Analyse und Modellierung von Informationssystemen

Dr. Klaus Höppner

Hochschule Darmstadt – Wintersemester 2014/2015

EAA Patterns: ORM

Relationen

Entwurfsmuster für Enterprise-Anwendungs-Architekturen

Neben den klassischen Entwurfsmustern der Gang of Four existieren weitere Entwurfsmuster für *Enterprise Application Architectures*. Diese gehen im Wesentlichen auf den britischen Informatiker Martin Fowler zurück, der auch als Pionier der so ganannten Agilen Softwareentwicklung bekannt ist.

Die EEA Patterns beziehen sich auf die Verteilung von Verantwortlichkeiten in Enterprise-Anwendungen (insbesondere die Trennung von Geschäftslogik, Datenmodell und Präsentationsschicht, beschrieben im Architekturmuster Model-View-Controller) und die Anknüpfung von Objekten an relationale Datenbanken. Letzteres stellt die Grundlage für das Objektrelationale Mapping (ORM) dar.

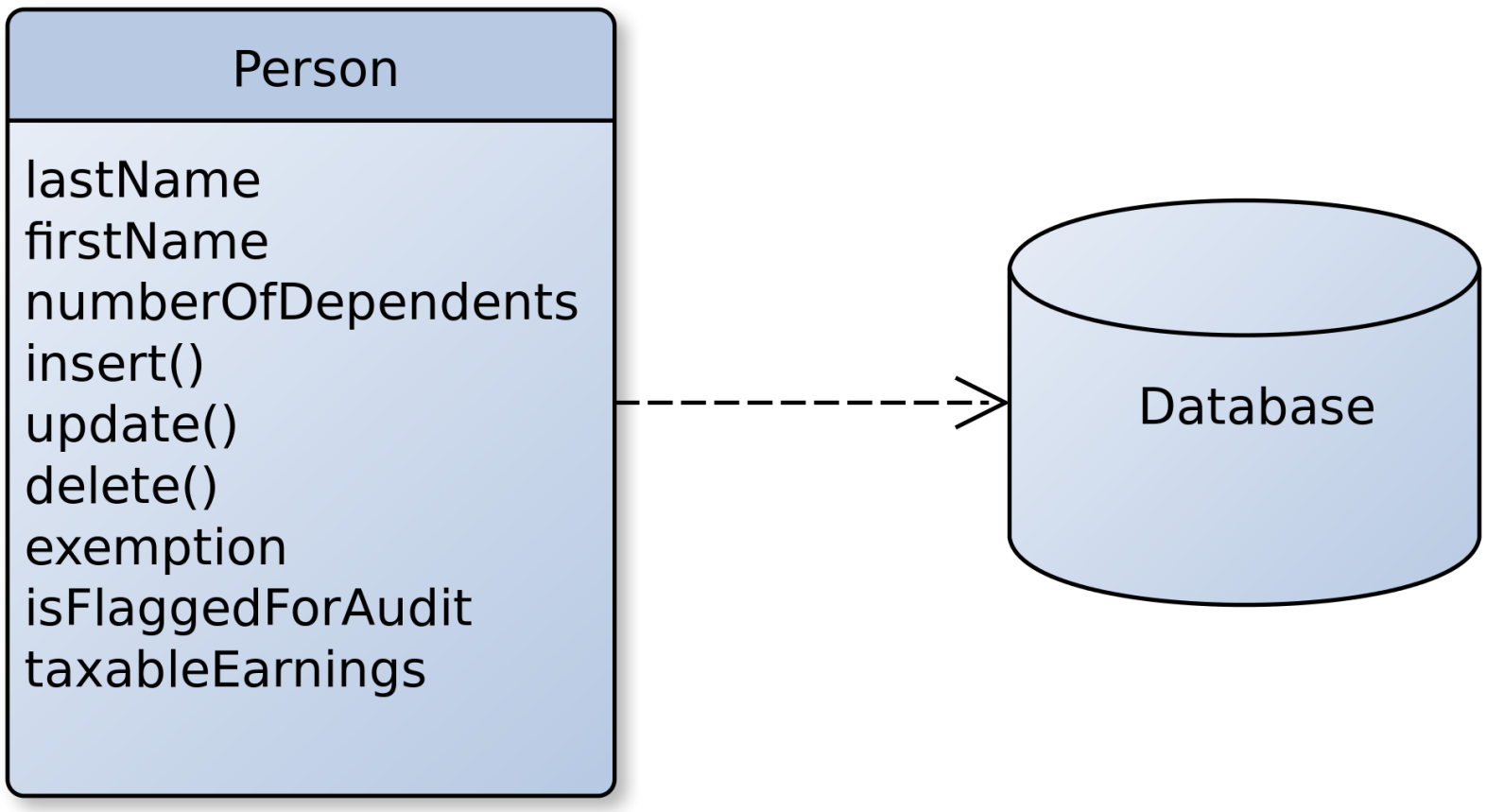
Persistenz von Objekten

Persistenz von Objekten bedeutet, dass Objekte in Java in einer Datenbank gespeichert werden. Hierbei wird i. A. eine Tabelle in der DB mit einer Klasse assoziiert, wobei jede Instanz der Klasse einer Tabellenzeile (Record) entspricht.

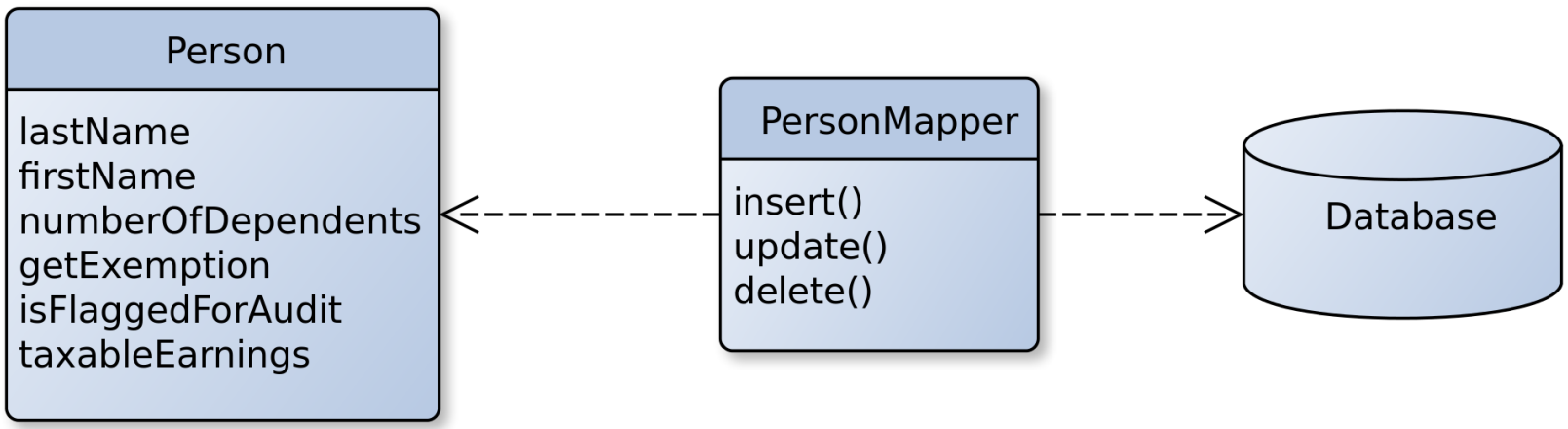
Für das Objektrelationale Mapping müssen hierbei Methoden zur Verfügung gestellt werden, die für Insert, Update, Delete und Query verantwortlich sind und die entsprechenden SQL-Befehle auf der Datenbank ausführen. Hierbei können diese Methoden

- in der Produktklasse selber befinden, d. h. es gibt keine Trennung zwischen den Daten, der eigentlichen Geschäftslogik der Klasse und dem ORM, in diesem Fall spricht man von einem *Active Record* oder
- in einer eigenen Klasse realisiert sein, die als *Data Mapper* fungiert.

Active Record



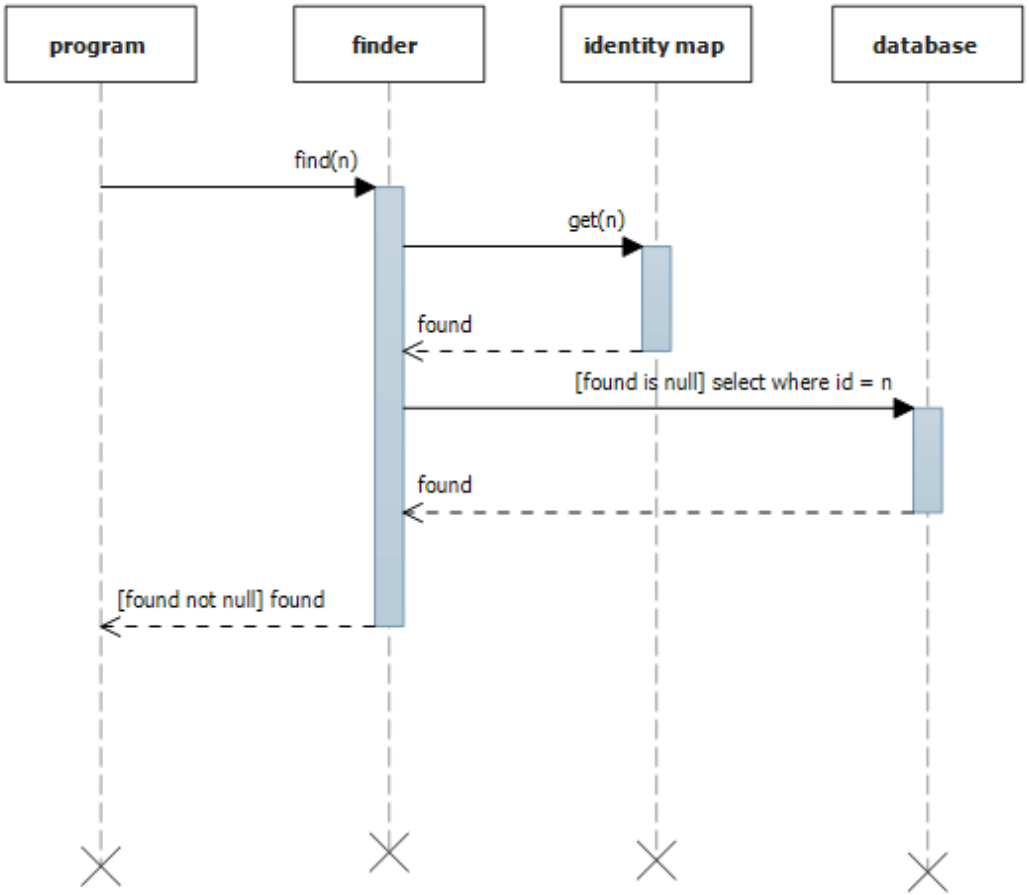
Data Mapper


Person
lastName
firstName
numberOfDependents
getExemption
isFlaggedForAudit
taxableEarnings
PersonMapper
insert()
update()
delete()
Database

Zwischenspeichern von Objekten

Statt bei jedem Laden eines Objektes die Daten neu aus der Datenbank zu holen und um zu vermeiden, dass für ein und den selben Datenbanksatz mehrere Instanzen der Produktklasse existieren, kann eine als *Identity Map* bezeichnete interne Abbildung des Primärschlüssels auf die Instanzen verwaltet werden.
Sobald anhand des Primärschlüssels ein Objekt gesucht wird, wird das zwischengespeicherte Objekt geliefert, wenn dieses schon geladen wurde, anderenfalls erfolgt ein Datenbankzugriff, eine neue Instanz der Klasse wird erzeugt und ein entsprechender Eintrag in der Identity Map angelegt.

Identity Map



Beispiel für Data Mapper mit Cache

```
public abstract class AbstractMapper<E> {
    protected Map<Integer,Object> identity_map =
            new HashMap<Integer,Object>();

    public abstract E load(Connection connection, Integer id);
    public abstract void insert(Connection connection, E e);
    public abstract void update(Connection connection, E e);
    public abstract void delete(Connection connection, E e);
    protected abstract E load(ResultSet res) throws SQLException;

    protected Collection<? super E>
    load(PreparedStatement st, Collection<? super E> target) throws SQLExc
        target.clear();
        ResultSet res = st.executeQuery();
        while (res.next()) {
            final E elem = load(res);
            target.add(elem);
        }
        return target;
    }
}
```

Abzubildende Klasse

Ausgehend von dem abstrakten Mapper soll nun ein Mapper für folgende Personen-Klasse entworfen werden:

```
public class Person {
    private Integer id;
    private String vorname;
    private String nachname;

    public Integer getId() {
        return id;
    }
    public void setId(Integer id) {
        this.id = id;

    // weitere Getter/Setter
    ...
}
```

Der Personen-Mapper

```
public class PersonMapper extends AbstractMapper<Person> {
    public final static String selectString =
        "select id,vorname,nachname from person";
    public final static String insertString =
        "insert into person (vorname,nachname) values (?,?)";
    public final static String updateString =
        "update person set vorname=?, nachname=? where id=?";
    public final static String deleteString =
        "delete from person where id=?";
```

Der Personen-Mapper (Forts.)

```
@Override
public Person load(Connection connection, Integer id) {
    Person p = (Person) identity_map.get(id);
    if (p!=null) return p;
    try {
        PreparedStatement st =
            connection.prepareStatement(selectString+
                                        " where id=?");
        st.setInt(1, id.intValue());
        List<Person> l =
            (List<Person>) load(st, new ArrayList<Person>());
        if (l.isEmpty()) return null;
        p = l.get(0);
        identity_map.put(p.getId(), p);
        return p;
    } catch (SQLException e) {
        e.printStackTrace();
        return null;
    }
}
```

Der Personen-Mapper (Forts.)

```
@Override
public void insert(Connection connection, Person p) {
    if (p.getId()==null) {
        try {
            PreparedStatement st =
                connection.prepareStatement(insertString);
            st.setString(1, p.getVorname());
            st.setString(2, p.getNachname());
            st.executeUpdate();
            st = connection.prepareStatement("call IDENTITY()");
            ResultSet res = st.executeQuery();
            res.next();
            connection.commit();
            Integer id = res.getInt(1);
            identity_map.put(id, p);
            p.setId(id);
        } catch (SQLException exc) {
            exc.printStackTrace();
        }
    } else {
        update(connection, p);
    }
}
```

Der Personen-Mapper (Forts.)

```
@Override
public void update(Connection connection, Person p) {
    if (p.getId()!=null) {
        try {
        PreparedStatement st =
            connection.prepareStatement(updateString);
        st.setString(1, p.getVorname());
        st.setString(2, p.getNachname());
        st.setInt(3, p.getId());
        st.executeUpdate();
        connection.commit();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    } else {
        insert(connection, p);
    }
}
```

Der Personen-Mapper (Forts.)

```
@Override
public void delete(Connection connection, Person p) {
    if (p.getId()==null)
        throw new RuntimeException("p not persistent");
    try {
        PreparedStatement st =
            connection.prepareStatement(deleteString);
        st.setInt(1, p.getId());
        st.executeUpdate();
        identity_map.remove(p.getId());
        p.setId(null);
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

Der Personen-Mapper (Forts.)

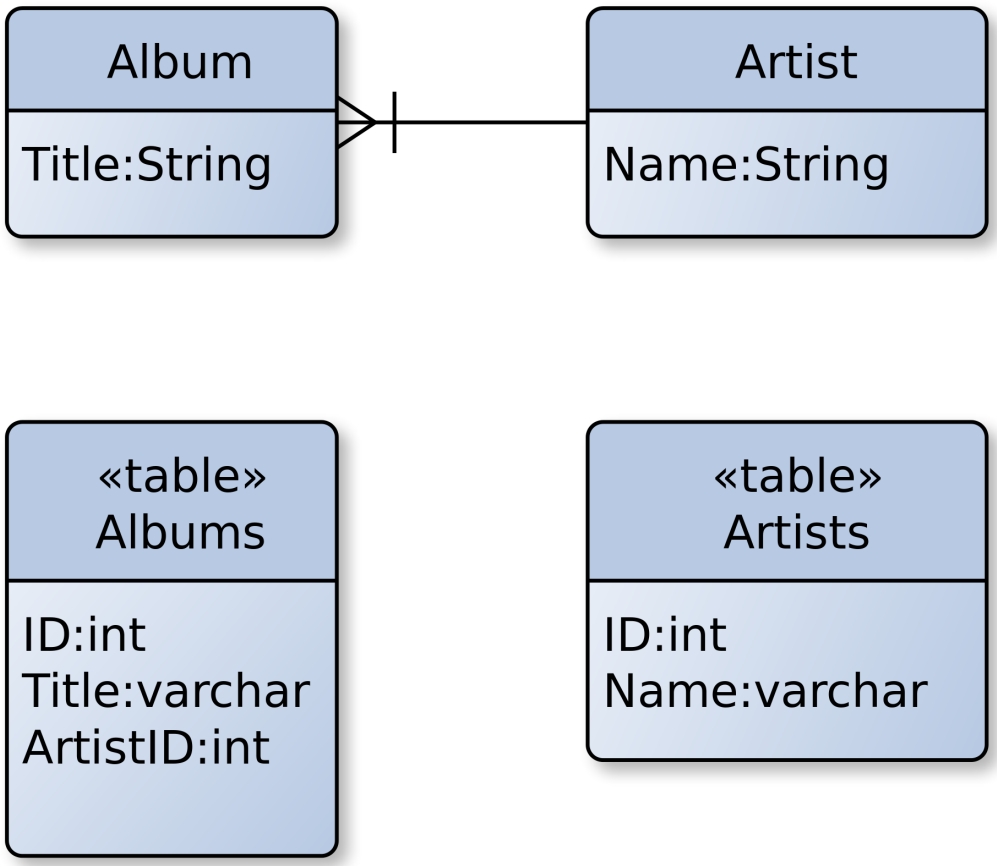
```
    @Override
    protected Person load(ResultSet res) throws SQLException {
        Integer id = res.getInt("id");
        Person p = (Person) identity_map.get(id);
        if (p!=null) return p;
        p = new Person();
        p.setId(id);
        p.setVorname(res.getString("vorname"));
        p.setNachname(res.getString("nachname"));
        identity_map.put(id, p);
        return p;
    }
}
```

Mapping von Relationen

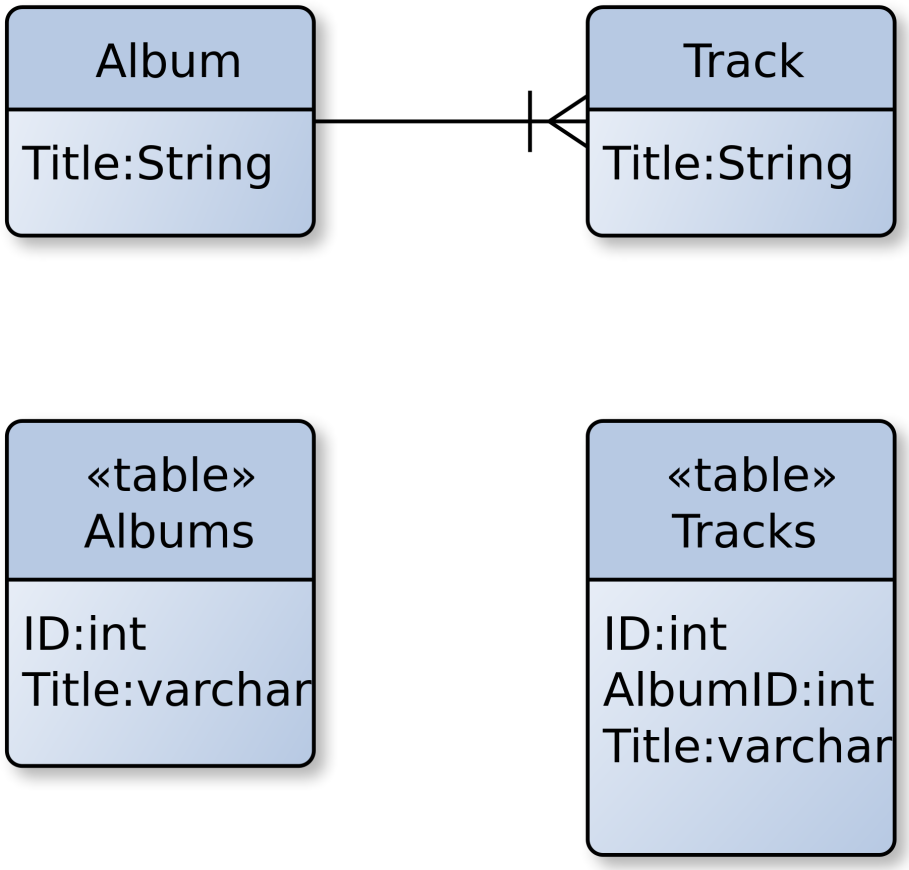
Zwischen den Tabellen in einer Datenbank bestehen Relationen über Fremdschlüssel, die im ORM entsprechend abgebildet werden müssen.
Beispiele:

- ein Musikalbum hat einen Interpreten (*n:1*), in der DB realisiert durch einen Fremdschlüssel auf die ID-Spalte der Interpreten-Tabelle. Hier muss die Klasse Album entsprechend die Daten des Interpreten laden.
- auf dem Album befindet sich eine Menge von Tracks (*1:n*), die anhand der Album-Id in der Track-Tabelle nachgeladen werden müssen (mit unterschiedlichen Möglichkeiten für den Ladezeitpunk, *eager* bzw. *lazy loading*).
- Abbildung der Form *n:m* über eine Assoziations-Tabelle.

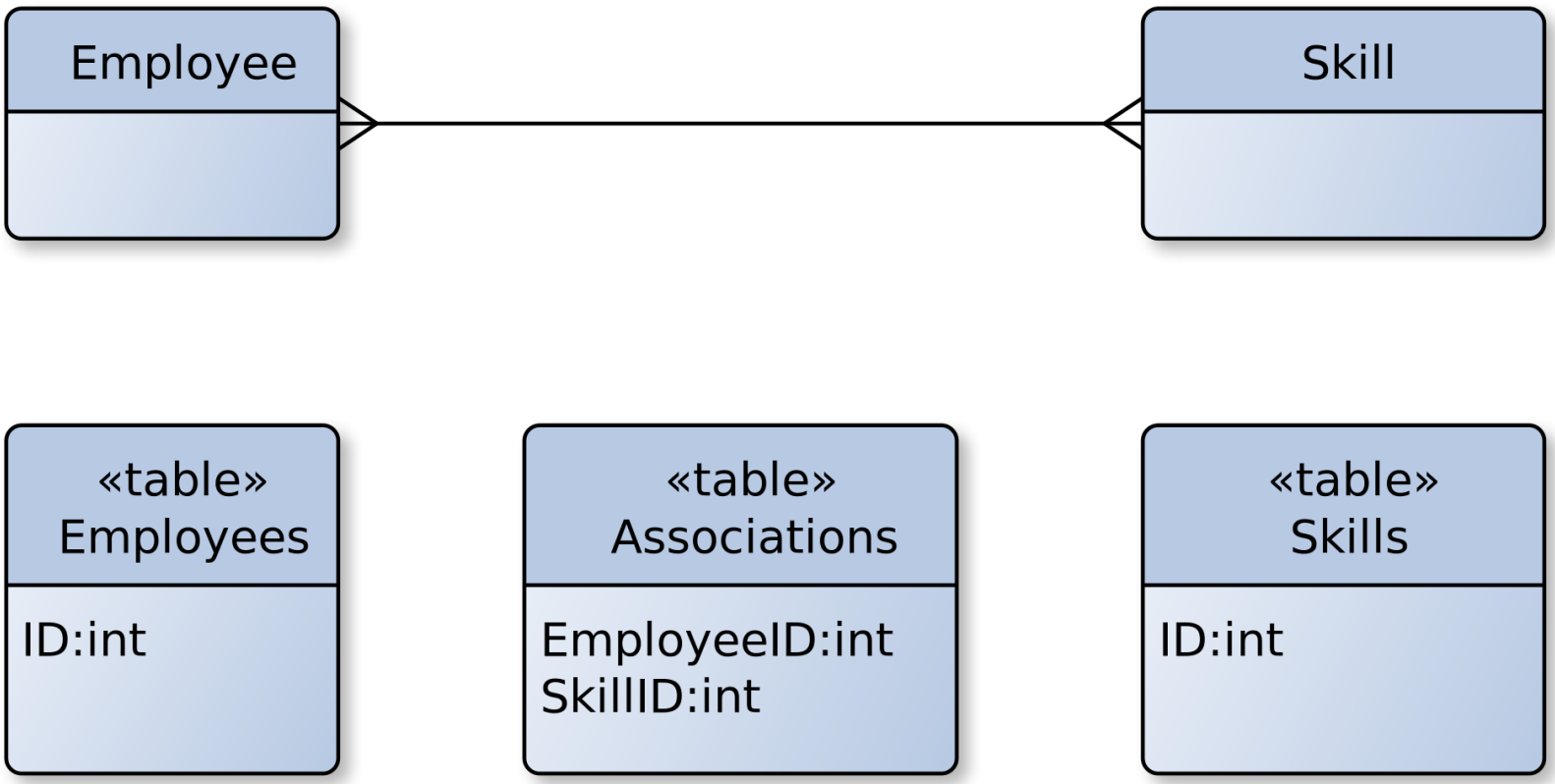
Foreign-Key-Mapping n:1



Foreign-Key-Mapping 1:n



Association-Table m:n



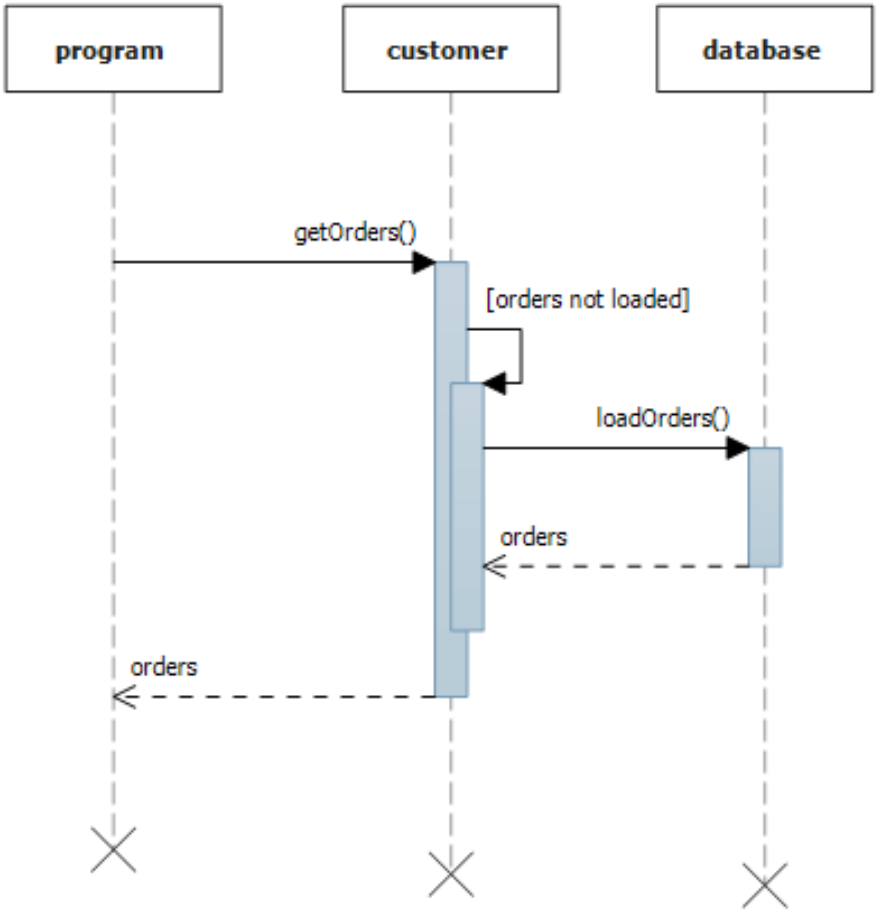
Ladestrategien

Bei einer Assoziation vom Typ n:1 wird es i. A. vorteilhaft sein, das assoziierte Objekt anhand seines Primärschlüssels direkt nachzuladen.

Bei Assoziationen vom Typ 1:n bzw. m:n kann ein direktes Nachladen (eager load) der assoziierten Objekte aufwändig sein (insbesondere, da hier noch 1nicht feststeht, ob auf diese tatsächlich zugegriffen werden soll).
In diesem Fall wird oft eine Proxy-Klasse benutzt, die die assoziierten Objekte beim ersten Zugriff nachlädt (lazy load).

Problematik: Hier muss beim ersten Zugriff die DB-Verbindung noch existieren!

Lazy Load


program
customer
database
getOrders()
[orders not loaded]
loadOrders()
orders
orders