

# **Verteilte Verarbeitung**

**Kapitel 13**

**JPA, Java Persistence API**

# Zugriff auf relationale Datenbanken in Java

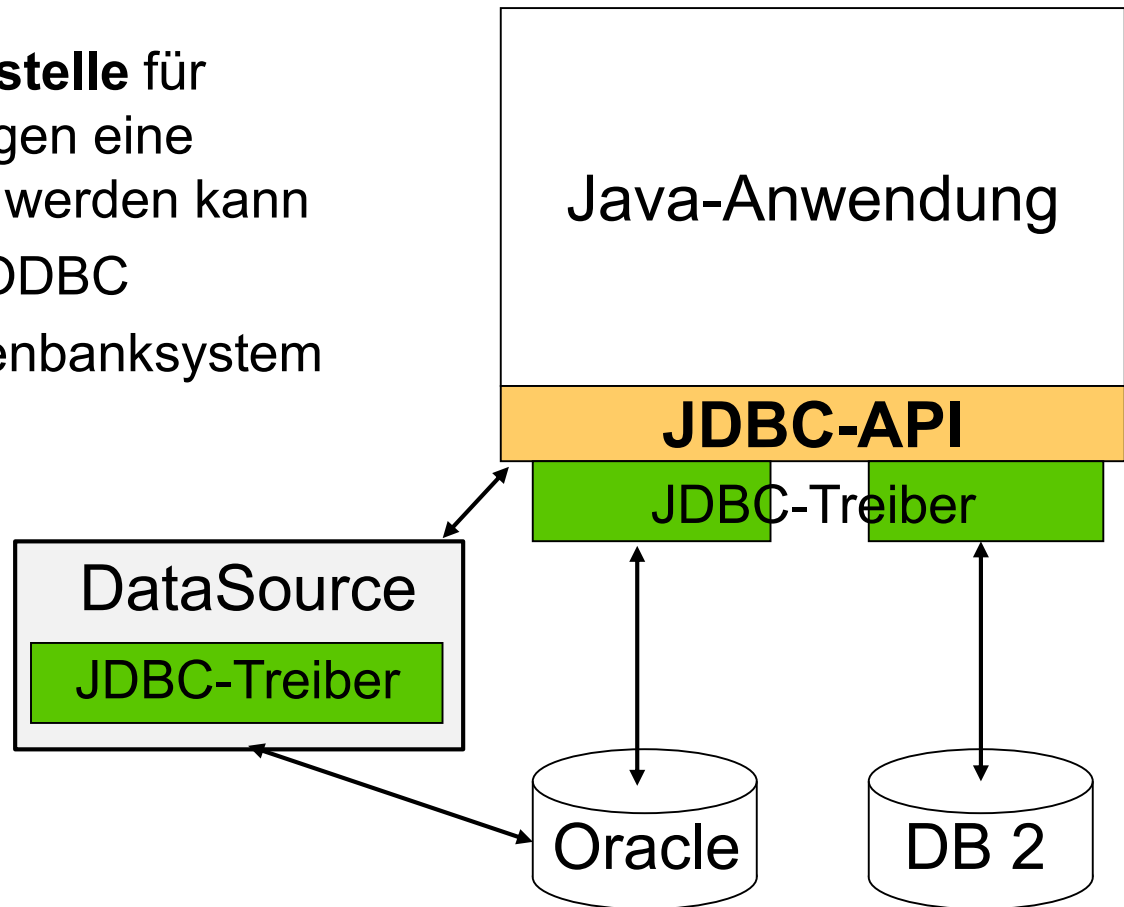
- ***Kein statisches SQL***  
(SQLJ hat sich nicht durchgesetzt)
- **Dynamisches SQL mit JDBC**
  - Java Database Connectivity
  - Liegt allen anderen Technologien zugrunde
  - API zum Versenden von Strings (SQL) und Auswerten des Ergebnisses (z.B. Datenbank-Cursor)
- **Persistenz-Schicht mit JPA**
  - Java Persistence API (sollte i.d.R. verwendet werden)
  - Implementierungen auf JDBC-Basis wie Hibernate oder EclipseLink
  - Implementiert auch Abbildung Objekte auf Relationen

# JDBC

# Java Database Connectivity

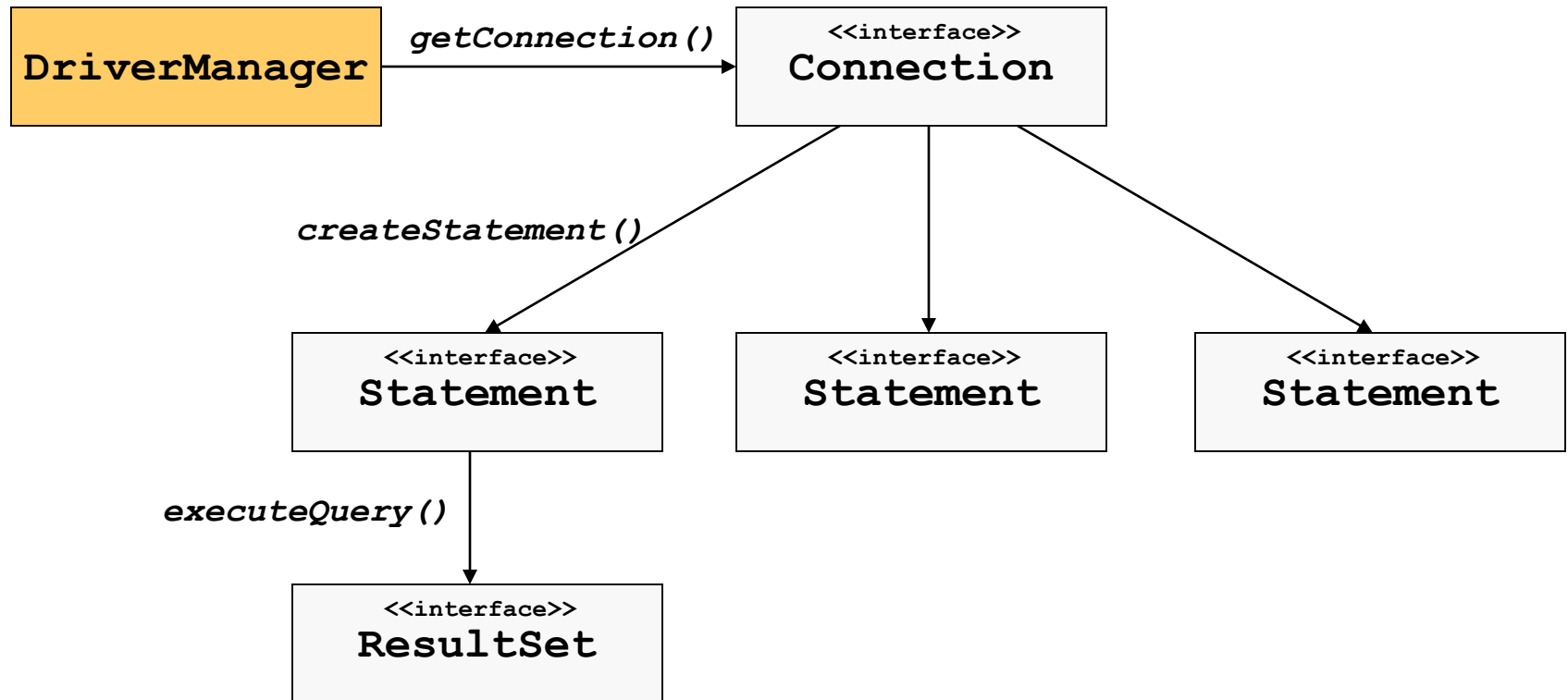
# Was ist JDBC ?

- **Programmierschnittstelle** für Java, mit der **SQL** gegen eine Datenbank abgesetzt werden kann
- konzeptuelle Basis: ODBC
- *unabhängig* vom Datenbanksystem
- dynamisches SQL

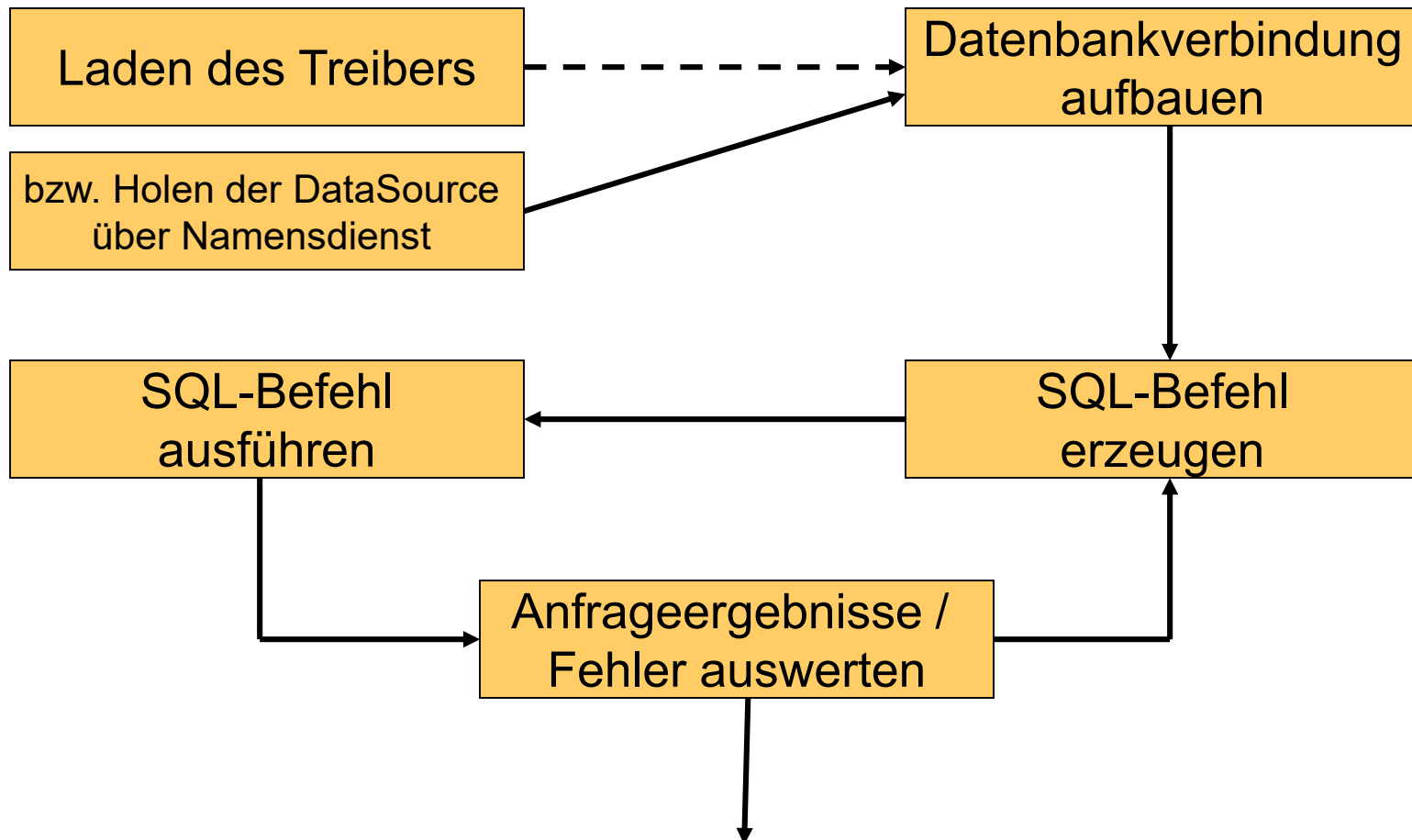


# JDBC-Struktur

(DriverManager nicht mehr verwenden!, Jetzt DataSource)



# Wie arbeitet JDBC ?



# Aufbau einer Datenbankverbindung (Leichtgewichtige Datenbank: h2database)

- Anmelden der benötigten Packages

```
import java.sql.*; // JDBC-Package
```

- JDBC-Treiber laden

Treiberklasse über den `ClassLoader` laden, der `DriverManager` verwendet diese automatisch

```
Class.forName("org.h2.Driver").newInstance();
```

- Verbindung öffnen

```
String url = "jdbc:h2:~/kundendb";
```

```
Connection dbc =  
    DriverManager.getConnection(url, "SA", "");
```

# Schreibender Datenbankzugriff (1)

- Erzeugen des SQL-Statements

```
Statement stmt = dbc.createStatement();
```

- Absetzen von SQL-Befehlen

```
(INSERT, UPDATE, DELETE, CREATE-, ALTER-, DROP TABLE)  
stmt.executeUpdate("CREATE TABLE CUSTOMER ("  
    + "id INTEGER, vorname VARCHAR(20), "  
    + "nachname VARCHAR(20), gebdat CHAR(10)");  
stmt.executeUpdate("insert into CUSTOMER values ("  
    + "100, 'Bernd', 'Brot', '1971-09-21')");
```

- Freigeben der belegten Ressourcen

```
stmt.close();
```



## Schreibender Datenbankzugriff (2)

- Erzeugen des SQL-Statements

```
PreparedStatement pstmt =  
    dbc.prepareStatement("INSERT INTO CUSTOMER (id,  
        vorname, nachname, gebdat) VALUES (?, ?, ?, ?) " );
```

- Binden der Parameter

```
pstmt.setInt(1, 103);  
pstmt.setString(2, "Harry");  
pstmt.setString(3, "Hirsch");  
pstmt.setString(4, "1971-03-05");
```

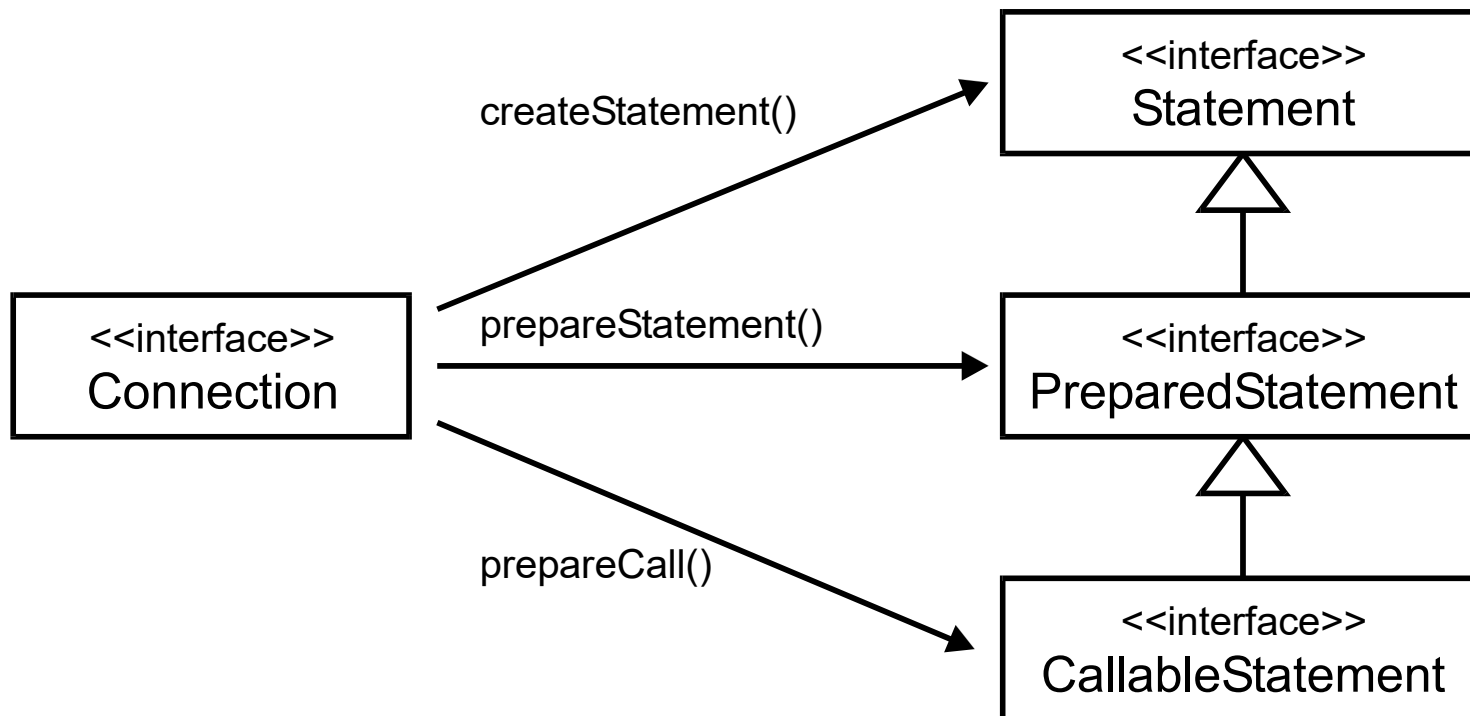
- Ausführen des Befehls

```
pstmt.executeUpdate();
```

- Freigeben der belegten Ressourcen

```
pstmt.close();
```

# Statements



# Lesender Datenbankzugriff (1)

- Instanzieren einer Anweisung

```
Statement stmt = dbc.createStatement();
```

- Absetzen der SQL-Anfrage (SELECT)

```
ResultSet rs = stmt.executeQuery("select VORNAME, NACHNAME"  
    + " from CUSTOMER WHERE GEBDAT > '1969-01-01'");
```

- Analyse des Ergebnisses

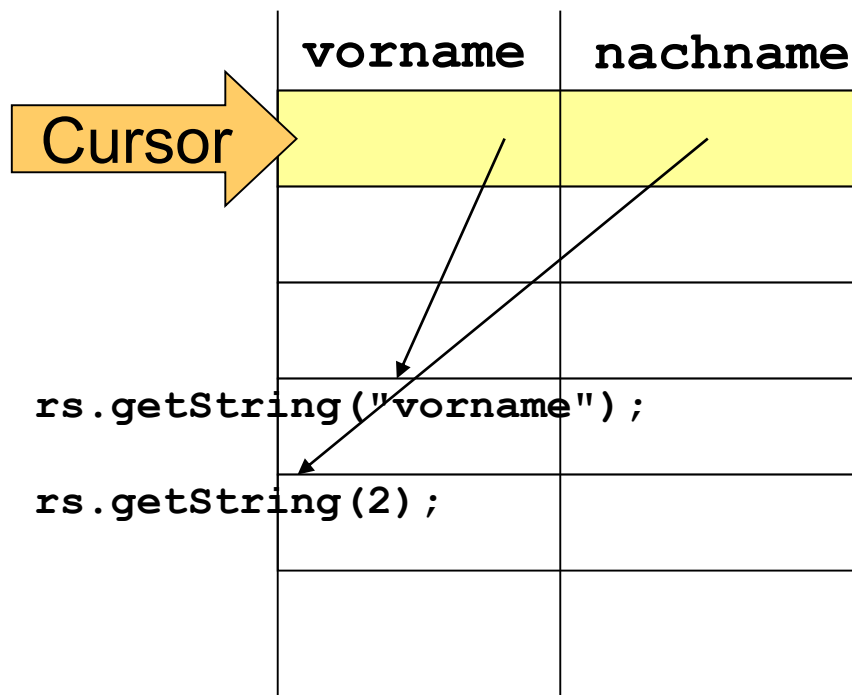
```
while (rs.next()) {  
    String vorname = rs.getString("vorname");  
    String nachname = rs.getString("nachname");  
    System.out.println("Vorname:" + vorname +  
        " Nachname: " + nachname);  
}
```

- Freigeben der belegten Ressourcen

```
rs.close();  
stmt.close();
```

## Lesender Datenbankzugriff (2)

```
ResultSet rs = stmt.executeQuery("select VORNAME, NACHNAME"  
    + " from CUSTOMER WHERE GEBDAT > '1969-01-01'");
```



- Auslesen über einen Cursor
- Bewegen des Cursors mit `next()`
- Auslesen der einzelnen Attribute mit `getXXX`-Methoden: `getInt`, `getString`, ...
- `getString` und `getObject` funktionieren bei jedem JDBC-Datentyp

# Datentypen

- Relationale Datenbanken haben eigenes Typsystem (z.B. `VARCHAR`, `INTEGER`, ...)
- JDBC-API: Abbildung SQL-Datentypen auf Java-Datentypen  
JDBC besitzt eine Standardzuordnung
- Die `setxxx`- bzw. `getxxx`-Methoden implementieren diese Abbildung ( `PreparedStatement` oder `ResultSet` )
- Innerhalb gewisser Grenzen sind Typumwandlungen möglich

## Datentypen (2)

SQL	Java-Datentyp	Get Methode	Set-Methode
CHAR	String	getString()	setString()
VARCHAR	String	getString()	setString()
INTEGER	Int	getInt()	setInt()
REAL	Float	getFloat()	setFloat()
DATE	java.sql.Date	getDate()	setDate()

# Transaktionen

- Automatisches Commit / manuelles Commit  
Zur manuellen Transaktionssteuerung muss das Automatische Commit ausgeschaltet werden:

```
dbc.setAutoCommit(false);
```

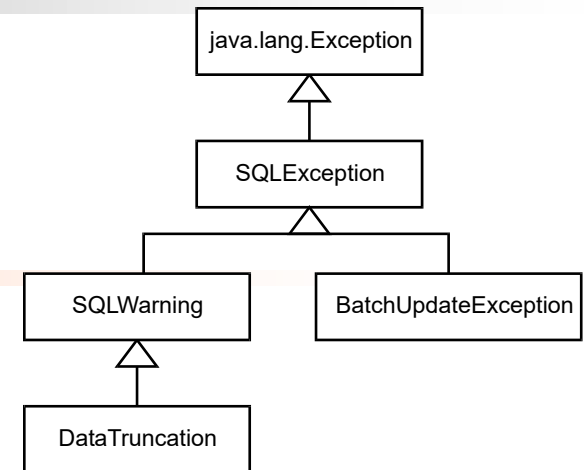
- Das automatische Commit kann wieder reaktiviert werden:

```
dbc.setAutoCommit(true);
```

- Transaktionen werden mit `commit` bestätigt und mit `rollback` verworfen:

```
if ( allesOK )  
    dbc.commit();  
else  
    dbc.rollback();
```

# Fehlerbehandlung



- Viele Quellen für mögliche Fehler: Netzwerkverbindung, SQL-Befehle, ...
- Fehler werden als `SQLException` dargestellt und müssen behandelt werden:

```
try {  
    // Statements, Connections, etc...  
}  
catch (SQLException ex) {  
    System.err.println("SQLException: " + ex.getMessage());  
}
```

- Unkritische Fehler: Darstellung als `SQLWarning`:

```
SQLWarning warn = stmt.getWarnings();
```



# Zusammenfassung:

## Besprechung eines vollständigen Beispiels

```
Statement s = null; Connection c = null;
try {
    Class.forName("org.h2.Driver").newInstance();
    c = DriverManager.getConnection("jdbc:h2:~/kundendb", "SA", "");

    s = c.createStatement();
    s.executeUpdate("insert into CUSTOMER values ("
        + "100, 'Bernd', 'Brot', '1971-09-21',"
        + "'36578', 'Osterode', 'Harzerstr. 7');");
} catch (Exception ex) {
    System.err.println("Exception:" + ex.getMessage());
}
finally { // ...
}
```

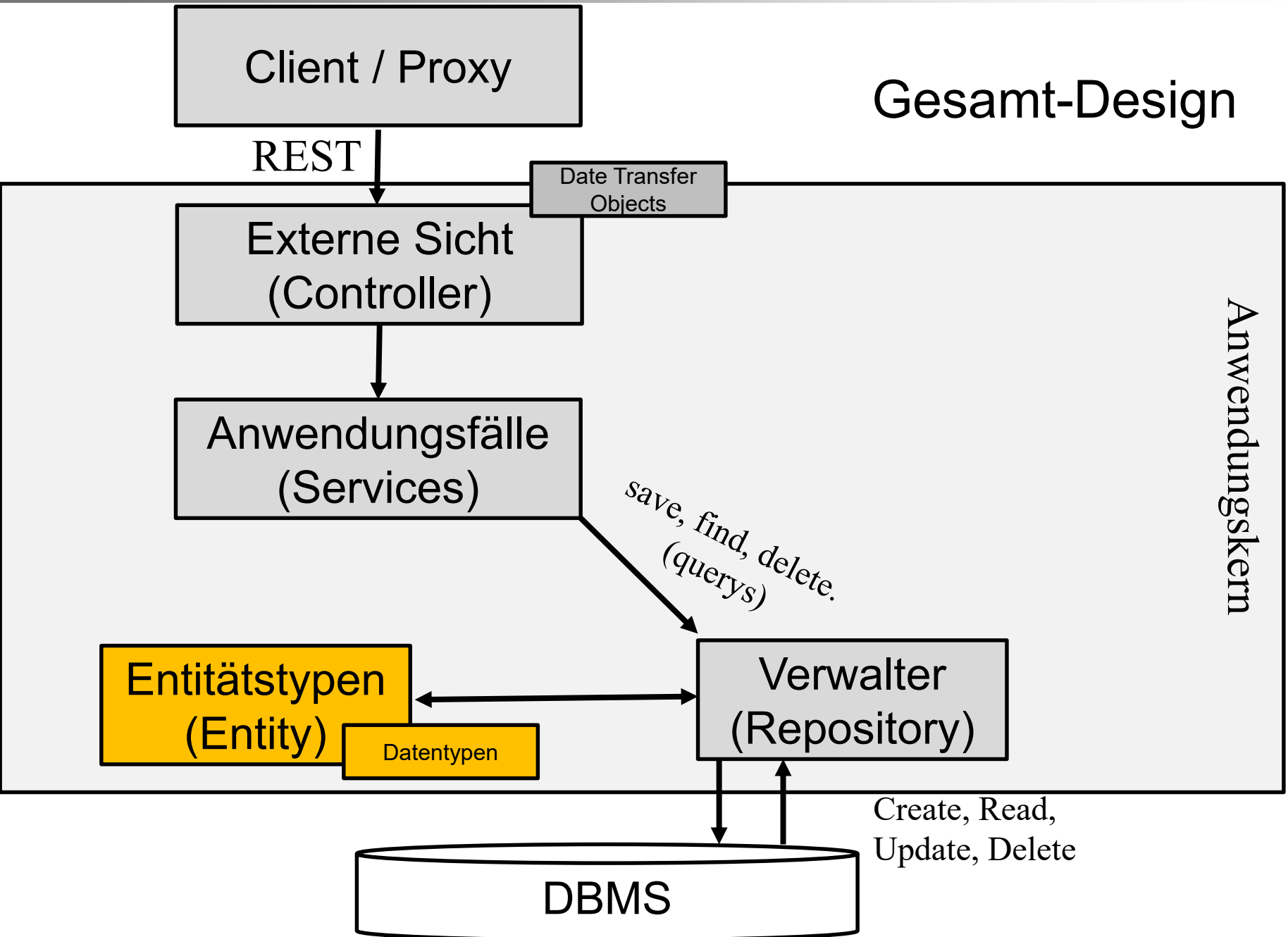
# Anwendungskern-Design

POJO lässt grüßen ...

# Ausprogrammiertes logisches Datenmodell

- = Domain Driven Design
- ***Datentypen (Pojo)***
  - Eigene Klasse pro fachlichem Datentyp, z.B. IBAN, Datum, Euro, Versicherungsnummer, ...
- ***Entitätstypen (Entity)***
  - Eigene Klasse pro Entitätstyp z.B. Kunde, Konto, Vertrag
  - 1:1 Übertragung der Beziehungen aus dem Modell
- ***Verwalter*** (= Data Access Objects, ***Repositorys***)
  - Kapseln das API der Datenbank, bei uns JPA
  - Bilden Entitäten auf Tabellen der Datenbank ab
- Anwendungsfälle (= Services)
  - Implementieren die Logik der Anwendungsfälle aus der Spez.

# Gesamt-Design



## Beispiel für einen Entitätstyp

```
public class Kunde {  
    private Long kundennummer;  
    private String vorname;  
    private String nachname;  
    private DateTime geburtsdatum;  
    private Adresse wohnort;  
    private List<Vertrag> vertraege;  
  
    ...  
}
```

Fachliches Datenmodell als Java-Klassen ausprogrammiert. Anwendungskern arbeitet mit Kunden, Verträgen, Konten etc.

# Beispiele für einen Datentypen

```
public class Adresse {  
    private String strasse;  
    private String plz;  
    private String stadt;  
  
    public Adresse(String strasse, String plz, String stadt) {  
        this.strasse = strasse;  
        this.plz = plz;  
        this.stadt = stadt;  
    }  
}
```

Strukturierte abhängiger Datentyp  
(ohne eigene Identität (PK))

```
org.joda.time.DateTime;
```

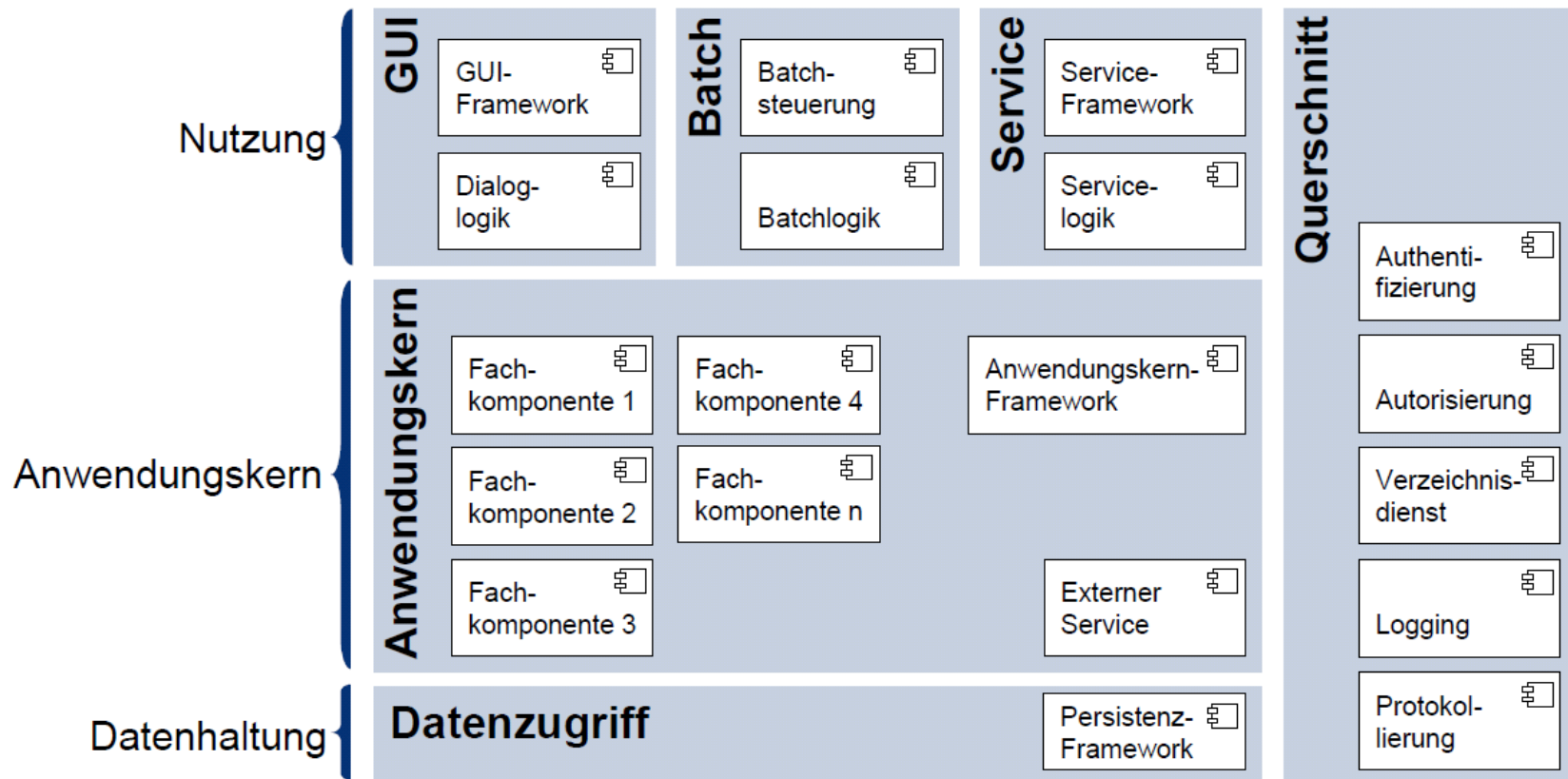
Intelligenter Datentyp

```
enum Vertragstyp {HAFTPFLICHT, UNFALL, HAUSRAT, LEBEN};
```

Aufzählung



# Drei-Schichten Architektur

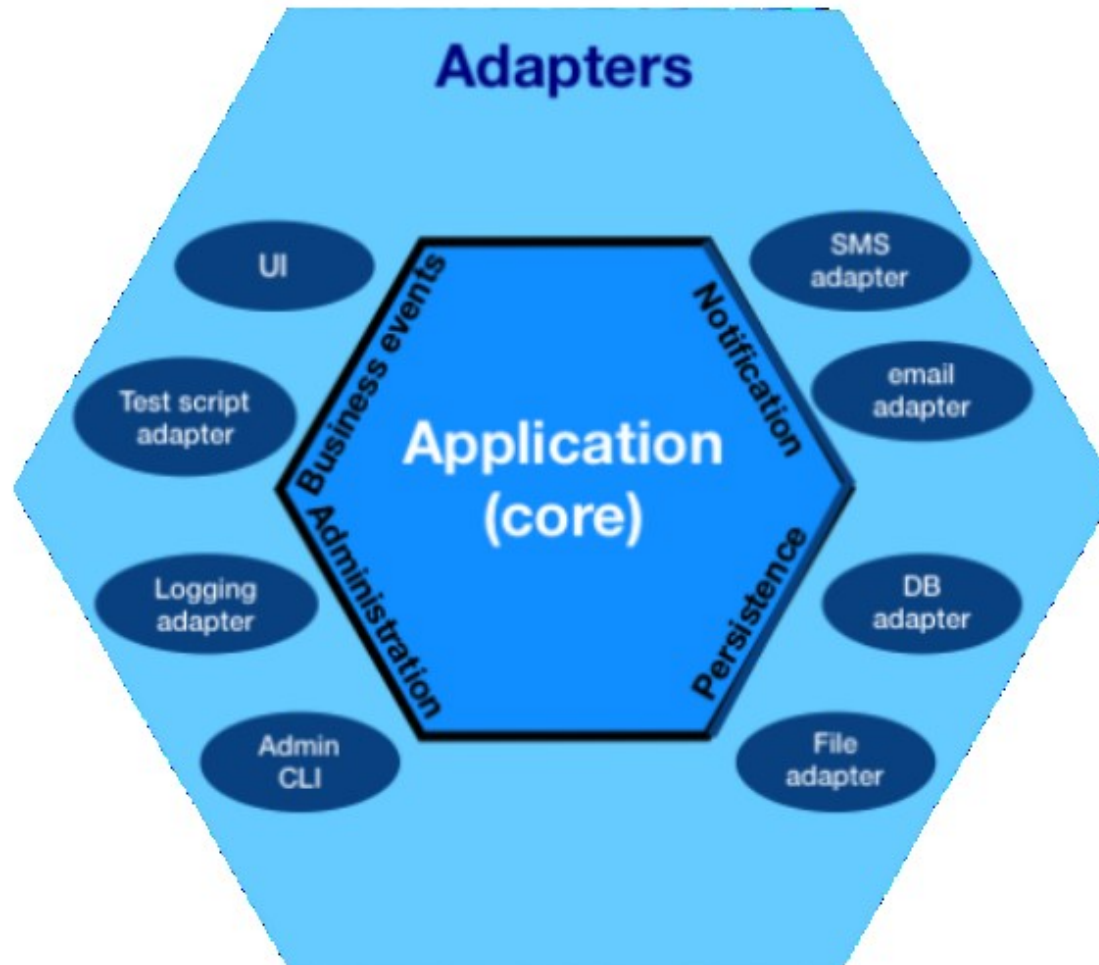


Originalquelle: Denert, Siedersleben: Software- Engineering. Methodische Projektabwicklung, Springer 1991  
 Abbildung aus Whitepaper zur Register Factory (Bundesverwaltungsamt)

# Hexagonal Architecture

Idee von A. Cockburn, Bild aus Wikipedia

[https://en.wikipedia.org/wiki/Hexagonal\\_architecture\\_\(software\)](https://en.wikipedia.org/wiki/Hexagonal_architecture_(software))





# Objekt / Relationales Mapping

## für den Zugriff auf

# Relationale Datenbanken

# Objektrelationaler Paradigmenbruch

- Objektorientierung und DBMS / Relationenalgebra:  
Große Unterschiede!
- Verschiedene elementare Datentypen
  - String unbegrenzter Länge vs. VARCHAR begrenzter Länge
  - java.util.Date / java.sql.Date / JodaTime vs. TIMESTAMP, DATE
  - Float / Double / Long / BigXXX vs. Flexible Zahlendarstellung im DBMS
  - List / Set / Map vs. ???
- Verschiedene Strukturen
  - Klasse vs. Tabelle (?)
  - Objektidentität vs. Primärschlüssel (?)
  - Objekt/Instanz vs. Zeile (?)
  - Vererbung vs. (?)
  - Referenzen: Assoziation / Aggregation / Komposition vs. Foreign Key (?)
  - Methoden vs. (Stored Procedures?)

# Objekt/Relationales Mapping

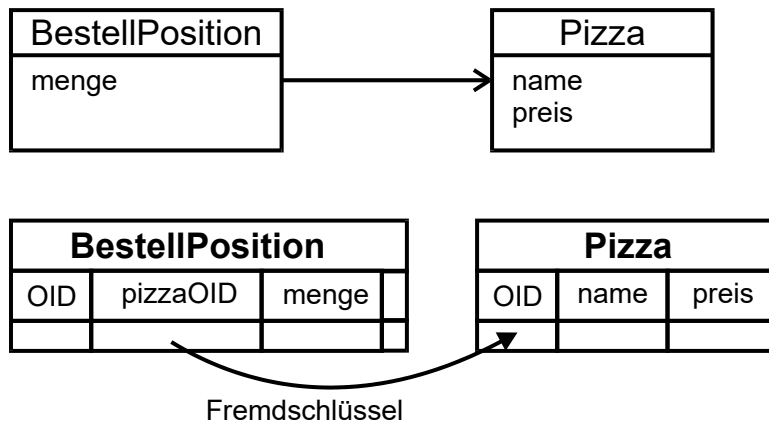
- Sie als Entwickler(in) entscheiden, wie bzw. ob Objekte auf Tabellen abgebildet werden!
  - Abhängig von der Größe / Lebensdauer der gebauten Software
  - Abhängig von den Zugriffsmustern auf die Objekte / das DBMS
- Einfachste Abbildung
  - Klasse <-> Tabelle
  - Attribut <-> Spalte / Attribut
  - Objektidentität <-> Primary Key
- Sonderfälle
  - Eine Klasse <-> Mehrere Tabellen (z.B. Head - Body Pattern)
  - Mehrere Klassen <-> Eine Tabelle (z.B. Abbildung 1:1 Relationen)

# Beziehungen über Referenzen

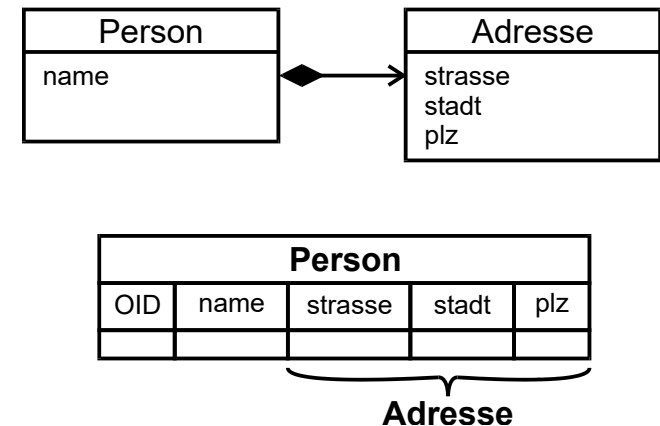
- Objektorientierte Beziehungen (etwas „ausdrucksstärker“ als ER-M)
  - Assoziation (Objekte kennen sich irgendwie)
  - Aggregation (Ein Objekt „hat“ ein anderes Objekt)
  - Komposition (Ein Objekt ist Teil eines anderen Objekts)
- Relationales DBMS: Referenzen über Foreign Keys nachbauen?
- Zu entscheiden: Navigation zwischen beiden Partnern bidirektional?

# Abbildung von 1:1 Beziehungen

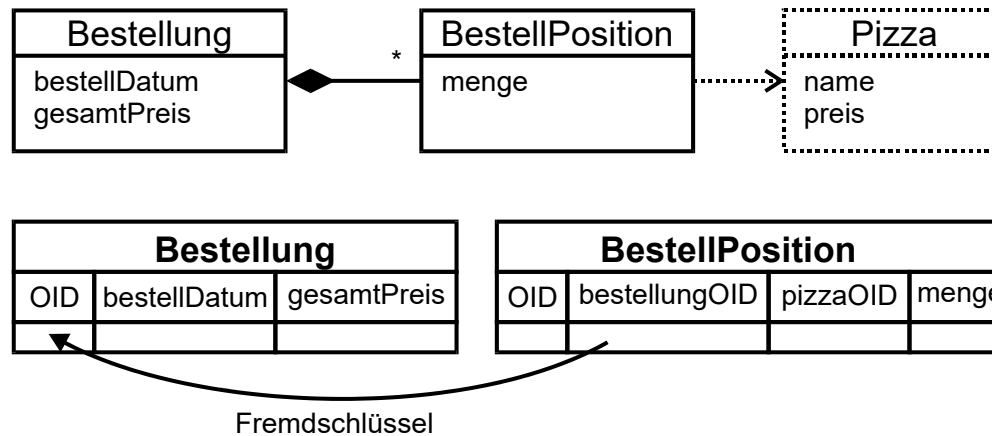
N:1 Beziehungen  
werden über  
Fremdschlüssel  
abgebildet



1:1 Komposition  
(Single Table  
Aggregation)



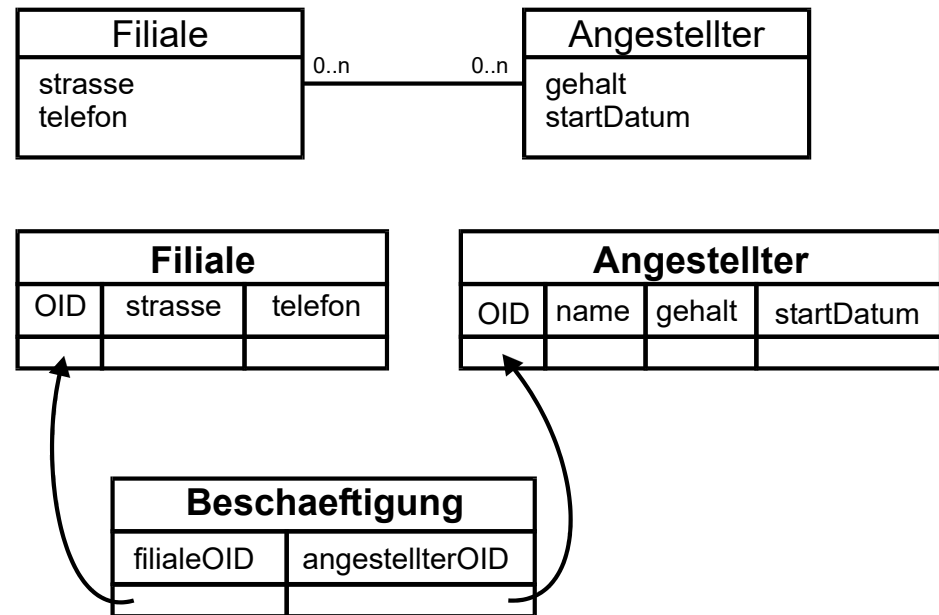
# Abbildung von 1:N Beziehungen



(Foreign Key Association)

# Abbildung von M:N Beziehungen

- Zusätzliche Fremdschlüssel-Tabelle, um die Beziehung darzustellen
- Fremdschlüssel jeweils in die Tabellen der beteiligten Entitäten
- = Association Table

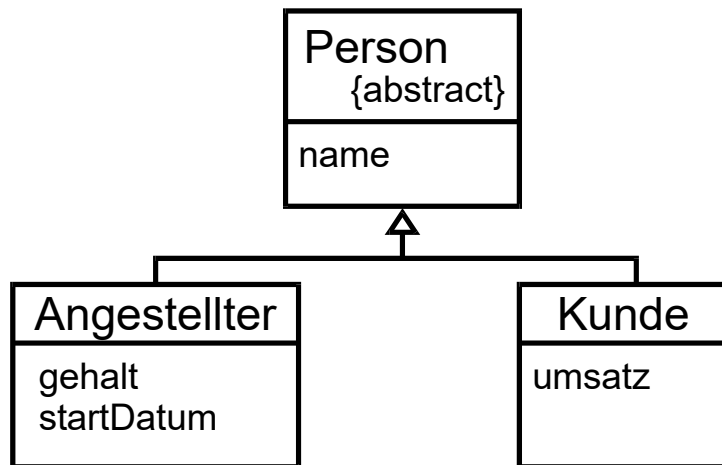


# Abbildung beliebiger Beziehungen: Zusammenfassung

- 1:1 Beziehung (idR. Komposition)
  - Abbildung idR. auf dieselbe Tabelle
  - Abbildung über Foreign Key wie bei abhängigen Entitäten
- 1:N Beziehung (Komposition/Aggregation/Assoziation)
  - Abbildung über eingebettete Fremdschlüssel (N-Teil/Tabelle „referenziert“ 1-Teil/Tabelle)
  - Abbildung über eigene Fremdschlüsseltabelle
  - Evtl. cascading delete bei Komposition
- M:N Beziehung (Aggregation / Assoziation)
  - Abbildung über eigene Fremdschlüsseltabelle
- Attribut-Tragende Beziehungen
  - Nur über Fremdschlüsseltabelle mit eigenen Attributen



# Abbildung von Vererbung



## 1. Typisierte Partitionierung

Person					
OID	name	gehalt	startDatum	umsatz	objTyp

## 2. Horizontale Partitionierung

Kunde		
OID	name	umsatz

Angestellter			
OID	name	gehalt	startDatum

## 3. Vertikale Partitionierung

Person		
OID	name	objTyp

Kunde	
OID (FS)	umsatz

Angestellter		
OID (FS)	gehalt	startDatum

# Wann nimmt man was?

(vgl. Keller: Mapping Objects To Tables, 1997)

Pattern	Performance			Space Consumption	Flexibility, Maintainability	Ad-hoc Queries
	Write/ Update	Single Read	Polymorphic Queries			
Single Table Aggregation	+	+	*	+	-	-
Foreign Key Aggregation	-	-	*	+	+	+
One Inheritance Tree One Table	<b>+o</b>	<b>+o</b>	<b>+</b>	-	+	+
One Class One Table	-	-	<b>-o</b>	+	+	-
One Inheritance Path One Table	+	+	-	+	-	-
Objects in BLOBs	<b>+o</b>	<b>+o</b>	<b>o</b>	+	-	-
Foreign Key Association	-	<b>o</b>	*	+	+	+
Association Table	-	<b>o</b>	*	+	+	+
+ good, - poor, * irrelevant, o depends, see detailed discussion						

# JPA

# Java Persistence API

# Datenbankzugriff in Java über JDBC und SQL

## Wollen wir das wirklich?

„In many ways this approach [OR-Mapping] treats the relational database *like a crazy aunt who's shut up in an attic* and whom nobody wants to talk about“

[M. Fowler, Patterns of Enterprise Application Architecture]

### ■ Zu lösende Probleme

- SQL sollte nicht im Quelltext des Programms verstreut werden (sonst z.B. Datenbankschema kaum noch änderbar)
- JDBC sollte nicht im Quelltext verstreut werden (Separation of Concerns)
- Optimierung des Zugriffs auf RDBMS?

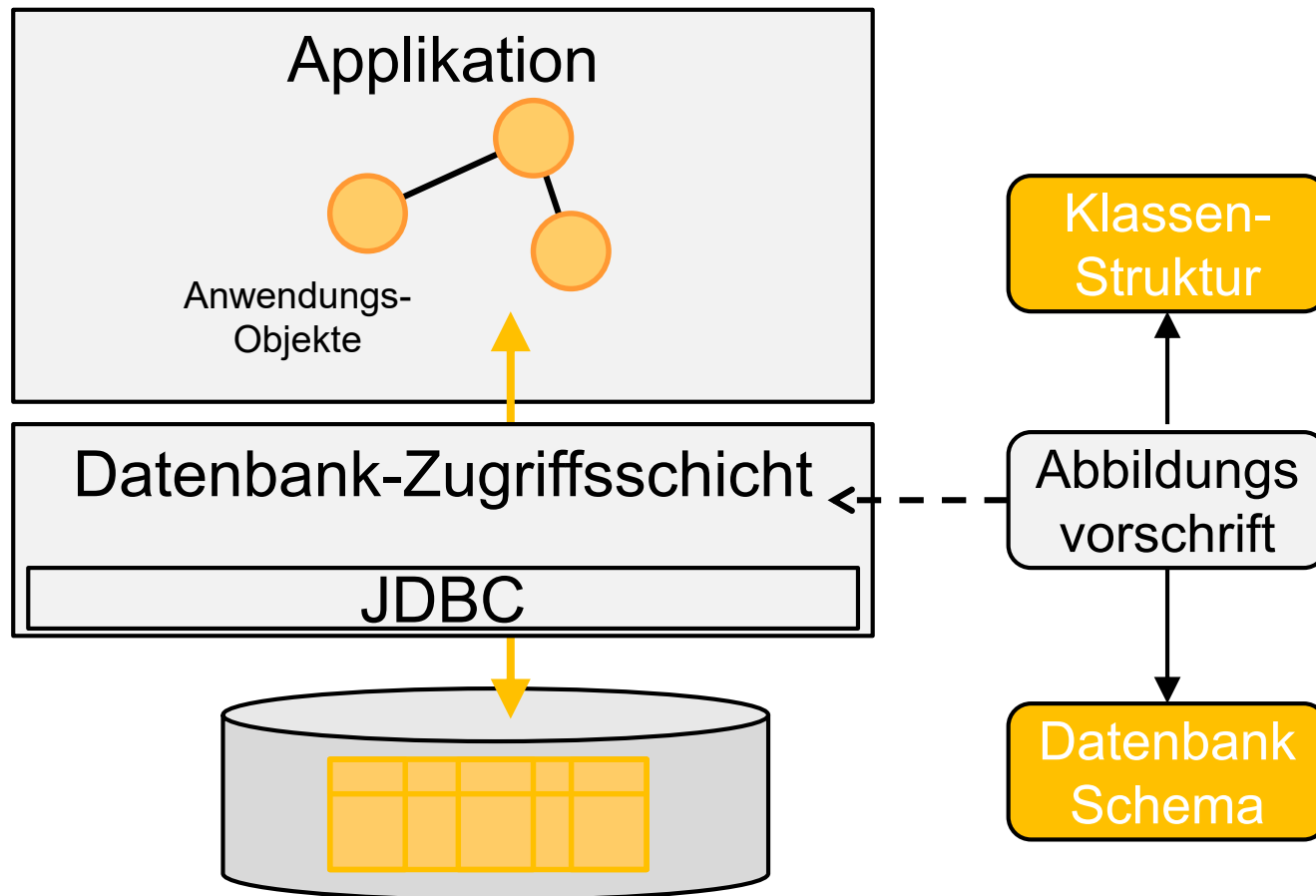
### ■ Zum Zitat:

- *Will der Programmierer mit Objekten des Anwendungskerns arbeiten? (Überwindung des OR-Paradigmenbruchs)*
- Verschiedene Designs des Anwendungskerns möglich (Domain Model, Transaction Script, Table Module)

# Motivation für JPA und Hibernate

- Mittels Low-level JDBC/ODBC lässt sich jede Persistenz-Notwendigkeit lösen –und diese ist immer schneller/effizienter als eine automatisierte Lösung.
- ABER: Programmcode ist aufwändig
  - fehleranfällig
  - (Programmierer-)Ressourcen fressend
  - Know-how intensiv (z.B. welche guten Cache-Alternativen existieren? welche Unterschiede zwischen Oracle, DB2, MS SQL,.. existieren? Etc.)
- Daher: Entwicklung von **Persistenz-Frameworks**
  - Standard für Java: **JPA** 2.x (javax.persistence.\*)
  - Referenzimplementierung JPA 2.x: **EclipseLink**
  - Orientieren sich an den Anforderungen der Programmierer
  - Marktführer bei Java: **Hibernate**
    - Hat JPA stark beeinflusst
    - Ist in den neuesten Versionen JPA kompatibel

# Persistenz-Frameworks: Grundsätzlicher Aufbau



# Hibernate - Eigenschaften

- Rahmenbedingungen
  - Erfüllt JPA 2.1 und EJB 3.1
  - Lizenz: Lesser GNU Public License
  - sehr weit verbreitet: > 2 Millionen Downloads in 5 Jahren
  - Unterstützte OO Sprachen: Java, C# (NHibernate)
  - Unterstützte DBMS: Oracle, DB2, MS SQL Server, Sybase, PostgreSQL, MySQL, HSQLDB, SAP DB, Informix, Interbase, Ingres, ...
- Legt mittels Meta-Daten fest
  - Welche Klassen in welche Tabellen abgebildet (gemapt) werden
  - Wie Objektidentität abgebildet wird
  - Wie Assoziationen abgebildet werden
  - Wie Vererbung abgebildet wird
  - ...

# Wie kommen Java, DBS und Mapping zustande?

## ■ Option 1: *Top down*

- Beginne mit dem *logischen Datenmodell* (ER-Modell) in Java. Erzeuge daraus die Mapping-Meta-Daten. Generiere daraus das DB-Schema.

## ■ Option 2: *Bottom up*

- Beginne mit dem *DB-Schema*. Reverse-Engineering (händisch/automatisch), um Mapping-Meta-Daten zu erzeugen. Generiere das Java-Business-Modell.

## ■ Option 3: *Middle out*

- Generiere sowohl die Java Klassen als auch das DB-Schema aus den Mapping-Meta Daten.



# Entitäten sind POJO (Plain Old Java Objects)

- JPA-Objekte sind normale Java-Objekte
  - Müssen kein Interface implementieren
  - müssen von keiner Oberklasse erben
  - Man sieht einer Klasse nicht an, dass sie persistent ist
  - Nötig i.allg. **JavaBeans-Konventionen**: Default Konstruktor, Getter, Setter
- Beispiel

```
public class Kunde {  
    private Long nummer;  
    public void setNummer(String nummer) {...}  
    public Long getNummer() {...}  
    Kunde() { // Default Konstruktor  
    }  
}
```

# Ergänzung der Mapping-Informationen als Annotations

**@Entity**

**@Table (name="CUSTOMER")**

```
public class Kunde {
```

**@Id**

**@GeneratedValue**

**@Column (name="kdnr")**

```
private Long    kundenummer;
```

**@Column (name="vorname")**

```
private String vorname;
```

```
...
```

```
}
```

**@Entity**: Objekte dieser Klasse können persistent gespeichert werden

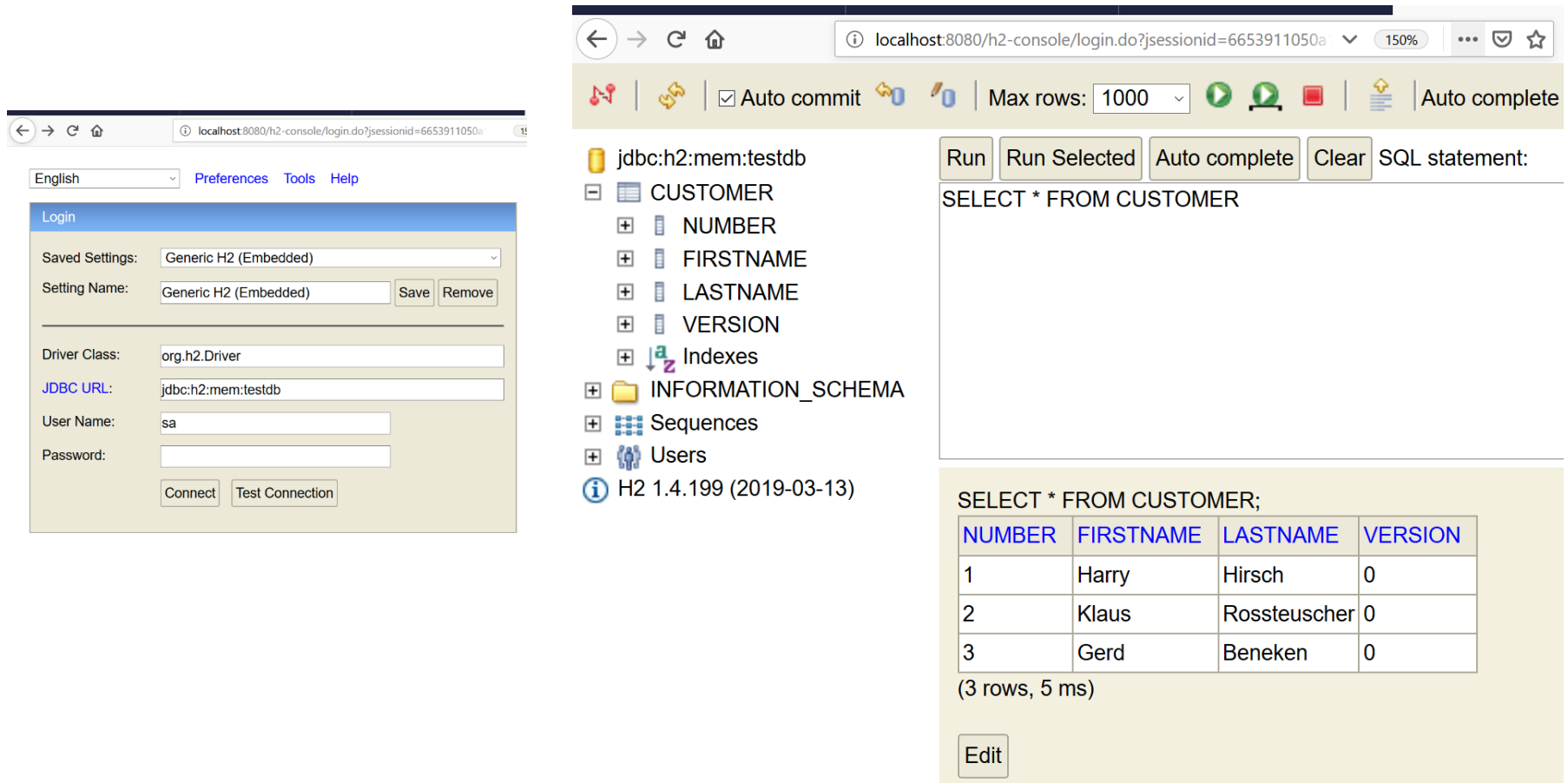
**@Table**: Optional: Tabelle kann angegeben werden, alternativ wird Klassenname genommen, bzw. Mapping-Datei verwendet

**@Id**: Primärschlüssel in der Tabelle

**@GeneratedValue**: JPA erzeugt eindeutige Werte

**@Column**: Optional: Spaltenname kann angegeben werden

# Daten in Spring Boot betrachten mit der h2-console: <http://localhost:8080/h2-console/>



The screenshot displays the H2 Console interface. On the left, the 'Login' tab is active, showing the configuration for the 'Generic H2 (Embedded)' database. The 'Driver Class' is set to 'org.h2.Driver', the 'JDBC URL' is 'jdbc:h2:mem:testdb', the 'User Name' is 'sa', and the 'Password' is empty. The 'Connect' and 'Test Connection' buttons are visible.

The main area shows the database structure for 'jdbc:h2:mem:testdb'. The 'CUSTOMER' table is expanded, showing columns: 'NUMBER', 'FIRSTNAME', 'LASTNAME', and 'VERSION'. Below the table structure, the 'H2 1.4.199 (2019-03-13)' version information is displayed.

On the right, the 'SQL statement' input field contains the query 'SELECT \* FROM CUSTOMER'. The 'Run' button is clicked, and the results are displayed in a table with 3 rows and 4 columns. The table shows the following data:

NUMBER	FIRSTNAME	LASTNAME	VERSION
1	Harry	Hirsch	0
2	Klaus	Rossteuscher	0
3	Gerd	Beneken	0

Below the table, the text '(3 rows, 5 ms)' indicates the execution time. An 'Edit' button is also present.

# JPA Annotationen

- **@Entity**: Angabe, welche Klasse persistent ist.
- **@Table**: Angabe des Tabellennamen (Default: wie Klasse)
- **@Id**: Angabe des Primärschlüssels.
- Default: alle Properties/Attribute der Klasse sind persistent
- **@Transient**: Kennzeichnen, dass Attribut NICHT persistent
- **@Column**: Angabe des Spaltennamens für eine Property (Default: wie Property) und vieler weiterer Eigenschaften
- **@ManyToOne**, **@OneToOne**, **@ManyToMany**, **@OneToMany**: Beziehungen zwischen Klassen

# Abhängige Entitäten / Objekte

```

@Entity
@Table(name="CUSTOMER")
public class Kunde {
    ...

    @Embedded
    private Adresse wohnort;

    ...
}

```

```

@Embeddable
public class Adresse {
    private String strasse;
    private String plz;
    private String stadt;

    ...
}

```

KDNr	GEBDAT	NACHNAME	VORNAME	PLZ	STADT	STRASSE
9	1965-10-22	Eberhard	Klaus	85636	Unterhaching	Steinstr 3

# 1:N Beziehungen abbilden

```

@Entity
@Table(name="CUSTOMER")
public class Kunde {
    @Id @GeneratedValue
    @Column(name="kdnr")
    private Long    kundennummer;

    @OneToMany(mappedBy="kunde")
    private List<Vertrag> vertraege;
    ...
}

```

```

@Entity
@Table(name="CONTRACT")
public class Vertrag {
    ...

    @ManyToOne
    private Kunde kunde;
    ...
}

```



VNR	TYP	KUNDE_KDNR
1234	0	9
4567	3	9

# Intelligente Datentypen abbilden

```
@Entity
public class Kunde {
    @Convert(converter=DateConverter.class)
    private LocalDate geburtsdatum;
    ...
}
```

```
@Converter
public class DateConverter
    implements AttributeConverter<LocalDate, String> {

    public String convertToDatabaseColumn(LocalDate datum) {
        return datum.toString();
    }

    public LocalDate convertToEntityAttribute(String datum) {
        return LocalDate.parse(datum);
    }
}
```

# Konfiguration in Spring Boot (Beispiel)

# H2

```
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password
spring.jpa.database-platform=
    org.hibernate.dialect.H2Dialect
```

# *H2-Console: http://localhost:8080/h2-console*

```
spring.h2.console.enabled=true
```

# JPA

```
spring.jpa.generate-ddl=true
spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.show-sql=true
```



# Konfiguration des EntityManagers

META-INF/persistence.xml

Spring Boot: application.properties

- Konfiguration von JPA über XML Datei
  - welches DBMS
  - welcher JDBC-Treiber (im Container eine DataSource)
  - welchen Server (Server-Instanz)
  - Authentifizierungsdaten
- Konfiguration von spezifischen Eigenschaften
  - Hibernate / EclipseLink – Properties

# Zugriff auf Datenbank über CrudRepository in Spring Boot

## CrudRepository<Customer, Long>

Long count() Zahl der Customer-Objekte, bzw. Zeilen in Tabelle

void delete(Customer c) Löscht den Kunden c

Void deleteAll() Löscht alle Kunden c

void deleteAll(Iterable<? extends Customer> customers)

void deleteById(Long pk) Löscht Customer mit Id PK

Boolean existsById(Long pk) Existiert der Customer mit Id pk

Iterable  
<Customer> findAll() Liefert alle Customer-Objekte

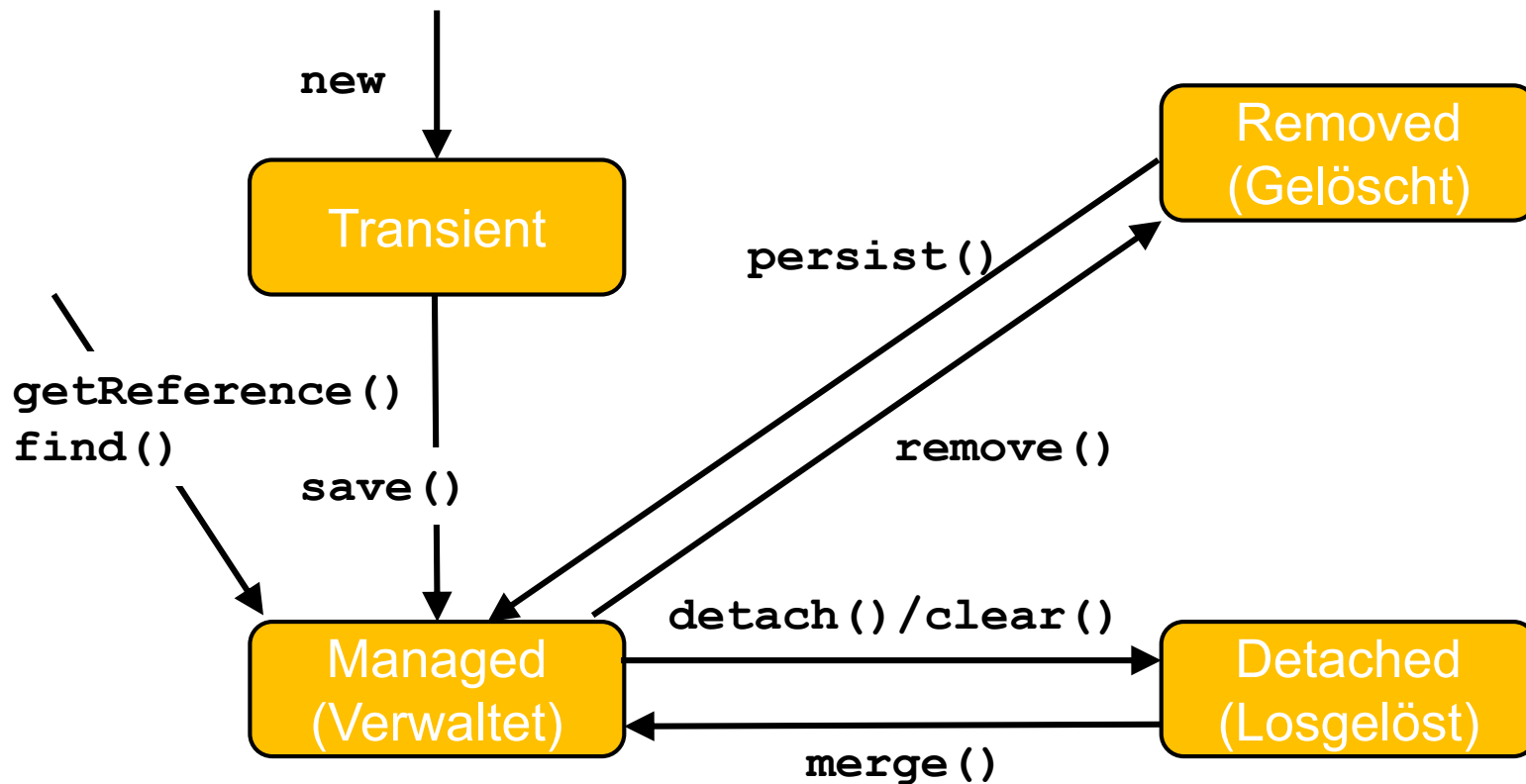
Iterable  
<Customer> findById(Iterable<Long> pks) liefert alle Customer mit den  
Ids pks

Optional  
<Customer> findById(Long pk) Liefert einen Customer über seine Id pk

<S extends  
Customer> S save(S entity) speichert Customer

<S extends  
Customer> saveAll(Iterable <S> entities) Saves all given entities.  
Iterable <S>

# Lebenszyklus einer Entität mit EntityManager (hier nicht relevant)



- = Zustandsmodell der Entitäten in Bezug auf EntityManager

# Queries in JPA

JPA bietet sehr viele Optionen für Queries

- „reines, low-level“ SQL
- JPA QL = „objektorientiertes SQL“ (verwendet Klassen- und Attributnamen, arbeitet mit Datentypen aus der Anwendung)

```
TypedQuery<Kunde> q = em.createQuery(  
    "select k from Kunde k where k.geburtsdatum > :aGebDat", Kunde.class);  
q.setParameter("aGebDat", new DateTime(1969,01,01,0,0,0));  
  
List<Kunde> neuerAls1969 = q.getResultList();  
  
for (Kunde fromDB : neuerAls1969) {  
    System.out.println("Neuer als 1969:" + fromDB);  
}
```

- Query Sprache mächtig: vgl. Dokumentation

# Named Queries Definition

Vordefinierte Queries  
sichern syntaktische  
Korrektheit und  
Wiederverwendbarkeit

```
@NamedQueries ({
    @NamedQuery (name=Kunde.FIND_ALL,
        query="SELECT k FROM Kunde k"),
    @NamedQuery (name=Kunde.FIND_BY_ID,
        query ="SELECT k FROM Kunde k WHERE kundennummer=:aKdnr")
})
@Entity
@Table (name="CUSTOMER")
public class Kunde {
    public static final String FIND_ALL = "Kunde.findAll";
    public static final String FIND_BY_ID = "Kunde.findById";
    // ...
}
```

# Named Queries Anwendung

Beispiel für Data Access  
Object, kapselt  
Datenbankanfragen

```
public class KundeDAO {  
    @Inject private EntityManager em;  
  
    public Kunde findByNumber(Long number) {  
        TypedQuery<Kunde> query =  
            em.createNamedQuery(Kunde.FIND_BY_ID, Kunde.class);  
        query.setParameter("aKdnr", number);  
        return query.getSingleResult();  
    }  
  
    public List<Kunde> findAll() {  
        TypedQuery<Kunde> query =  
            em.createNamedQuery(Kunde.FIND_ALL, Kunde.class);  
        return query.getResultList();  
    }  
}
```

# Pattern: Data Access Object

- Ideen:
  - Jeder *Entitätstyp wird als Klasse in Java* ausprogrammiert
  - Anwendungskern entspricht weitgehend dem ER-Modell / logischen Datenmodell  
(Fowler nennt dieses Design: „Domain Model“)
- DAO übernimmt die Abbildung der Entitätstypen auf Datenquelle (z.B. RDBMS), kapselt JPA, SQL oder JDBC
- DAO kümmert sich auch um die fachliche / technische Identität der geladenen Objekte

\*) Ausführliche Version: Siedersleben: Moderne Software-Architektur, dpunkt, 2004

# Pattern: Data Access Object

## Spring Boot: CrudRepository

```
public class KundeDAO { // Kapselt SQL / JDBC / OR-Mapping
    // ...
    public Kunde create( String vorname, String nachname,
                        DateTime geburtstag) { ... }
    public Kunde find(Integer id) { ... }
    public boolean update(Kunde k) { ... }
    public boolean delete(Kunde k) { ... }
}
```

```
public class Kunde{ // Rein fachliche Klasse
    private Integer kundennummer;
    private String vorname;
    private String nachname;
    private DateTime geburtstag;
    // ...
}
```