Saarbrücken, 02.07.2015
Information Systems Group

Vorlesung „Informationssysteme“

Vertiefung Kapitel 9: Objektrelationales Mapping

Erik Buchmann (buchmann@cs.uni-saarland.de)

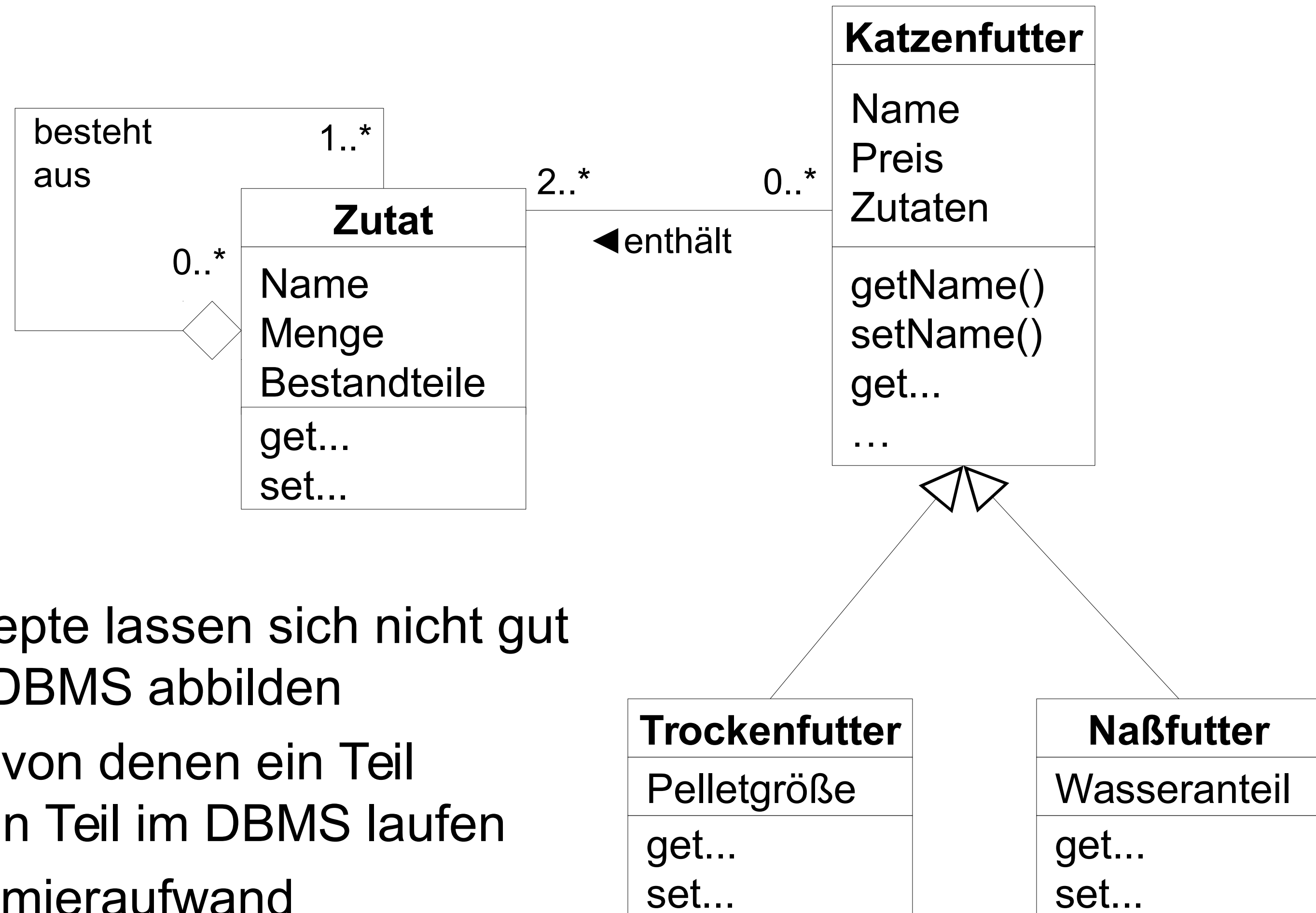
Foto: M. Strauch

Aus den Videos wissen Sie...

- ...dass relationale DBMS effizient und performant arbeiten
  - allerdings nur auf relationalen Daten
- …wie ein Zugriff auf ein DBMS per JDBC aussieht
  - mit sehr viel händischer Programmierarbeit

- Vertiefung heute:
  - Abbildung von Objekthierarchien auf Relationale Datenbanken
  - Konzepte für Objektrelationales Mapping

Ziel: (Java-) Objekte in der Datenbank speichern



- Viele OO-Konzepte lassen sich nicht gut auf relationale DBMS abbilden
- Transaktionen, von denen ein Teil im Client und ein Teil im DBMS laufen
- Hoher Programmieraufwand

Herausforderungen

1) Identität
   - in DBMS Prüfung auf Primärschlüssel-Attribute
   - OO: attributunabhängige Prüfung auf Objektidentität

```
String x = „test“;
String y = (String)x.clone();
```
← neues Objekt, neue Objektreferenz

2) Assoziationen
   - Fremdschlüsselbeziehungen auf Attributebene vs. Objektreferenzen in OO

3) Vererbung
   - in DBMS aufwendig über Fremdschlüsselbeziehungen nachzubilden

4) Verschachtelung/Kollektionen/Listen
   - Keine *einfache* Möglichkeit, folgendes in relationalen DBMS abzubilden:

```
List l = new List(); List j = new List();
l.add(new Rectangle());
l.add(j);
j.add(new String(„Test“));
j.add(new Integer(15));
```

Datenbankanbindung per JDBC

aus einer Video-Lecture

- Hier fehlt noch der Typecast vom ResultSet auf Java-Objekte

```
 7  import java.sql.Connection;
 8  import java.sql.DriverManager;
 9  import java.sql.ResultSet;
10  import java.sql.ResultSetMetaData;
11  import java.sql.SQLException;
12  import java.sql.Statement;
13  import static jdbcexample.JDBCExample.outputResultSet;
14
15  public class JDBCParameterExample {
16
17      public static void outputResultSet(ResultSet rset) throws SQLException {...}
27
28
29      public static void main(String[] args) {
30          Connection conn = null;
31
32          try {
33              System.out.println(Class.forName("org.postgresql.Driver"));
34              conn = DriverManager.getConnection("jdbc:postgresql://localhost/Fotoagentur?user=postgres&password=.....");
35          } catch (Exception e) {
36              System.out.println("Fehler: " + e);
37              System.exit(-1);
38          }
39          if (conn != null) {
40              try {
41                  Statement sql_stmt = conn.createStatement();
42                  ResultSet rset = sql_stmt.executeQuery("select * from FotographenKomplett2 where id = " + args[0]);
43                  outputResultSet(rset);
44                  rset.close();
45                  conn.close();
46              } catch (SQLException se) {
47                  System.out.println("Fehler: " + se);
48              }
49          }
50      }
51  }
```

Programmierer muss sich SQL-Kommandos aus Strings zusammenbasteln

Fragestellungen im Folgenden

- Objekthierarchien effizient in objektorientierten DBMS speichern?

```
List l = new List();
List j = new List();
l.add(new Rectangle());
l.add(j);
j.add(new String(„Test“));
j.add(new Integer(15));
...
```

  - mehrere Varianten mit unterschiedlichen Stärken und Schwächen

- Frameworks als <u>einfache</u> Schnittstelle von OO (Java) zum DBMS
  - Objekte speichern mit Hibernate:

```
Transaktion t = session.beginTransaction();
session.save(object);
t.commit();
```

Hierarchien in Relationalen DBMS

Foto: M. Strauch

Simple Variante: BLOBs

- Objekthierarchie im Client vollständig serialisieren und in ein Binary Large Object (BLOB) speichern

  ```
  CREATE TABLE object (
          id              SERIAL PRIMARY KEY,
          bin_data        BLOB
  );
  ```

  Anm: in Postgres heißt der BLOB-Datentyp „BYTEA“ (Byte Array)

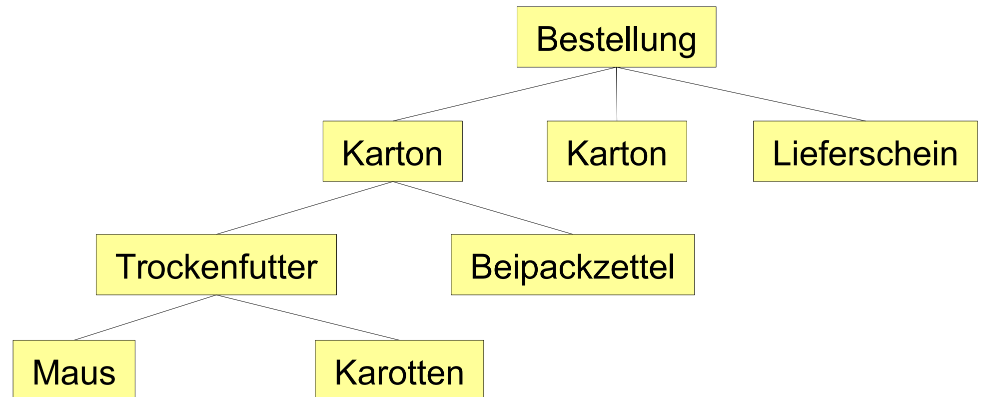
- Mit Extensions oder Zusatz-Packages des DBMS zum Teil Operationen und Anfragen im BLOB
  - Postgres: JSON (Java Script Object Notation) – Funktionen

- Keine Integritätsüberwachung, viele Operationen erfordern vollständiges Einlesen und Neuschreiben der Hierarchie
  - Für kleinere Datenmengen ganz nett

Beispiel JSON

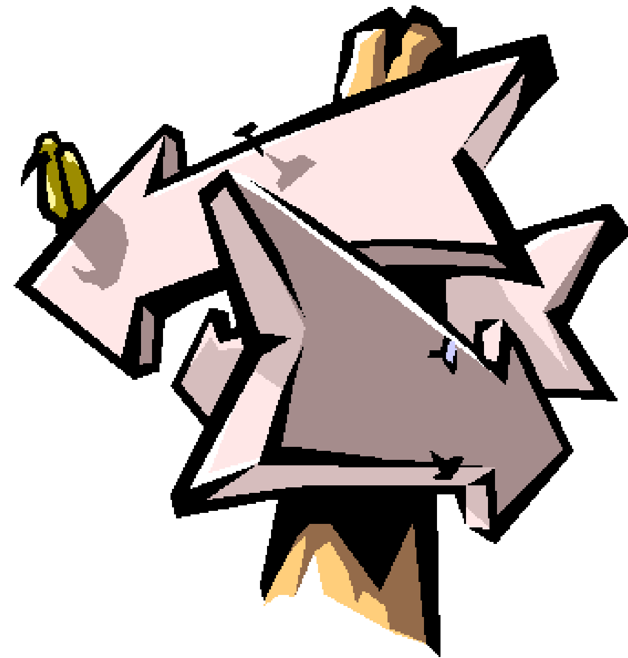
- Struktur etwas einfacher als XML
  - Im BLOB gespeichert:

```
{
„Bestellung“: {
    „Karton“: {
        „Trockenfutter“: {
            „Maus“,
            „Karotten“
            },
        „Beipackzettel“
        }
    },
    „Karton“,
    „Lieferschein“
}
```



Varianten im Folgenden

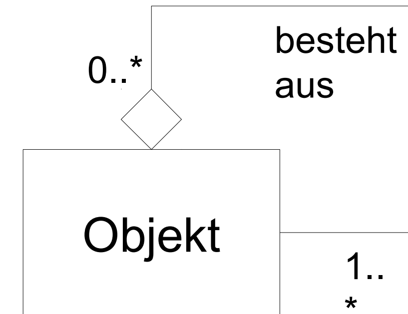
- Adjacence Lists
  (alle mir bekannten ORM-Frameworks)
- Path Enumeration
- Nested Sets
- Closure Tables



Dr.-Ing. Erik Buchmann (buchmann@cs.uni-saarland.de)

Adjacence Lists

- Objekthierarchie als Baumstruktur über ID
  - Knoten über Parent navigierbar




```
CREATE TABLE object (
        id              SERIAL PRIMARY KEY,
        parent          INT,
        value           TEXT,
        FOREIGN KEY (parent)
                REFERENCES object(id)
                ON DELETE CASCADE
);
```

id	parent	value
1	NULL	Bestellung
2	1	Karton
3	1	Karton
4	1	Karton
5	2	Trockenfutter
6	5	Mausaroma
7	5	Karotten
8	5	Flugasche
9	2	Beipackzettel
10	1	Lieferschein
...	...	...

Abfragen in Adjacence Lists (1/2)

- Hole ein Objekt und alle seine Nachfolger

```
SELECT o1.*, o2.*, o3.* … on.*
FROM object AS o1                              Level 1
    LEFT OUTER JOIN object AS o2
    ON o2.parent = o1.id                       Level 2
    LEFT OUTER JOIN object AS o3
    ON o3.parent = o2.id                       Level 3
    …
    LEFT OUTER JOIN object AS on
    ON o_n.parent = o_n-1.id                   Level n
    WHERE o1.id = (irgendwas);
```

  - unpraktisch bei Hierarchien unbekannter Tiefe

id	parent	value
1	NULL	Bestellung
2	1	Karton
3	1	Karton
4	1	Karton
5	2	Trockenfutter
6	5	Mausaroma
7	5	Karotten
8	5	Flugasche
9	2	Beipackzettel
10	1	Lieferschein
...	...	...

- Alternativ: setze die Hierarchie im Client zusammen

```
SELECT * FROM object
```

  - funktioniert nur bei sehr kleinen Datenbanken, extrem ineffizient wenn nur eine Teilhierarchie gebraucht wird

Abfragen in Adjacence Lists (2/2)

- Wenn das DBMS rekursive Sichten beherrscht

```
WITH hierarchy (id, parent, value, depth)
AS (
    SELECT id, parent, value, 0 AS depth
    FROM object WHERE parent IS NULL
UNION ALL
    SELECT o.id, o.parent, o.value, h.depth+1 AS depth
    FROM hierarchy h JOIN object o ON (h.id = o.parent)
)

SELECT * FROM hierarchy WHERE (irgendwas);
```

id	parent	value
1	NULL	Bestellung
2	1	Karton
3	1	Karton
4	1	Karton
5	2	Trockenfutter
6	5	Mausaroma
7	5	Karotten
8	5	Flugasche
9	2	Beipackzettel
10	1	Lieferschein
...	...	...

Achtung, ungetesteter Code!

Insert, Update, Delete

- Neuer Blattknoten ist effizient einzufügen
  INSERT INTO object(parent, value)
  VALUES (1, 'Rechnung');

- Knoten sind effizient zu verschieben
  UPDATE object SET parent = 2
  WHERE id = 6;

- Knoten incl. Nachfolgern sind leicht zu löschen
  DELETE FROM object WHERE id = 2;
  - jedenfalls wenn ON DELETE CASCADE gesetzt ist, sonst sehr mühsam

id	parent	value
1	NULL	Bestellung
2	1	Karton
3	1	Karton
4	1	Karton
5	2	Trockenfutter
6	5	Mausaroma
7	5	Karotten
8	5	Flugasche
9	2	Beipackzettel
10	1	Lieferschein
...	...	...

Wann Adjacence Lists nutzen?

- Viele Schreiboperationen
- Leseoperationen hauptsächlich nach dem direkten Kind/dem Vater eines Knotens
- performante Integritätssicherung auf DB-Ebene ist essentiell
  - In Adjacence Lists simpel über referentielle Integrität von Fremdschlüsselbeziehungen

id	parent	value
1	NULL	Bestellung
2	1	Karton
3	1	Karton
4	1	Karton
5	2	Trockenfutter
6	5	Mausaroma
7	5	Karotten
8	5	Flugasche
9	2	Beipackzettel
10	1	Lieferschein
...	...	...

Path Enumeration

- Idee: speichere zu jedem Knoten den Pfad bis Wurzel

```
CREATE SEQUENCE objectid START 0;
CREATE TABLE object (
        path        TEXT PRIMARY KEY,
        value       TEXT
);
```

  - Objekt-ID steht am Pfadende
  - Mengenwertige Attribute mit Separator: **1. Normalform verletzt**

- Anfragen nach Teilhierarchien mit LIKE
  - z.B. alle Objekte, die in id=2 enthalten sind

```
SELECT * FROM object
WHERE path LIKE '1>2>%';
```

findet 1>2>5, 1>2>5>6, 1>2>5>7, 1>2>5>8, 1>2>9

path	value
1	Bestellung
1>2	Karton
1>3	Karton
1>4	Karton
1>2>5	Trockenfutter
1>2>5>6	Mausaroma
1>2>5>7	Karotten
1>2>5>8	Flugasche
1>2>9	Beipackzettel
1>10	Lieferschein
	...

Insert, Update, Delete

- Neuer Blattknoten ist effizient einzufügen

```
INSERT INTO object(path, value)
        VALUES ('1>2>' || nextval(objectid) , 'Rechnung');
```

  - *Anm.: Konstruktion des Pfades aus gegebenem Teilbaum und Sequenz objectid*

- Knoten sind effizient zu verschieben

```
UPDATE object SET path = '1>3>6'
        WHERE path = '1>2>5>6';
```

- Knoten incl. Nachfolgern sind leicht zu löschen

```
DELETE FROM object
        WHERE path LIKE '1>3>%';
```

path	value
1	Bestellung
1>2	Karton
1>3	Karton
1>4	Karton
1>2>5	Trockenfutter
1>2>5>6	Mausaroma
1>2>5>7	Karotten
1>2>5>8	Flugasche
1>2>9	Beipackzettel
1>10	Lieferschein
	...

Wann Path Enumerations nutzen

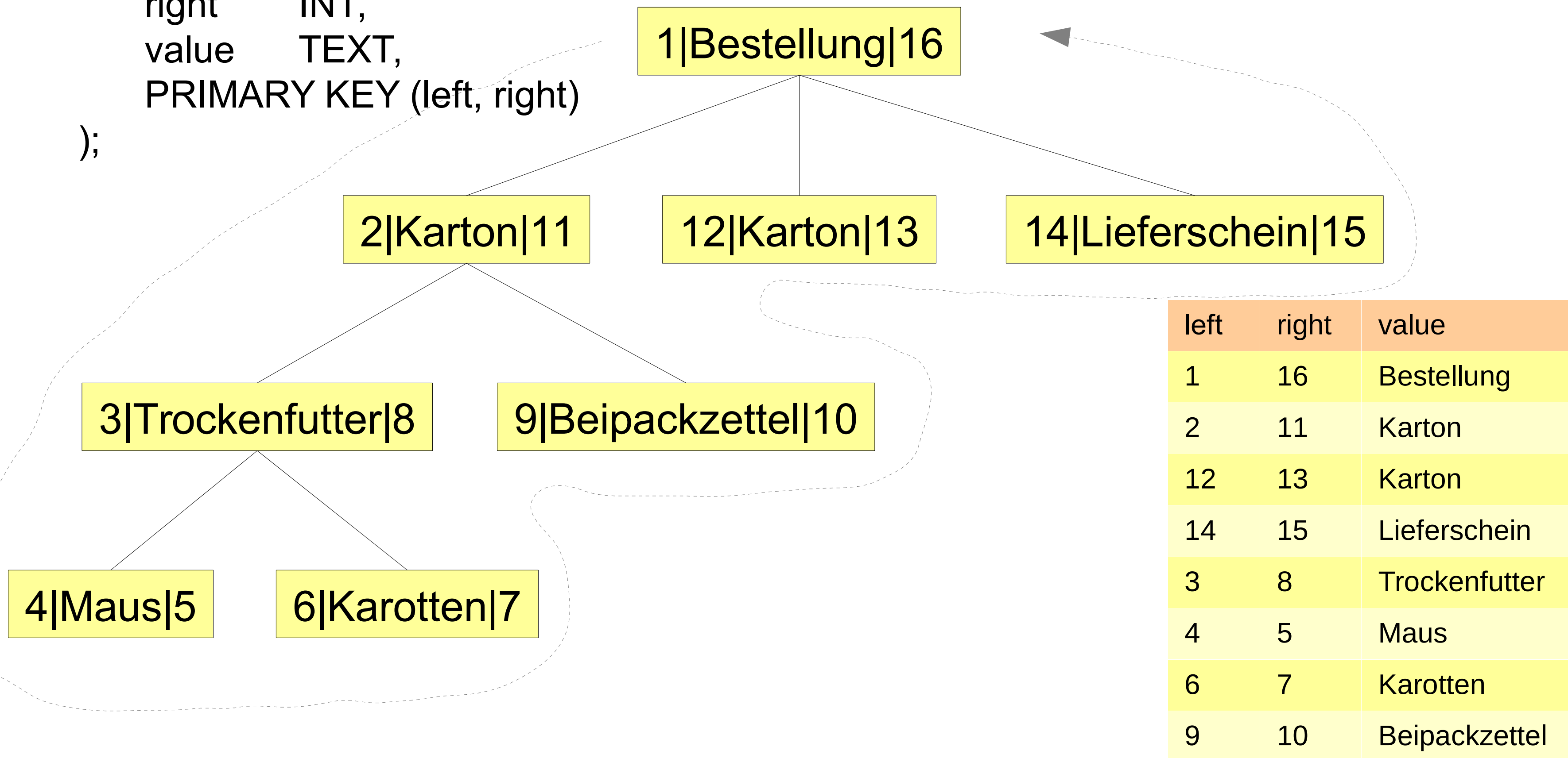
- DBMS kann
  - sicherstellen, dass es einen Pfad nur einmal gibt
  - Korrektheit des Ausdrucks in path <u>nicht garantieren</u>
  - Existenz aller Knoten in path <u>nicht garantieren</u>
    - (oder beides höchstens über ineffiziente Trigger mit String-Processing)
- Teile der Datenbank-Logik liegen im Client-Code
  - Schwierigkeiten bei Transaktionen, da Teile davon beim Client ablaufen
  - **Niemals gegen die 1. Normalform verstoßen,
es sei denn man hat einen sehr guten Grund dafür!**
- Path Enumerations **nur** wenn Performanz wichtiger als Konsistenz

Nested Sets

Anm: left und right sind in SQL verbotene Schlüsselworte, hier nur zur besseren Verständlichkeit verwendet!

- Idee: speichere einen Tiefensuche-Durchlauf in *left,right* Attributen

```
CREATE TABLE object (
    left      INT,
    right     INT,
    value     TEXT,
    PRIMARY KEY (left, right)
);
```

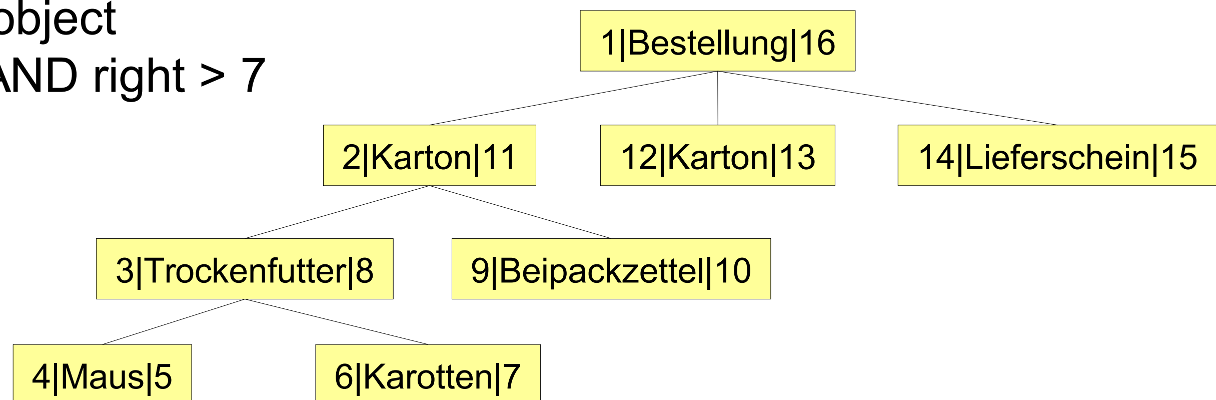


left	right	value
1	16	Bestellung
2	11	Karton
12	13	Karton
14	15	Lieferschein
3	8	Trockenfutter
4	5	Maus
6	7	Karotten
9	10	Beipackzettel

Anfragen in Nested Sets (1/3)

- Suche nach allen enthaltenen Objekten
  - alles innerhalb der Schranken von left und right
  - Karton (2,11) und alles was darin ist
    SELECT * FROM object
      WHERE left >= 2 AND right <= 11;
- Suche nach allen übergeordneten Objekten
  - alles außerhalb der Schranken von left und right
  - Wozu gehören Karotten (6,7)
    SELECT * FROM object
      WHERE left < 6 AND right > 7

left	right	value
1	16	Bestellung
2	11	Karton
12	13	Karton
14	15	Lieferschein
3	8	Trockenfutter
4	5	Maus
6	7	Karotten
9	10	Beipackzettel



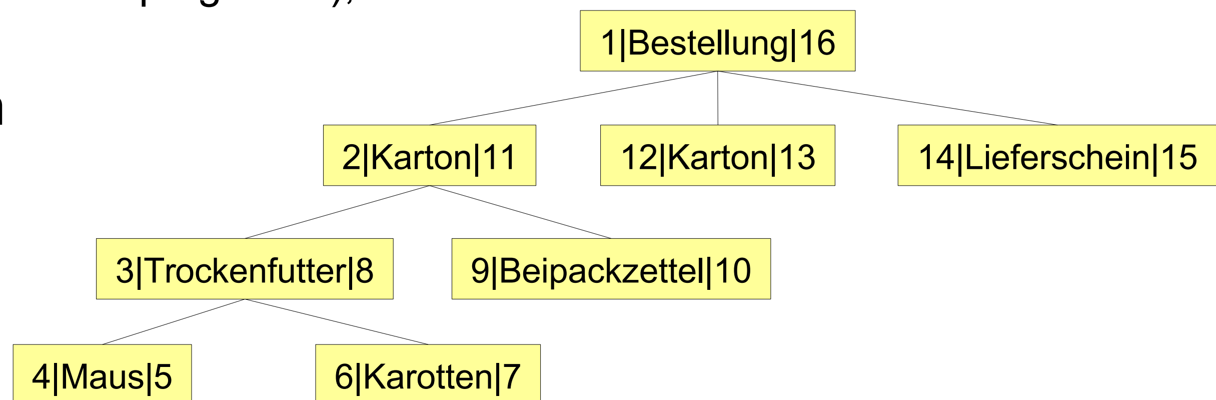
Anfragen in Nested Sets (2/3)

- Suche nach direktem Vorgänger (Join)
  - Vorgänger von Karotten (6,7) hat Schranken außerhalb von left und right, kein weiterer Knoten existiert zwischen Vorgänger und Karotten

```
SELECT o.* FROM object AS o
WHERE o.left < 6 AND o.right > 7
    AND NOT EXISTS (
            SELECT p.left FROM object AS p
            WHERE o.left < p.left AND o.right > p.right
            AND p.left < 6 AND p.right > 7);
```

- Suche nach direktem Nachfolger ebenso

left	right	value
1	16	Bestellung
2	11	Karton
12	13	Karton
14	15	Lieferschein
3	8	Trockenfutter
4	5	Maus
6	7	Karotten
9	10	Beipackzettel



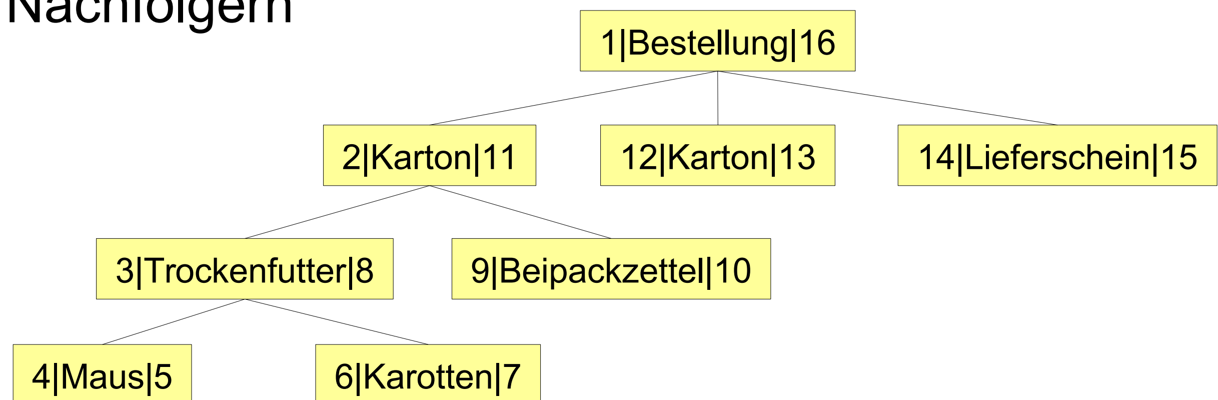
Anfragen in Nested Sets (3/3)

- Suche nach direktem Vorgänger (Sortierung)
  - Vorgänger von Karotten (6,7) hat Schranken außerhalb von left und right und maximalen Wert für left

```
SELECT o.* FROM object AS o
WHERE o.left < 6 AND o.right > 7
ORDER BY o.left DESC
LIMIT 1;
```

left	right	value
1	16	Bestellung
2	11	Karton
12	13	Karton
14	15	Lieferschein
3	8	Trockenfutter
4	5	Maus
6	7	Karotten
9	10	Beipackzettel

- Suche nach direkten Nachfolgern so nicht möglich, mehrere Nachfolger denkbar



Insert, Update, Delete

- Neuer Blattknoten ist schwierig einzufügen
  - Neuberechnung aller left, right-Werte größer als der left-Wert des eingefügten Knotens

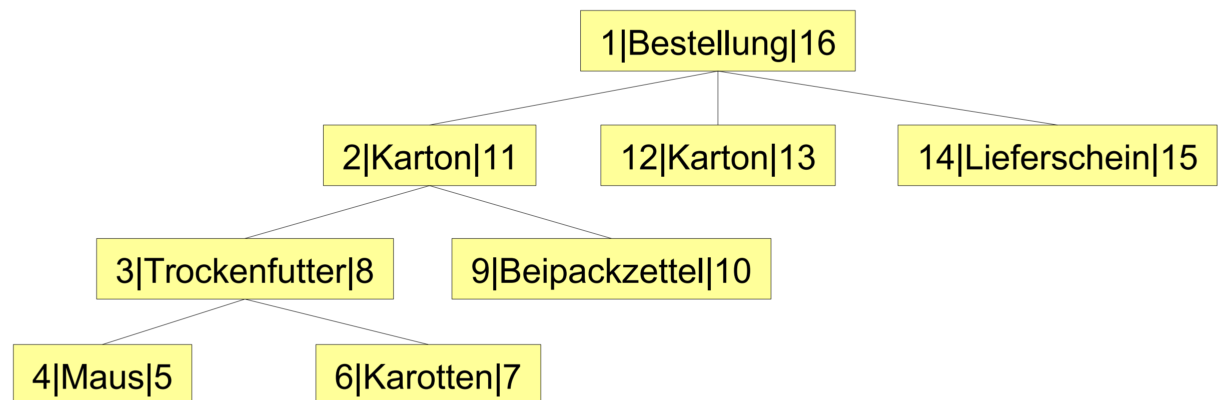
- Knoten sind kompliziert verschieben
  - es sei denn zwei Knoten können Plätze tauschen

- Knoten incl. Nachfolgern sind leicht zu löschen
  DELETE FROM object WHERE left >= 2 AND right <= 11;

- einzelne Knoten sind leicht zu löschen
  DELETE FROM object WHERE left = 2;
  - Kinder von Knoten mit left=2 werden automatisch Kinder des Vaterknotens

Wann Nested Sets nutzen

- DBMS kann über Uniqueness-Constraints und Trigger die Korrektheit der left, right-Werte überwachen
- Aber: Berechnung der left, right-Werte beim Einfügen und Updaten liegen im Client-Code
  - Schwierigkeiten bei Transaktionen, da Teile davon beim Client ablaufen
- Gut für häufige Anfragen, seltene Einfüge- und Update-Operationen



Closure Tables

- Idee: speichere ALLE möglichen Pfade als Vorgänger-Nachfolger-Beziehung in separater Relation

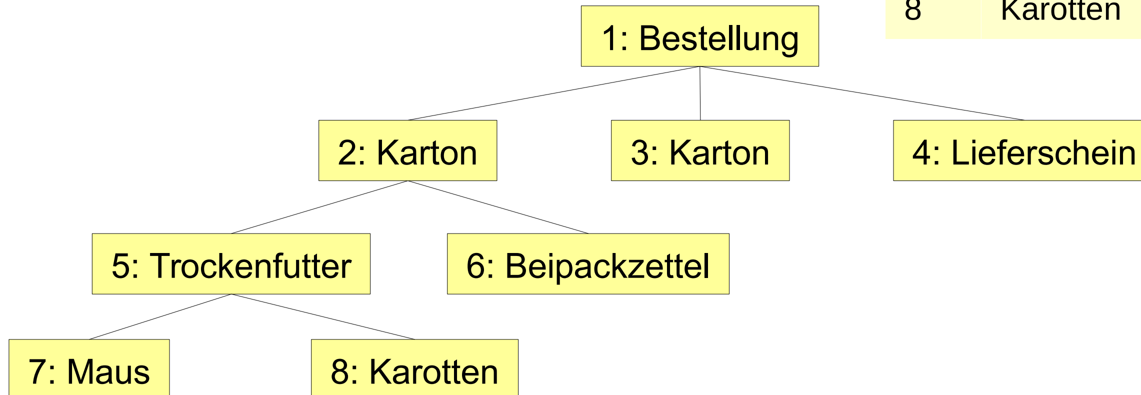
```
CREATE TABLE object (
        id          SERIAL PRIMARY KEY,
        value       TEXT
);
CREATE TABLE path (
        anc         INT,
        desc        INT,
        PRIMARY KEY (anc, desc),
        FOREIGN KEY (anc) REFERENCES object(id),
        FOREIGN KEY (desc) REFERENCES object(id)
);
```

object

id	value
1	Bestellung
2	Karton
3	Karton
4	Lieferschein
5	Trockenfutter
6	Beipackzettel
7	Maus
8	Karotten

path

anc	desc
1	1
1	2
1	3
1	4
1	5
1	6
1	7
1	8
2	2
2	5
2	6
2	7
2	8
3	3
...	...



Anfragen in Closure Tables

- Suche nach enthaltenen Objekten sehr einfach
  - Alles im Karton mit ID=2:
    ```
    SELECT * FROM object
         JOIN path ON id = desc
    WHERE anc = 2;
    ```
- Suche nach Vorgängern ebenso einfach
  - Worin ist Beipackzettel 6 enthalten?
    ```
    SELECT * FROM object
      JOIN path ON id = desc
    WHERE desc = 6;
    ```

object

id	value
1	Bestellung
2	Karton
3	Karton
4	Lieferschein
5	Trockenfutter
6	Beipackzettel
7	Maus
8	Karotten

path

anc	desc
1	1
1	2
1	3
1	4
1	5
1	6
1	7
1	8
2	2
2	5
2	6
2	7
2	8
3	3
...	...

Insert

- Neue Blattknoten einfügen:
  - Kopiere alle Pfade des Vorgängerknotens und setze dabei desc auf eigene ID, füge eigenen Pfad hinzu
  - z.B. Rechnung in Karton 2 unter ID 9:

```
INSERT INTO object VALUES (9, 'Rechnung');

INSERT INTO path (anc, desc)
     SELECT (anc, 9)
     FROM path
     WHERE desc = 2;

INSERT INTO path (anc, desc) VALUES (9,9);
```

object

id	value
1	Bestellung
2	Karton
3	Karton
4	Lieferschein
5	Trockenfutter
6	Beipackzettel
7	Maus
8	Karotten

path

anc	desc
1	1
1	2
1	3
1	4
1	5
1	6
1	7
1	8
2	2
2	5
2	6
2	7
2	8
3	3
...	...

Update, Delete

- Verschieben eines Knotens
  - relativ kompliziert, weil Pfade vielfach redundant gespeichert
- Knoten incl. Nachfolgern löschen ist einfach

  DELETE FROM path WHERE desc IN
  (SELECT desc FROM path WHERE anc = 2);

  - jedenfalls wenn ON DELETE CASCADE gesetzt ist, sonst müssen die Objekte separat gefunden und gelöscht werden

object

id	value
1	Bestellung
2	Karton
3	Karton
4	Lieferschein
5	Trockenfutter
6	Beipackzettel
7	Maus
8	Karotten

path

anc	desc
1	1
1	2
1	3
1	4
1	5
1	6
1	7
1	8
2	2
2	5
2	6
2	7
2	8
3	3
...	...

Wann Closure Tables nutzen?

- Closure Tables im Vergleich zu Nested Sets
  - Closure Tables sind beinahe ebenso performant abzufragen
  - Closure Tables haben effizientere Einfügeoperationen
    - kein Client-Code zum Berechnen der left,right-Werte nötig, alles innerhalb einer DBMS-Transaktion
  - Closure Tables benötigen **wesentlich** mehr Speicherplatz
    - Redundante Informationen, ggf. Konsistenzprobleme
  - Verschieben von Knoten in beiden Strukturen eher aufwendig

Zusammenfassung Hierarchien

	Effiziente Anfragen	Effiziente Änderungen	Speicher-verbrauch	Integritätssicherung
BLOBs	nein	nein	gering	nein
Adjacence Lists	nein	ja	gering	ja
Path Enumeration	ja	ja	gering	nein (oder Trigger mit String-Verarbeitung)
Nested Sets	ja	nein	gering	ja (Trigger)
Closure Tables	ja	ja	hoch	ja

Anmerkung: ORM-Frameworks verwenden Adjacence Lists wegen der sehr einfachen Integritätssicherung

Foto: M. Strauch

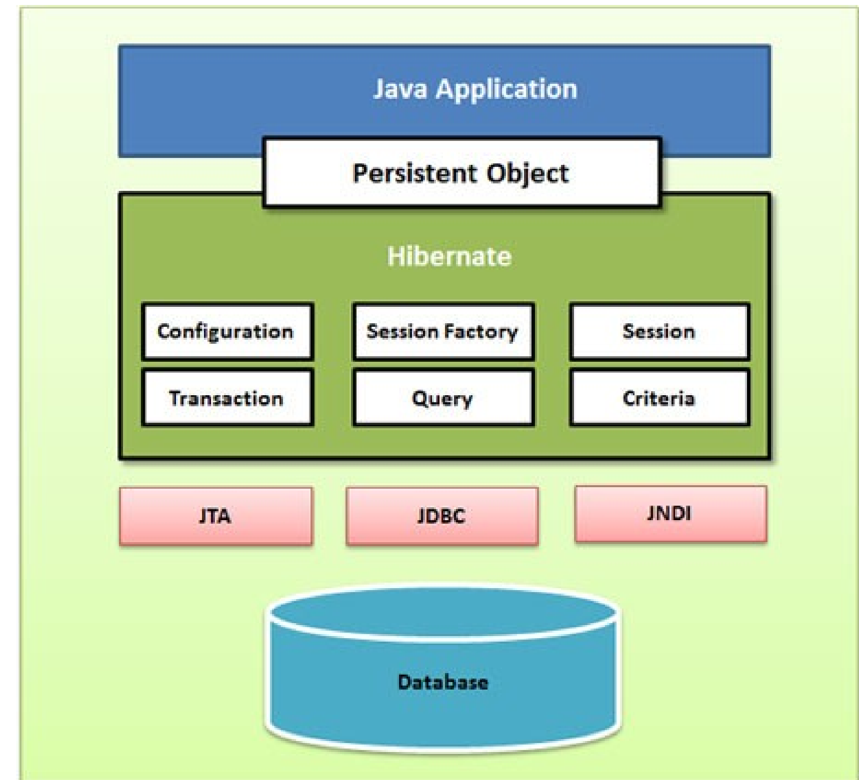
Objektrelationales Mapping

- Ziel: DBMS-Logik vor dem Programmierer kapseln
  - Abbildung von Objekten auf Relationen, automatische Schlüsselverwaltung
  - DBMS-Client-übergreifendes Transaktionsmanagement
  - bedarfsabhängige Lese/Schreiboperationen unterstützen (nur Objekte aus der DB lesen die tatsächlich verwendet werden)
  - Unabhängigkeit vom verwendeten DBMS
- Java ORM Frameworks
  - DataNucleus
  - Enterprise Java Beans
  - **Hibernate** ← *im Folgenden*
  - Java Data Objects
  - und viele mehr...
    http://en.wikipedia.org/wiki/List_of_object-relational_mapping_software

Überblick über Hibernate

Hibernate als Persistenz-Layer zwischen DBMS und OO:

- deklarative Konfiguration von DBMS-Zugang und Zugriffsstrategien
  - hibernate.cfg.xml
- Session-Management für Datenbank-Verbindung
- Transaktionsmanagement
- Query, Criteria bieten trans-parentes Anfragemanagement



Grafik: tutorialspoint.com

Ein einfaches Beispiel

- Folgende Klasse soll in PostgreSQL gespeichert werden

```
1  package lecture;
2
3  public class Katzenfutter {
4      private int kid;
5      private String name;
6      private double preis;
7
8⊕     public Katzenfutter() {...
10
11⊕    public Katzenfutter(String name, double preis) {...
15
16⊕    public int getKid() {...
19
20⊕    public void setKid(int kid) {...
23
24⊕    public String getName() {...
27
28⊕    public void setName(String name) {...
31
32⊕    public double getPreis() {...
35
36⊕    public void setPreis(double preis) {...
39
40 }
```

Katzenfutter
Name Preis
getName() setName() get... ...

Vorbereiten der Datenbank

- Nutzer anlegen
  CREATE ROLE futtermeister WITH PASSWORD '123' LOGIN;

- Relation anlegen, auf die 'futtermeister' Schreibrechte hat

```
CREATE TABLE katzenfutter (
        kid        integer PRIMARY KEY,
        name       character varying(255),
        preis numeric(5,2)
);
```

- Sequenz anlegen, auf die 'futtermeister' zugreifen kann
  CREATE SEQUENCE katzenfutter_id_seq;

Konfiguration

- hibernate.cfg.xml
  - SQL-Dialekt, JDBC-Treiber, Ort der Datenbank, Nutzername, Passwort
  - Verweis auf die Dateien mit dem objektrelationalen Mapping

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration SYSTEM
"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
  <session-factory>

    <property name="hibernate.dialect">org.hibernate.dialect.PostgreSQLDialect</property>
    <property name="hibernate.connection.driver_class">org.postgresql.Driver</property>
    <property name="hibernate.connection.url">jdbc:postgresql://localhost/lecture</property>
    <property name="hibernate.connection.username">futtermeister</property>
    <property name="hibernate.connection.password">123</property>

    <mapping resource="lecture/Katzenfutter.hbm.xml" />

  </session-factory>
</hibernate-configuration>
```

Das Mapping

- Klassen und Vererbungshierarchien
  - ggf. Objekt mit JOIN aus mehreren Tabellen zusammensetzen
- Objektattribute
  - Attribut kann keiner, einer oder mehreren Tabellenspalten entsprechen
  - einfachster Fall: ein Objektattribut entspricht einer Spalte in der DB
  - Meta-Informationen (Constraints, Primärschlüsselattribute)
- Objektbeziehungen
  - 1:1, 1:N, M:N-Beziehungen, uni- und bidirektional, Containertypen
  - in der DB: Fremdschlüsselbeziehungen + Hilfstabellen
- Ladestrategie
  - Lazy: lade nur Objekt selbst, verbundene Objekte holt Programmierer
  - Eager: lade bei Objektzugriff alle damit verbundenen Objekte

Mapping für Klassen ohne Vererbungsbeziehung

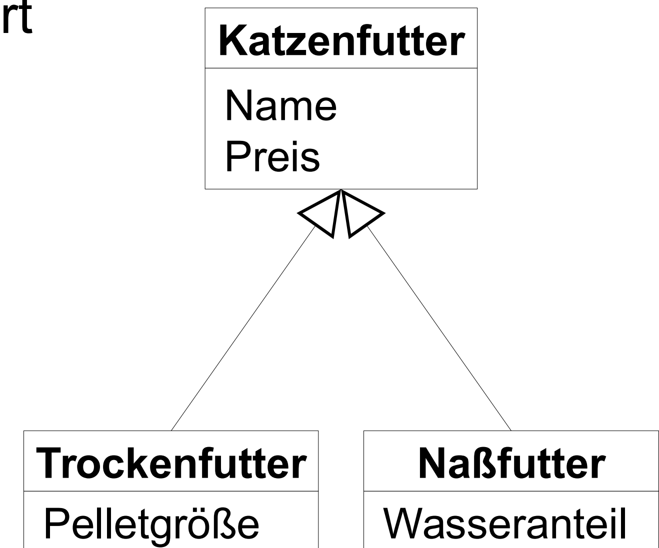
- Katzenfutter.hbm.xml
  - Welche SEQUENCE für den Primärschlüssel nutzen?
  - Wie heißt der Primärschlüssel?
  - Auf welche DB-Spalten werden die Attribute abgebildet?

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
 "-//Hibernate/Hibernate Mapping DTD//EN"
 "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
    <class name="lecture.Katzenfutter" table="katzenfutter">
        <id name="kid" type="integer" column="kid">
            <generator class="sequence">
                <param name="sequence">katzenfutter_id_seq</param>
            </generator>
        </id>
        <property name="name" column="name" type="string" />
        <property name="preis" column="preis" type="double" />
    </class>
</hibernate-mapping>
```

Mapping von Vererbung: Eine Relation pro Hierarchie

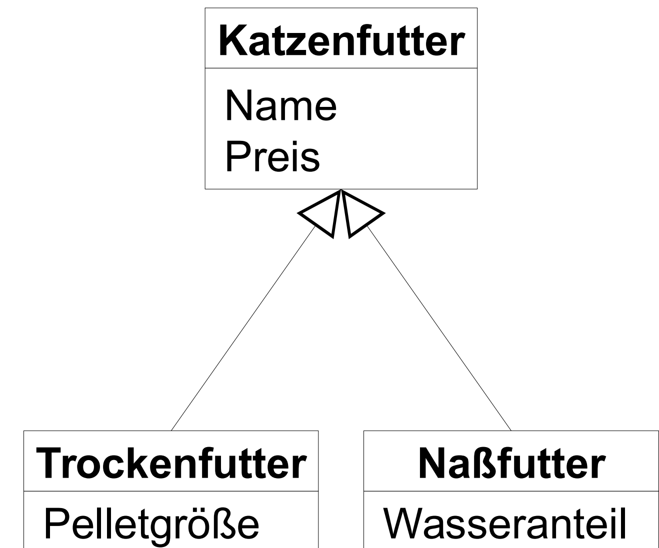
- Alle Attribute in einer Basisrelation gespeichert
  Mapping: <subclass … >
- Viele NULL-Werte
- NOT NULL-Constraints in den Unterklassen nicht möglich



kid	name	preis	pelletgröße	wasseranteil
1	Mausi	5.30	NULL	NULL
2	Saarlands Bestes	12.48	5	NULL
3	Cheapy	1.27	3	NULL
4	Blubb	6.70	NULL	0.70

Mapping von Vererbung: Eine Relation pro Unterklasse

- Eine Relation für die Basisklasse, für jede Unterklasse eine eigene weitere Relation mit den zusätzlichen Attributen der Unterklasse
  Mapping: <joined-subclass … >
- Objekte werden mit JOIN von Hibernate automatisch zusammengesetzt



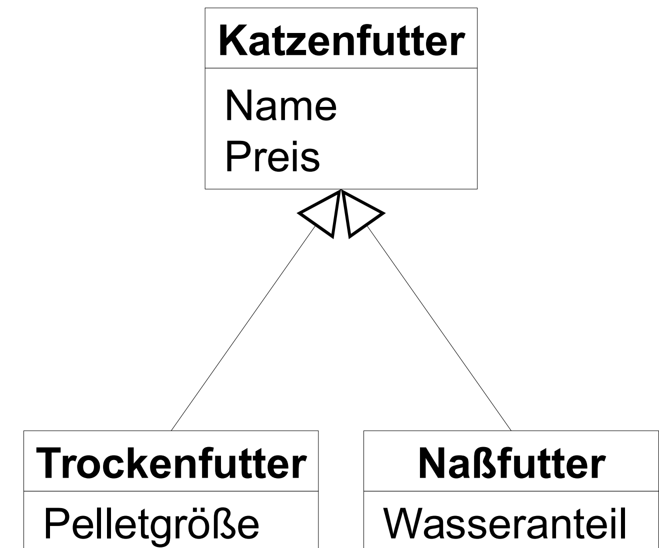
kid	name	preis
1	Mausi	5.30
2	Saarlands Bestes	12.48
3	Cheapy	1.27
4	Blubb	6.70

kid	pelletgröße
2	5
3	3

kid	wasseranteil
4	0.70

Mapping von Vererbung: Relation pro konkreter Klasse

- Für jede Klasse eine eigene Relation mit allen Attributen der Klasse
  Mapping: <union-subclass … >
- Objekte müssen nicht zusammengesetzt werden
- Änderungen am Vererbungsschema schwierig



kid	name	preis
1	Mausi	5.30

kid	name	preis	pelletgröße
2	Saarlands Bestes	12.48	5
3	Cheapy	1.27	3

kid	name	preis	wasseranteil
4	Blubb	6.70	0.70

Objekte schreiben mit Hibernate

```
// Konfiguration einlesen
SessionFactory factory = null;
try {
    Configuration cfg = new Configuration().configure("hibernate.cfg.xml");
    StandardServiceRegistryBuilder sb = new StandardServiceRegistryBuilder();
    StandardServiceRegistry registry = sb.applySettings(cfg.getProperties()).build();
    factory = cfg.buildSessionFactory(registry);
} catch (Exception e) {
    System.err.println(e);
    System.exit(-1);
}

// Datenobjekt anlegen
Katzenfutter k = new Katzenfutter("Maunz", 10.75);

// Datenbankverbindung öffnen, Transaktion beginnen, Objekt speichern
Transaction tx = null;
try {
    Session session = factory.openSession();
    tx = session.beginTransaction();
    session.save(k);
    tx.commit();
    session.close();

} catch (Exception e) {
    // Wenn etwas schief gegangen ist, Rollback
    if (tx != null) {tx.rollback();}
    System.err.println(e);
    System.exit(-1);
}
System.exit(1);
```

das ist der spannende Teil

Anfragen in Hibernate

- Am einfachsten über Criteria-Objekt
  - bekommt Klasse des Objekts, ggf. Prädikate als Restrictions
  - liefert List-Objekt mit den Ergebnissen zurück

```
// Datenbankverbindung öffnen, Transaktion beginnen, Objekt laden
Transaction tx = null;
try {
    Session session = factory.openSession();
    tx = session.beginTransaction();

    Criteria c = session.createCriteria(Katzenfutter.class);
    c.add(Restrictions.gt("preis", 5.0));
    List<Katzenfutter> l = c.list();

    for (int i = 0; i < l.size(); i++) {
        System.out.println(l.get(i));
    }
    tx.commit();
    session.close();

} catch (Exception e) {
```

das ist der spannende Teil

Zusammenfassung Hibernate

- Sie haben gesehen
  - Konfiguration von Hibernate
  - flexibles Mapping von Objekten auf Relationen
  - einfache Lese- und Schreiboperationen
- Folgendes wurde ausgelassen
  - Mapping von Lists, Collections und Bags (Sets)
  - Uni- und bidirektionales Mapping von 1:1, 1:N und N:M-Assoziationen
  - Lazy- und Eager-Loading
  - Hibernate Query Language (HQL, ähnlich mit SQL ohne SELECT)
    Query q = session.createQuery(„FROM katzenfutter k WHERE k.preis > 5“);

Zum Abschluss

Foto: M. Strauch

Wie geht es weiter?

- bis Montag, 06.07., 12 Uhr
  - Quiz: Normalformen
- Dienstag, 07.07., GHH 12-14 Uhr: Tutoriumstermin
  - kurze Besprechung von Aufgabenblatt 9
  - nächstes Aufgabenblatt: Trigger
- Donnerstag, 09.07.: Präsenztermin
  - Sensornetze als relationales DBMS