JPA bidirectional relations

René Anderes | 2015 | Rev.51

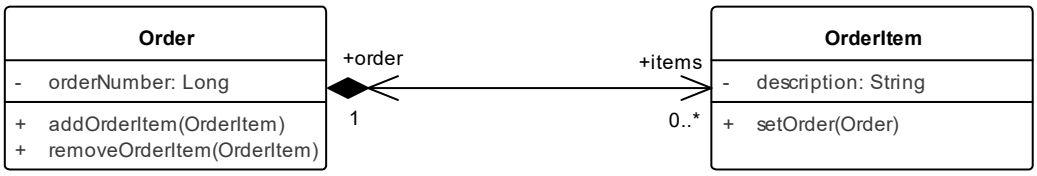
Allgemein

Die Erwartung, dass JPA die Synchronität bei Entität/Objekte in einer bidirektionalen Beziehung sicherstellt, kann JPA nicht erfüllen. Dies ginge sowieso nur solange die Entitäten den Status MANAGED haben. Sobald sie den Status DETACHED haben (z.B. durch Serialisierung) wäre es unmöglich.

Bei einer bidirektionalen Beziehung liegt es daher in der Verantwortlichkeit der Applikation die Synchronität zu jedem Zeitpunkt sicher zu stellen.

Folgendes Pattern kann eingesetzt werden:

Beispiel



Java-Code

```
@Entity(name = "ORDERING")
public class Order {

    @Id
    @GeneratedValue
    private Long id;
    @Column(nullable = false)
    private Long orderNumber;

    @OneToMany(
        mappedBy="order",
        cascade = { CascadeType.ALL },
        orphanRemoval = true)
    private Set<OrderItem> items = new HashSet<>();

    Order() { super(); }

    public Order(Long orderNumber) {
        super();
        this.orderNumber = orderNumber;
    }

    public Long getOrderNumber() {
        return orderNumber;
    }
```

```
    public Collection<OrderItem> getOrderItems() {
        return Collections.unmodifiableSet(items);
    }

/* --------- Pattern für JPA bidirektionale Beziehung -------- */

    public void addOrderItem(OrderItem orderItem) {
        orderItem.setOrder(this);
    }

    public void removeOrderItem(OrderItem orderItem) {
        if (!items.contains(orderItem)) {
            throw new IllegalArgumentException("...");
        }
        orderItem.setOrder(null);
    }

    void internalRemoveOrderItem(OrderItem orderItem) {
        items.remove(orderItem);
    }

    void internalAddOrderItem(OrderItem orderItem) {
        items.add(orderItem);
    }

    @PreRemove
    public void preRemove() {
        final HashSet<OrderItem> orderItems =
                  new HashSet<OrderItem>(getOrderItems());
        for (OrderItem orderItem : orderItems) {
            removeOrderItem(orderItem);
        }
    }

/* /-------- Pattern für JPA bidirektionale Beziehung --------- */

    @Override
    public int hashCode() {
        ...
    }

    @Override
    public boolean equals(Object obj) {
        ...
    }

}

@Entity
public class OrderItem {

    @Id
    @GeneratedValue
    private Long id;
    @Column(nullable = false)
    private String description;

    @ManyToOne
    private Order order;
```

```
    OrderItem() {
        super();
    };

    public OrderItem(String item) {
        this.description = item;
    }

    public String getDescription() {
        return description;
    }

    public Order getOrder() {
        return order;
    }

/* ----------- Pattern für JPA bidirektionale Beziehung ---------- */

    public void setOrder(Order order) {
        if (this.order != null) {
            this.order.internalRemoveOrderItem(this);
        }
        this.order = order;
        if (order != null) {
            order.internalAddOrderItem(this);
        }
    }

    @PreRemove
    public void preRemove() {
        setOrder(null);
    }

/* /---------- Pattern für JPA bidirektionale Beziehung ---------- */

    @Override
    public int hashCode() {
        ...
    }

    @Override
    public boolean equals(Object obj) {
        ...
    }

}
```

Es versteht sich, dass eine entsprechende Implementation von `equals()` und `hashCode()` vorhanden sein muss, welche hier zu Gunsten der Übersichtlichkeit geschuldet ist.

Es werden im obigen Beispiel entsprechende ***add…(), set…() und remove…() Methoden*** implementiert, welche sicherstellen, dass beide Objekte immer synchron sind egal woher (mittels Order oder OrderItem) ein entsprechender Wert gesetzt, hinzugefügt oder entfernt wird.

Diese Lösung stellt eine Möglichkeit dar. Folgende Überlegungen können dazu angestellt werden:

- Die Methoden "internal…" können nur dann die Sichtbarkeit Package haben, wenn sie sich im selben Package befinden.
- Werden keine OrderItem's einer Order entfernt, so kann ein "removeOrderItem" entfallen.
- Die Idee funktioniert natürlich auch anders herum; Die "internal" Methoden sind im OrderItem untergebracht etc.
- Das Pattern funktioniert nur wenn für JPA "field access" verwendet wird (Annotation auf den Klassenmembers nicht auf den Methoden)

@PreRemove

Die JPA-Annotation ***@PreRemove*** wiederum stellt sicher, dass die Beziehungen synchron gehalten werden, wenn eine Entität gelöscht wird.
Z.B. mittels:

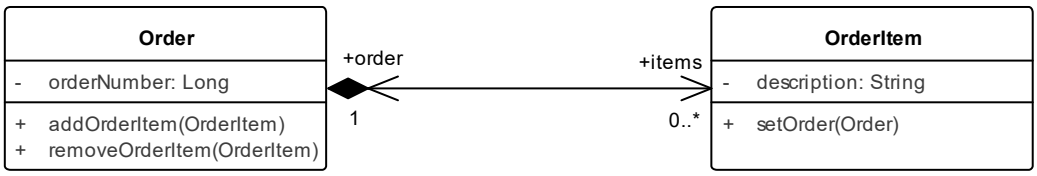
- EntityManager.remove(orderItem);
- EntityManager.remove(order);

In diesem Beispiel werden alle OrderItem zu einer Order sowieso gelöscht, da "orphanRemoval = true" gesetzt ist. Das Pattern funktioniert so jedoch für alle möglichen Beziehungen.

bidirectional by example

Bei den folgenden Beispielen gilt es sich im Hinterkopf zu behalten, wie die Zustände von Entitäten funktionieren.

OneToMany - ManyToOne (Composition)



Java-Code

```
@Entity
public class Order {

    @OneToMany(
        mappedBy="order",
        cascade = { CascadeType.ALL },
        orphanRemoval = true)
    private Set<OrderItem> items = new HashSet<>();

    ...
}

@Entity
public class OrderItem {

    @ManyToOne
    private Order order;

    ...
}
```

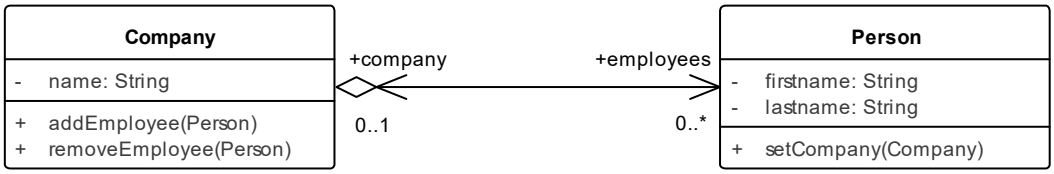
Anwendungsfall

Der Owner der Beziehung ist die Order (Bestellung). Eine Instanz der Klasse OrderItem kann nicht ohne Beziehung zu einer Order existieren.

Aus fachlicher Sicht macht es auch kaum Sinn mittels OrderItem eine Bestellung zu persistieren. Schauen wir uns trotzdem an, wie sich persist und merge in unterschiedlichen Szenarien verhalten:

Szenario	merge(order)	persist(order)	merge(orderItem)	persist(orderItem)
Order neue Instanz OrderItem neue Instanz	Order und OrderItem werde persistiert	Order und OrderItem werde persistiert	OrderItem und Order werden persistiert	IllegalStateException
Order existiert OrderItem neue Instanz	Order wird aktualisiert, der neue OrderItem wird hinzugefügt	Order wird aktualisiert, der neue OrderItem wird hinzugefügt	Order wird aktualisiert und *zwei* neue OrderItem werden hinzugefügt	Order wird aktualisiert, der neue OrderItem wird hinzugefügt

OneToMany - ManyToOne (Aggregation)



Java-Code

```
@Entity
public class Person {

    @ManyToOne
    private Company company;

    ...
}

@Entity
public class Company {

    @OneToMany(mappedBy="company")
    private Collection<Person> employees = new HashSet<>();

    ...
}
```

Anwendungsfall

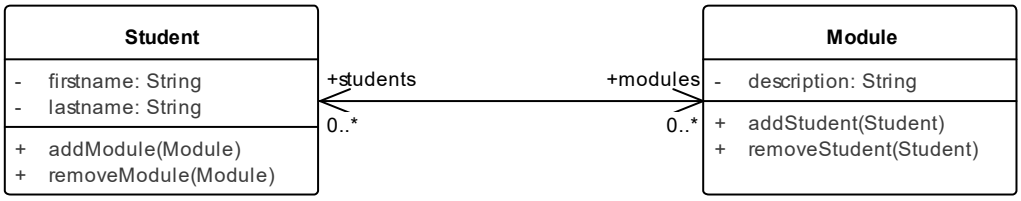
Sowohl Instanzen der Klasse Company wie auch Person sollen unabhängig voneinander gespeichert und gelöscht werden können. Beziehungen sollen jederzeit hergestellt und auch gelöscht werden können.

Auch hier verhalten sich merge(…) und persist(…) unterschiedlich. Sehen wir uns ein paar Szenarien dazu an:

Es wird eine neue Instanz Person und eine neue Instanz Company erstellt.

A) Mittels merge(person) oder merge(company) werden bei einem commit jeweils beide Entitäten persistiert: Nicht das erwartete Verhalten (kein cascade angegeben)
B) Ein persist(person) oder persist(company) löst bei einem commit (wie erwartet) eine Exception aus, da jeweils die zur Beziehung gehörende Instanz nicht in der Datenbank vorliegt.
C) Mittels persist(person) und persist(company) kann sichergestellt werden, dass bei einem commit beide Entitäten persistiert werden.

ManyToMany



Java-Code

```
@Entity
public class Module {

    @ManyToMany(mappedBy = "modules")
    private Set<Student> students = new HashSet<>();

    ...
}

@Entity
public class Student {

    @ManyToMany
    @JoinTable(name="STUDENT_MODULE",
        joinColumns = @JoinColumn(name="S_ID"),
        inverseJoinColumns = @JoinColumn(name="M_ID"))
    private Set<Module> modules = new HashSet<>();

    ...
}
```

Anwendungsfall

Ein Student kann sich bei beliebig vielen Module anmelden, eine Beziehung haben. Ein Module kann beliebig viele Beziehungen zu Instanzen der Klasse Student haben.

Wie aus den bisherigen Beispielen hervorgeht, verhaltet sich auch in diesem Fall merge(…) in Zusammenhang mit neuen Instanzen nicht den Erwartungen entsprechend.

Ausgangslage analog vorherigen Beispiel: Es wird eine neue Instanz der Klasse Module und eine neue Instanz der Klasse Student gemacht.

A) Mittels merge(student) oder merge(module) werden bei einem commit jeweils beide Entitäten persistiert: Nicht das erwartete Verhalten (kein cascade angegeben)
B) Ein persist(student) oder persist(module) löst bei einem commit (wie erwartet) eine Exception aus, da jeweils die zur Beziehung gehörende Instanz nicht in der Datenbank vorliegt.
C) Mittels persist(student) und persist(module) in derselben Transaktion kann sichergestellt werden, dass bei einem commit beide Entitäten persistiert werden.

Save-Pattern

Als Resultat aus den obigen Betrachtungen kann folgende Empfehlung abgegeben werden:

- Bei neuen Instanzen einer Entity (und nur dann) soll ausschliessliche persist(…) verwendet werden.
- Werden Entitäten aktualisiert, die den Zustand MANAGED haben, ist kein merge(…) notwendig. Der Entity-Manager wird beim nächste commit/flush die Änderungen automatisch in die Datenbank schreiben.
- Zum Aktualisieren von Entitäten die den Zustand DETACHED haben (z.B. durch Serialisierung etc.) soll die Methode merge(…) verwendet werden.

Anhang

Die obigen Beispiele inkl. Test's sind hier zu finden:

- https://github.com/rene-anderes/jpa-bidirectional-relations