Vector relatedObjects){
      this.object = object;
      this.relatedObjects = relatedObjects;
   }
   Persistent getObject()    { return this.object;   }
   Vector getRelatedObjects()  { return this.relatedObjects;  }
}
```

Relationen in Transaktionen (2)

- Aufruf-Hierarchie: t.transactionCommit() → this.handleRelation() → relBroker.saveAssociations()



Verantwortung der Klasse `AssociationsDbBroker`

- gleicht Oids in Bezeihungsvektor Vector mit der DB Beziehungstabelle ab
- nutzt die übergebene Connection der Transaktion

```
public void saveAssociations(AssociationTable tab, Persistent fromObject,
                  Vector toObjects, Connection c) throws SQLException {
   String fromOid = fromObject.getOid();
   Vector oidsInRuntime = getOidsFromVector(toObjects);//oid's aus Vector
   Vector oidsInDb = getOidsFromDb(tab, fromOid, c);//oid's direkt aus DB

   String runtimeOid;
   for (int i=0; i < oidsInRuntime.size(); i++) {// in DB fehlende Beziehungen
      runtimeOid = (String)oidsInRuntime.get(i);
      // oid in Laufzeit-Vektor noch nicht in DB
      if (! oidsInDb.contains(runtimeOid))
          insert(tab, fromOid, runtimeOid, c);  // Helper-Methode     }

   String dbOid;                                 // in DB überflüssige Beziehungen
   for (int i = 0; i < oidsInDb.size(); i++){
      dbOid = (String)oidsInDb.elementAt(i);
      //oid aus DB nicht mehr im Laufzeit-Vektor
      if (! oidsInRuntime.contains(dbOid))
            delete(tab, fromOid, dbOid, c);     // Helper-Methode
}  }
```