Objekt-Relationale Abbildung

Lehrveranstaltung Datenbanktechnologien

Prof. Dr. Ingo Claßen   Prof. Dr. Martin Kempa

Hochschule für Technik und Wirtschaft Berlin

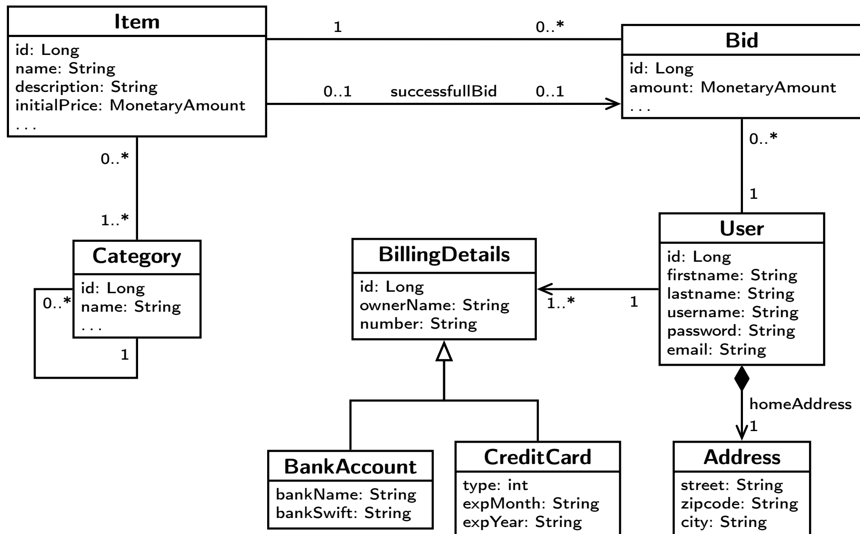
Objekt-Relationale Abbildung
- Allgemeine Abbildung
- Java Persistence API

© Ingo Claßen, Martin Kempa

Abbildung von objekt-orientierten Strukturen auf Tabellen

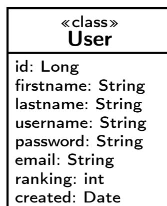
- Das objekt-orientierte Modell enthält reichere Strukturierungsmechanismen als das relationale Modell
  - Eine Transformation auf struktureller Ebene ist notwendig
- Folgende Konzepte müssen transformiert werden
  - Klassen
  - Komplexe Datentypen
  - Assoziationen
  - Kompositionen
  - Vererbung

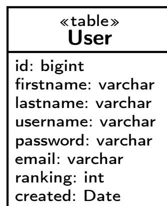
Domänen-Modell für Internet-Auktionen



Transformation von Klassen

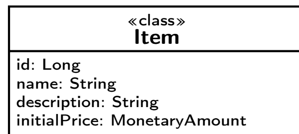
- Klassen werden zu Tabellen
  - Standardvariante: Eine Klasse wird eine Tabelle
  - Andere Verhältnisse denkbar: z. B. eine Klasse auf mehrere Tabellen
- Attribute werden zu Spalten
  - Standarddatentypen werden auf korrespondierende Typen in der Datenbank abgebildet
  - Komplexe Typen, die keine Korrespondenz im Typsystem der Datenbank haben, müssen gesondert behandelt werden, z. B. durch programmtechnische Transformation

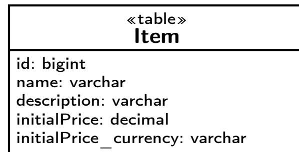




Transformation von komplexen Attributen

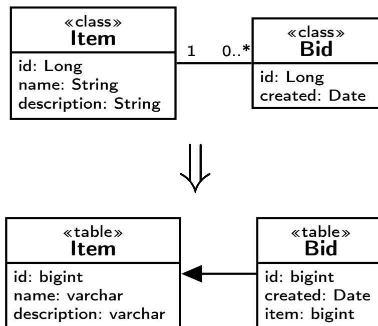
- `MonetaryAmount` ist eine Klasse die Geldbeträge inklusive Währungsangabe abbildet
- Wird in der Datenbank durch zwei Attribute gespeichert
  - Transformation erfolgt durch Programmcode





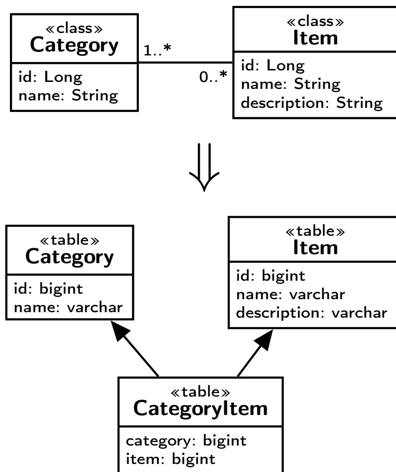
Transformation von Assoziationen (1)

- 1:n-Assoziationen
  - Fremdschlüssel auf der n-Seite



Transformation von Assoziationen (2)

- n:m-Assoziationen
  - Zwischentabelle
  - Fremdschlüssel auf beide Seiten



Unidirektionale Beziehungen

- Bid enthält eine Referenz auf Item

```
class Item {
  ...
}
```

```
class Bid {
  ...
  private Item item;

  ...
  public Item getItem() {
    return item;
  }
  public void setItem(Item item) {
    this.item = item;
  }
}
```

Bidirektionale Beziehungen

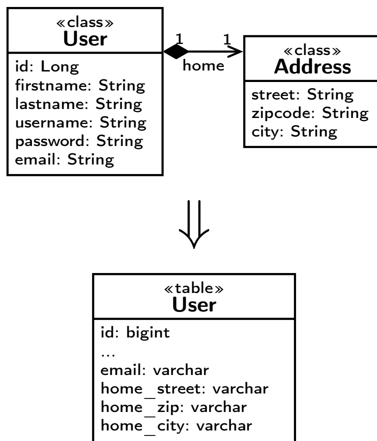
- Bidirektionale Version zwischen `Item` und `Bid`

```
class Item {
  ...
  private Set<Bid> bids;

  ...
  public Set<Bid> getBids() {
    return bids;
  }
  // kein setBids, sondern
  // addBid zur korrekten Ab-
  // bildung der 1:n-Semantik
  public void addBid(Bid bid) {
    bid.getItem().getBids()
      .remove(bid);
    bid.setItem(this);
    bids.add(bid);
  }
}
```

```
class Bid {
  ...
  private Item item;

  ...
  public Item getItem() {
    return item;
  }
  public void setItem(Item item) {
    this.item = item;
  }
}
```

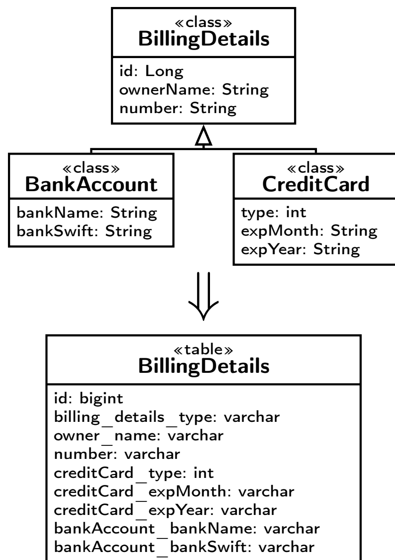
Transformation von Kompositionen

- Unterscheidung zwischen Entity-Typen und Wert-Typen
- Entity-Typen
  - Haben eine eigene Datenbankidentität und einen Lebensyklus
  - Existieren unabhängig von anderen Entity-Typen
- Wert-Typen
  - Haben keine Datenbankidentität
  - Ihre Daten werden in die Daten der besitzenden Entität eingebettet
  - Sind abhängig von anderen Entity-Typen



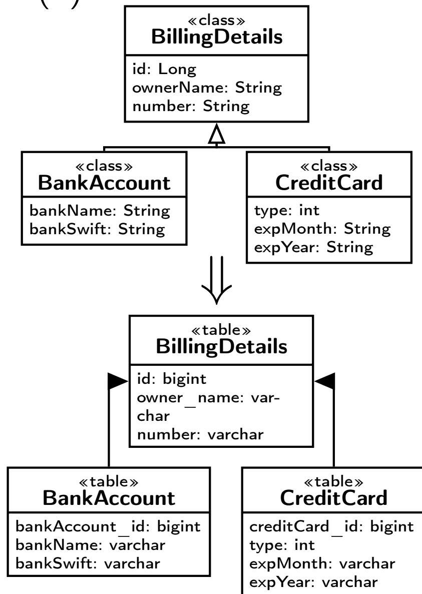
Transformation von Vererbungen (1)

- Eine Tabelle für die gesamte Klassenhierachie
- Vorteile
  - Einfache Struktur
  - Keine Verbundoperationen notwendig
  - Polymorphe Beziehungen und Abfragen möglich
- Nachteile
  - Datenkonsistenz schwieriger zu gewährleisten, da Spalten aus den abgeleiteten Klassen keine `not-null`-Beschränkung haben können



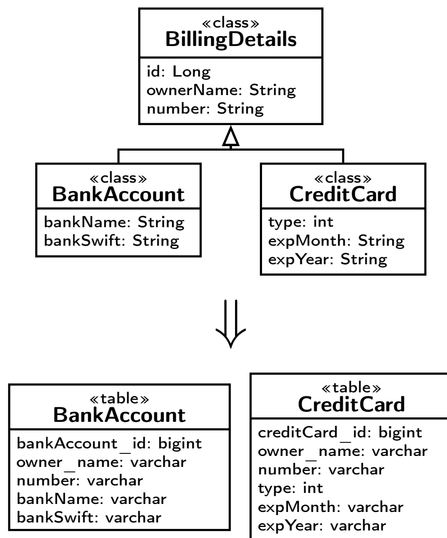
Transformation von Vererbungen (2)

- Eine Tabelle pro Unterklasse
- Vorteile
  - Polymorphe Beziehungen und Abfragen möglich
  - Datenkonsistenz in Bezug auf `not-null`-Spalten bleibt erhalten
- Nachteile
  - Verbundoperationen notwendig



Transformation von Vererbungen (3)

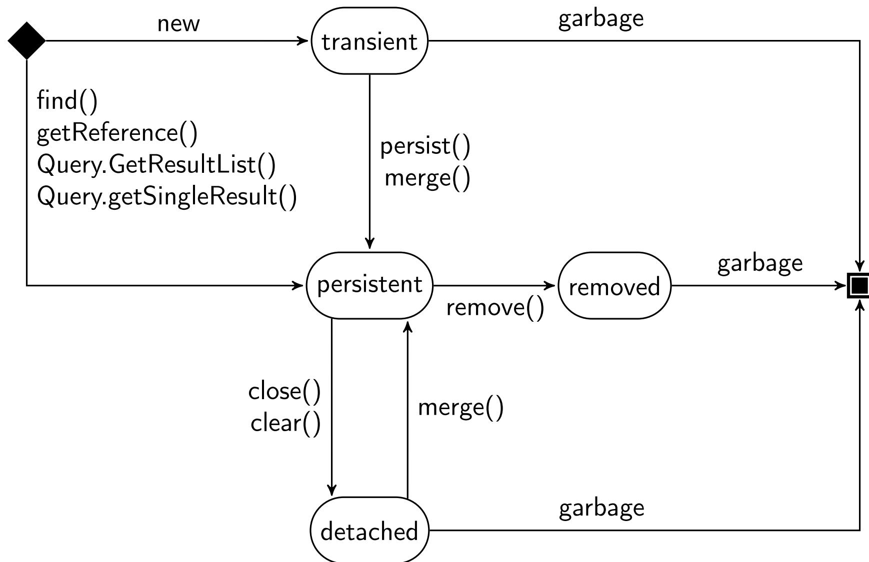
- Eine Tabelle pro konkreter Unterklasse
- Vorteile
  - Datenkonsistenz in Bezug auf `not-null`-Spalten bleibt erhalten
  - Keine Verbundoperationen notwendig
- Nachteile
  - Polymorphe Beziehungen und Abfragen nicht möglich



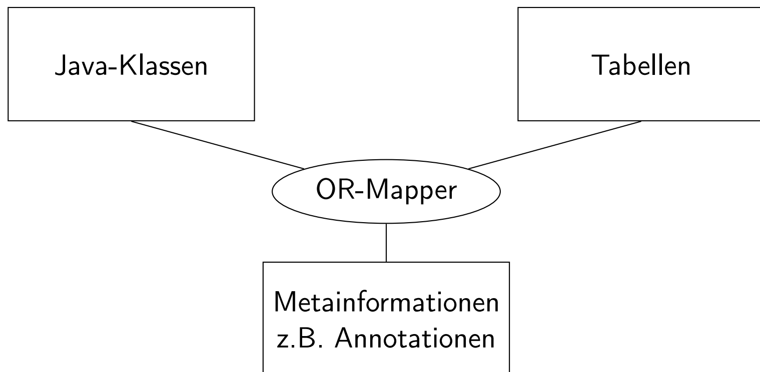
Java Persistence API (JPA)

- Persistenz für Java-Objekte
  - Einfaches Klassenmodell (POJOs - Plain Old Java Objects)
- Transformation von OO-Konzepten
  - Klassen, komplexe Datentypen
  - Assoziationen, Kompositionen, Vererbung
- Funktionalität
  - Laden von Objekten, Ladestrategien für Assoziationen, Dirty-Management
  - Speichern von Objekten, Speicherstrategien für Assoziationen, Persistence by Reachability
  - Beziehungsverwaltung
  - Zustandsverwaltung und Identifikation von Objekten
  - Objektorientierte Abfragesprache
  - Transaktionen
  - Caching

Persistenz-Lebenszyklus für Geschäftsobjekte



OR-Mapper



OR-Mapping für Klasse Item (1)

```
@Entity @Table(name = "TBL_ITEM")
@javax.persistence.SequenceGenerator(
  name="seq_Item", sequenceName="SEQ_ITEM")
public class Item {
  ...
  @Id
  @GeneratedValue(
    strategy=GenerationType.SEQUENCE,
    generator="seq_Item")
  @Column(name = "ITEM_ID")
  public Long getId() { ... }
  ...
}
```

OR-Mapping für Klasse Item (2)

```
...
public class Item {
  ...
  @Column(name = "START_DATE", nullable = false,
    updatable = false)
  public Date getStartDate() { ... }

  @ManyToOne
  @JoinColumn(name = "APPROVED_BY_USER_ID", nullable = true)
  public User getApprovedBy() { ... }

  @ManyToOne
  @JoinColumn(name = "SELLER_ID", nullable = false,
    updatable = false)
  public User getSeller() { ... }
  ...
}
```

OR-Mapping für Klasse Item (3)

```
...

public class Item {
  ...

  @OneToMany(cascade = CascadeType.ALL,
    mappedBy = "category")
  @org.hibernate.annotations.Cascade(
    org.hibernate.annotations.CascadeType.DELETE_ORPHAN)
  public Set<CategorizedItem> getCategorizedItems() { ... }

  ...

}
```

OR-Mapping für Klasse BillingDetails

```
@Entity @Table(name = "BILLING_DETAILS")
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class BillingDetails {
  ...
  @Id  @GeneratedValue(...)
  @Column(name = "BILLING_DETAILS_ID")
  public Long getId() { ... }
}

@Entity @Table(name = "BANK_ACCOUNT")
public class BankAccount extends BillingDetails { ... }

@Entity @Table(name = "CREDIT_CARD")
public class CreditCard extends BillingDetails { ... }
```

Objektorientierte Abfragesprache

- Das JPA hat eine eigene Abfragesprache (JPAQL)
- syntaktische Ähnlichkeiten zu SQL
- erweitert um Objektorientierung
- Beispiel: `select i from Item as i`
- kurze Schreibweise: `from Item as i`

Aufbau einer JPAQL-Abfrage

- Wesentliche Schritte:
  1. Abfrage erzeugen
     ```
     Query query = em.createQuery("from User");
     Query query = em.createQuery(
       "from Category as c where c.name like 'Fahrzeug%'");
     ```
  2. Laufzeitargumente setzen
     ```
     String queryString =
       "from Item as i where i.description like :search";
     Query query = em.createQuery(queryString).setParameter(
       "search", searchString);
     Query query = em.createQuery(
       "from Item as i where i.seller = :seller").setParameter(
         "seller", theSeller);
     ```
  3. Abfage ausführen
     ```
     List<Item> l = query.getResultList();
     Bid maxBid = (Bid) em.createQuery(
       "from Bid as b order by b.amount desc").setMaxResults(1)
       .getSingleResult();
     ```

Abfragekonzepte

- Einfache Abfragen
```
from Item
```
- Polymorphe Abfrage
```
from BillingDetails as bd
from java.lang.Object as o
```
- Suchbedingungen
```
from User as u where u.email = 'foo@hibernate.org'
```
- Vergleichsausdrücke
```
from Bid as b where b.amount.value between 1 and 10
from Bid as b where b.amount.value > 100
from User as u where u.email in ('foo@bar', 'bar@foo')
from User as u where u.email is null
from Item as i where i.successfulBid is not null
from User as u where u.firstname like 'C%'
from User as u where u.firstname not like '%sen%'
  and u.email in ('foo@hibernate.org', 'bar@hibernate.org')
```

Operatoren

Operator	Beschreibung
.	Pfad-Ausdruck-Operator zur Navigation
+, -	Einstelliger Vorzeichenoperator
*, /	Multiplikation und Division für numerische Werte
+, -	Addition und Subtraktion für numerische Werte
=, <>, <, >, >=, <=, [not] between, [not] like, [not] in, is [not] null	Zweistellige Vergleichsoperatoren (analog zu SQL)
is [not] empty, [not] member [of]	Zweistellige Operatoren für Collections
not, and, or	Logische Operatoren zur Verknüpfung von boolschen Ausdrücken

Abfragekonzepte

- Vergleichsausdrücke

```
from Bid as b where (b.amount.value / 0.71) - 100.0 > 0.0
from User as u
where (u.firstname like 'S%' and u.lastname like 'C%')
  or u.email in ('foo@hibernate.org', 'bar@hibernate.org')
```

- Ausdrücke mit Listen

```
from Item as i where i.bids is not empty
```

- Funktionen

```
from User as u where lower(u.email) = 'foo@hibernate.org'
from User as u where size(u.billingDetails) = 2
```

- Sortierung

```
from User as u order by u.username
from User as u order by u.lastname asc, u.firstname asc
```

Projektion

- select-Klausel

```
from Item as i, Bid as b

select i.id, i.description, i.initialPrice
from Item as i
where i.endDate > current_timestamp()
```

- distinct

```
select distinct i.description from Item as i
```

- Funktionen

```
select i.startDate, current_date() from Item as i

select i.startDate, i.endDate, upper(i.name) from Item as i
```

Funktionen

Operator	Beschreibung
`upper(S)`, `lower(S)`	Umwandlung eines Stings S in Groß-/Kleinschreibung
`concat(S1, S2)`	Konkatenation zwei Strings S1 mit S2
`substring(S, OFFSET, LENGTH)`	Bildung eines Teilstrings von S mit der Länge LENGTH ab OFFSET
`trim([[both| leading| trailing] C [from]] S)`	Entfernt Leerzeichen oder ein anderes Zeichen C am Anfang oder am Ende oder auf beiden Seiten des Strings S
`length(S)`	Länge des Strings S
`locate(SS, S, OFFSET)`	Bestimmt den Index des Suchstrings SS in S ab der Position OFFSET
`abs(N)`, `sqrt(N)`, `mod(DIVIDEND, DIVISOR)`	Bestimmen den Absolutbetrag, die Wurzel, den Modulo für numerische Werte
`size(C)`	Anzahl der Elemente der Collection C

Weitere Funktionen

Operator	Beschreibung
`bit_lenght(S)`	Liefert die Anzahl von Bits in S
`current_date()`, `current_time()`, `current_timestamp()`	Liefert das Datum und/oder Zeit des DBMS-Rechners
`second(D)`, `minute(D)`, `hour(D)`, `day(D)`, `month(D)`, `year(D)`	Extrahiert Zeitangaben eines Datums D
`cast(O as T)`	Castet ein Objekt O in den Typ T
`index(E)`	Liefert den Index des Elements E aus einer gejointen Kollektion
`minelement(C)`, `maxelement(C)`, `minindex(C)`, `maxindex(C)`, `elements(C)`, `indices(C)`	Liefert ein Element oder einen Index einer indexbasierten Kollektion (Array, List, Map)
Erweiterungen in `org.hibernate.Dialect`	Erweiterbar um zusätzliche Funktionen des DBMS

Join (Verbund) und Unterabfragen

- Join durch implizite Assoziation

```
from User as u where u.homeAddress.city = 'Berlin'
```

- Join in der FROM-Klausel

```
from Item as i inner join i.bids b
where i.description like '%Car%'
and b.amount.value > 100

from Item as i
left outer join i.bids b
with b.amount.value > 100
where i.description like '%Car%'
```

Join (Verbund) und Unterabfragen

- Dynamisches-Laden durch Join (Performanz-Optimierung)

```
from Item as i
left outer join fetch i.bids
where i.description like '%Car%'
```

- Theta-style-Join (für nicht Fremdschlüssel-Beziehungen)

```
from User, Category
from Item as i, Bid as b where i.seller = b.bidder
```

- Unterabfrage

```
from Bid as b1 where b1.amount.value + 1 >= (
  select max(b2.amount.value) from Bid as b2)
```

- Prädikate mit Unterabfragen `some`, `all`, `in`

```
from Item as i where 100 in
  (select b.amount.value from i.bids b)
```

Aggregation und Gruppierung

- Aggregationsfunktionen
  `count`, `min`, `max`, `sum` und `avg`
- Gruppierung
  ```
  select u.lastname, count(u) from User as u group by u.lastname
  ```
- Suchbedingung auf aggregierter Spalte
  ```
  select u.lastname, count(u)
  from User as u
  group by u.lastname
  having u.lastname like 'C%'
  ```
- Konstruktor
  ```
  select new ItemBidSummary(b.item.id, count(b), avg(b.amount))
  from Bid as b
  where b.item.successfulBid is null
  group by b.item.id
  ```