

UNIT 0x0D

ORM UND HIBERNATE

Motivation

Idee Object-Relational Mapping (ORM)

- Abbildung/Persistierung der Geschäftslogikobjekte in einem relationalen DBMS.

Q&A

- Vor- bzw. Nachteile?
- Wo liegen die Schwierigkeiten
- Wären andere DBMS besser?

Bekannte ORM

- **Hibernate** – Ein Beispiel.
- Eine Liste aktueller ORMs für unterschiedliche Programmiersprachen findet sich schnell online.

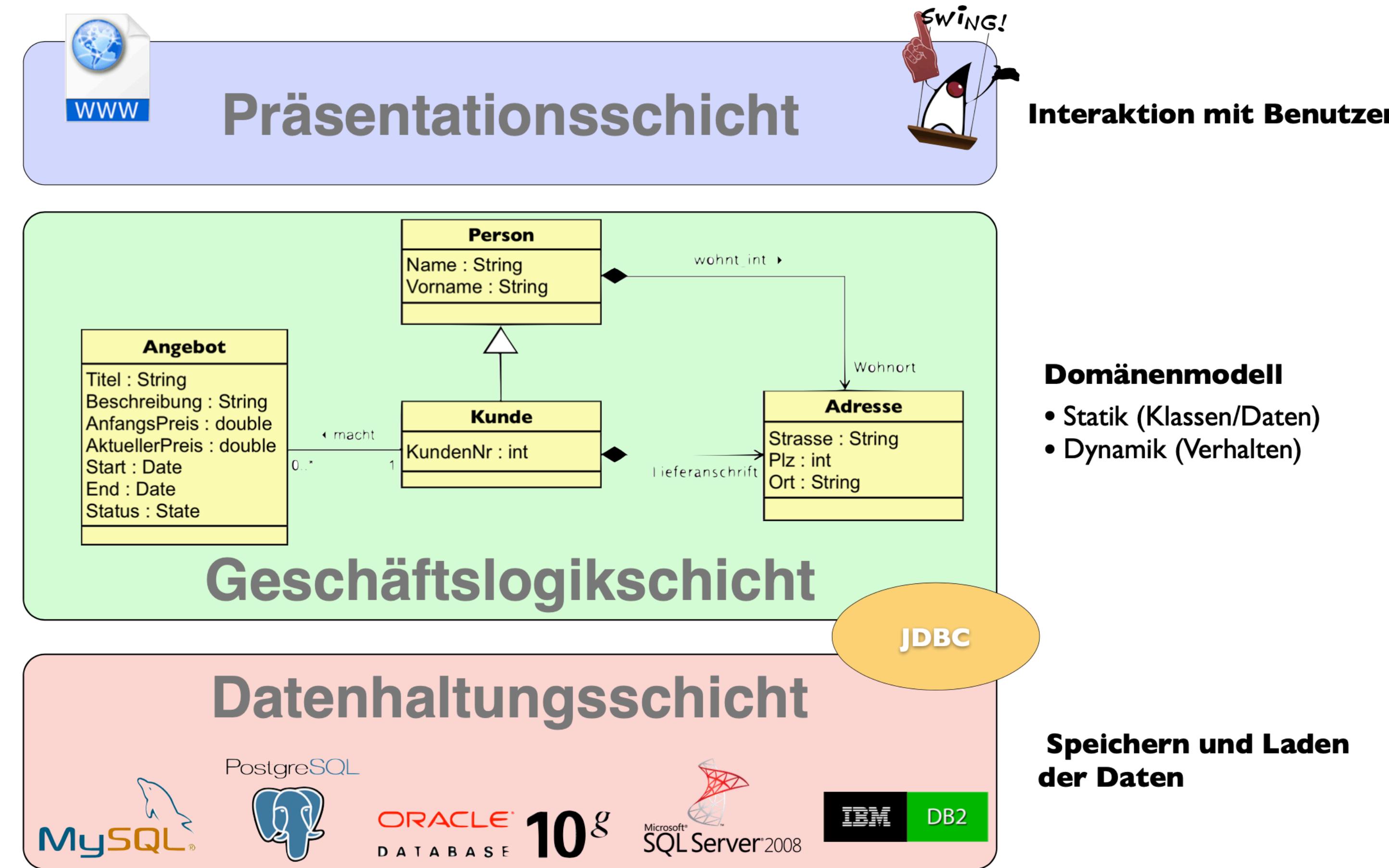
Hibernate

Hintergrund

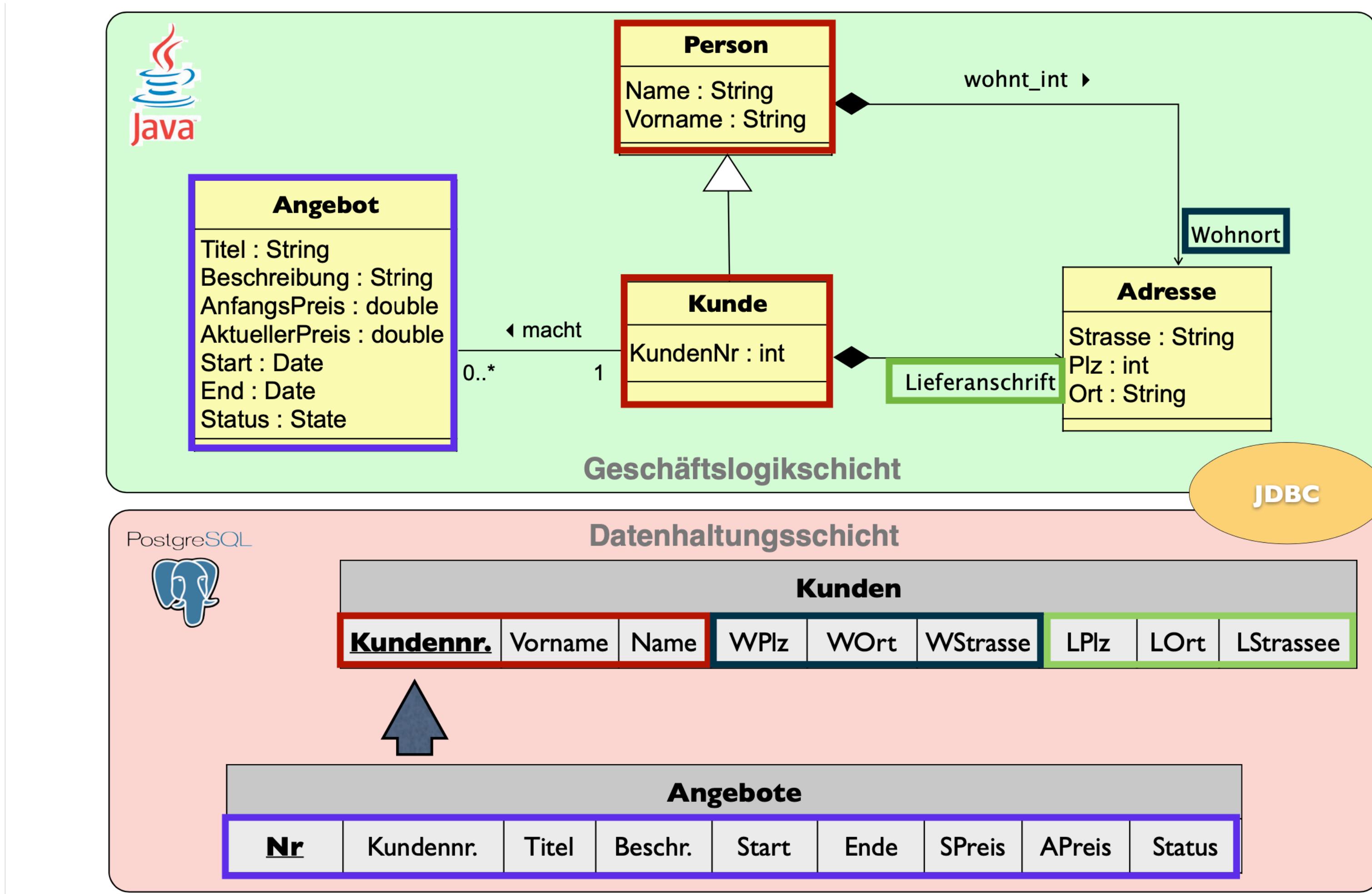
- Konfiguration
- Der Entity-Manager
- Lebenszyklus eines Entity-Objekts

3-Schichten-Modell

- Typisches Architekturmuster für JAVA-Applikationen



3-Schichten-Modell – Entwurf



3-Schichten-Modell – Implementierung

```

public class Kunde extends Person {
    protected Vector<Gegenstand> Angebote;
    protected Adresse LieferAnschrift;
    protected int KundenNr;
    /* Interaktion mit DBMS */
    public Kunde create(String Name,
                        String Vorname,
                        int KundenNr);
    public Kunde read(int KundenNr);
    public void update(Kunde p);
    public void delete(Kunde p);
}

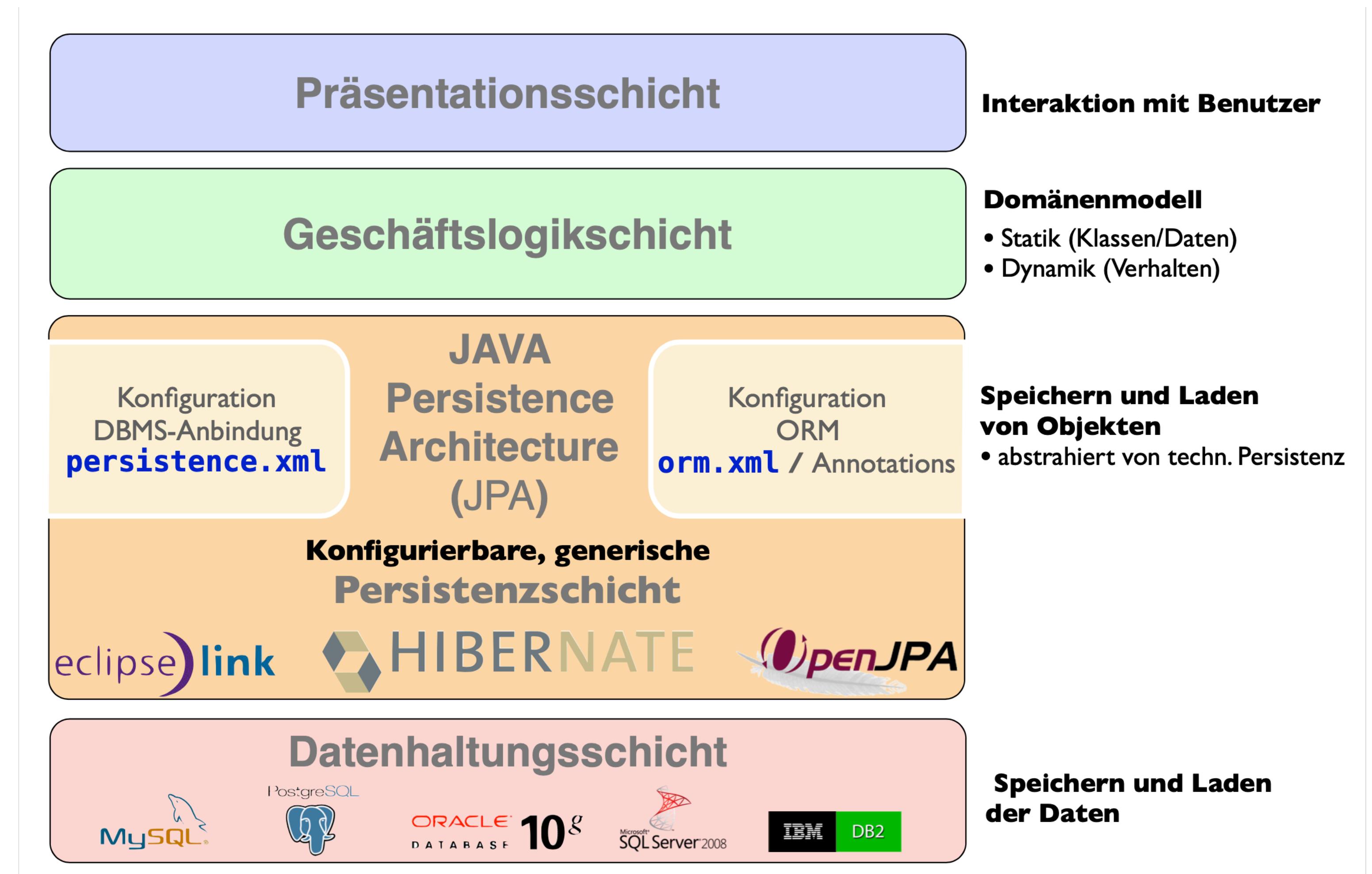
```

The diagram illustrates a UML Class Diagram. At the top is a pink-bordered box labeled 'Person' containing attributes 'Name : String' and 'Vorname : String'. Below it is a yellow-bordered box labeled 'Kunde' containing an attribute 'KundenNr : int'. A red-bordered box contains Java code for the Kunde class. A relationship line connects 'Person' to 'Wohnort' with the name 'wohnt_int'. Another relationship line connects 'Kunde' to 'Adresse' with the name 'Lieferanschrift'.

IBM
Redbooks
ca. 750 Seiten

- **Aspekte der Datenhaltungsschicht jetzt in Geschäftslogik!**
⇒ SQL in Geschäftslogik!
Verstoss gegen Prinzip „**Separation of Concerns**“:
- **Konsequenz:**
Pro **DBMS / Relationenmodell** eigene Geschäftslogikschicht
Änderungen an eigentlicher Geschäftslogik bedeutet Änderung in mehreren Implementierungen der Geschäftslogikschicht
⇒ Hohe Anforderungen an Konfigurationsmanagement

JAVA Persistence Architecture (JPA)



Hibernate – Konfiguration

XML

- Pro: Trennung der „Geschäftslogik“ und „Datenhaltung“.
- Contra: Dateien können sehr groß und unübersichtlich werden.

Annotationen

- Pro: Kürzer in der Formulierung, direkt am Code.
- Contra: Prinzip „Separation of Concerns“ verletzt, DBMS spezifisch.

Hibernate – Konfiguration

Beispiel

- **Konfiguration:**

Verzeichnis **META-INF** im Wurzelverzeichnis des **classpath** mit

- **persistence.xml** (Konfiguration der Persistenzschicht)
- **orm.xml** (optional - Beschreibung des Mappings)

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd" version="1.0">

    <persistence-unit name="Persistence_Test">
        <provider>org.hibernate.ejb.HibernatePersistence</provider>
        <properties>
            <!-- SQL-Dialekt -->
            <property name="hibernate.dialect"
                value="org.hibernate.dialect.PostgreSQLDialect" />

            <!-- Verbindung zum DBMS -->
            <property name="hibernate.connection.url"
                value="jdbc:postgresql://localhost/TestDB" />
            <property name="hibernate.connection.driver_class"
                value="org.postgresql.Driver" />
            <property name="hibernate.connection.username" value="testuser" />
            <property name="hibernate.connection.password" value="123" />
        </properties>
    </persistence-unit>
</persistence>
```

Hibernate – EntityManager

- **EntityManager:**

Verwaltet persistente Objekte (laden, speichern, ändern und löschen)

Stellt Persistenz-Kontext bereit (z.B. Cache)

Schnittstelle zur Persistenzschicht (z.B. Transaktionsmanagement)

```
EntityManagerFactory f = Persistence.createEntityManagerFactory("Persistence_Test");
EntityManager em      = f.createEntityManager();

/* Transaktion beginnen */
em.getTransaction().begin();
/* Laden von Objekten mithilfe von Primärschlüsselwerten */
Kunde k1 = em.find(Kunde.class, 1);
Kunde k2 = em.find(Kunde.class, 2);
/* Ändern persistenter Objekte -> Änderungen landen nach commit in DB */
k1.getAngebote().add( .... );
k1.getAngebote().get(2).setTitel( ... );

Person Jan = new Kunde( ... ); /* Neues Objekt anlegen ...
em.persist( Jan ); /* ... und unter Verwaltung des Entity-Managers stellen */
/* Löschen eines Objektes aus der Datenbank - k2 bleibt gültiges JAVA-Objekt */
em.remove( k2 );

/* Ändern: nach commit() werden alle Änderungen am Objektgraph persistent */
em.getTransaction().commit();

em.close(); f.close();
```

Lebenszyklus persistenter Objekte

- Objekte können in vier Zuständen sein

transient: Objekte, die mit new erzeugt wurden.

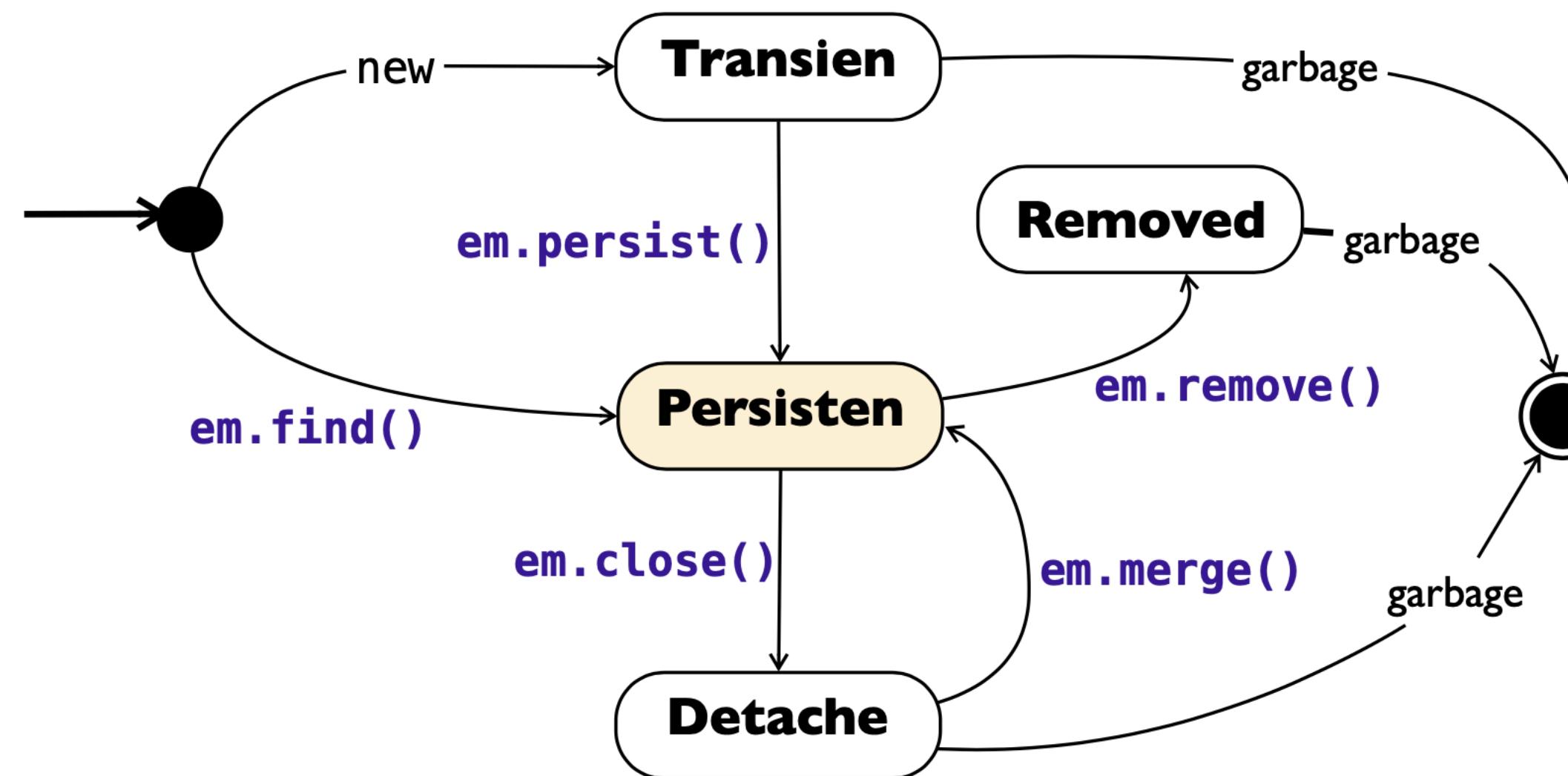
persistent: Objekte, die eindeutig einer Zeile in einer DB-Tabelle zugeordnet sind.

Änderungen am Objekt werden in der Datenbank sichtbar (und umgekehrt).

detached: Objekte, die eindeutig einer Zeile in einer DB-Tabelle zugeordnet sind.

Änderungen am Objekt werden **nicht** in Datenbank sichtbar

removed: Objekte, die aus Datenbank entfernt wurden.



In Anlehnung an: Christian Bauer, Gavin King: JAVA Persistence With Hibernate, S. 386

ORM – Mapping

Statisches Mapping

- Klassen
- Vererbung
- Aggregation, Komposition und Assoziation

Dynamisches Mapping

- Constraints und Trigger
- Performance: Dirty Checking, Lazy Fetching und Caching
- hier nicht behandelt.

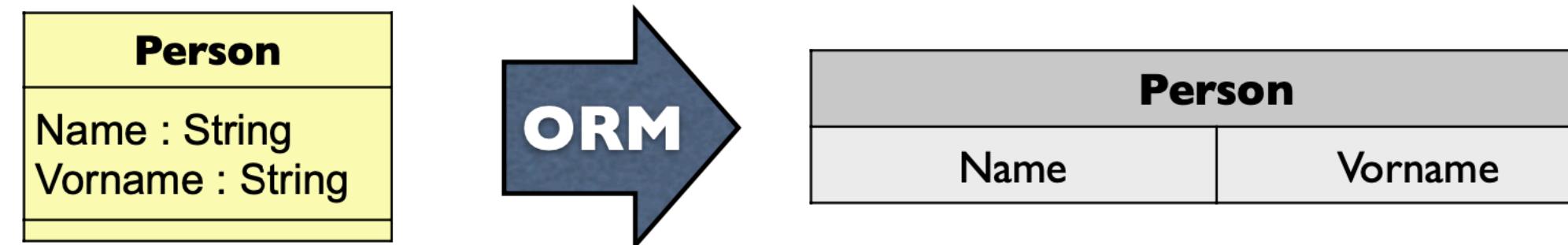
ORM – Grundvoraussetzungen

Entity-Klassen

- sind Klassen, deren Objekte durch die Persistenzschicht verwaltet werden können.
- Muss-Bedingungen:
 - entweder Annotation @Entity oder <entity>-Eintrag in orm.xml-Konfiguration.
 - Top-Level-Klasse (keine inner-class, kein interface und kein enum).
 - Default-Konstruktor (Konstruktor ohne Argumente, public oder protected).
 - Klasse und Attribute dürfen nicht final sein.
 - Primärschlüssel muss definiert sein.
- Kann-Bedingungen:
 - Implementierung des Interfaces „Serializable“.
 - Get- und Set-Methoden für zur speichernde Attribute.

Klassen – Basics

- Klassen werden direkt auf Tabellen abgebildet, wenn**
sie von keiner anderen Klasse (ausser Object) abgeleitet sind
sie nur primitive oder serialisierbare Attribute haben (keine Collections)



- Ein Objekt entspricht immer einer Zeile in einer Tabelle**
- Klassen- und Attributnamen werden übernommen**
- Typen werden automatisch zugeordnet**
Abhängig vom eingestellten SQL-Dialekt

B Type-Mapping in PostgreSQL

Java	PostgreSQL
<code>byte, short</code>	<code>int2</code>
<code>int</code>	<code>int4</code>
<code>long</code>	<code>int8</code>
<code>float</code>	<code>float4</code>
<code>double</code>	<code>float8</code>

Java	PostgreSQL
<code>boolean</code>	<code>bool</code>
<code>String</code>	<code>varchar(255)</code>
<code>Integer</code>	<code>int4</code>
<code>Double</code>	<code>float8</code>
<code>Serializable</code>	<code>bytea</code>

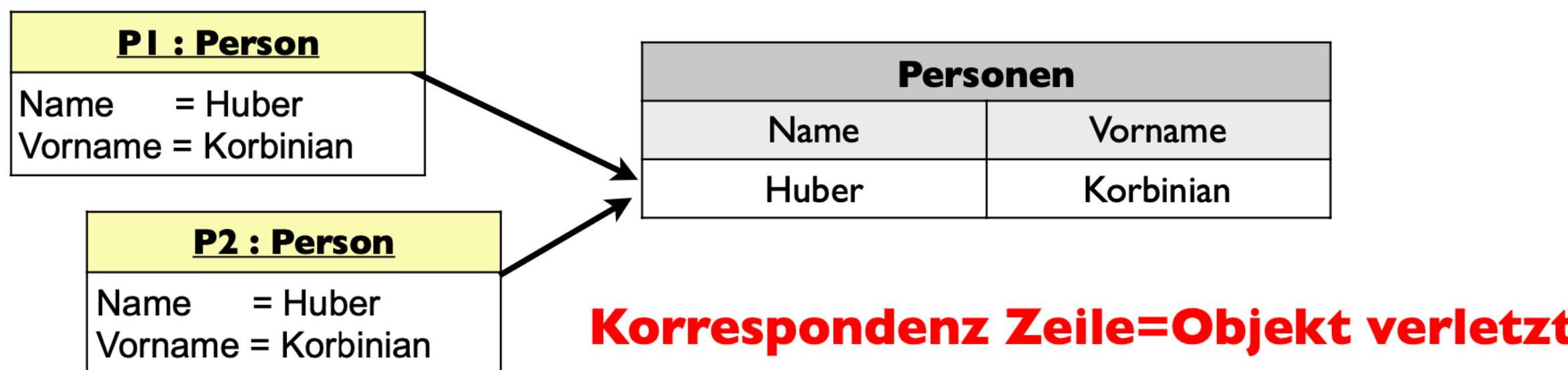
Klassen - Objektidentität

- **Problem: Objektidentität**

JAVA: Verschiedene Objekte dürfen gleiche Attributwerte haben

DBMS: Zeilen mit identischen Attributwerten nicht erlaubt

Daher: persistente Objekte mit gleichen Attributwerten nicht mehr unterscheidbar



- **Lösung: definiere Attribute, die Objekt eindeutig auszeichnen**
synthetische ID-Schlüssel (z.B. Sequenzen) mit neuem Attribut (z.B. int id)
fachlicher ID-Schlüssel (auch zusammengesetzt - dazu später)

Beispiel: ID-Schlüssel

B Entity-Klasse beschrieben durch Annotationen

```
@Entity  
public class Person {  
  
    @Id  
    int Id;  
  
    String Nachname;  
    String Vorname;  
  
    public Person() {}  
}
```



Person		
Id	Vorname	Nachname

B Entity-Klasse beschrieben in orm.xml

```
<entity-mappings  
    xmlns="http://java.sun.com/xml/ns/persistence/orm"  
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm  
    http://java.sun.com/xml/ns/persistence/orm_1_0.xsd" version="1.0">  
    <package>model</package>  
    <entity class="Person">  
        <attributes>  
            <id name="id" />  
        </attributes>  
    </entity>  
</entity-mappings>
```

Klassen – Synthetische ID-Schlüssel

- Können von **Applikation** verwaltet werden

Dann: neben @Id keine weiteren Angaben nötig!

Vorsicht! Applikation ist u.U. für Erzeugung von Schlüsseln verantwortlich

Mit new erzeugtes Objekt kann nur dann „persistent“ werden, wenn @Id noch nicht in DB!

B Probleme mit selbstverwalteten Schlüsseln

```
EntityManager em = ...;
em.getTransaction().begin();

/* Ok! Id 2 noch nicht in Tabelle */
Person Heidi = new Person("Heidi","von der Alm");
Heidi.id = 2;
em.persist( Heidi );

/* Fehler! Id 1 schon in Verwendung ... */
Person Sepp = new Person("Sepp","Huber");
Sepp.id = 1;
em.persist( Sepp );

em.getTransaction().commit();
```

Person		
ID	Vorname	Nachname
1	Hein	Bollo

- **Besser: Verwaltung durch DBMS oder Hibernate**

@GeneratedValue: mit Strategien Identity, Sequence, Table und Auto

Klassen - Synthetische ID-Schlüssel

- Können von **DBMS** verwaltet werden

DBMS erzeugt Schlüssel und sorgt für Eindeutigkeit (z.B. Auto-Increment-Felder)

Dann: ID-Attribut als @GeneratedValue mit Strategie IDENTITY markieren

B Datentyp „serial“ für synthetische Schlüssel mit PostgreSQL

Effekt der Verwendung von serial:

```
CREATE SEQUENCE person_id_seq;

CREATE TABLE Person (
    Id integer DEFAULT nextval(person_id_seq) NOT NULL,
    Vorname varchar(255),
    Nachname varchar(255),
    CONSTRAINT pk_Person PRIMARY KEY (Id)
);
```

```
@Entity
public class Person {
    @Id /* Nur kompatibel für echten „IDENTITY“-Typ in DB! */
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    int Id;
    String Nachname;
    String Vorname;

    public Person() {}
}
```

Klassen - Synthetische ID-Schlüssel

- Können von **Hibernate** verwaltet werden

Hibernate erzeugt Schlüssel und sorgt für Eindeutigkeit

Strategie wird mit Annotation `@GeneratedValue` z.B. auf SEQUENCE oder TABLE gesetzt

B Synthetischer Schlüssel mittels SEQUENCE

Ohne Verwendung von `serial`:

```
CREATE SEQUENCE person_id_seq;

CREATE TABLE Person (
    Id integer NOT NULL, /* Beachte! DEFAULT-Angabe fehlt */
    Vorname varchar(255),
    Nachname varchar(255),
    CONSTRAINT pk_Person PRIMARY KEY (Id)
);
```

```
@Entity
public class Person {
    @Id
    @SequenceGenerator(name="person_seq", sequenceName="person_id_seq")
    @GeneratedValue(strategy=GenerationType.SEQUENCE, generator= "person_seq")
    int Id;
    String Nachname;
    String Vorname;

    public Person() {}
}
```

Klassen - Namen, Typen und transiente Attribute

- **Standardmäßig (ohne Annotation)**

Tabellenname entspricht Klassenname; Spaltenname entspricht Attributname

Typzuordnung automatisch (siehe Folie zu Basics)

Alle Attribute werden abgebildet - **Ausnahme:** Markierung mit @Transient

B Mapping von Typen / Namen für Felder und Namen für Klassen

```
CREATE DOMAIN german_plz AS text  
CONSTRAINT german_plz_check CHECK ((VALUE ~ '^\d{5}$'::text));
```

Kunde			
Id:serial	Vorname:text	Nachname:text	WPlz : german_plz

```
@Entity @Table(name="Kunde")/* Name-Map! */  
public class Person {  
    @Id @GeneratedValue  
    int Id;  
    /* Name- und Type-Map! Plz kommt aus Feld WPlz */  
    @Column(name="WPlz", length=5, columnDefinition="german_plz")  
    String Plz;  
    String Nachname;  
    String Vorname;  
    /* Nur zur Laufzeit relevant */  
    @Transient  
    boolean activeTransaction;  
    public Person() {}  
}
```

Klassen – Aufgespaltene Relationen

- Entities können Ergebnis eines Joins repräsentieren

Join-Attribut muss definiert sein (nur 1:1 Beziehung)

Für Attribute muss ggf. angeben werden, aus welcher Tabelle sie kommen

B | 1:1 Beziehung mit @SecondaryTable

Person		
Id	Vorname	Nachname

Adresse				
Id	Plz	Ort	Strasse	Person

Fremdschlüssel

```
@SecondaryTable(name="Adresse",
    pkJoinColumns=@PrimaryKeyJoinColumn(name="Person"))
public class Person {
    @Id @GeneratedValue
    int Id;
    String Vorname;
    String Nachname;
    @Column(table="Adresse")
    String Plz;
    @Column(table="Adresse")
    String Ort;
    @Column(table="Adresse")
    String Strasse;
    public PersonA() {}
}
```

Vorsicht!
Geht so nur für 1:1 Beziehungen

Vererbung – Einfachster Fall

- **Der einfachste Fall:**

Oberklasse nur zur Modellierung (z.B. abstrakte Klasse)

Objekte sollen **nicht** gespeichert werden

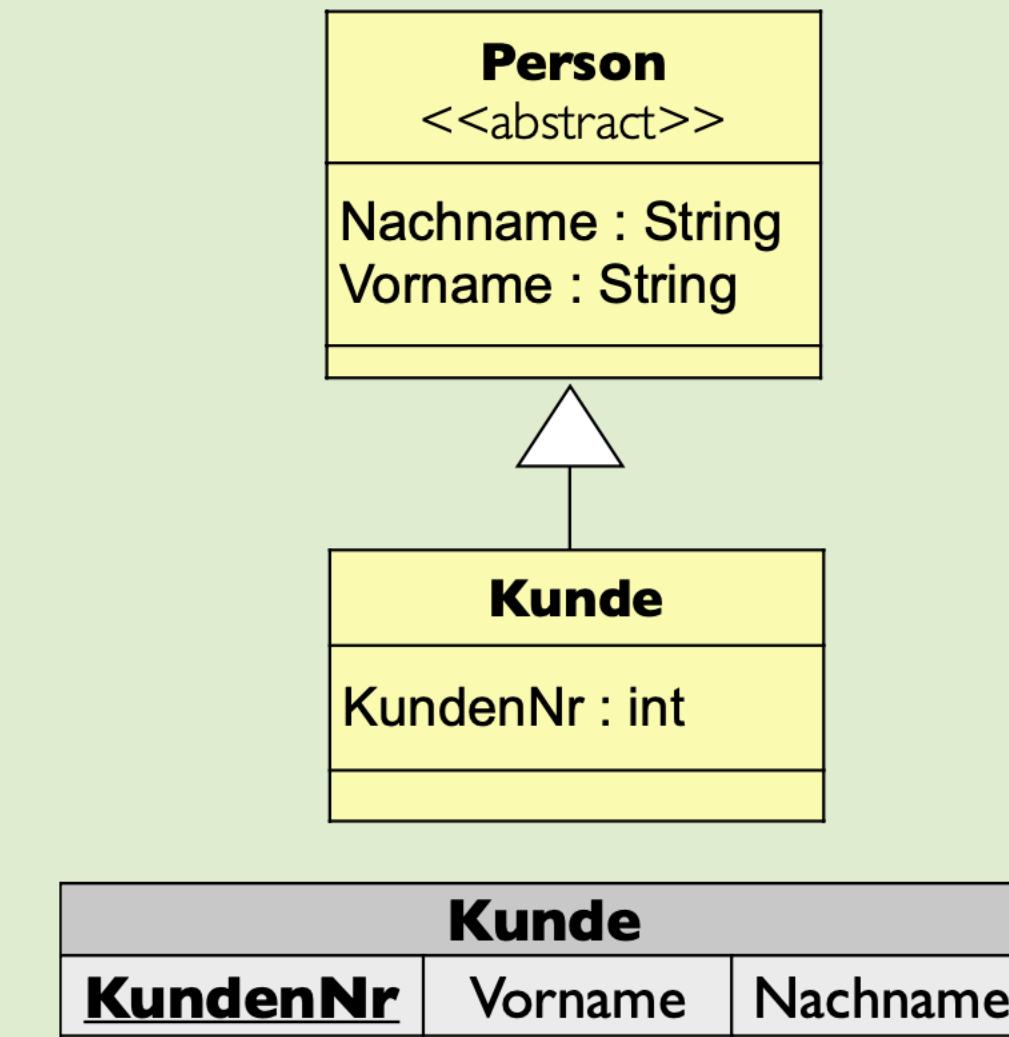
Dann: markiere Oberklasse mit `@MappedSuperclass` und Attribute wie gewohnt

B Vererbung ohne Speicherung von Objekten der Superklasse

```
@MappedSuperclass
public abstract class Person {
    /* Beachte: @Id nicht erforderlich! */
    String Nachname;
    String Vorname;
}

/* „Normale“ Entity-Klasse */
@Entity
public class Kunde extends Person {
    @Id @GeneratedValue
    int KundenNr;

    public Kunde() {}
}
```



Vererbung – Strategien

- **Strategie wird in Wurzel der Entity-Hierarchie gewählt**

Annotation hierzu: **@Inheritance**

Kann in abgeleiteten **nicht** mehr gewechselt werden

Darüber nur **@MappedSuperclass** erlaubt

**Entity-Hierarchie ist
Auschnitt aus
Klassenhierarchie**

- **Drei Varianten pro Klassenhierarchie**

TABLE_PER_CLASS Jede Klasse in eine Tabelle
(**mit** geerbten Attributen)

JOINED Jede Klasse in eine Tabelle
(**ohne** geerbte Attribute, Fremdschlüsselbeziehung zu Oberklasse)

SINGLE_TABLE Alle Klassen in einer Tabelle
(Diskriminator-Spalte zur Bestimmung des Objekt-Typs)

Polymorphe Abfragen

- **Polymorphe Abfragen**

Polymorphe Abfragen berücksichtigen Objekte von Kindklassen

B Polymorphe „select“

```

classDiagram
    class Person {
        Id : int
        Name : String
        Vorname : String
    }
    class Freelancer {
        Available : bool
    }
    class Angestellter {
        Gehalt : double
        Eintritt : Date
    }
    Freelancer <|-- Person
    Angestellter <|-- Person
  
```

The diagram illustrates a polymorphic query scenario. It shows a base class **Person** with attributes **Id**, **Name**, and **Vorname**. Two subclasses inherit from it: **Freelancer** (with attribute **Available**) and **Angestellter** (with attributes **Gehalt** and **Eintritt**). A polymorphic query **select p from Person p;** is demonstrated, which returns three results corresponding to the objects Hein, Erik, and Jan.

:Person		:Person		:Person	
Hein : Person	Id = 1 Name = Bollo Vorname = Hein	Erik : Freelancer	Id = 2 Name = Roth Vorname = Erik Available = true	Jan : Angestellter	Id = 3 Name = Cux Vorname = Jan Gehalt = 3217,55 Eintritt = 1.5.2003

select p from Person p;

:Person		:Person		:Person	
Hein : Person	Id = 1 Name = Bollo Vorname = Hein	Erik : Freelancer	Id = 2 Name = Roth Vorname = Erik	Jan : Angestellter	Id = 3 Name = Cux Vorname = Jan

Vererbung – TABLE_PER_CLASS

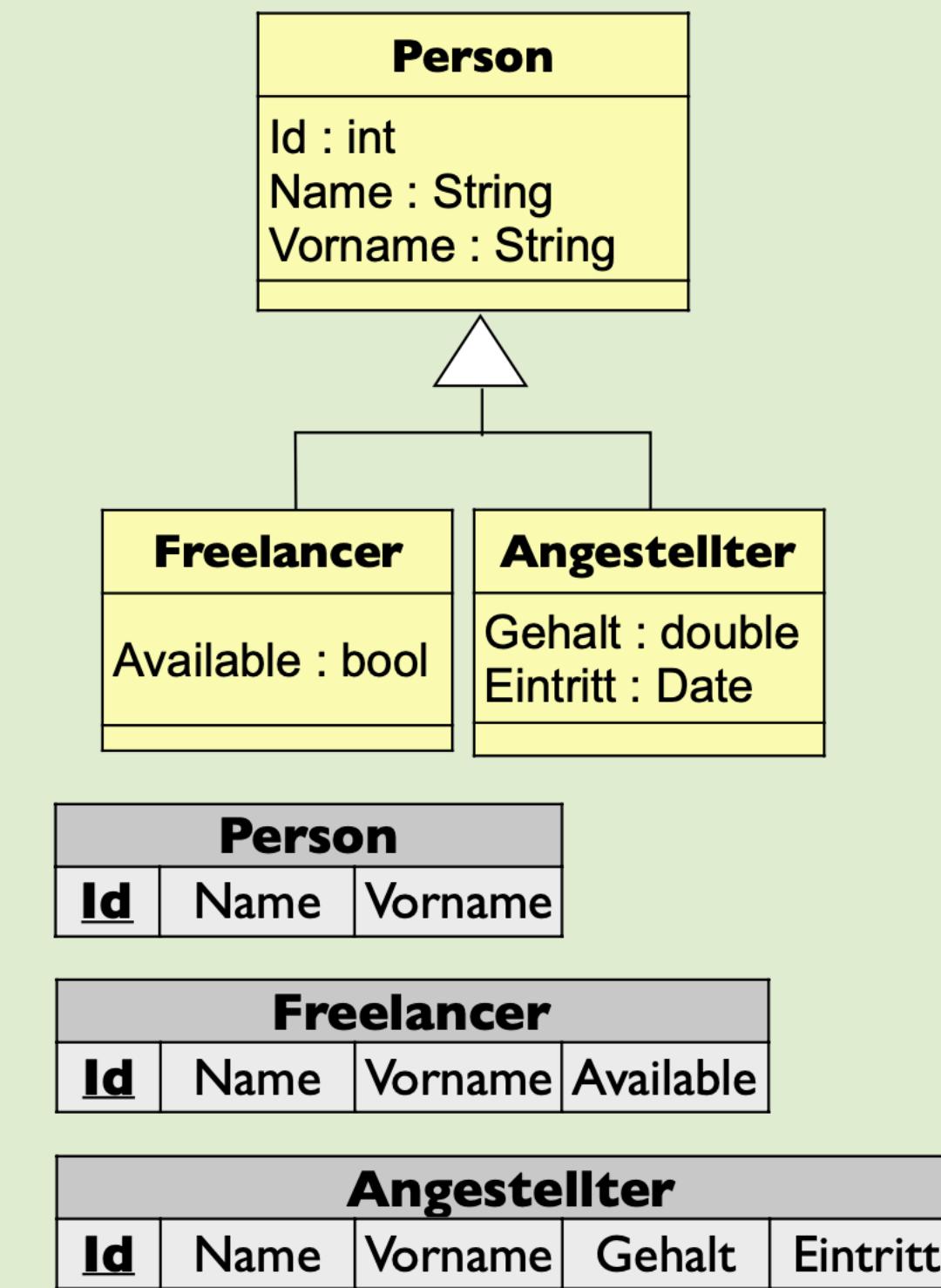
B Anwendung von TABLE_PER_CLASS

```

@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public class Person {
    @Id @GeneratedValue
    int id;
    String Name;
    String Vorname;
    public Person() {}
}
/* Beachte: @Id wird geerbt */
@Entity
public class Freelancer extends Person {
    boolean Available;
    public Freelancer() {}
}

@Entity
public class Angestellter extends Person {
    double Gehalt;
    Date Eintritt;
    public Angestellter() {}
}

```



- Vererbung datenbankseitig nicht erkennbar

Polymorphes Select – TABLE_PER_CLASS

B Polymorphe Anfrage - TABLE_PER_CLASS

Hein : Person	Person
Id = 1 Name = Bollo Vorname = Hein	Id Name Vorname 1 Bollo Hein

Jan : Angestellter	Freelancer
Id = 3 Name = Cux Vorname = Jan Gehalt = 3217,55 Eintritt = 1.5.2003	Id Name Vorname Available 2 Roth Erik true

Erik : Freelancer	Angestellter
Id = 2 Name = Roth Vorname = Erik Available = true	Id Name Vorname Gehalt Eintritt 3 Cux Jan 3217,55 152.003

```
select p from Person p;
```

```

SELECT Id,Name,Vorname FROM Person UNION
SELECT Id,Name,Vorname FROM Freelancer UNION
SELECT Id,Name,Vorname FROM Angestellter;

```

Bewertung – TABLE_PER_CLASS

Person		
Id	Name	Vorname
1	Bollo	Hein

Freelancer			
Id	Name	Vorname	Available
2	Roth	Erik	true

Angestellter				
Id	Name	Vorname	Gehalt	Eintritt
3	Cux	Jan	3217,55	152.003

- **Polymorphes select und update**
 - Select: langsam, drei selects (u.U. mit Projektionen) oder ein select mit union
 - Update: langsam, mehrere Tabellen betroffen
- **Monomorpher Zugriff**
 - schnell, alle Daten in einer Tabelle
- **Änderungen am Design**
 - Neue Klasse: unproblematisch, falls Blatt in Hierarchie
 - Neues Attribut: Tabellen aller Kindklassen müssen geändert werden
- **Sonstiges**
 - passt gut zur Philosophie „Eine Entity-Klasse entspricht einer Tabelle“

Vererbung – JOINED

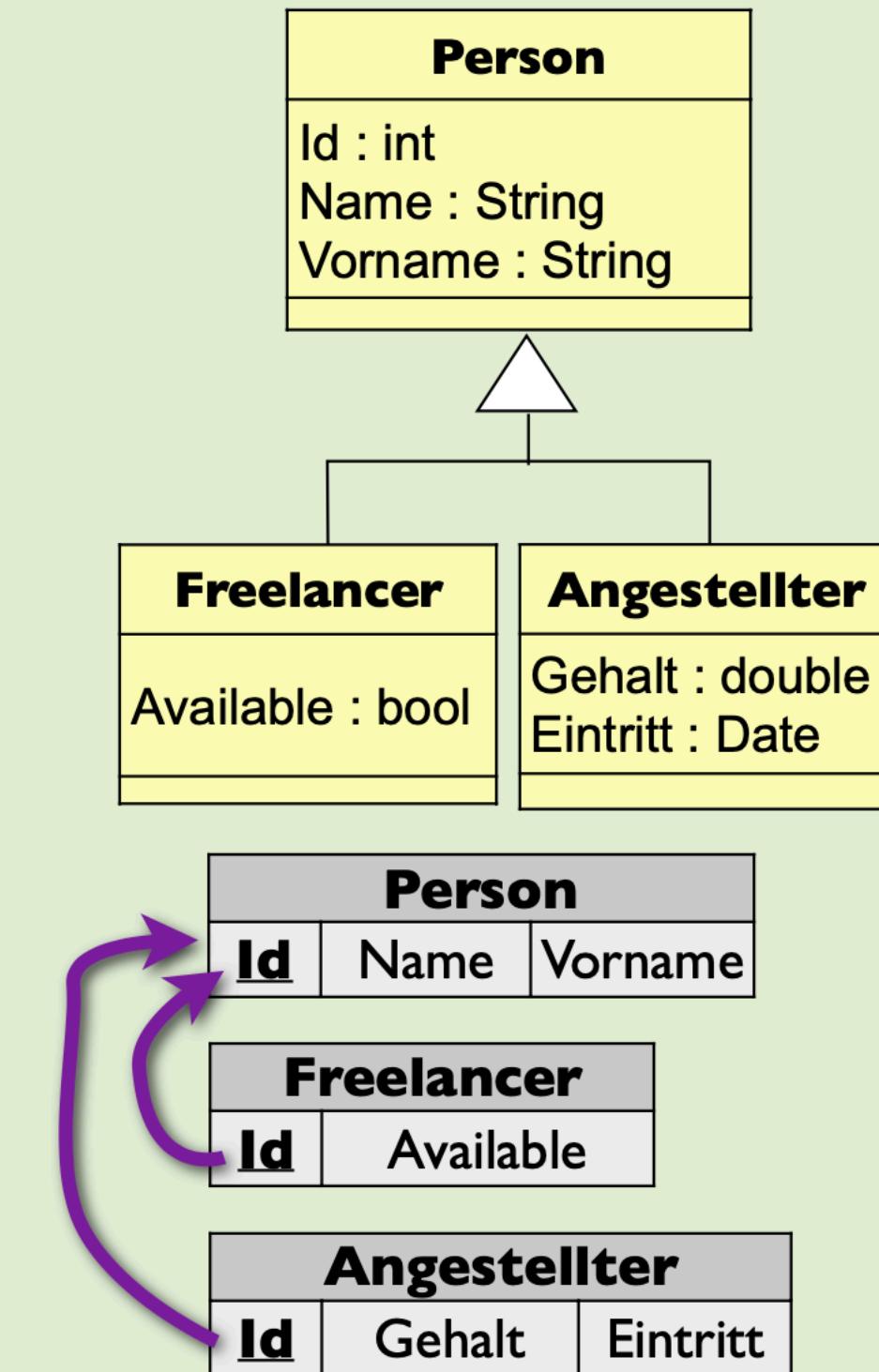
B Anwendung von JOINED

```

@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public class Person {
    @Id @GeneratedValue
    int id;
    String Name;
    String Vorname;
    public Person() {}
}
/* Beachte: @Id wird geerbt */
@Entity
public class Freelancer extends Person {
    boolean Available;
    public Freelancer() {}
}

@Entity
public class Angestellter extends Person {
    double Gehalt;
    Date Eintritt;
    public Angestellter() {}
}

```



- Vererbung datenbankseitig als Fremdschlüssel-Beziehung

Polymorphes Select - JOINED

B Polymorphe Anfrage - JOINED

Hein : Person	Jan : Angestellter	Erik : Freelancer	Person	Freelancer	Angestellter
Id = 1 Name = Bollo Vorname = Hein	Id = 3 Name = Cux Vorname = Jan Gehalt = 3217,55 Eintritt = 1.5.2003	Id = 2 Name = Roth Vorname = Erik Available = true	Id Name Vorname 1 Bollo Hein 2 Roth Erik 3 Cux Jan	Id 2 true	Id Gehalt Eintritt 3 3217,55 152.003

select p from Person p; **SELECT Id,Name,Vorname FROM Person;**

Bewertung – JOINED

Person		
Id	Name	Vorname
1	Bollo	Hein
2	Roth	Erik
3	Cux	Jan

Freelancer	
Id	Available
2	true

Angestellter		
Id	Gehalt	Eintritt
3	3217,55	152.003

- **Polymorphes select und update**
 - Select: für Wurzel ein select, ansonsten Join mit allen Oberklassen
 - Update: langsam, mehrere Tabellen betroffen
- **Monomorpher Zugriff**
 - langsam, Join mit allen Oberklassen
- **Änderungen am Design**
 - Neue Klasse: unproblematisch, neue Tabelle
 - Neues Attribut: nur eine Tabelle betroffen
- **Sonstiges**
 - besonders geeignet für wechselnde Designs, polymorphe selects

Vererbung – SINGLE_TABLE

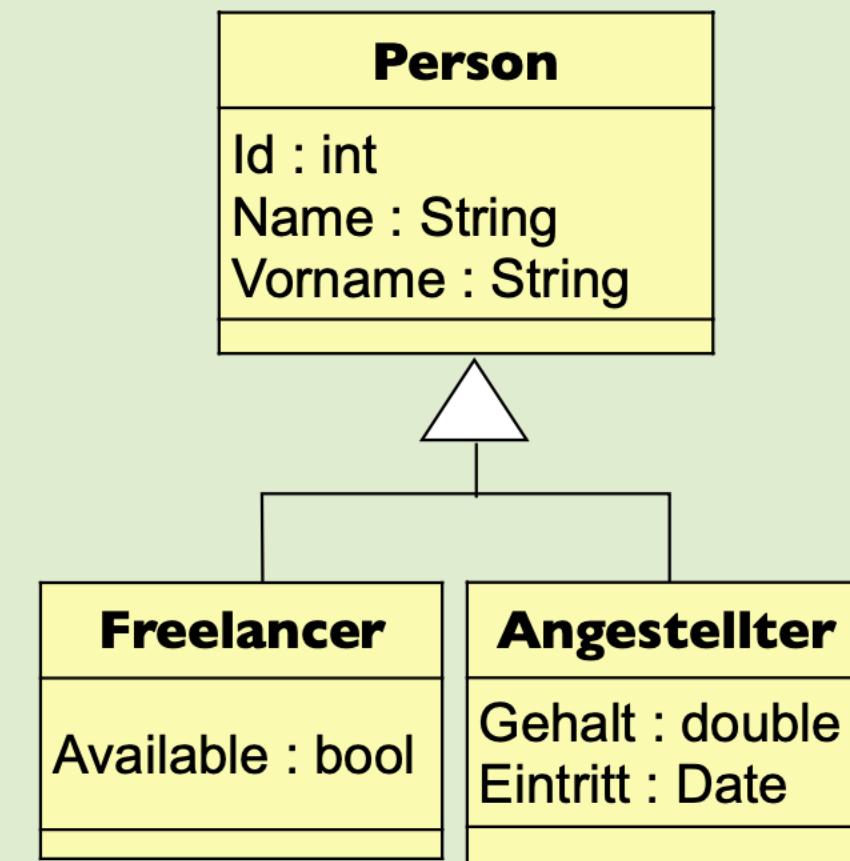
B Anwendung von SINGLE_TABLE

```

@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(
    name = "Art",
    discriminatorType = DiscriminatorType.STRING)
@DiscriminatorValue("P")
public class Person {
    @Id @GeneratedValue
    int id;
    String Name;
    String Vorname;
    public Person() {}
}
/* Beachte: @Id wird geerbt */
@Entity
@DiscriminatorValue("F")
public class Freelancer extends Person {
    boolean Available;
    public Freelancer() {}
}

@Entity
@DiscriminatorValue("A")
public class Angestellter extends Person {
    double Gehalt;
    Date Eintritt;
    public Angestellter() {}
}

```



Person						
Id	Art	Name	Vorname	Gehalt	Eintritt	Available

Polymorphes Select – SINGLE_TABLE

B Polymorphe Anfrage - TABLE_PER_CLASS						
Person						
Id	Art	Name	Vorname	Gehalt	Eintritt	Available
1	P	Bollo	Hein			
2	F	Roth	Erik			true
3	A	Cux	Jan	3217,55	15.2.2003	

Hein : Person

Id = 1
Name = Bollo
Vorname = Hein

Jan : Angestellter

Id = 3
Name = Cux
Vorname = Jan
Gehalt = 3217,55
Eintritt = 1.5.2003

Erik : Freelancer

Id = 2
Name = Roth
Vorname = Erik
Available = true

select p from Person p;

SELECT Id,Name,Vorname FROM Person;

Bewertung – SINGLE_TABLE

Person						
Id	Art	Name	Vorname	Gehalt	Eintritt	Available
1	P	Bollo	Hein			
2	F	Roth	Erik			true
3	A	Cux	Jan	3217,55	15.2.200	

- **Polymorphes select und update**
 - Select: schnell, ein select mit Projektion
 - Update: schnell, eine Tabelle
- **Monomorpher Zugriff**
 - schnell: eine Tabelle
- **Änderungen am Design**
 - Neue Klasse: Umorganisation einer u.U. großen Tabelle
 - Neues Attribut: Umorganisation einer u.U. großen Tabelle
- **Sonstiges**
 - Attribute müssen nullable sein
 - Interne Darstellungsform in Hibernate

Zusammenfassung Vererbung

Aktion / Strategie	TABLE_PER_CLASS	JOINED	SINGLE_TABLE
Hinzunehmen von Klassen	Neue Tabelle.	Neue Tabelle; evtl. Fremdschlüsselbeziehung von Kindklassen anpassen.	Aufwändig, da neue Spalten einzufügen sind.
Ändern von Attributen	Alle Kindklassen bzw. deren Tabellen betroffen.	Nur eine Tabelle betroffen.	Nur eine Tabelle betroffen.
Polymorphes select	Aufwändig: SQL-UNION über alle Kindklassen.	Nur eine Tabelle betroffen	Nur eine Tabelle betroffen; aber: Projektion erforderlich
Monomorpher Zugriff auf einzelnes Objekt	Alle Daten in einer Tabelle.	Daten müssen aus Tabellen der Oberklassen gesammelt werden.	Alle Daten in einer Tabelle.
Sonstiges	Passt zum Konzept „Klasse entspricht Tabelle“		Attribute in Kindklassen müssen nullable sein.

Einbettung

- 1:1-Beziehungen wird in eine Tabelle eingebettet

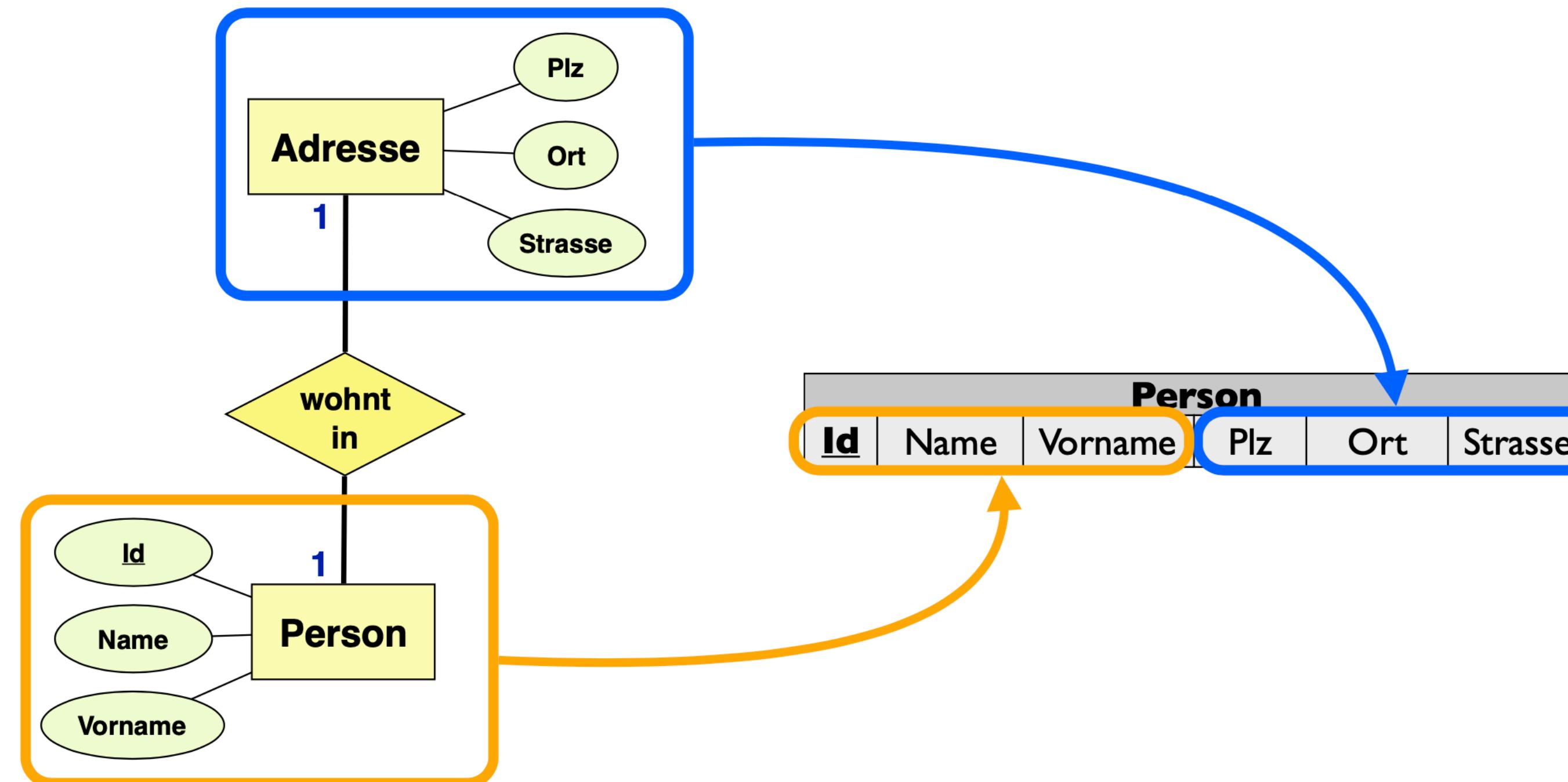
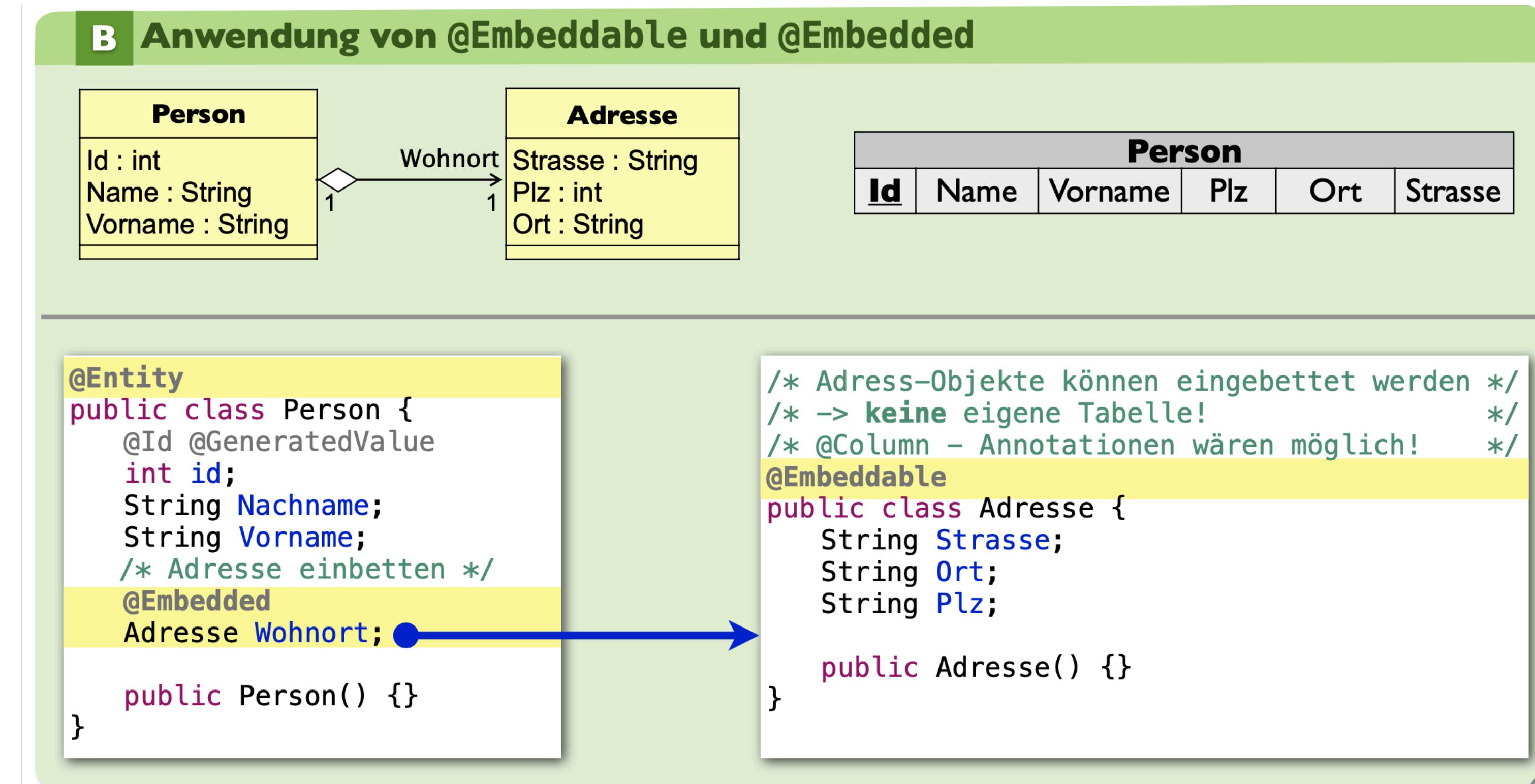


Abbildung einer UML-Komposition

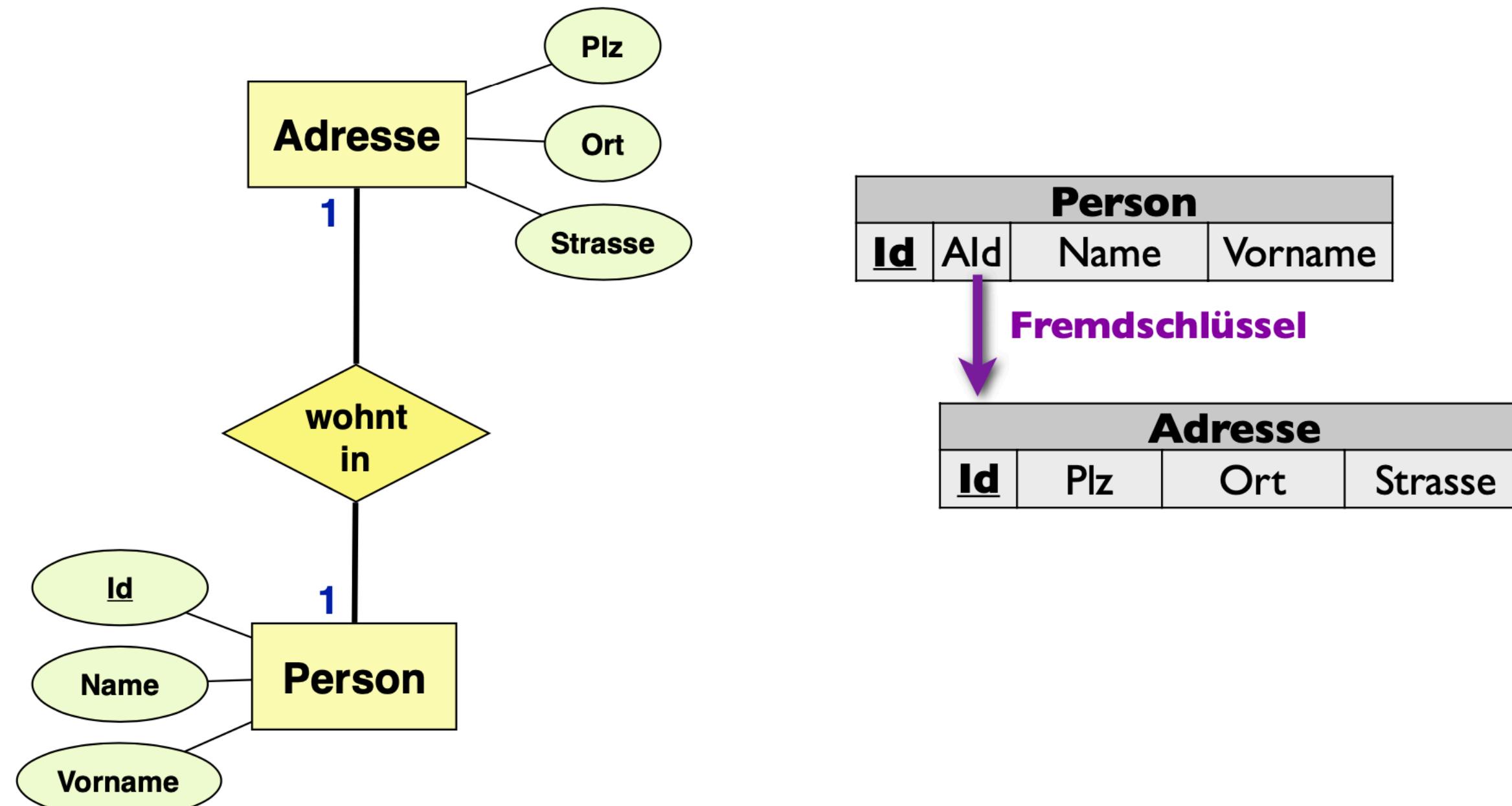


- **@EmbeddedId: Einbettung als Primärschlüssel sein, wenn**
 - Methoden **equal** und **hashCode** im Embeddable implementiert sind und
 - das **Serializable** interface realisiert wird

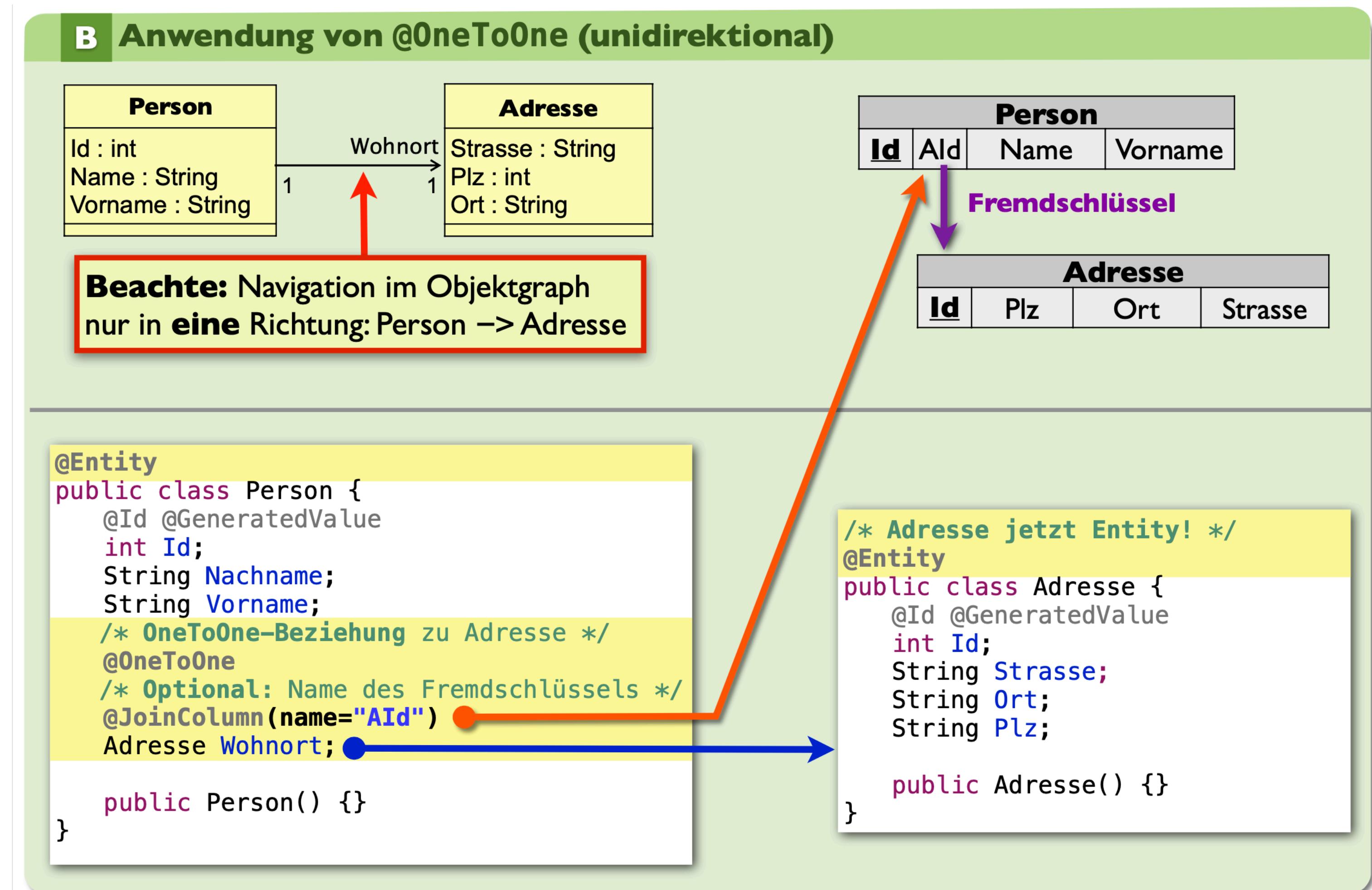
1:1-Beziehung über Fremdschlüssel

- **Jeder Teilnehmer in eine eigene Tabelle**
- **Beziehung wird über Fremdschlüssel hergestellt**

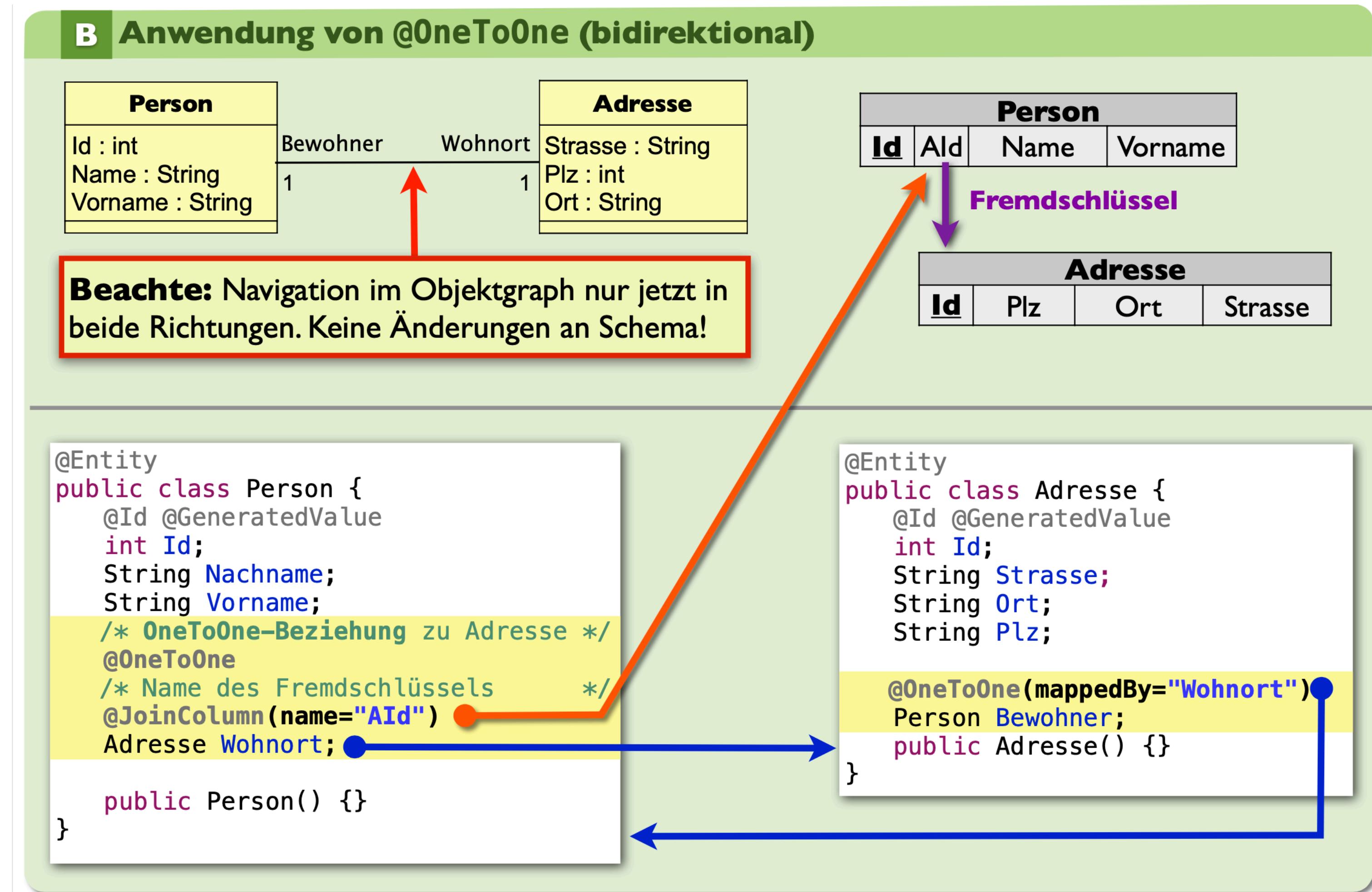
Re-Kombination via Join über **zwei** Tabellen



1:1-Beziehung über Fremdschlüssel



1:1-Beziehung über Fremdschlüssel

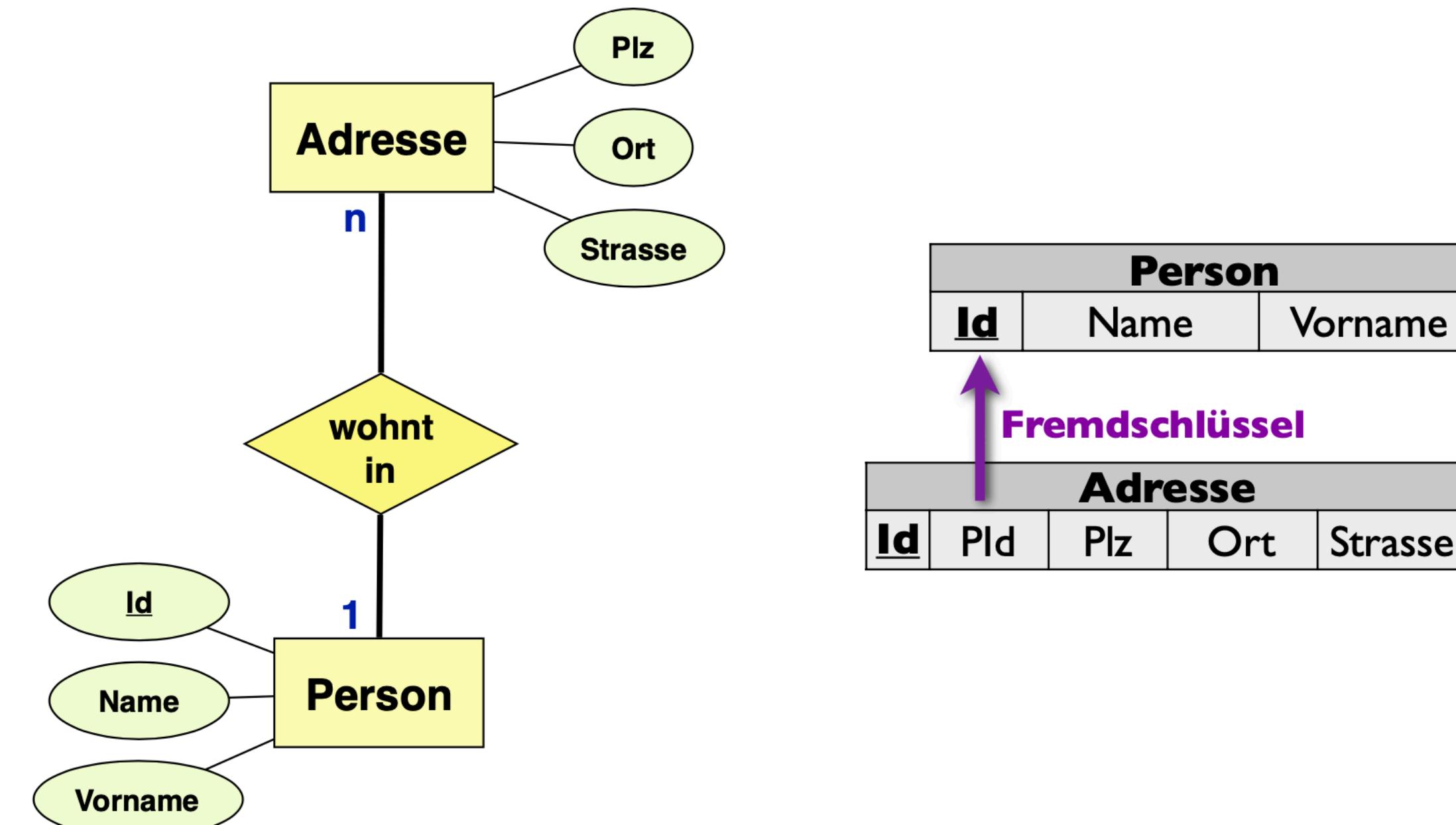


1:n-Beziehung

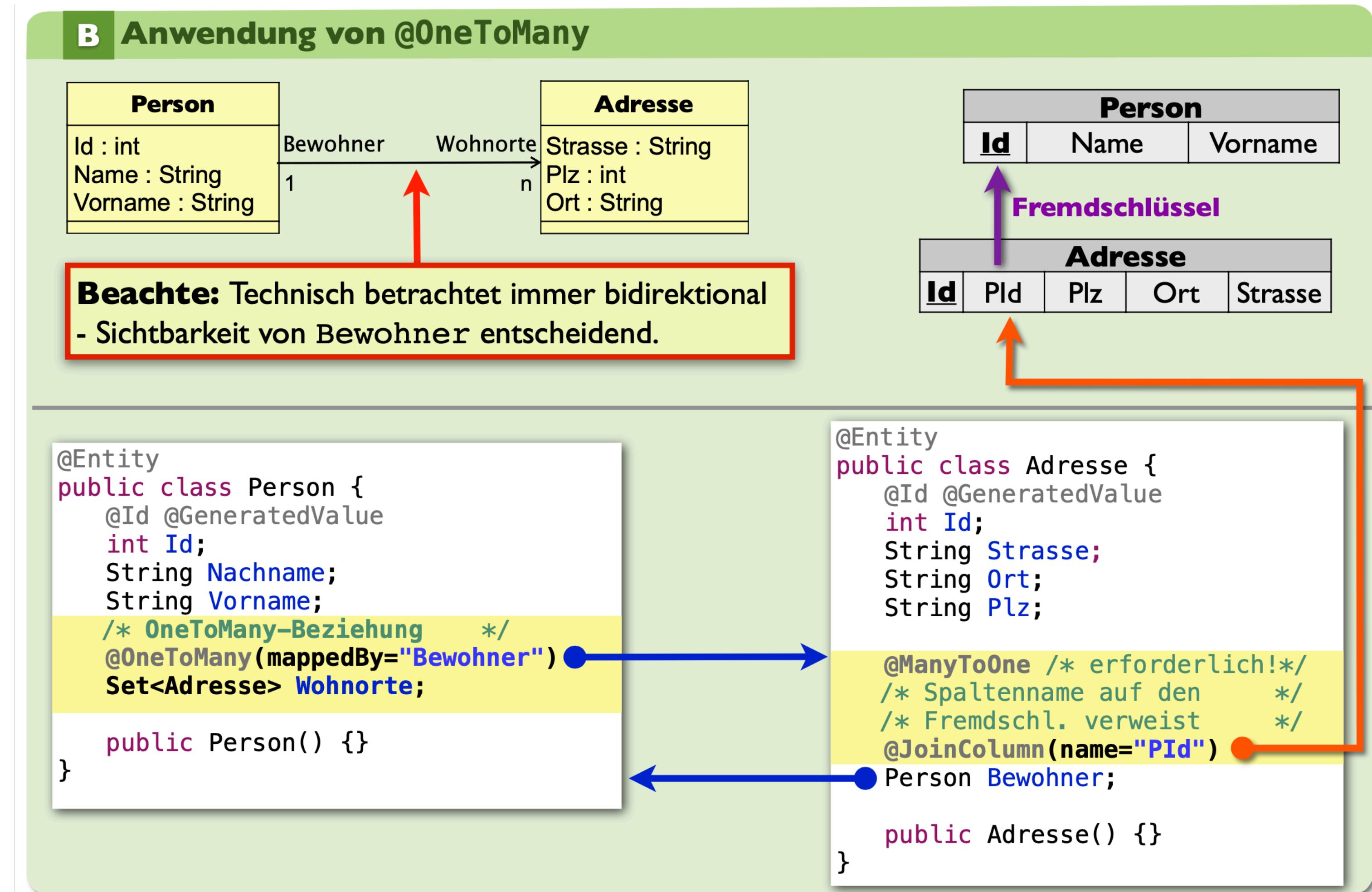
- **Jeder Teilnehmer in eigene Tabelle**
- **Beziehung direkt über Fremdschlüssel in n-Tabelle**

n-Tabelle ist Besitzer ; Re-Kombination via Join über **zwei** Tabellen

Am „1-Ende“ als interface-type (Collection, Set, List, Map)



1:n-Beziehung

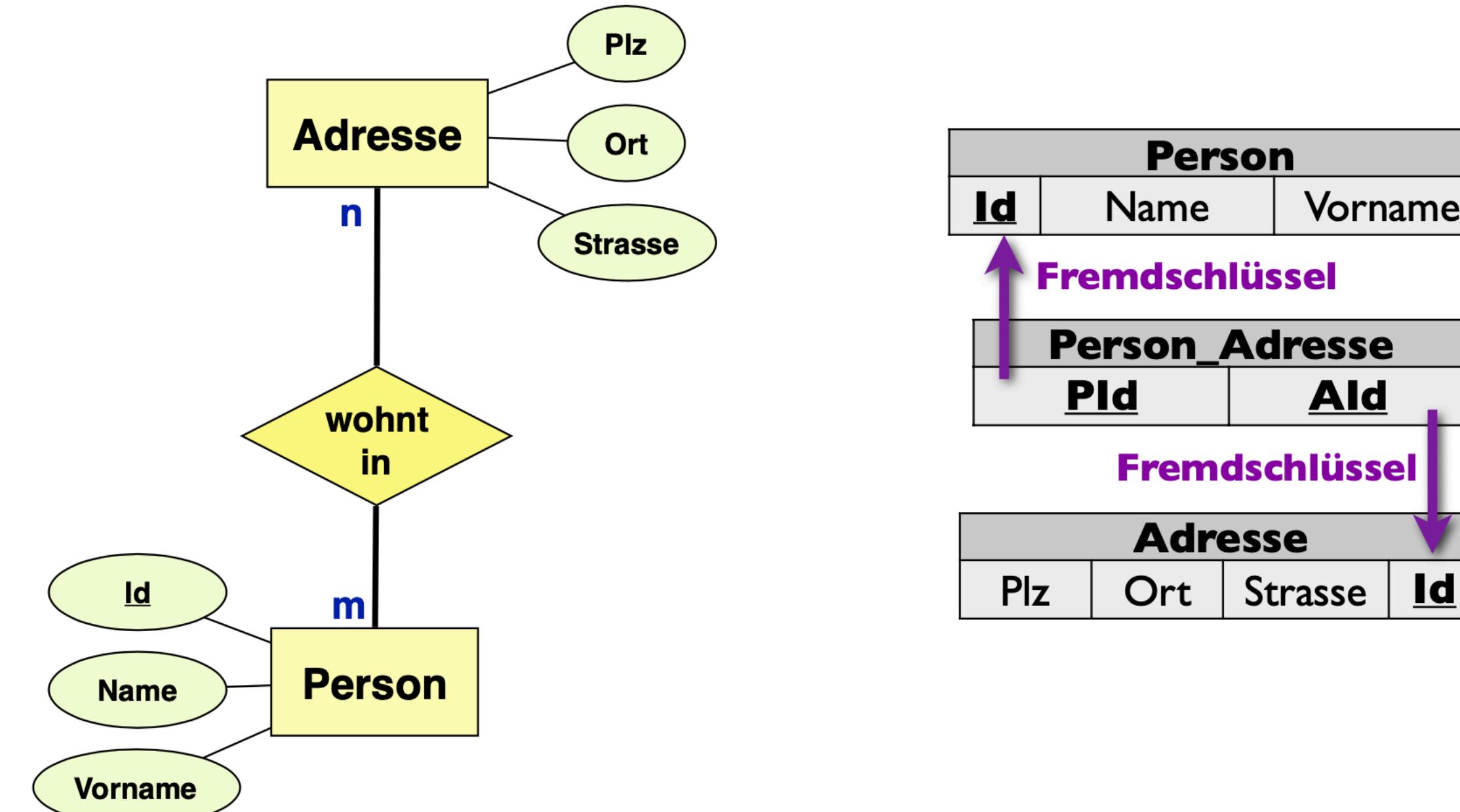


n:m-Beziehung über Join

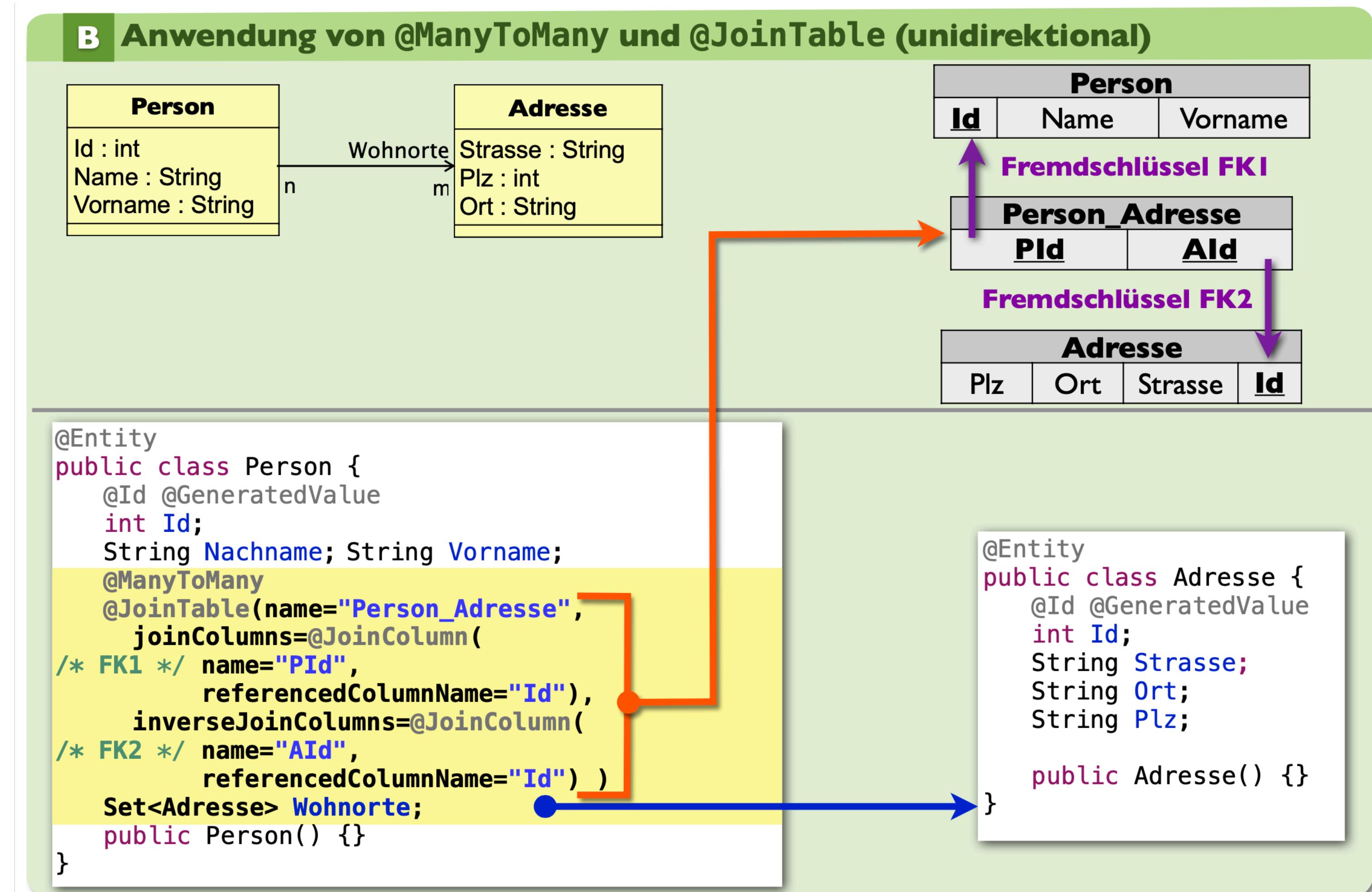
- **Jeder Teilnehmer in eine eigene Tabelle**
- **Beziehung über Assoziationsstabelle**

Re-Kombination via Join über **drei** Tabellen

Darstellung an beiden Enden als Collection (Set, ArrayList, Map etc.)



n:m-Beziehung über Join



ORM

Anmerkungen

- Idee eines ORM sollte bekannt sein.
- Hibernate nur ein Beispiel. Die Wege, Datenbank und Klassen zu verbinden, also etwa über externe Beschreibungen oder Attribute, findet man auch in anderen ORM.
- Abstrahiert generell die Anbindung an verschiedene DBMS.
- Abwägung, ORM zu nutzen und wenn welcher, ist nicht trivial.