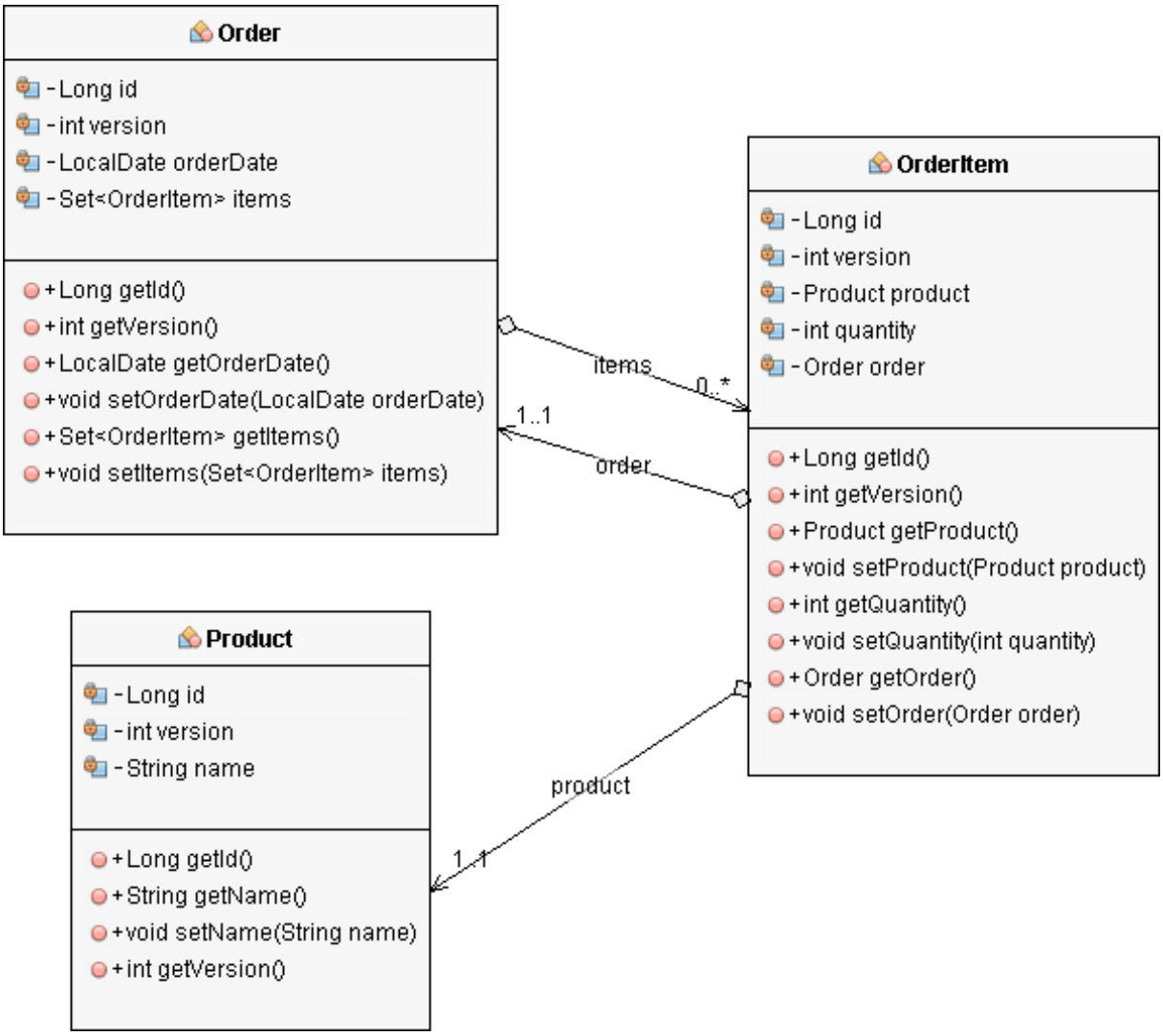
Implementing the builder pattern for your entities can massively improve the readability of your business code. In contrast to the fluent interface pattern, there is nothing in the JPA specification or the Hibernate documentation that prevents you from implementing a builder for an entity class.

The Domain Model

Let's create builders to create an Order with multiple OrderItems comfortably. Each OrderItem references a Product from the product catalog.



Creating Builders for a Graph of Entities

A meaningful builder API needs to help you to create a graph of entity objects. For the example in this article, that means that you not only need to provide a builder for the *Order* and the *OrderItem* entity. You also need to support the creation of a *Set* of *OrderItem* objects for a given *Order*.

If you do that, you will be able to create a new *Order* with 2 *OrderItem*s like this:

```
Order o = new Order.OrderBuilder()
        .withOrderDate(LocalDate.now())
        .withItems()
            .addItem().withProduct(p1).withQuantity(1).addToList()
            .addItem().withProduct(p2).withQuantity(2).addToList()
            .buildItemList()
        .buildOrder();
em.persist(o);
```

OK, let's take a look at the code of the builder classes that I use in the code snippet.

The OrderBuilder

The *Order* entity is the root of the small graph of entities. When you create an *Order* object, you can ignore the primary key and the version attribute. These are generated attributes that you don't need to provide when you instantiate a new object.

When you ignore these attributes, there are only 2 attributes left:

- The *orderDate* of type *LocalDate* and
- A *Set* of *OrderItem* entities.

Due to that, you only need to provide a *withOrderDate(LocalDate orderDate)*, a *withItems(Set<OrderItem>items)* and a *buildOrder()* method to be able to build an *Order* entity object.

Within these methods, you can perform additional validations, e.g., check that the *orderDate* isn't in the past or that the *Set* of *OrderItem* isn't empty.

They also enable you to hide technical details. I use that in the *buildOrder* method, to hide the double linking between the *Order* and *OrderItem* objects that's required to manage the bidirectional one-to-many association.

```
public static final class OrderBuilder {
  private LocalDate orderDate;

  private OrderItemListBuilder itemListBuilder;

  public OrderBuilder withOrderDate(LocalDate orderDate) {
    if (orderDate.isBefore(LocalDate.now())) {
      throw new IllegalArgumentException("OrderDate can't be in the past.");
    }

    this.orderDate = orderDate;
    return this;
  }

  public OrderBuilder withItems(Set<OrderItem> items) {
    if (items.isEmpty()) {
      throw new IllegalArgumentException("Order has to have at least 1 item.");
    }
    this.itemListBuilder = new OrderItemListBuilder(this);
    this.itemListBuilder.items = items;
    return this;
  }

  public OrderItemListBuilder withItems() {
    this.itemListBuilder = new OrderItemListBuilder(this);
    return this.itemListBuilder;
  }

  public Order buildOrder() {
    Order o = new Order();
    o.setOrderDate(this.orderDate);
```

```
// Create Set<OrderItem> and link with order
    Set<OrderItem> items = this.itemListBuilder.items;
    for (OrderItem item : items) {
       item.setOrder(o);
    }
    o.setItems(items);

    return o;
  }
}
```

Technically, we don't need any additional methods. But as you can see in the code snippet, I also implement the *withItems*() method that returns an *OrderItemListBuilder* and doesn't take any *OrderItem* entities as parameters.

The *withItems*() method and the *OrderItemsBuilder* class make the API much easier to use because you can use them to create new *OrderItem* objects and add them to the *Order*.

The *OrderItemListBuilder*

The *OrderItemListBuider* class bridges the gap between the *Order* and the *OrderItemBuilder* by managing the *Set* of *OrderItems*.

```
public static class OrderItemListBuilder {

 private Set<OrderItem> items = new HashSet<OrderItem>();

 private OrderBuilder orderBuilder;

 public OrderItemListBuilder (OrderBuilder orderBuilder) {
    this.orderBuilder = orderBuilder;
 }

 public OrderItemListBuilder addItem(OrderItem item) {
    this.items.add(item);
    return this;
 }

 public OrderItemBuilder addItem() {
    return new OrderItem.OrderItemBuilder(this);
 }
```

```
public OrderBuilder buildItemList() {
    return this.orderBuilder;
 }
}
```

In contrast to the 2 other builders in this example, this class doesn't implement any logic. It only provides the required glue code so that you can chain the method calls required to create multiple *OrderItem*s and to add them to an *Order*.

There are 2 important things that I want to point out:

1. The reference to the *OrderBuilder* that gets provided as a constructor parameter.
2. The *buildItemList()* method that you need to call to get back to the *OrderBuilder* when you're done adding *OrderItem*s to the *Set*.

The *OrderItemBuilder*

The *OrderItemBuilder* implements the required methods to build an *OrderItem*.

```
public static final class OrderItemBuilder {

  private Product product;

  private int quantity;

  private OrderItemListBuilder itemListBuilder;

  public OrderItemBuilder() {
     super();
  }

  public OrderItemBuilder(OrderItemListBuilder itemListBuilder) {
     super();
     this.itemListBuilder = itemListBuilder;
  }

  public OrderItemBuilder withProduct(Product product) {
     this.product = product;
```

```
return this;
  }

  public OrderItemBuilder withQuantity(int quantity) {
     this.quantity = quantity;
     return this;
  }

  public OrderItem build() {
     OrderItem item = new OrderItem();
     item.setProduct(this.product);
     item.setQuantity(this.quantity);
     return item;
  }

  public OrderItemListBuilder addToList() {
     OrderItem item = build();
     this.itemListBuilder.addItem(item);
     return this.itemListBuilder;
  }
}
```

The only method that's not strictly required is the *addToList()* method. It creates a new *OrderItem* object and returns the *OrderItemListBuilder* so that you can keep adding *OrderItems* to the *Set*.

Using the Cascaded Builders to Create a Graph of Entities

After you've seen the code of the 3 builder classes, let's take another look at the business code that creates an *Order* with 2 *OrderItems*.

```
<pre class="wp-block-preformatted brush: java; gutter: true">Order o = new
Order.OrderBuilder()
        .withOrderDate(LocalDate.now())
        .withItems()
            .addItem().withProduct(p1).withQuantity(1).addToList()
            .addItem().withProduct(p2).withQuantity(2).addToList()
            .buildItemList()
        .buildOrder();
em.persist(o);
```

In the first line, I instantiate a new *OrderBuilder* which I then use to provide the date of the order.

Then I want to add *OrderItems* to the *Order*. To do that, I first call the *withItems*() method. It returns an *OrderItemListBuilder* on which I call the *addItem*() method to get an OrderItemBuilder that's linked to the *OrderItemListBuilder*. After I've set the reference to the *Product* entity and the quantity that the customer wants to order, I call the *addToList*() method. That method builds an *OrderItem* object with the provided information and adds it to the *Set<OrderItem>* managed by the *OrderItemListBuilder*. The method also returns the *OrderItemListBuilder* object. That allows me to either add another *OrderItem* to the *Set* or to call the *buildItemList*() to complete the creation of the *Set*.

In the final step, I call the *buildOrder*() method on the *OrderBuilder* to create the *Order* method. Within that method, a new *Order* object gets created, the *Set* of *OrderItems* gets filled, and each OrderItems gets associated with the *Order* entity.

After I've created the *Order* object, I provide it as a parameter to the *persist* method of the *EntityManager*. I've set the *CascadeType* of the *orderItems* association to *PERSIST* so that Hibernate automatically persists the associated *OrderItem* entities when I persist the *Order*entity.

Conclusion

You can easily apply the builder pattern to your entities. In contrast to the fluent interface pattern, you don't need to work around any technical requirements defined by the JPA specification or the Hibernate documentation.

To implement a good builder API, you need to cascade your builders so that you can comfortably create one or more associated entities.