Analyse und Modellierung von Informationssystemen

Dr. Klaus Höppner

Hochschule Darmstadt – Wintersemester 2014/2015

Mapping von Relationen

Mapping von Vererbung

Model-View-Controller

Mapping einer Relation n:1

Im Beispiel wird die Klasse `Person` nun um ein Attribut `abteilung` ergänzt:

```
public class Person {
    private Integer id;
    ...
    private Abteilung abteilung;

    ...
    public Abteilung getAbteilung() {
        return abteilung;
    }
    public void setAbteilung(Abteilung abteilung) {
        this.abteilung = abteilung;
    }
}
```

Mapping in der Klasse PersonMapper

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

Mapping in der Klasse PersonMapper (Forts.)

```
@Override
public void insert(Object o) {
    if (! (o instanceof Person)) throw new RuntimeException();
    Person p = (Person) o;
    Connection connection = data.getConnection();
    PreparedStatement st = connection.prepareStatement(insertString);
    st.setString(1, p.getVorname());
    st.setString(2, p.getNachname());
    if (p.getAbteilung()==null) {
        st.setNull(3, java.sql.Types.INTEGER);
    } else {
        st.setInt(3, p.getAbteilung().getId());
    }
    st.executeUpdate();
    connection.commit();
    st = connection.prepareStatement("call IDENTITY()");
    ResultSet res = st.executeQuery();
    res.next();
    Integer id = res.getInt(1);
    identity_map.put(id, p);
    p.setId(id);
}
```

Mapping in der Klasse PersonMapper (Forts.)

```
@Override
public void update(Object o) {
    if (! (o instanceof Person)) throw new RuntimeException();
    Person p = (Person) o;
    Connection connection = data.getConnection();
    PreparedStatement st = connection.prepareStatement(updateString);
    st.setString(1, p.getVorname());
    st.setString(2, p.getNachname());
    if (p.getAbteilung()==null) {
        st.setNull(3, java.sql.Types.INTEGER);
    } else {
        st.setInt(3,  p.getAbteilung().getId());
    }
    st.setInt(4, p.getId());
    st.executeUpdate();
    connection.commit();
}
```

Mapping in der Klasse PersonMapper (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"));
        int abteilung_id = res.getInt("abteilung_id");
        if (abteilung_id==0) {
            p.setAbteilung(null);
        } else {
            p.setAbteilung(data.load(Abteilung.class, abteilung_id));
        }
        identity_map.put(p.getId(), p);
        return p;
    }
}
```

Mapping einer Relation 1:n

```
public class Abteilung {
    private Integer id;
    ...
    private List<Person> personen = new ArrayList<Person>();

    ...
    public List<Person> getPersonen() {
        return personen;
    }
    public void setPersonen(List<Person> personen) {
        this.personen = personen;
    }
}
```

Proxy für die Personen-Liste

Für die Relation vom Typ 1:n wird nun nach dem Prinzip *lazy loading* verfahren.
Hierfür werden ein Stellvertreter für eine Liste und eine Schnittstelle zum Laden aus der DB bei Bedarf eingeführt:

```
public interface VirtualListLoader<E> {
    public List<E> getData();
}
```

Proxy für die Personen-Liste (Forts.)

```
public class VirtualList<E> extends AbstractList<E> {
    private List<E> source=null;
    private VirtualListLoader<E> dataloader;

    public VirtualList(VirtualListLoader<E> dataloader) {
        this.dataloader = dataloader;
    }
    private List<E> getSource() {
        if (source==null)
            source = dataloader.getData();
        return source;
    }
    @Override
    public E get(int index) {
        return getSource().get(index);
    }
    @Override
    public int size() {
        return getSource().size();
    }

}
```

Eigentliches Lazy Loading

Beim Laden einer Abteilung wird nun eine Virtuelle Liste verwendet, die beim ersten Zugriff die zur Abteilung gehörigen Mitarbeiter aus der Datenbank lädt.

```
public class AbteilungMapper extends AbstractMapper<Abteilung> {
    ...
    @Override
    protected Abteilung load(ResultSet res) throws SQLException {
        final Integer id = res.getInt("id");
        Abteilung a = (Abteilung) identity_map.get(id);
        if (a!=null) return a;
        a = new Abteilung();
        a.setId(id);
        a.setName(res.getString("name"));
```

Eigentliches Lazy Loading (Forts.)

```
        VirtualList<Person> vl =
        new VirtualList<Person>(new VirtualListLoader<Person>() {
            @Override
            public List<Person> getData() {
                Connection connection = data.getConnection();
                PersonMapper mapper =
                    (PersonMapper) data.getMapper(Person.class);

                PreparedStatement st =
                    connection.prepareStatement(mapper.selectString+
                        " where abteilung_id=? order by id");
                st.setInt(1, id);
                return (List<Person>) mapper.load(st,
                    new ArrayList<Person>());
            }
        });
        a.setPersonen(vl);
        identity_map.put(a.getId(), a);
        return a;
    }
}
```

Mapping von Vererbung

Das Abbilden von Vererbung im Klassendiagramm auf Tabellen in der Datenbank erfolgt nach folgendem Prinzip:

- Entweder verwenden einer einzigen Tabelle (*single table inheritance*), die Spalten für alle Attribute aller Kindklassen besitzt, wobei die nicht passenden Spalten jeweils auf `NULL` gesetzt werden, oder
- eine Tabelle je Klasse (*class table inheritance*), in der Tabelle der Elternklasse befinden sich Spalten für die Attribute der Elternklasse, individuelle Attribute der Kindklassen in den entsprechenden Tabellen, samt einem Verweis auf den Primärschlüssel der Tabelle für die Elternklasse,
- mit Verwenden einer Diskriminator-Spalte, die entscheidet, von welcher Klasse der Datensatz instantiiert wurde.

Schaubild: Single Table Inheritance

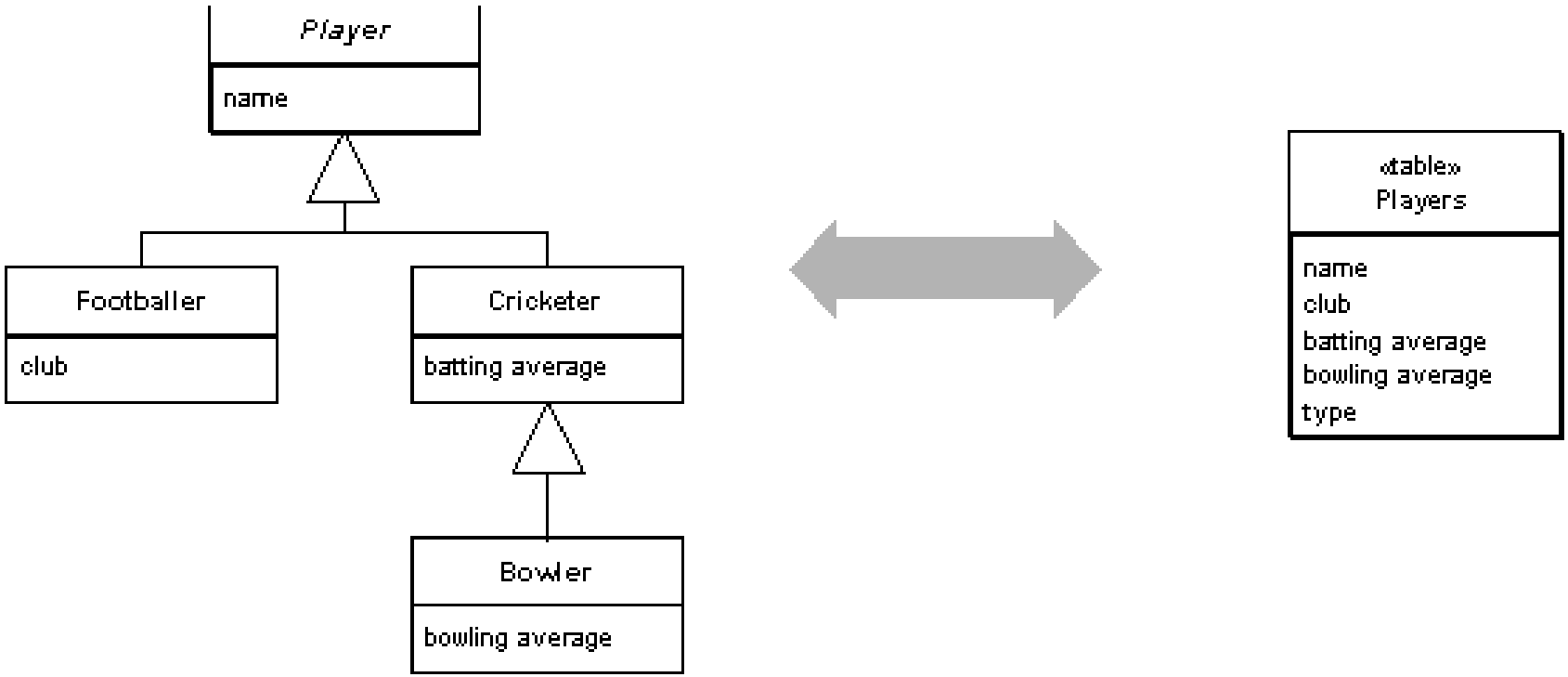
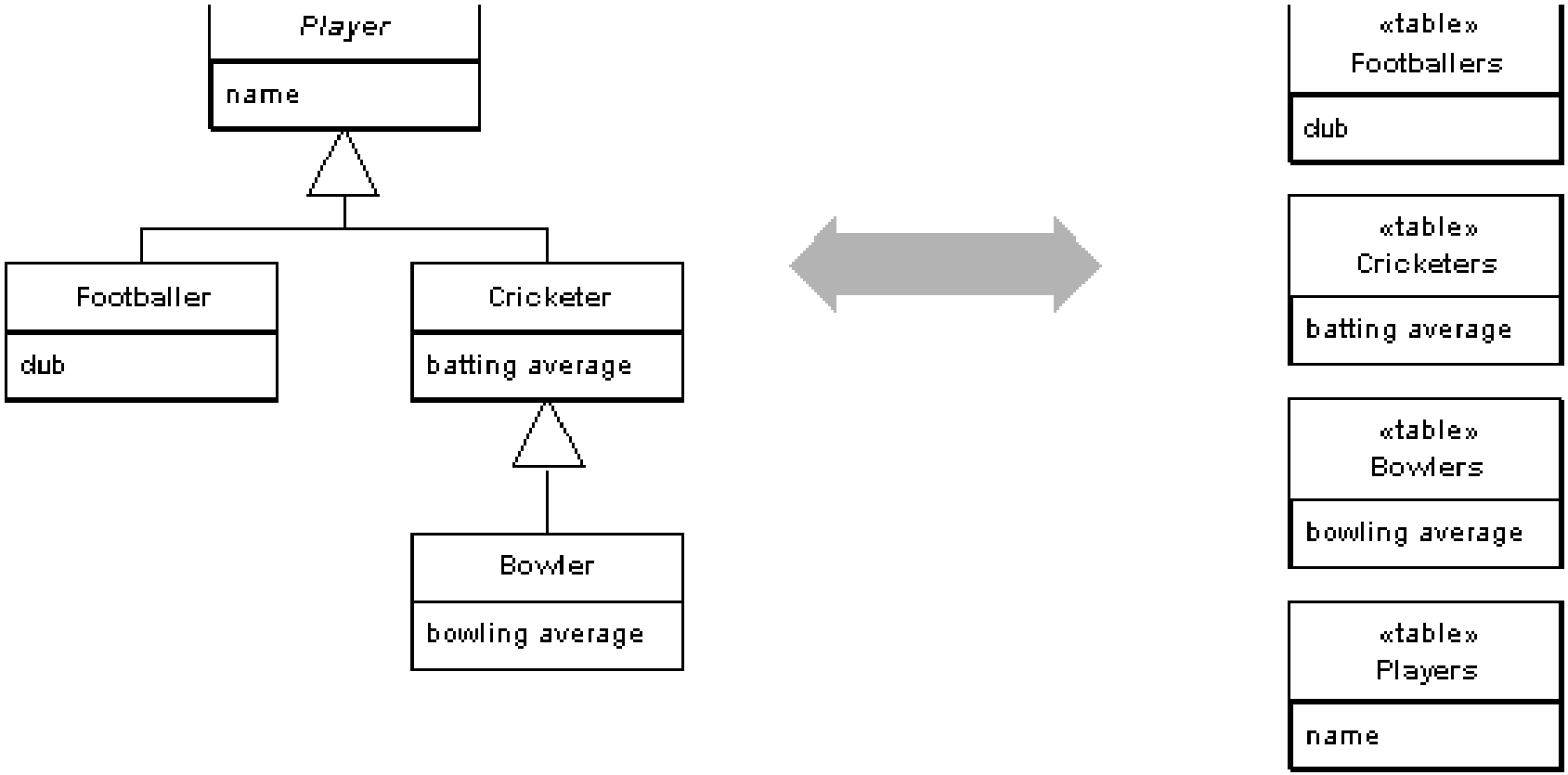


Schaubild: Class Table Inheritance



Beispiel

Von der Klasse `Person` erben nun die beiden Klassen `Angestellter` und `Chef`:

```
public class Angestellter extends Person {
    private Person chef;

    public Person getChef() {
        return chef;
    }

    public void setChef(Person chef) {
        this.chef = chef;
    }
}
```

Beispiel (Forts.)

```
public class Chef extends Person {
    private int budget;

    public int getBudget() {
        return budget;
    }

    public void setBudget(int budget) {
        this.budget = budget;
    }
}
```

Beide Klassen werden in der Tabelle Person abgebildet, mit Hilfe der Spalte DTYPE mit Wert 1 für Angestellte, Wert 2 für Chefs.

Beispiel (Forts.)

```
public class PersonMapper extends AbstractMapper<Person> {
    public final static String selectString = "select id,dtype,vorname,"+
        "nachname,abteilung_id,boss_id,budget from person";
    public Mapper<? extends Person> getMapper(Integer dtype) {
        switch (dtype) {
        case 1:
            return data.getMapper(Angestellter.class);
        case 2:
            return data.getMapper(Chef.class);
        default:
            return null;
        }
    }
    @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;
        Integer dtype = res.getInt("dtype");
        PersonMapper mapper = (PersonMapper) getMapper(dtype);
        return mapper.load(res);
    }
    ...
}
```

Beispiel (Forts.)

```
public class AngestellterMapper extends PersonMapper {
    @Override
    protected Angestellter load(ResultSet res) throws SQLException {
        Integer id = res.getInt("id");
        Angestellter a = (Angestellter) identity_map.get(id);
        if (a!=null) return a;
        a = new Angestellter();
        a.setId(id);
        a.setVorname(res.getString("vorname"));
        a.setNachname(res.getString("nachname"));
        int abteilung_id = res.getInt("abteilung_id");
        if (abteilung_id==0) a.setAbteilung(null);
        else a.setAbteilung(data.load(Abteilung.class, abteilung_id));
        int boss_id = res.getInt("boss_id");
        if (boss_id==0) a.setChef(null);
        else a.setChef(data.load(Person.class, boss_id));
        identity_map.put(a.getId(), a);
        return a;
    }
}
```

Model-View-Controller

MVC ist eines der bekanntesten Entwurfsmuster in der Entwicklung von Anwendungen. Es beschreibt die Trennung von

- Datenmodell, häufig in Form von persistenten Objekten über ORM,
- Präsentationsschicht, die für die Darstellung der Daten gegenüber dem Anwender verantwortlich ist,
- Controller, der die eigentliche Anwendungslogik ausführt, indem eine Anfrage interpretiert, die Daten zusammenstellt und an die Präsentationsschicht übergibt.

Schaubild: Model-View-Controller