Zürcher Hochschule
für Angewandte Wissenschaften

School of
Engineering
InIT Institut für angewandte
Informationstechnologie

Bachelor of Science (BSc) in Informatik

Modul Software-Entwicklung 1 (SWEN1)

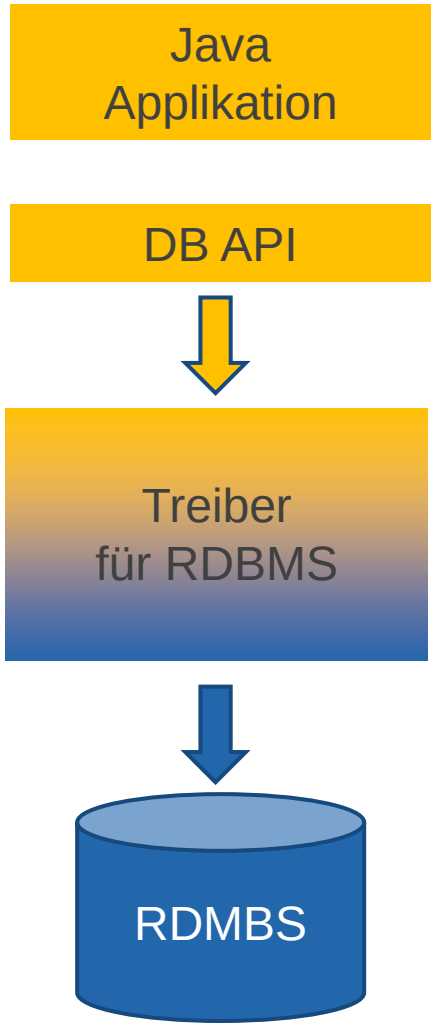
V3 – Persistenz

SWEN1/PM3 Team:
R. Ferri (feit), D. Liebhart (lieh), K. Bleisch (bles), G. Wyder (wydg)

Ausgabe: HS24

Um was geht es?

- Wie kann ich meine Java Objekte dauerhaft speichern?
- Welche Arten von Datenspeicherung gibt es?
- Welche Design Patterns stehen für die Realisierung von Persistenz in einer Applikation zur Verfügung?
- Wie kann ich mit Hilfe von den Java APIs JDBC (Java Database Connectivity) und JPA (Java Persistence API) meine Objekte dauerhaft in einer Datenbank speichern?



Lernziele LE 12 – Persistenz

- Sie sind in der Lage
  - die Varianten der Datenspeicherung zu nennen,
  - die unterschiedlichen Design Patterns für die Persistenz zu erklären,
  - mit Hilfe des Design Patterns DAO (Data Access Object) und JDBC eine Persistenz in Java umzusetzen,
  - mit JPA ein Objekt-Relationales-Mapping (O/R-Mapping) in Java anzuwenden.

Agenda

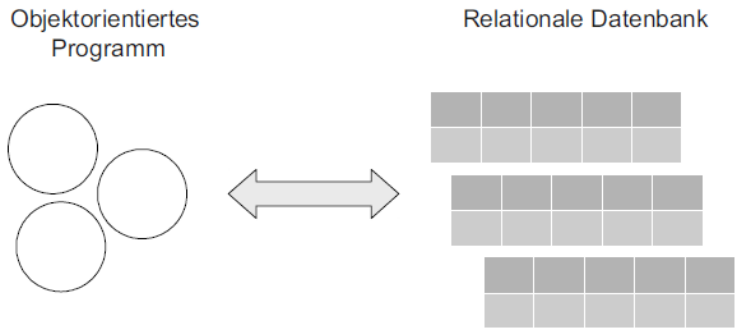
1. **Einführung in Persistenz**
2. Design-Optionen für Persistenz
3. Persistenz mit JDBC
4. O/R-Mapping mit DAO
5. O/R-Mapping mit JPA
6. Wrap-up und Ausblick

Problemstellung Persistenz (1/2)

- In vielen Applikationen müssen unterschiedliche Daten verarbeitet, verwaltet und dauerhaft, d.h. über das Programmende hinaus gesichert werden.
- Letzteres bezeichnet man als Persistenz.
- Die dauerhafte Speicherung erfolgt in Datenbankmanagementsystemen (DBMS).
- Übliche Datenbanksysteme sind sogenannte Relationale Datenbanksysteme (RDBMS) und sogenannte NoSQL-Datenbanken.
- NoSQL-Datenbanken speichern Daten ohne fixes Schema und in verschiedenen Formaten (Dokument Stores, Key-Value Stores, Graph DB, …).

Problemstellung Persistenz (2/2)

- Die Abbildung zwischen Objekten und Datensätzen in Tabellen einer relationalen Datenbank wird auch als O/R-Mapping (Object Relational Mapping, ORM) bezeichnet.
- Verhältnismässig viel Java-Code wird benötigt, um die Datensätze des Ergebnisses zu verarbeiten und in Java-Objekte zu transformieren.
- Es besteht ein Strukturbruch (engl. Impedance Mismatch) aufgrund der unterschiedlichen Repräsentationsformen von Daten (flache Tabellenstruktur – Java-Objekte).



Denkpause

Aufgabe 12.1 (5’)

Diskutieren Sie in Murmelgruppen folgende Frage:

- Was ist aktuell die vorherrschende Technologie zum Speichern von Daten im Enterprise-Umfeld?
- Recherchieren Sie dazu unter https://db-engines.com/en/ranking.
- Was sind die Gründe für dieses Ranking?

Agenda

1. Einführung in Persistenz
2. **Design-Optionen für Persistenz**
3. Persistenz mit JDBC
4. O/R-Mapper mit DAO
5. O/R-Mapper mit JPA
6. Wrap-up und Ausblick

Herausforderung: Der O/R-Mismatch (1/2)

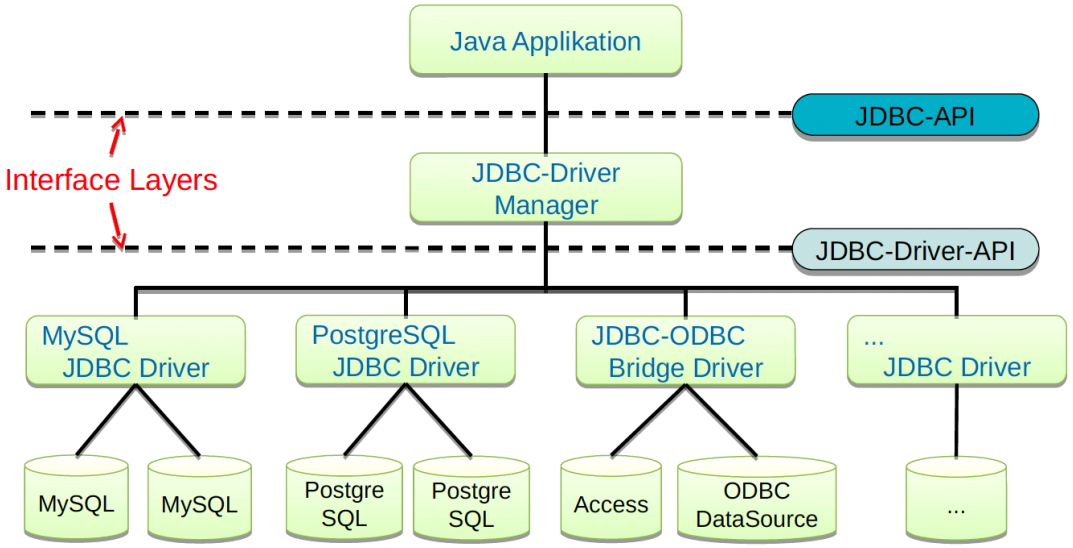
- Der O/R-Mismatch ist ein Fakt.
- Der O/R-Mismatch folgt aus konzeptionellen Unterschieden der zugrundeliegenden Technologien.
- Es gibt viele verschiedene Möglichkeiten (Patterns) den O/R-Mismatch zu überwinden.
- Active Record, O/R-Mapping resp. O/R-Mapping Frameworks oder Repositories (aus Domain Driven Design, DDD) sind ein möglicher Lösungsansatz.

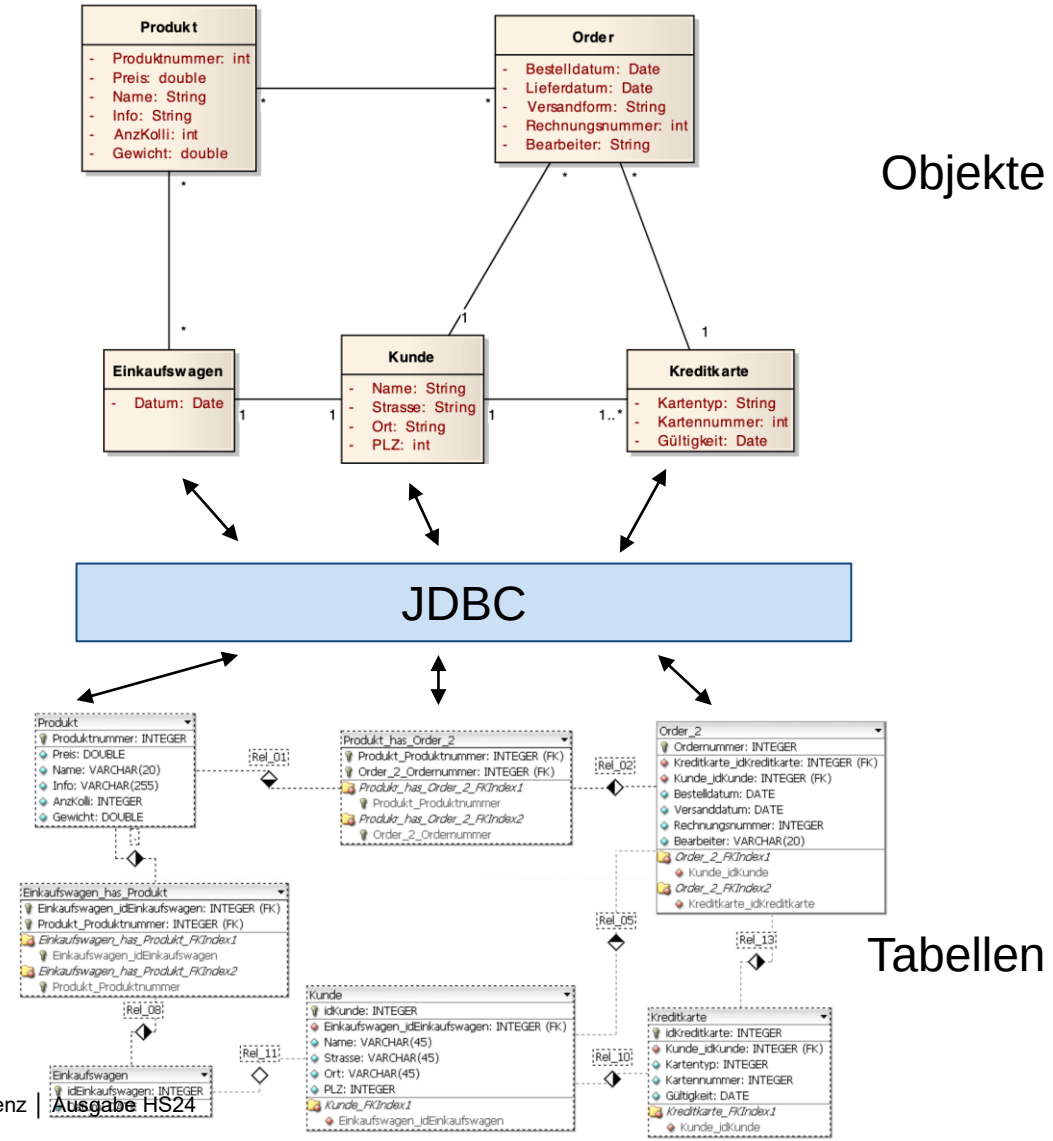
Herausforderung: Der O/R-Mismatch (2/2)

- Typen-Systeme
  - Null
  - Datum/Zeit
- Abbildung von Beziehungen
  - Richtung
  - Mehrfachbeziehungen
- Vererbung
- Identität
  - Objekte haben eine implizite Identität
  - Relationen haben eine explizite Identität (Primary Key)
- Transaktionen

JDBC: Java Database Connectivity (1997)

- JDBC verbindet die Objektwelt mit der relationalen Datenbankwelt
  - Herausforderung: Objekte vs. Tabellen,
  - Verschiedene Datentypen etc.
  - Die Programmierung ist aufwändig





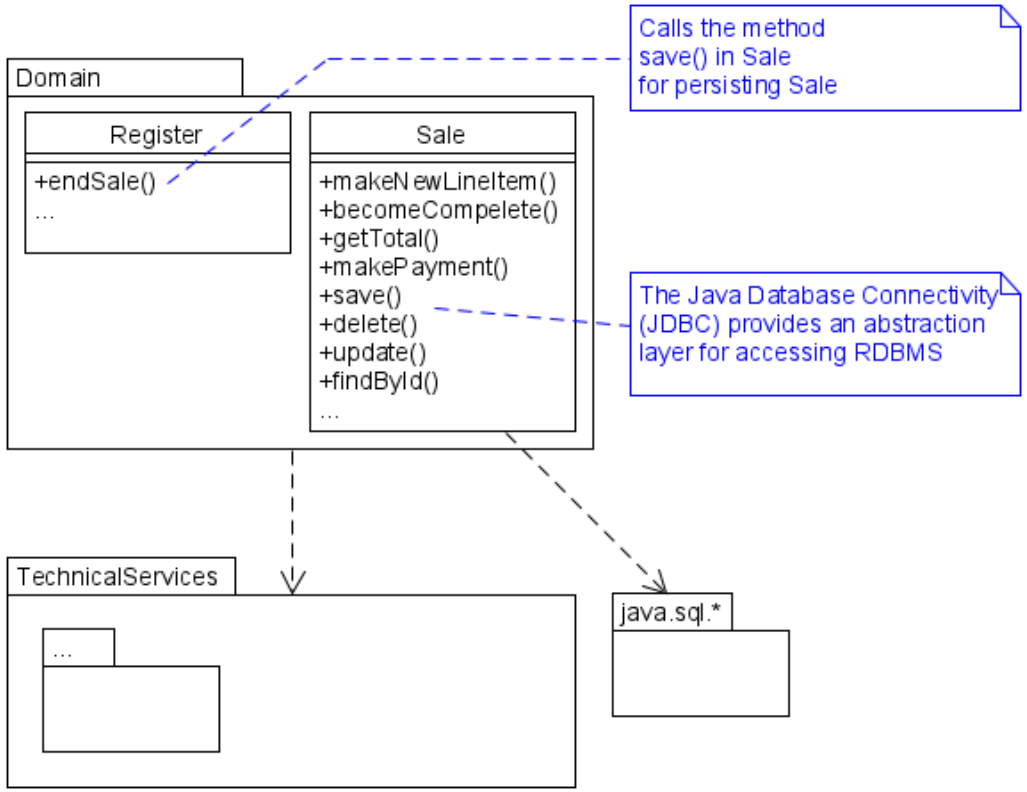
Design Pattern für Persistenz

Für eine Persistenz-Strategie muss eine Entscheidung getroffen werden, wo die Zuordnung (Mapping) zwischen Objekten und Tabellen stattfinden soll:

- Active Record (Anti Pattern): Jede Entität ist selber dafür zuständig
- Data Access Object (DAO): Abstrahiert und kapselt den Zugriff auf die Datenquelle
- O/R Data Mapper: Separate Klasse für das Mapping oder Einsatz eines ORM

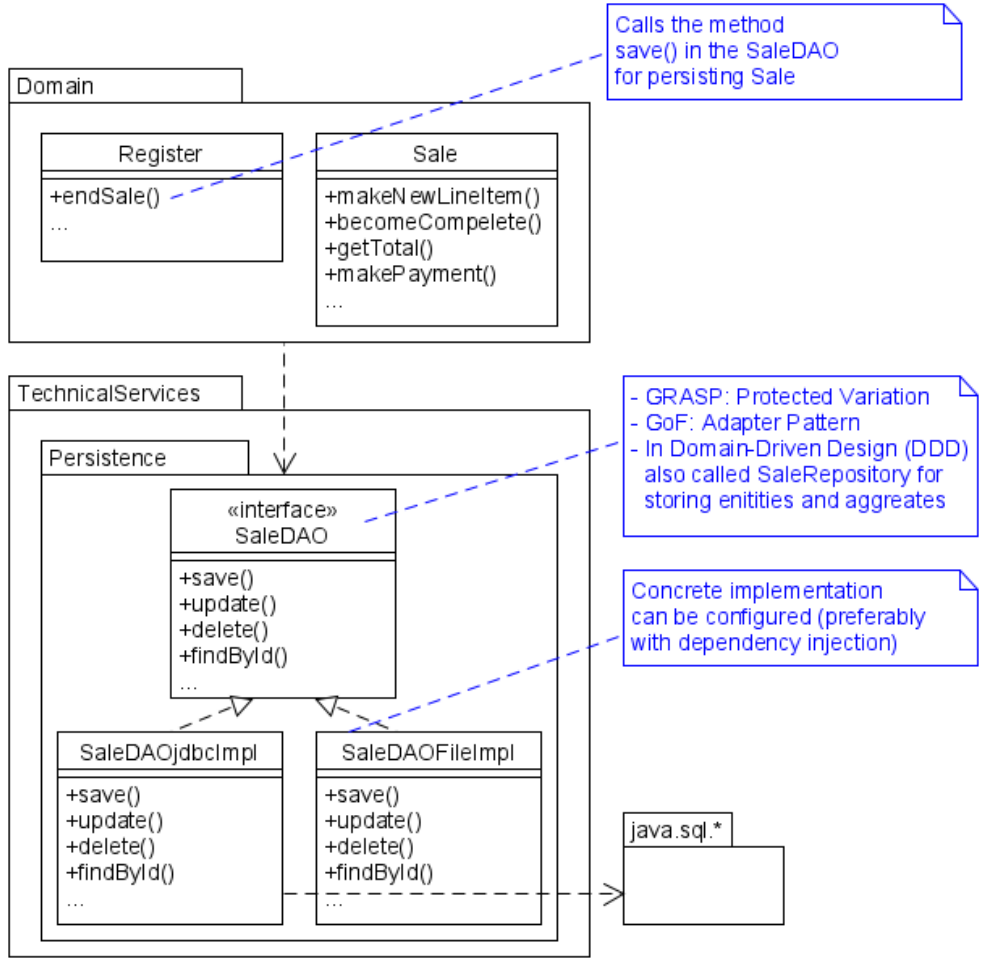
Active Record -> Anti Pattern

- Zugriffscode auf Datenbank ist in der Domänenklasse
- Wrapper für eine Zeile einer Datenbanktabelle
- Spiegelt die Datenbankstruktur
- Enthält Daten und Verhalten
- Fachlichkeit und Technik alles in einer Klasse (GRASP: Information Expert?)
- Schlechte Testbarkeit der Domänenlogik ohne Datenbank
- Schlechte Wartbarkeit und Erweiterbarkeit (No separation of concerns!)



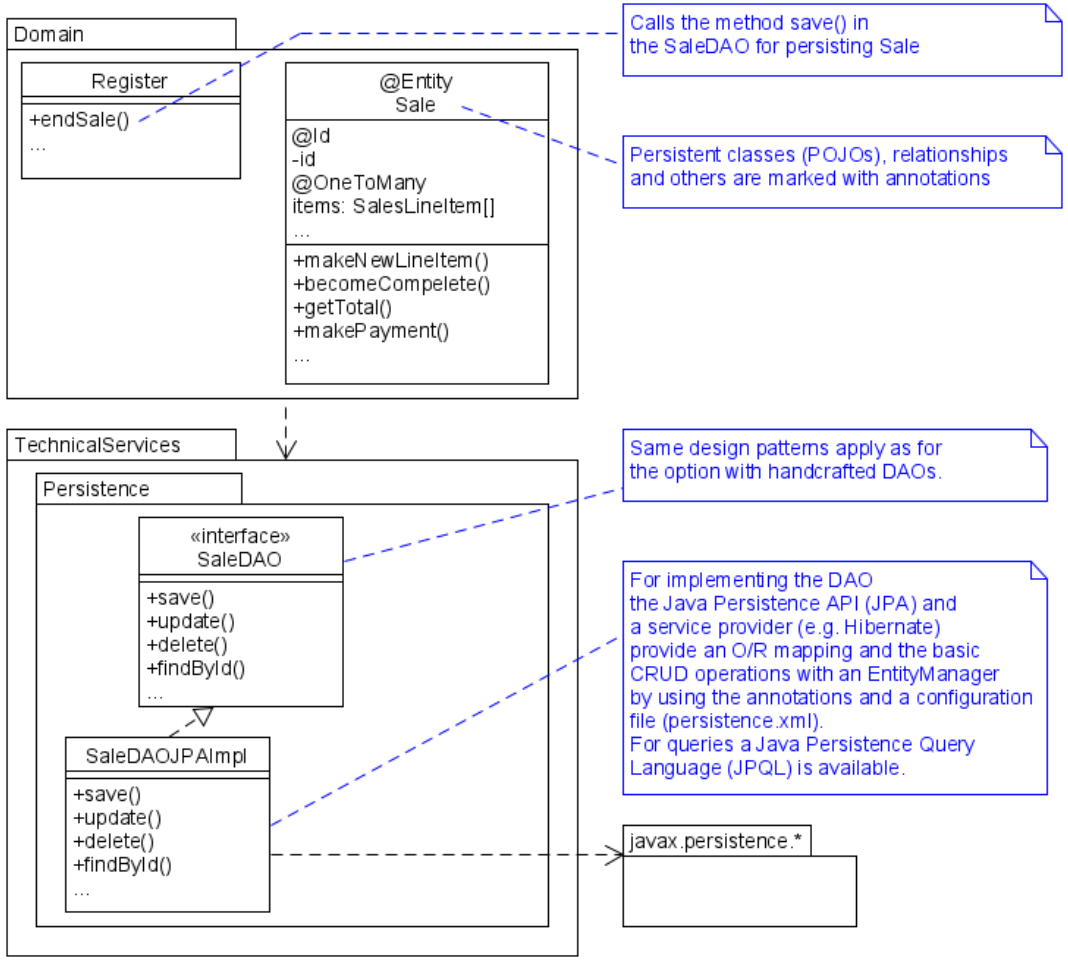
O/R-Mapping von Hand mit Hilfe eines Data Access Objects (DAO)

- Trennung von Fachlichkeit und Technik (Domänenklasse hat hohe Kohäsion)
- Gute Testbarkeit und Mocking der Persistenz
- Bevorzugtes Design ohne Einsatz eines O/R-Mappers



Verwendung eines O/R-Mappers (JPA mit Hibernate/Eclipselink o.a. Framework)

- Viel weniger Aufwand bzw. Code und standardisierte Schnittstelle
- Trennung von Fachlichkeit und Technik (Domänenklasse hat hohe Kohäsion)
- Gute Testbarkeit und Mocking der Persistenz
- DAO ist auch beim Einsatz von JPA empfehlenswert (Trennung von Fachlichkeit und Technik) - JPA könnte aber auch ohne DAO verwendet werden



Denkpause

Aufgabe 12.2 (5’)

Diskutieren Sie in Murmelgruppen folgende Fragen:

- Wo befindet sich Persistenz-Logik in einer geschichteten Architektur?
- Welche der 3 oben beschriebenen Design Patterns erfüllen die Prinzipen und Patterns von GRASP?

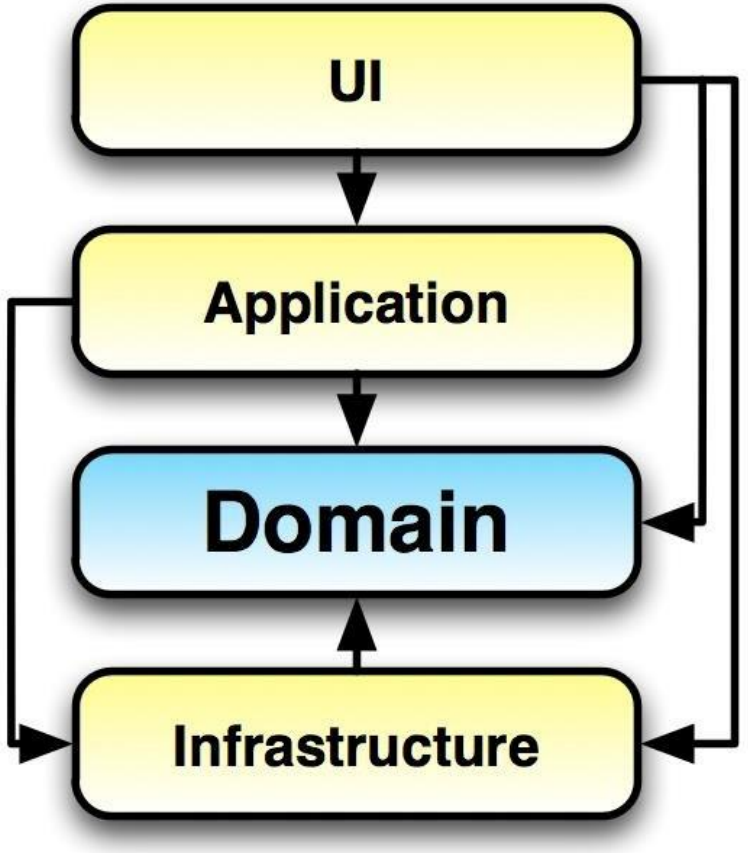
Architektur: Domain Layer

Aufgabe 12.2 - **Musterlösung**

Data Mapper und *Repository*

Zentralisierung und Lokalisierung von Business Logik (DRY)
Umsetzung der Persistenz können sich über die Zeit ändern
Das Domain Model bleibt.

DESHALB: die Persistenz aus dem Domain Model entfernen und in eine separate Schicht verschieben.



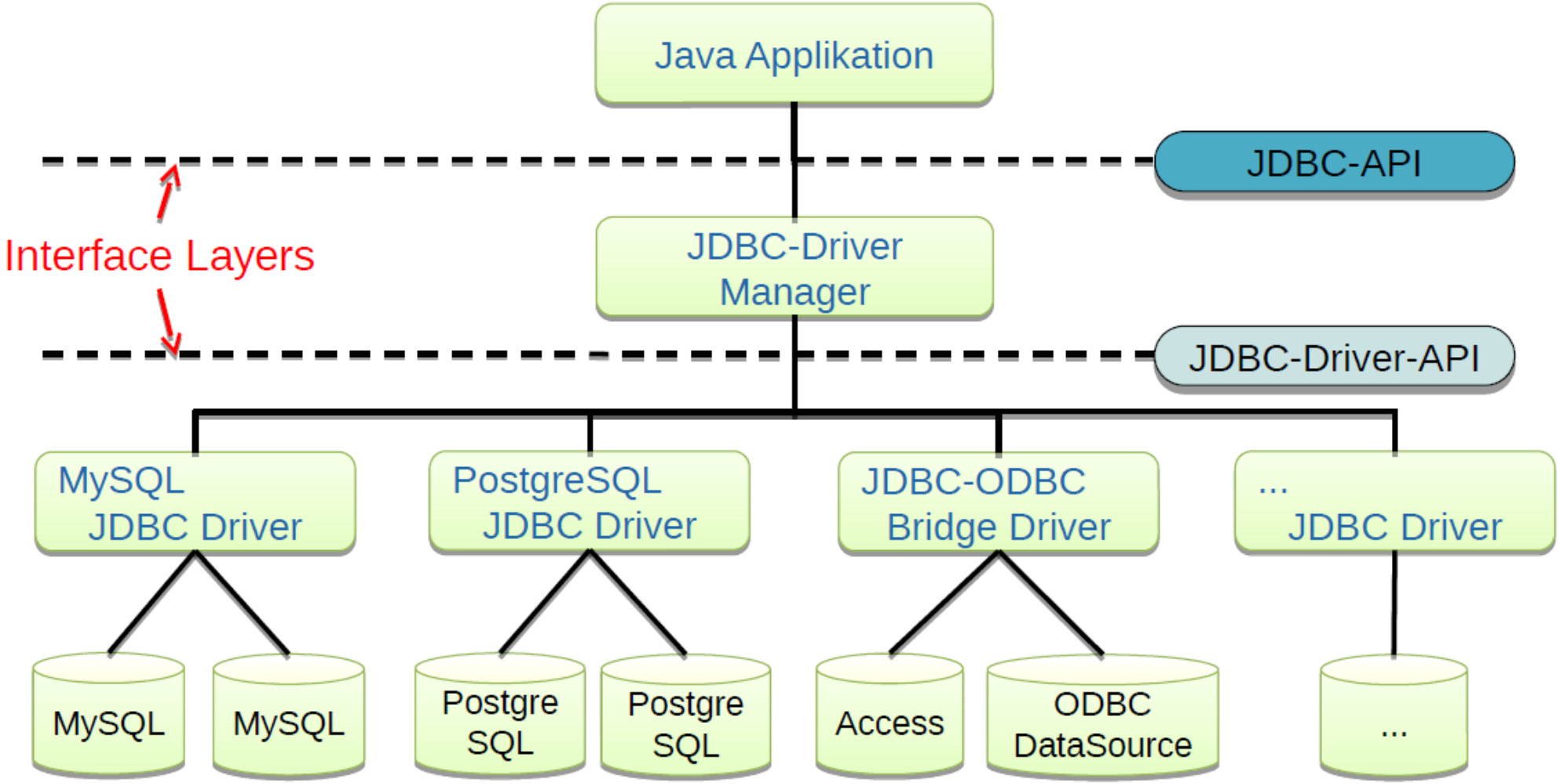
Agenda

1. Einführung in Persistenz
2. Design-Optionen für Persistenz
3. **Persistenz mit JDBC**
4. O/R-Mapper mit DAO
5. O/R-Mapper mit JPA
6. Wrap-up und Ausblick

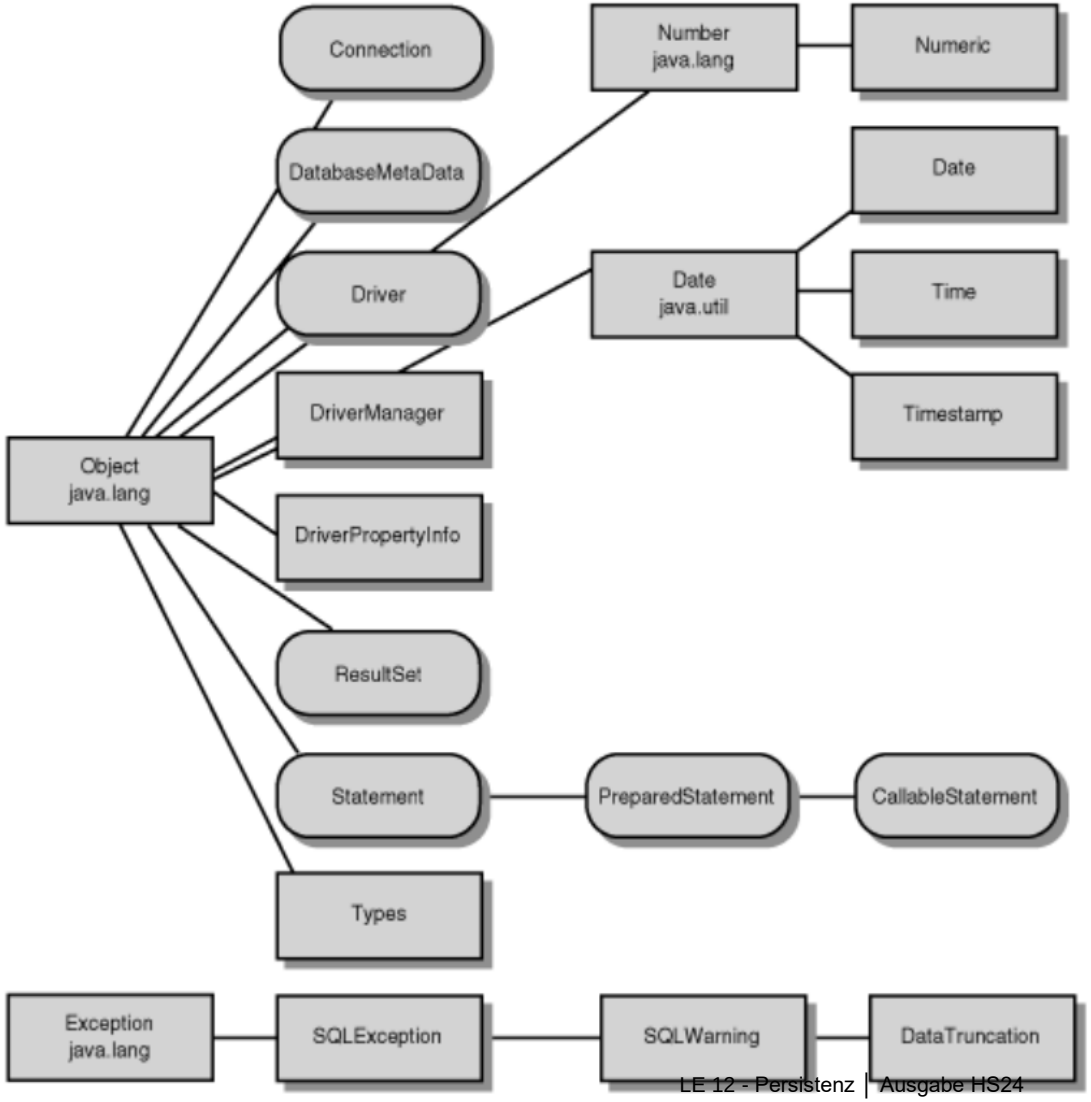
Was genau ist JDBC?

- JDBC = Java Data Base Connectivity
- Standardisierte Schnittstelle, um auf relationale Datenbanken mit Hilfe von SQL zuzugreifen
- Cross-Plattform und DB-independent
- JDBC-API ist Teil der Java-Plattfrom seit JDK1.1 (1997)
- Die aktuelle Version ist 4.2

JDBC API: Basic -Architecture



JDBC-API: Interfaces and Classes



Anwendung von JDBC

Basisanweisungen:

1. Install and load JDBC driver
2. Connect to SQL database
3. Execute SQL statements
4. Process query results
5. Commit or Rollback DB updates
6. Close Connection to database

```
import java.sql.*;

public class DbTest {
  public static void main(String[] args)
    throws ClassNotFoundException, SQLException {
    Connection con = DriverManager.getConnection(
         "jdbc:postgresql://test.zhaw.ch/testdb",
         "user", "password");
    Statement st = con.createStatement();
    ResultSet rs = st.executeQuery(
         "SELECT * FROM test ORDER BY name");
    while (rs.next()) {
      System.out.println(
         "Column 1 contains '" +
          rs.getString(2) +"'");
    }
    con.close();
  }
}
```

Eine Verbindung zur Datenbank öffnen (1/2)

- Die URL definiert den Zugriff auf die Datenbank:

  Form jdbc:<subprotocol>:<dbSource>

  <subprotocol>:: Name des DB-Treibers (+ optionaler Zugriffstyp)

  <dbsource>:: Treiber-spezifischer Pfad um die Datenbank anzuwählen
- Beispiele:
  - `jdbc:odbc:odbcsourcename` `// ODBC-Connection Type1`
  - `jdbc:oracle:thin:@test.zhaw.ch:1521:mydb` `// Oracle Type 4 Driver`
  - `jdbc:mysql://localhost:3306/mydb` `// MySQL Type 4 Driver`

Infos z.B. unter https://docs.oracle.com/cd/E13157_01/wlevs/docs30/jdbc_drivers/usedriver.html

Eine Verbindung zur Datenbank öffnen (2/2)

- Beispiel: Eine Verbindung zur Postgres Datenbank öffnen

```
import java.sql.*; // required to access JDBC classes String

url = "jdbc:postgresql://localhost:5432/mydb";
String user = "musterfelix";
String passwd = "secret";
Connection con = DriverManager.getConnection(url, user, passwd);
```

SQL Query ausführen

- Um einen SQL-Befehl mittels einer offenen Connection an die DB zu senden:
  - Eine Instanz von **Statement** oder **PreparedStatement** ist erforderlich
  - Das komplette Ergebnis einer Query wird als ein **ResultSet** Objekt zurückgegeben.

```
Statement st = con.createStatement();
String query = "SELECT * FROM mytable WHERE x=500";
ResultSet rs = st.executeQuery(query);
while (rs.next()) { // read the ResultSet
  System.out.print("Column 1 contains ");
  System.out.println(rs.getString(1));
}
rs.close();
st.close();
```

Prepared Statements

- Prepared Statements optimieren das Laufzeitverhalten, weil die Queries vorkompiliert werden können.
  - SQL-Injection kann vermieden werden

```
int id = 500;
String query = "SELECT * FROM meinetabelle WHERE x = ?";
PreparedStatement st = con.prepareStatement(query);
st.setInt(1, id); // replace 1st placeholder by value of id
// various setter for different datatypes exist e.g. setString(), setDate(),
setBlob(), ...
ResultSet rs = st.executeQuery();
while (rs.next()) {
  System.out.print("Spalte 1 ergab ");
  System.out.println(rs.getString(1));
}
rs.close(); st.close();
```

Statement Execute-Methoden

- Die Statement Schnittstelle definiert drei Execute-Methoden, um SQL-Befehle auszuführen:
  - **`executeQuery`**`()` für SELECT-Statements,
    → return eines ResultSet
  - **`executeUpdate`**`()` für INSERT, UPDATE, DELETE, CREATE, DROP, ALTER, ...-Statements,
    → return eines int-Wertes (number of affected tuples)
  - **`execute`**`()` um Stored-Procedures auszuführen (scripts running inside the DB).

Close Connection

- Um alle Ressourcen frei zu geben, soll die Connection am Ende der Transaktion geschlossen werden.

```
con.close();
```

- Dies bewirkt automatisch das Schliessen:
  - aller offenen Statements
  - aller offenen ResultSets

Transaktionen

- Eine Transaktion ist eine Sequenz von SQL-Statements, welche Zusammenhängend («in einem Schritt») ausgeführt werden müssen
  `con.commit()`
- Oder zurück zum Zustand vor der Ausführung
  `con.rollback()`
- Transactions are ACID (Atomic, Consistent, Independent, Durable)
  - Falls `auto-commit` freigeschaltet ist (Standard), wird jedes «Execute-Statement» automatisch «commited».
  - Auto-Commit-Mode kann mittels der Methode setAutoCommit gesetzt werden
    `con.setAutoCommit(boolean)`

Beispiel Transaktionen


```
try {
    Connection con = DriverManager.getConnection(...);
    con.setAutoCommit(false); // disable auto-commit
    Statement s = con.createStatement();

    s.executeUpdate("... SQL statement 1 ... ");
    s.executeUpdate("... SQL statement 2 ... ");
    s.executeUpdate("... SQL statement 3 ... ");

   con.commit(); // transaction (3 statements) is committed
} catch (SQLException e) {
   con.rollback(); // an error occurred -> rollback
} finally {
   con.close(); // in any case close the connection
}
```

Agenda

1. Einführung in Persistenz
2. Design-Optionen für Persistenz
3. Persistenz mit JDBC
4. **O/R-Mapping mit DAO**
5. O/R-Mapping mit JPA
6. Wrap-up und Ausblick

O/R-Mapping Pattern

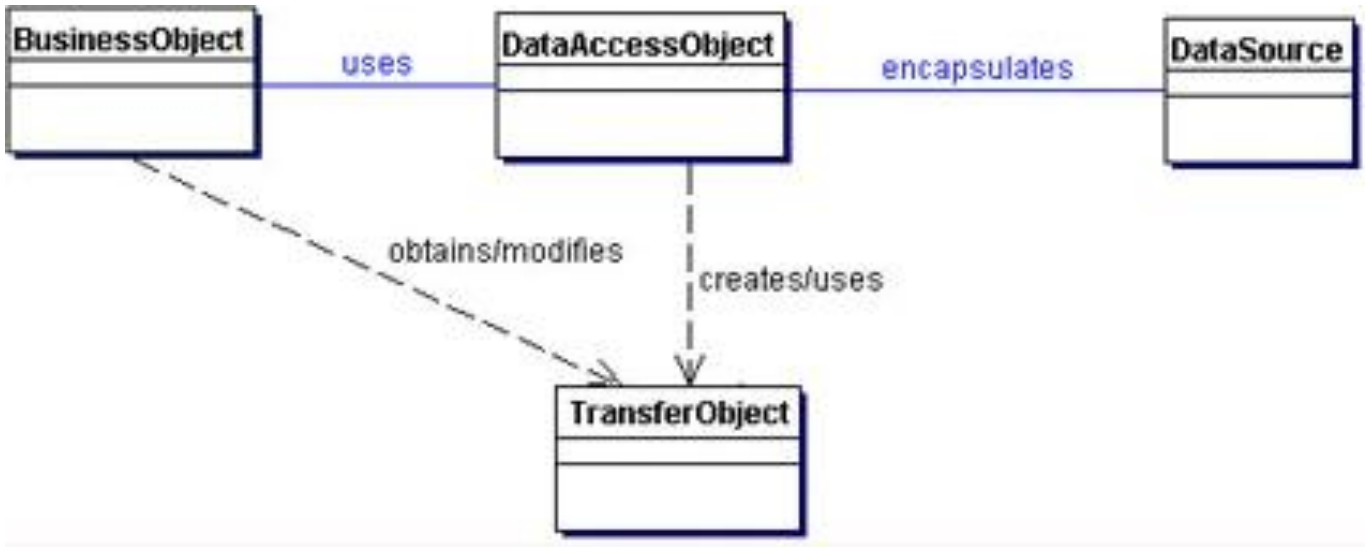
Es sollen beide Varianten des O/R-Mapper Patterns anhand eines praktischen Beispiels betrachtet werden:

- DAO (Data Acess Object) ohne ein ORM (Object Relational Mapper)
- Umsetzung von DAO mit Hilfe von JPA (Java Persistence API)

DAO - Data Access Object Pattern

- Das Artikel-Objekt repräsentiert das Domain-Model-Objekt.
- Die Verbindung zur Datenbank wird durch das DAO sichergestellt.
  - Enthält die üblichen CRUD-Methoden wie create, read, update und delete.
  - Kann auch Methoden enthalten wie findAll, findByName, findByID um eine Kollektion von Daten aus der Datenbank abzufragen.

DAO - Data Access Object Pattern



Sun Developer Network - Core J2EE Patterns
http://www.oracle.com/technetwork/java/dataaccessobject-138824.html

Das TransferObject aka. DTO kann zusätzlich für den Transport der Daten in einem verteilten System verwendet werden.

Beispiel Article und ArticleDAO

Business Object

```
public class Article {
  private long id;
  private String name;
  private float price;
  public long getId(){
    return id;
  }
  public void setId(long id){
    this id = id
  };
  ...
}
```

Data Access Object (DAO)

```
//Interface to be implemented by all ArticleDAOs
public interface ArticleDAO {
  public void insert(Article item);
  public void update(Article item);
  public void delete(Article item);
  public Article findById(int id);
  public Collection<Article> findAll();
  public Collection<Article> findByName (String name);
  public Collection<Article> findByPrice (float price);
  ...
}
```

Agenda

1. Einführung in Persistenz
2. Design-Optionen für Persistenz
3. Persistenz mit JDBC
4. O/R-Mapping mit DAO
5. **O/R-Mapping mit JPA**
6. Wrap-up und Ausblick

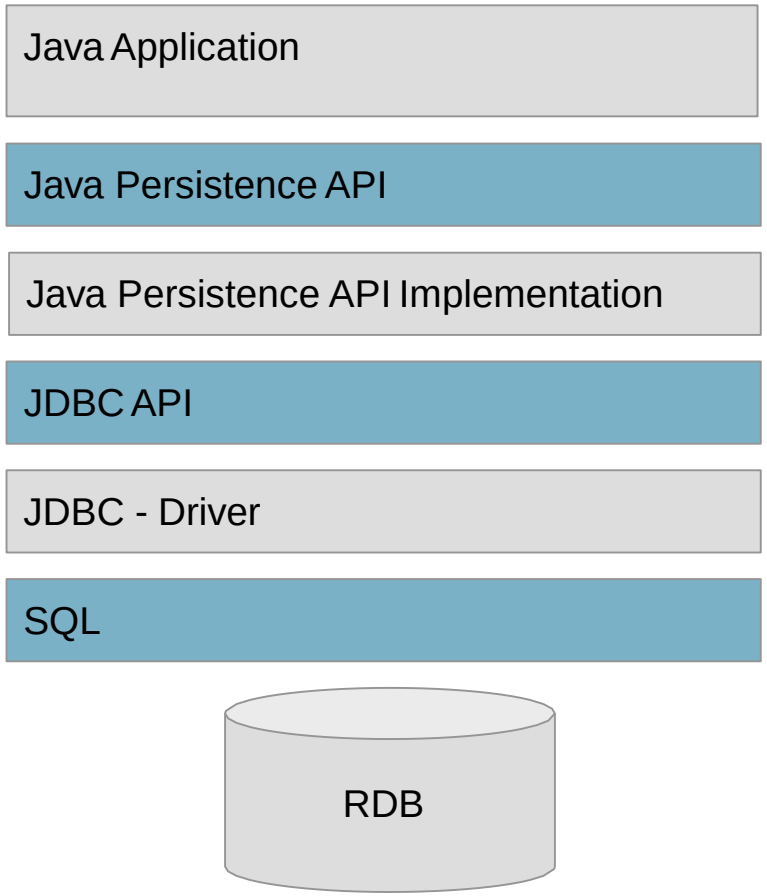
Versprechen von automatischem O/R-Mapping

- Die Applikation wird von der DB entkoppelt
  - Applikationsentwickler muss kein SQL beherrschen.
  - Das relationale Modell der Datenbank hat keinen Einfluss auf das OO-Design.
- Automatische Persistenz
  - Automatisierte Abbildung der Objekte in die relationalen Strukturen.
  - Die Applikationsentwickler muss sich nicht um die «low-level»-Details kümmern.
- Transparente Persistenz / Persistence Ignorance
  - Die Klassen des Domain-Models wissen nicht, dass sie persistiert und geladen werden können und haben keine Abhängigkeit zur Persistenz-Infrastruktur.
- JPA ist ein Java Standard für O/R-Mapping
  - Verschiedene Implementationen, Hibernate vermutlich die bekannteste

JPA (Java Persistence API) Überblick

- Es folgt eine kurze, unvollständige Auflistung der wichtigsten Konzepte von JPA.
- Starke Entkopplung der Anwendungslogik von der (relationalen) Datenbank.
- Die Domänenklassen sind ganz normale Java Klassen (POJO)
  - Ausser Annotationen enthalten Sie keinen JPA spezifischen Code.
- Referenzen
  - Werden entweder mit der referenzierenden Klasse (eager loading) oder erst, wenn die Referenz gebraucht wird (lazy loading), geladen.
  - Referenzen können direkt traversiert werden, JPA erledigt das Laden des referenzierten Objekts im Hintergrund.
- Transaktionshandling und das Absetzen von Queries müssen über JPA spezifische Klassen abgewickelt werden.
  - EntityMangerFactory, EntityManager, EntityTransaction

Technologie-Stack



Java 5+

JPA Spezifikation

EclipseLink (TopLink), Hibernate, OpenJPA etc.

JDBC 4.0

Herstellerspezifisch

SQL (und Dialekte)

Entity Metadata

- Kennzeichnung mit Annotation @Entity oder Mapping mit XML
- Klasse kann Basisklasse oder abgeleitet sein
- Klasse kann abstrakt oder konkret sein
- Serialisierbarkeit ist bezüglich Persistenz nicht erforderlich

Beispiel Entity

- Minimale Anforderung an eine Entity

```
@Entity
public class Employee {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
    private String name;
    private String lastName;
..
..
```

Primärschlüssel

- Primärschlüssel können in Zusammenarbeit mit der Datenbank generiert werden. Strategien sind Identity, Table, Sequence und Auto

```
@Entity public class Employee {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    public Integer id;
}
```

```
public class Employee {
    @TableGenerator(name = "Emp_Gen", table = "ID_GEN",  pkColumnName = "GEN_NAME",
                    valueColumnName = "GEN_VAL")
    @Id
    @GeneratedValue(strategy = GenerationType.TABLE, generator = "Emp_Gen")
    private int id;
...
}
```

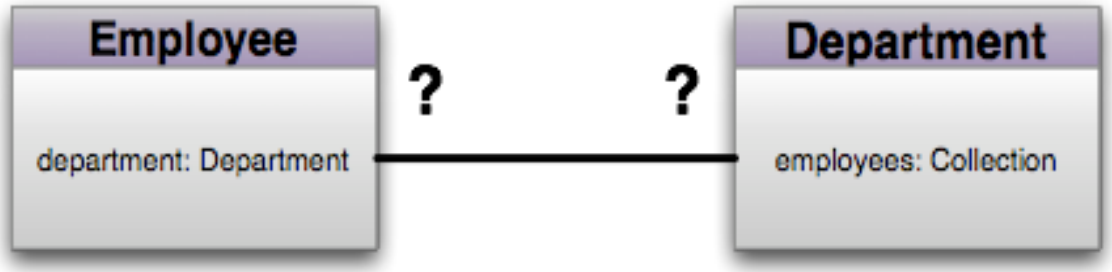
Mapping

- Es wird immer vom Default-Verhalten ausgegangen
- Das Default-Verhalten kann übersteuert werden

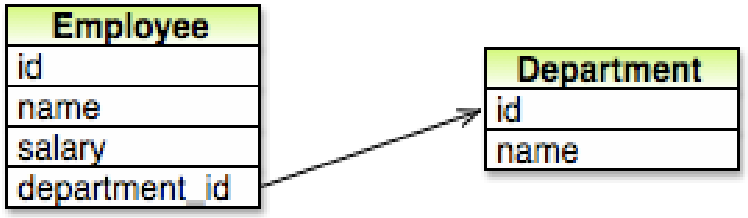
```
@Entity
@Table(name = "EMP")  public class Employee {

    @Id
    @Column(name = "EMP_ID")
...
}
```

```
@Column(name = "sender")
protected String sender;
```

Parent-Child Beziehung



- Mapping des Klassenmodells auf das DB-Schema mittels JPA: Metadata ist erforderlich.
  - Je nach Klassenmodell wird entweder eine many-to-one Beziehung oder eine one-to-many Beziehung gemappt.
  - Falls beide Richtungen gemappt werden sollen, so muss definiert werden, dass für beide derselbe Foreign-Key zugrunde liegt.



Parent-Child Beziehung

```
@Entity
public class Employee {
   …
   @ManyToOne
   private Department department;
   …
```

Mapping der many-to-one Beziehung

```
@Entity
public class Department {
   …
   @OneToMany
   @JoinColumn (name="department_id")
   private Set<Employee> employees =
             new HashSet<Employee>();
   …
```

Variante:
Mapping der one-to-many Beziehung mit Foreign Key

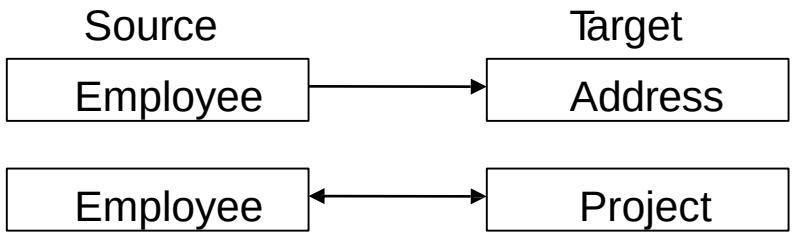
```
@Entity
public class Department {
   …
   @OneToMany (mappedBy = "department")
   private Set<Employee> employees =
             new HashSet<Employee>();
   …
```

Variante:
Mapping der bidirektionalen Beziehung

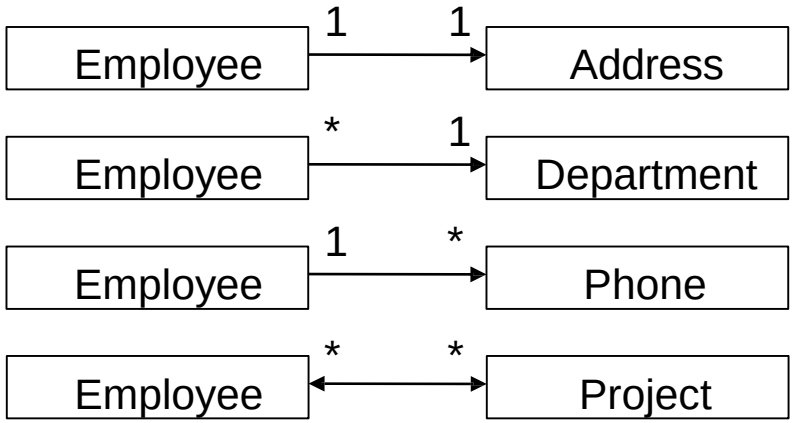
- JPA muss wissen, dass nur ein Foreign-Key für beide Richtungen existiert.

Parent-Child Beziehung: Collection Types

- Richtung
  - Unidirektional
  - Bidirektional



- Kardinalität
  - One-to-one
  - Many-to-one
  - One-to-many
  - Many-to-many



JPA Schlüsselkonzepte

- Persistence Unit
  - Konfiguration für das Mapping der Entitäten mit einer relationalen Datenbank
- Entity Manager
  - Schnittstelle für die Interaktion mit der Persistence Engine
  - API um den Lebenszyklus einer Entität zur verwalten
- Persistence Context
  - Ein Set von Entitäten, welche durch den Entity Manager verwaltet werden
- Transactions

Insert Entity

- Eine Entität mit vorgegebenem employee-Objekt in der Datenbank speichern.
- `entityManager` erstellen und Transaktion beginnen und mittels der Methode `persist()` in der DB speichern.
- Transkation muss mittels `commit()` bestätigt werden.

```
public void insertEntity(Employee employee) {
    EntityManager entityManager = JPAUtil.getEntityManagerFactory().createEntityManager();
    EntityTransaction entityTransaction = entityManager.getTransaction();
    entityTransaction.begin();
    entityManager.persist(employee);
    entityManager.getTransaction().commit();
    entityManager.close();
}
```

Find Entity

- Eine Entität mit vorgegebener id von der Datenbank lesen.
- `entityManager` erstellen und mittels der Methode `find()` von der DB lesen.

```
public Employee findEntityById(long id) {
    EntityManager entityManager = JPAUtil.getEntityManagerFactory().createEntityManager();
    Employee employee = entityManager.find(Employee.class, id);
    entityManager.close();
    return employee;
}
```

Anwendung JPQL – Java Persistence Query Language

- Mittels JPQL können datenbankunabhängige Queries formuliert werden.
- Beispiel: Anzahl Datensätze in der DB ermitteln.
- Mittels createQuery ein Query-Objekt erstellen und mittels getSingleResult() ein skalares Ergebnis einlesen.

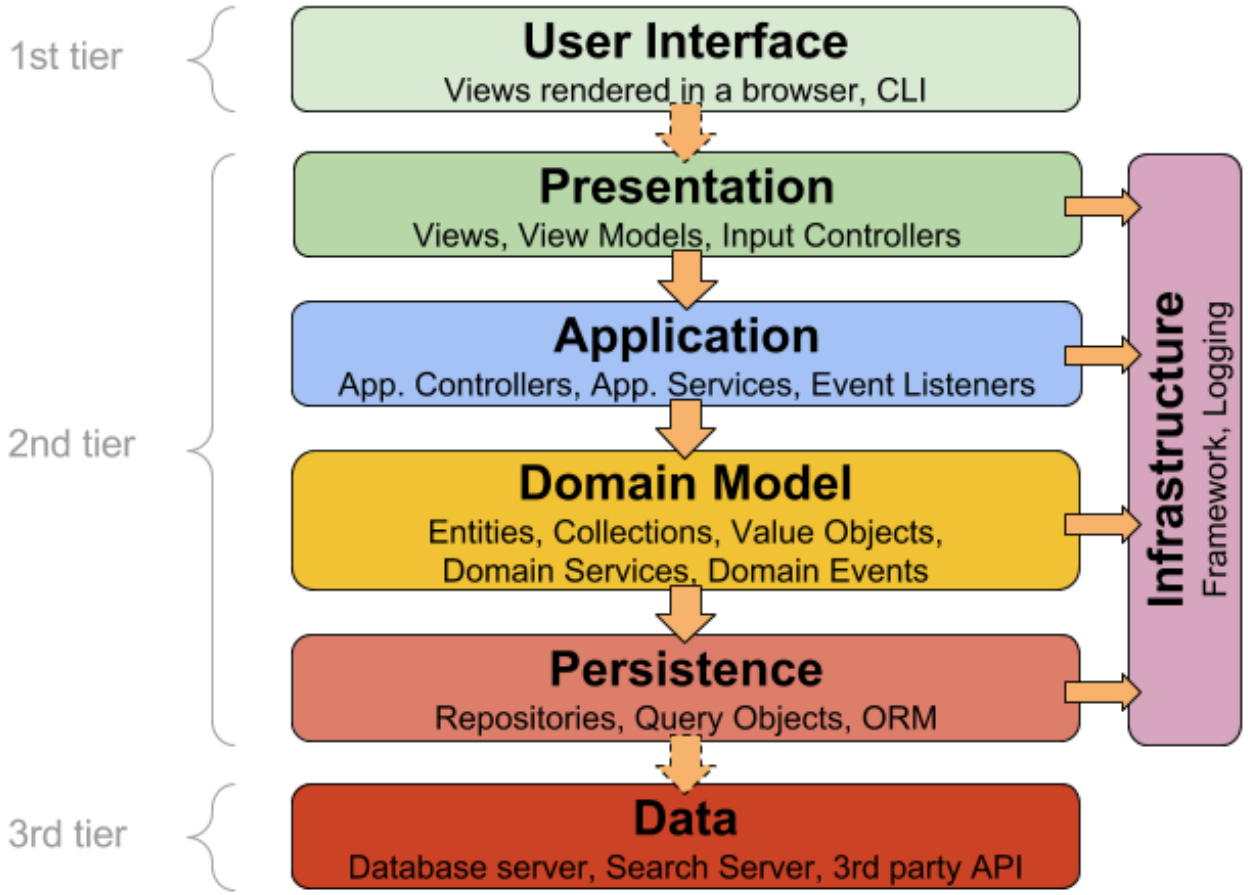
```
public long count () {
    EntityManager entityManager = JPAUtil.getEntityManagerFactory().createEntityManager();
    entityManager.getTransaction().begin();

    Query query=entityManager.createQuery("SELECT COUNT (emp.id) FROM Employee emp");
    long result =(long) query.getSingleResult();
    entityManager.getTransaction().commit();
    entityManager.close();
    return result;
}
```

Ausblick Design Pattern Repository

- Ein System mit einem komplexen Domänen-Model profitiert wie vorher beschrieben von einer Data-Mapper-Schicht (mit JPA und DAOs), um die Details des Datenbankzugriffcodes zu isolieren.
- Eine zusätzliche Abstraktionsschicht oberhalb des Data-Mappers kann helfen um die Konstruktion von Datenbank-Abfragen (Queries) an einem Ort zu konzentrieren.
- Diese zusätzliche Schicht wird um so wichtiger je mehr Domänen-Klassen vorhanden sind, die viele Zugriffe auf die Datenbank vornehmen.
- Die zusätzliche Schicht wird als Repository bezeichnet
- Das Konzept stammt aus Domain Driven Design, DDD (Eric Evans).
  - Wird in den Wahlpflichtmodulen ASE1/2 anhand von Spring Data behandelt.

Design Pattern Repository: Schichtenmodell

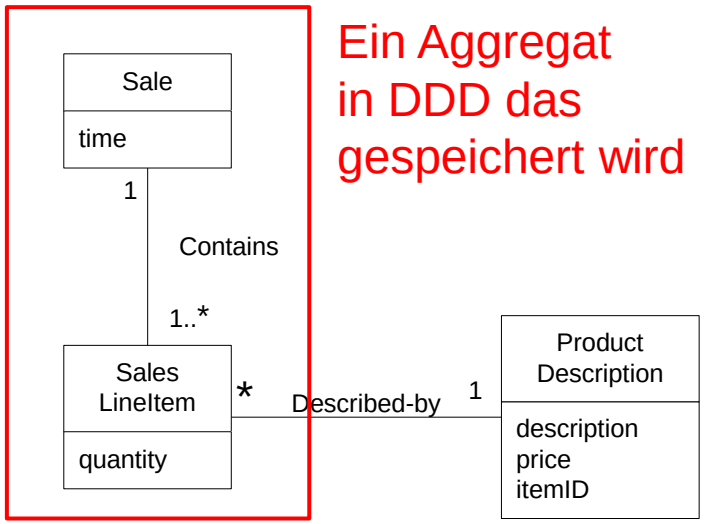
- 3-Tier Architecture
- Persistenz kann mittels Repositories umgesetzt werden



Idee und Beispiel Repository Pattern

- Eine Repository vermittelt zwischen Domänen- und Data-Mapping Schicht

```java
/**
 * The GRASP controller for the use case process sale.
 */
public class ProcessSaleHandler {
    private ProductDescriptionRepository catalog;
    private SaleRepository saleRepository;
    private Sale currentSale;

    public ProcessSaleHandler(ProductDescriptionRepository catalog, SaleRepository saleRepository) {

    public void makeNewSale() {

    public void enterItem(String id, int quantity) {

    public Money getTotalOfSale() {
    @Transactional
    public void endSale() {
        assert(currentSale != null && !currentSale.isComplete());
        this.currentSale.becomeComplete();
        this.saleRepository.save(currentSale);
    }

    public Money getTotalWithTaxesOfSale() {

    public void makePayment() {
```



Ein Aggregat in DDD das gespeichert wird

```java
/**
 * Repository for Sale.
 * An implementation of CRUD and common search methods
 * is automatically generated by Spring Data.
 */
@Repository
public interface SaleRepository extends CrudRepository<Sale, String> {
    public List<Sale> findOrderByDateTime();
    public List<Sale> findByDateTime(final LocalDateTime dateTime);
}
```

Agenda

1. Einführung in Persistenz
2. Design-Optionen für Persistenz
3. Persistenz mit JDBC
4. O/R-Mapping mit DAO
5. O/R-Mapping mit JPA
6. **Wrap-up und Ausblick**

Wrap-up

- Viele Applikationen verlangen, dass Daten dauerhaft gesichert werden müssen nach dem Programmende.
- Bei kleineren Applikationen kann diese Persistenz auch selber ausprogrammiert werden.
- Dabei sollte aber das Design Pattern Data Access Object (DAO) oder Repository angewendet werden.
- Für grössere Applikationen werden heute sogenannte O/R-Mapper eingesetzt.
- Java bietet mit dem Java Persistence API (JPA) eine standardisierte Schnittstelle für das O/R-Mapping, für die es viele Provider gibt (z.B. Hibernate).

Ausblick

- In der nächsten Lerneinheit werden wir:
  - das Thema Design von Frameworks vertiefen.

Quellenverzeichnis

[1] Oracle: Core J2EE Patterns - Data Access Object, https://www.oracle.com/java/technologies/dataaccessobject.html

[2] Oracle: Core J2EE Patterns - JPA, https://www.oracle.com/java/technologies/persistence-jsp.html

[3] Hibernet: Documentation, https://hibernate.org/orm/documentation/5.4/

[4] Ranking Database: https://db-engines.com/en/ranking