Inheritance

lesson4

Inheritance is a practical way to reuse code in the object oriented world. Using inheritance with JPA and OpenXava is as easy as using it in plain Java. We are going to use inheritance to remove repeated and boring code, like the UUID definition. Also, we'll add a new entity, `Order`, and we'll use inheritance to do it with minimal code. Moreover, you'll learn how practical it is to use inheritance even for the code used for testing.

4.1 Inheriting from a mapped superclass

The `Author`, `Category`, `Detail` and `Invoice` classes have some common code. This code is the `oid` field definition (listing 4.1) and it is just the same for all those classes. You know that copy and paste is a mortal sin, so we have to look for a solution to remove the code repetition, to avoid our way to hell.

Listing 4.1 Oid property: a common code for Author, Category, Detail and Invoice

```
@Id @GeneratedValue(generator="system-uuid") @Hidden
@GenericGenerator(name="system-uuid", strategy = "uuid")
@Column(length=32)
private String oid;

public String getOid() {
    return oid;
}

public void setOid(String oid) {
    this.oid = oid;
}
```

An elegant solution for this situation is inheritance. JPA allows you to exploit the concept of inheritance. One of them is to inherit from a mapped superclass. A mapped superclass is a Java class with JPA mapping annotations, but it is not an entity itself. Its only goal is to be used as superclass for an entity. Let's use it, and you'll see its utility quickly.

First, we move this common piece of code to a class marked as `@MappedSuperclass`. We name it `Identifiable`[6]. See it in listing 4.2.

Listing 4.2 Identifiable: a mapped superclass with UUID generation

```
package org.openxava.invoicing.model;

import javax.persistence.*;
import org.hibernate.annotations.*;
import org.openxava.annotations.*;

@MappedSuperclass   // Marked as mapped superclass instead of entity
public class Identifiable {

    @Id @GeneratedValue(generator="system-uuid") @Hidden
```

6 Since OpenXava 4.0 the Identifiable superclass is included in OpenXava

```
    @GenericGenerator(name="system-uuid", strategy = "uuid")
    @Column(length=32)
    private String oid;  // Property definition includes OX and JPA annotations

    public String getOid() {
        return oid;
    }

    public void setOid(String oid) {
        this.oid = oid;
    }

}
```

Now you can define `Author, Category, Detail and Invoice` entities in a more succinct way. To see an example you have the new code for `Category` in listing 4.3.

Listing 4.3 Category entity refactored to extend from Identifiable

```
@Entity
public class Category extends Identifiable {  // Extends from Identifiable
                                            // so it does not need to have an id property

    @Column(length=50)
    private String description;

    public void setDescription(String description) {
        this.description = description;
    }

    public String getDescription() {
        return description;
    }

}
```

The refactoring is plain vanilla. `Category` now extends from `Identifiable` and we have removed the `oid` field and the respective `getOid()` and `setOid()` methods. This way, not only is your code shorter, but it's more elegant, because you declare that your class is identifiable (the what, not the how), and you removed from your business logic class code that is somewhat technical.

Now, you can apply this same refactoring to the `Author, Detail` and `Invoice` entities, moreover, from now on you'll extend most of your entities from the `Identifiable` mapped superclass.

You have learned that a mapped superclass is a regular class with JPA mapping annotations that you can use as a base class for your entities. Moreover, you have learned how to use mapped superclasses to simplify your code.

4.2 Entity inheritance

An entity may inherit from another entity. This entity inheritance is a useful tool to simplify your model. We are going to use it to add a new `Order` entity to your Invoicing application.

4.2.1 New Order entity

We want to add a new concept to the Invoicing application, i. e., order. While an invoice is something you want to charge your customer, an order is something the customer has ordered from us. These two concepts are strongly related, because you will charge the things your customer has ordered from you, and you actually have served him.

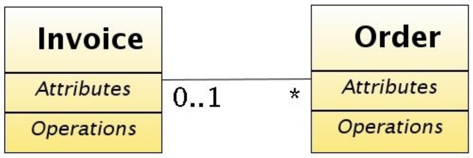


Figure 4.1 UML for Order and Invoice

It would be nice to manage orders in your application, and to associate those orders with its corresponding invoices. Just as shown in the UML diagram in figure 4.1 and the Java code in listing 4.4.

Listing 4.4 Relationship between Invoice and Order

```
@Entity
public class Invoice {

    @OneToMany(mappedBy="invoice")
    private Collection<Order> orders;
    ...
}

@Entity
public class Order {

    @ManyToOne   // Without lazy fetching (1)
    private Invoice invoice;

}
```

That is, an invoice has several orders, and an order can reference an invoice. Note that we do not use lazy fetching for the `invoice` reference (1), this is because of a Hibernate bug when the reference owns the bidirectional relationship (that is, it's declared in the `mappedBy` attribute of the `@OneToMany`).

What shape does `Order` have? Well, it has a customer, several detail lines with product and quantity, a year and a number, something like figure 4.2

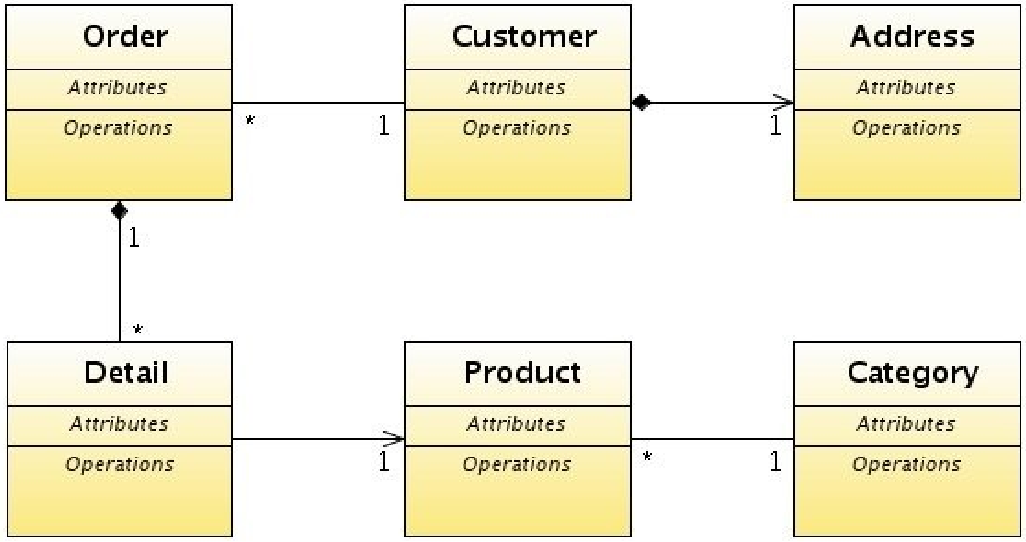


Figure 4.2 UML diagram for Order

Incidentally, this UML diagram is identical to the `Invoice` diagram (look at it in the figure 2.1 of section 2.1). That is, to create your `Order` entity you can just copy and paste the `Invoice` class, and the work is done. But, wait a moment! “Copy and paste” is a mortal sin. So, we have to find a way to reuse `Invoice` code for `Order`.

4.2.2 *CommercialDocument as an abstract entity*

A practical way to reuse the code for `Invoice` and `Order` is by using inheritance, moreover it's an excellent opportunity to learn how easy it is to use inheritance with JPA and OpenXava.

In most object oriented cultures you have to observe the *is a* rule[7]. It means that we cannot do `Invoice` extends `Order`, because an `Invoice` is not an `Order`, and because the same rule cannot be done like `Order` extends `Invoice`. The solution for this case is creating a base class for both `Order` and `Invoice`. We are going to call this class `CommercialDocument`.

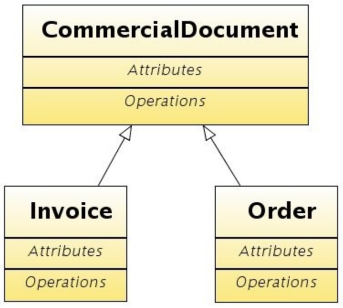


Figure 4.3 UML for CommercialDocument

In figure 4.3 you see the UML diagram for `CommercialDocument`, and in listing 4.5 you see the

7 Eiffel culture does not observe the *is a* rule. So *is a* is not an absolute rule in OO universe

same idea expressed with Java.

Listing 4.5 Using inheritance to define Order and Invoice

```
public class CommercialDocument { … }
public class Order extends CommercialDocument { … }
public class Invoice extends CommercialDocument { … }
```

Let's start to refactor your current code. First, rename (using *Refactor > Rename*) `Invoice` as `CommercialDocument`. Now, edit the `CommercialDocument` code to declare it as an abstract class, as shown in listing 4.6.

Listing 4.6 CommercialDocument is an abstract class

```
abstract public class CommercialDocument  // We add the abstract modifier
```

We want to create instances of `Invoice` and `Order`, but we do not want to create instances of `CommercialDocument` directly, so we declare it as abstract.

4.2.3 *Invoice refactored to use inheritance*

Now, you have to create the `Invoice` entity extending it from `CommercialDocument`. See the new `Invoice` code in listing 4.7.

Listing 4.7 The Invoice entity now extends from CommercialDocument

```
@Entity
public class Invoice extends CommercialDocument {

}
```

`Invoice` now has very succinct code, indeed the body of the class is, by now, empty.

This new code requires a slightly different database schema, now invoices and orders will be stored in the same table (the CommercialDocument table) using a discriminator column. Therefore you need to remove the old tables executing the SQL statements in listing 4.8.

Listing 4.8 SQL sentence to remove the old tables for Invoice

```
drop table detail;
drop table invoice;
```

You can execute these SQL statements from the Database Development perspective of Eclipse, just as explained in lesson 1.

After removing the old tables, execute the `updateSchema` ant target to regenerate all the needed tables. And now you can execute the `Invoice` module and see it working in your browser. Also, execute the `InvoiceTest`. It must be green.

4.2.4 Creating Order using inheritance

Thanks to CommercialDocument creating the Order entity is dead easy. See the code in listing 4.9.

Listing 4.9 Order entity that extends from CommercialDocument

```
@Entity
public class Order extends CommercialDocument {

}
```

After writing the Order class in listing 4.9, although it is empty by now, you can use the Order module from your browser. Yes, creating a new entity with a structure like Invoice, that is any commercial document entity, is very easy and rapid. You see how good use of inheritance is an elegant way to have simpler code.

Order module works perfectly, but it has a little problem. The new order number is calculated from the last invoice number, not from the last order number. This is because the calculator for the next number is read from the Invoice entity. An obvious solution is to move the number property definition from CommercialDocument to Invoice and Order. Although, we are not going to do it in this way, because in lesson 5 we'll refine the way to calculate the next number, for now we simply do a little adjustment in the current code so that it does not fail. Edit the NextNumberForYearCalculator class and in the query change "Invoice" for "CommercialDocument", leaving the calculate() method as in listing 4.10.

Listing 4.10 NextNumberYearCalculator.calculate() using CommercialDocument

```
public Object calculate() throws Exception {
    Query query = XPersistence.getManager()
       .createQuery(
       "select max(i.number) from " +
       "CommercialDocument i " +  // CommercialDocument instead of Invoice
       "where i.year = :year");
    query.setParameter("year", year);
    Integer lastNumber = (Integer) query.getSingleResult();
    return lastNumber == null?1:lastNumber + 1;
}
```

Now we search for the maximum number of any commercial document of the year in order to calculate the new number, therefore the numbering is shared across all the commercial documents. We'll improve it in lesson 5 for having separate numbering for invoices and orders and to have better support for a multiuser environment.

4.2.5 Naming convention and inheritance

Note that you do not need to change the name of any property of `Invoice` to do the refactoring. This is because we follow the next pragmatic principle: *Do not use the class name in member names*, e.g., inside an `Account` class do not use the “Account” word in any method or property. See listing 4.11.

Listing 4.11 Do not use the class name in member names

```
public class Account {   // We'll not use Account in member names

    private int accountNumber;   // Bad, because it uses “account”
    private int number;   // Good, it does not use “account”

    public void cancelAccount() { }   // Bad, because it uses “Account”
    public void cancel() { }   // Good, it does not use “account”
    ...
}
```

Using this nomenclature you can refactor `Account` in an inheritance hierarchy without renaming its members, and you can write polymorphic code with `Account`.

4.2.6 Associating Order with Invoice

By now, `Order` and `Invoice` are exactly the same. We are going to do the first extensions on them, that is to associate `Order` with `Invoice`. Just as shown in figure 4.4. You only need to add a reference from `Order` to `Invoice`, just as shown in listing 4.12.

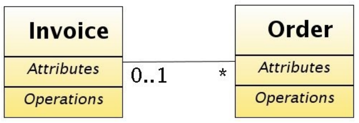


Figure 4.4 Association between Invoice and Order

Listing 4.12 Full code for Order with a reference to Invoice

```
@Entity
public class Order extends CommercialDocument {

    @ManyToOne
    private Invoice invoice;   // Reference to invoice added

    public Invoice getInvoice() {
       return invoice;
    }

    public void setInvoice(Invoice invoice) {
       this.invoice = invoice;
    }

}
```

Conversely in `Invoice` we add a collection of `Order` entities. See it in listing 4.13.

Listing 4.13 Full code for Invoice with a collection of orders

```
@Entity
public class Invoice extends CommercialDocument {

    @OneToMany(mappedBy="invoice")
    private Collection<Order> orders;   // Collection of Order entities added

    public Collection<Order> getOrders() {
       return orders;
    }

    public void setOrders(Collection<Order> orders) {
       this.orders = orders;
    }

}
```

After writing this simple code you are ready to test the newly created relationship. First, update the database schema, and then open your browser and explore the `Order` and the `Invoice` modules. Note that at the end of the `Order` user interface you have a reference to `Invoice`. The user can use this reference to associate an invoice with the current order. On the other hand, if you explore the `Invoice` module, you will see a collection of orders at the end. The user can use it to add orders to the current invoice.

Try to add orders to invoice, and to associate an invoice to an order. It works, although the user interface is a little ugly.

4.3 View inheritance

You can use inheritance not only for reusing the Java code and mapping, but also for reusing the user interface definition, the `@View` definitions. This section shows how view inheritance works.

4.3.1 The extendsView attribute

Both `Order` and `Invoice` use a user interface generated by default with all its members one in each line. Note that the annotation `@View` that we have declared in `CommercialDocument` is not inherited by default. That is, if you do not define a view for an entity a default one is generated, and the `@View` of the parent entity is not used. Just as shown in listing 4.14.

Listing 4.14 Views are not inherited by default

```
@View(members = "a, b, c;")  // This view is used to display Parent, but not for Child
public class Parent { … }
public class Child extends Parent { … }     // Child is displayed using an automatically
                                            // generated view, not the view from Parent
```

Usually the view of the parent entity "as is" is not very useful because it does not contain the new properties that the current entity has. So this behavior is good for default behavior.

Although, in a non-trivial entity you may need to refine the user interface, and it might be useful to inherit (instead of copy and paste) the view from the parent. You can do it using the `extendsView` attribute in `@View`. See this in listing 4.15.

Listing 4.15 Using view inheritance by means of extendsView

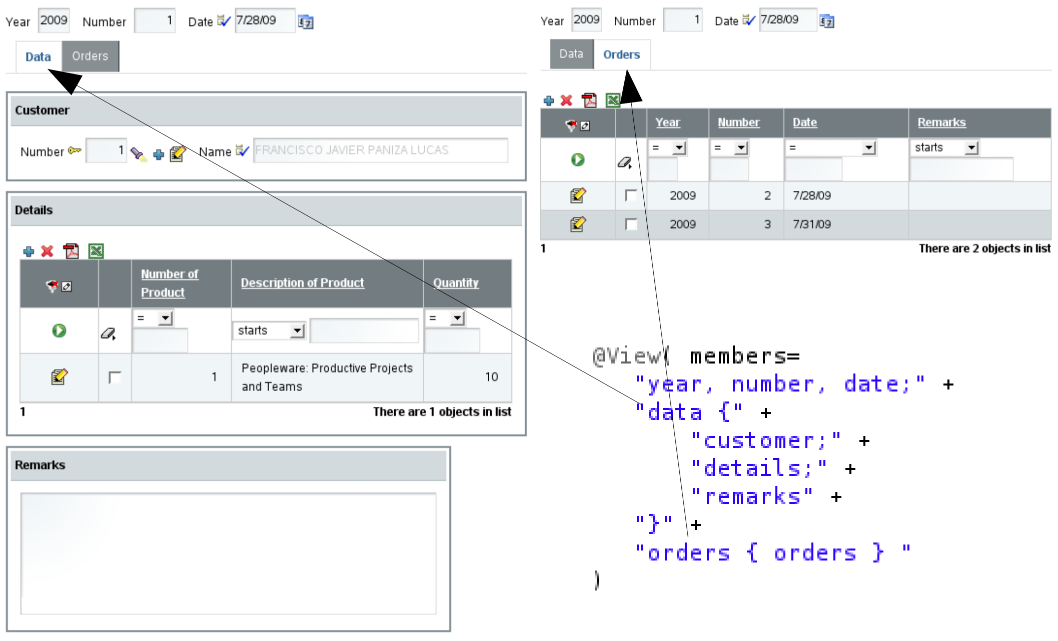
```
@View(members = "a, b, c;")   // This view with no name is the DEFAULT view
public class Parent { … }

@Views({
    @View(name="A" members = "d",    // Adds d to the inherited view
       extendsView = "super.DEFAULT"),   // Extends the default view from Parent
    @View(name="B" members = "a, b, c; d")   // View B is equals to view A
})
public class Child extends Parent { … }     // Child is displayed using an automatically
                                            // generated view, not the view from Parent
```

Using `extendsView` the members that appear will be those of the extended view plus those declared in `members` of the current one.

We are going to use this feature for defining the views for `CommercialDocument`, `Order` and `Invoice`.

4.3.2 View for Invoice using inheritance

Given that the `@View` of `CommercialDocument` has not inherited the current user interface for `Invoice` it's pretty ugly to see, all the members, each in a line. We are going to define a better user interface. A view similar to the one we defined in lesson 2, but adding a tab for orders. See it in figure 4.5.



Year 2009 Number 1 Date 7/28/09

Number 1 Name FRANCISCO JAVIER PANIZA LUCAS

Number of Product	Description of Product	Quantity
1	Peopleware: Productive Projects and Teams	10

There are 1 objects in list

Year	Number	Date	Remarks
2009	2	7/28/09	
2009	3	7/31/09	

There are 2 objects in list

```
@View( members=
    "year, number, date;" +
    "data {" +
        "customer;" +
        "details;" +
        "remarks" +
    "}" +
    "orders { orders } "
)
```

Figure 4.5 User interface for Invoice with orders

Note that we have put all the members from the `CommercialDocument` part of the `Invoice` in the header and the first tab (data), and the collection of orders in the other tab. Listing 4.16 shows the way to define this view without inheritance.

Listing 4.16 View for Invoice without view inheritance

```
@View( members=
    "year, number, date;" +
    "data {" +
       "customer;" +
       "details;" +
       "remarks" +
    "}" +
    "orders { orders } "
)
public class Invoice extends CommercialDocument {
```

You can note how all except the orders part is common for every `CommercialDocument`. Therefore, we are going to move this part to `CommercialDocument` and then redefine this view using view inheritance.

Remove the old `@View` in `CommercialDocument`, and write the one in listing 4.17.

Listing 4.17 View for CommercialDocumment

```
@View(members=
    "year, number, date," +   // The members for the header part in one line
    "data {" +   // A tab 'data' for the main data of the document
```

```
        "customer;" +
        "details;" +
        "remarks" +
    "}"
)
abstract public class CommercialDocument extends Identifiable {
```

This view indicates how to layout the common data for all commercial documents. Now we can redefine the view for `Invoice` from this one. Look at it in listing 4.18.

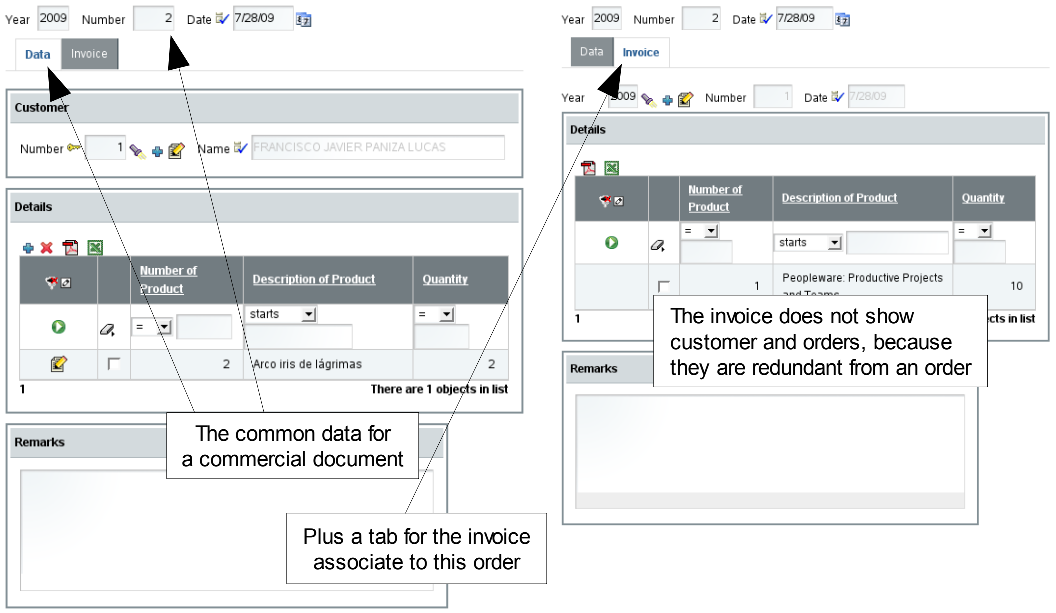
Listing 4.18 View definition for Invoice using view inheritance

```
@View(extendsView="super.DEFAULT",   // Extends from the CommercialDocumment view
    members="orders { orders }"   // We add the orders inside a tab
)
public class Invoice extends CommercialDocument {
```

In this way declaring the view for `Invoice` is shorter. What's more, the common layout for `Order`, `Invoice` and all other possible `CommercialDocument` objects are all in one place. So if you add a new property to `CommercialDocument` you only need to touch the view for `CommercialDocument`.

4.3.3 View for Order using inheritance

Now that you have a suitable view in `CommercialDocument`, declararing the view for `Order` is plain vanilla. We want something like figure 4.6.



Year 2009 Number 2 Date 7/28/09

Data | Invoice

Customer

Number 1 Name FRANCISCO JAVIER PANIZA LUCAS

Details

		Number of Product	Description of Product	Quantity
		=	starts	=
		2	Arco iris de lágrimas	2

1 There are 1 objects in list

Remarks

Year 2009 Number 2 Date 7/28/09

Data | Invoice

Year 2009 Number 1 Date 7/28/09

Details

		Number of Product	Description of Product	Quantity
		=	starts	=
		1	Peopleware: Productive Projects and Teams	10

1 ...cts in list

Remarks

The common data for a commercial document

Plus a tab for the invoice associate to this order

The invoice does not show customer and orders, because they are redundant from an order

Figure 4.6 User interface for Order with a reference to invoice

To get this result, you can define the view for `Order` by extending the default view for `CommercialDocument`, adding the referenced invoice in a new tab, just as shown in listing 4.19.

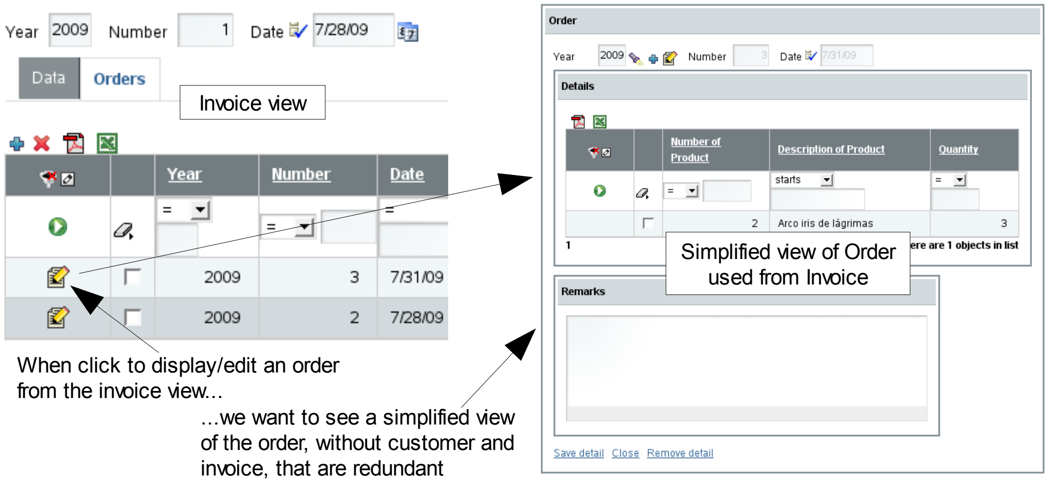
Listing 4.19 View definition for Order using view inheritance

```
@View(extendsView="super.DEFAULT",    // Extends from the CommercialDocumment view
    members="invoice { invoice } "    // We add the invoice inside a tab
)
public class Order extends CommercialDocument {
```

With this we get all the data from `CommercialDocument` plus a tab with the invoice.

4.3.4 Using @ReferenceView and @CollectionView to refine views

When an order is to be edited from the `Invoice` user interface, we want the view used to do so to be simple, with no customer or invoice information, because this data is redundant in this case. See figure 4.7.



When click to display/edit an order from the invoice view...

...we want to see a simplified view of the order, without customer and invoice, that are redundant

Figure 4.7 The view for editing Order from Invoice must be simpler

To get this result define a simpler view in `Order` (listing 4.20), and reference it from the `orders` collection in `Invoice` (listing 4.21).

Listing 4.20 New view NoCustomerNoInvoice in Order to be used from Invoice

```
@Views({    // In order to declare more than one view
    @View( extendsView="super.DEFAULT",    // The default view
        members="invoice { invoice } "
    ),
    @View(   name="NoCustomerNoInvoice",    // A view named NoCustomerNoInvoice
        members=                            // that does not include customer and invoice.
        "year, number, date;" +     // Ideal to be used from Invoice
        "details;" +
        "remarks"
    )
})
public class Order extends CommercialDocument {
```

This new view defined in `Order` named `NoCustomerNoInvoice` can be referenced from `Invoice` to display/edit the individual elements of the `orders` collection using `@CollectionView`. See it in listing 4.21.

Listing 4.21 Using the NoCustomerNoInvoice view in Order from Invoice

```
@OneToMany(mappedBy="invoice")
@CollectionView("NoCustomerNoInvoice")  // This view is used to display orders
private Collection<Order> orders;
```

And with this code only, the `orders` collection will use a more appropriate view from `Invoice` to edit individual elements (figure 4.7).

Moreover, we do not want to display the customer and order information from the `Order` user interface because it is redundant data in this case. To do so, we are going to define a simpler view in `Invoice` (listing 4.22), and then reference it from the `invoice` reference in `Order` (listing 4.23).

Listing 4.22 New view NoCustomerNoOrders in Invoice to be used from Order

```
@Views({   // In order to declare more than one view
    @View(  extendsView="super.DEFAULT",   // The default view
        members="orders { orders } "
    ),
    @View(  name="NoCustomerNoOrders",    // A view named NoCustomerNoOrders
        members=                          // that does not include customer and orders
        "year, number, date;" +           // Ideal to be used from Order
        "details;" +
        "remarks"
    )
})
public class Invoice extends CommercialDocument {
```

This new view defined in `Invoice` named `NoCustomerNoOrders` can be referenced from `Order` to display the reference to `Invoice` using `@ReferenceView`. See it in listing 4.23.

Listing 4.23 Using the NoCustomerNoOrders view of Invoice from Order

```
public class Order extends CommercialDocument {

    @ManyToOne
    @ReferenceView("NoCustomerNoOrders")  // This view is used to display invoice
    private Invoice invoice;

    ...

}
```

Now the `invoice` reference will be displayed appropriately from `Order` and you will get the desired user interface.

4.4 Inheritance in JUnit tests

`Invoice` has been refactored to use inheritance, also we have used inheritance to add a new `Order` entity. Moreover, this `Order` entity has a relationship with `Invoice`. You have a lot of new functionality, therefore you must test all the new features.

Because `Invoice` and `Order` have a lot of common stuff (the `CommercialDocument` part) we can refactor the tests in order to use inheritance, thus you avoid the harmful “copy and paste” in your test code as well.

4.4.1 Creating an abstract module test

If you examine the test for creating an invoice, from the `testCreate()` method of `InvoiceTest` (section 3.6). You can note that testing the creation of an invoice is exactly the same to testing the creation of an order. Because, both have year, number, date, customer, details and remarks. So, here inheritance is a good tool for code reuse.

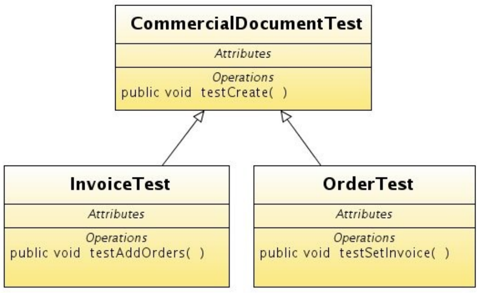


Figure 4.8 CommercialDocumentTest hierarchy

We are going to rename `InvoiceTest` as `CommercialDocumentTest`, then we'll create `InvoiceTest` and `OrderTest` from it. You can see the UML diagram of this idea in figure 4.8.

First rename your current `InvoiceTest` class to `CommercialDocumentTest`, then make the changes indicated in listing 4.24.

Listing 4.24 CommercialDocumentTest created renaming the old InvoiceTest

```
abstract public class CommercialDocumentTest  // Add abstract to class definition
    extends ModuleTestBase {

    private String number;

    public CommercialDocumentTest(
       String testName,
       String moduleName)  // moduleName added as constructor argument
    {
       super(testName, "Invoicing", moduleName);   // Send moduleName
    }

    public void testCreate() throws Exception { … }   // As original

    private void addDetails() throws Exception {
        // Adding a detail line
        assertCollectionRowCount("details", 0);
        execute("Collection.new",
           "viewObject=xava_view_section0_details");  // Change xava_view_details for
                                                       // xava_view_section0_details

        // The remaining code of the method as original
        ...
    }

    private String getNumber() {
        if (number == null) {
           Query query = getManager().
              createQuery(
        "select max(i.number) from " +
        "CommercialDocument i " +  // Invoice changed for CommercialDocument
        "i where i.year = :year");
           query.setParameter("year",
              Dates.getYear(new Date()));
           Integer lastNumber = (Integer)
              query.getSingleResult();
           if (lastNumber == null) lastNumber = 0;
           number = Integer.toString(lastNumber + 1);
        }
        return number;
    }

    private void remove() throws Exception { … }  // As original

    private void verifyCreated() throws Exception { … }  // As original

    private void save() throws Exception { … }  // As original

    private void setOtherProperties()
        throws Exception { … }  // As original

    private void chooseCustomer() throws Exception { … }  // As original

    private void verifyDefaultValues()
        throws Exception { … }  // As original

    private String getCurrentYear() { … }  // As original

    private String getCurrentDate() { … }  // As original

}
```

As you see in listing 4.24 you have to do a few changes in order to adapt `CommercialDocumentTest`. First, you declared it as an abstract class. This way this class is not executed by Eclipse as a JUnit test. It is only valid as base class for creating tests, but it is not a test itself.

Another important change is in the constructor, where you now use `moduleName` instead of “Invoice”, thus you can use this test for `Order`, `Invoice` or whatever other module you want. The other changes are simple details: you have to use `xava_view_section0_details` for `viewObject` when you call collection actions, because now the `details` collection is in the first tab (section0), and you have to change “Invoice” for “ComercialDocument” in the query to get the next number.

Now you have a base class ready to create the module tests for `Order` and `Invoice`. Let's do it.

4.4.2 Using the abstract module test to create concrete module tests

Creating your first version of `OrderTest` and `InvoiceTest` is just a matter of extending from `CommercialDocumentTest`. Nothing more. See `InvoiceTest` in listing 4.25.

Listing 4.25 InvoiceTest using inheritance

```
public class InvoiceTest extends CommercialDocumentTest {

    public InvoiceTest(String testName) {
       super(testName, "Invoice");
    }

}
```

And `OrderTest` in listing 4.26.

Listing 4.26 OrderTest using inheritance

```
public class OrderTest extends CommercialDocumentTest {

    public OrderTest(String testName) {
       super(testName, "Order");
    }

}
```

Execute these two tests and you will see how `testCreate()`, inherited from `CommercialDocumentTest`, is executed in both cases, against its corresponding module. With this we are testing the common behavior of `Order` and `Invoice`. Let's test the different cases for them.

4.4.3 Adding new tests to the extended module test

So far we tested how to create an invoice and an order. In this section we'll test also how to add orders to an invoice in the `Invoice` module, and how to set the invoice to an order from the `Order` module.

To test how to add an order to an invoice, add the `testAddOrdersMethod()` method in listing 4.27 to `InvoiceTest`.

Listing 4.27 Testing adding orders to an invoice in InvoiceTest

```java
public void testAddOrders() throws Exception {
    assertListNotEmpty();  // This test assumes that some invoices already exist
    execute("List.orderBy", "property=number"); // To always work with the same order
    execute("Mode.detailAndFirst");  // Goes to detail mode editing the first invoice
    execute("Sections.change", "activeSection=1");  // Changes to tab 1
    assertCollectionRowCount("orders", 0);  // This invoice has no orders
    execute("Collection.add",  // Clicks on the button for adding a new order, this takes
       "viewObject=xava_view_section1_orders");  // you to a list of orders
    execute("AddToCollection.add", "row=0");  // Chooses the first order in the list
    assertCollectionRowCount("orders", 1);  // Order added to the invoice
    checkRowCollection("orders", 0);  // Checks the order, to remove it
    execute("Collection.removeSelected",  // Removes the recently added order
       "viewObject=xava_view_section1_orders");
    assertCollectionRowCount("orders", 0);  // The order has been removed
}
```

In this case we rely on the fact that there is at least one invoice, and that the first invoice in the list has no orders. Before running this test, if you have no invoices, create one with no orders, or if you already have invoices, be sure that the first one has no orders.

To test how to assign an invoice to an order add the `testSetInvoice()` method in listing 4.28 to `OrderTest`.

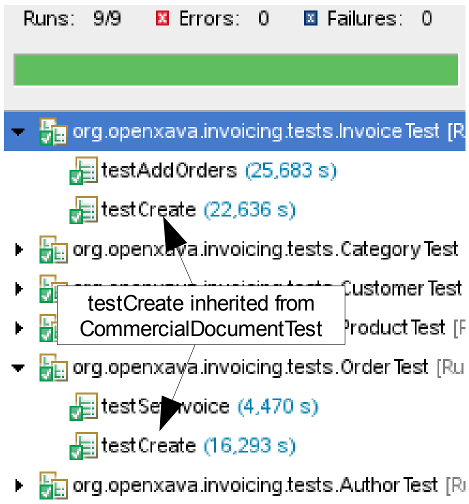
Listing 4.28 Testing assigning an invoice to an order in OrderTest

```java
public void testSetInvoice() throws Exception {
    assertListNotEmpty();  // This test assumes that some orders already exist
    execute("Mode.detailAndFirst");  // Goes to detail mode editing the first invoice
    execute("Sections.change", "activeSection=1");  // Changes to tab 1
    assertValue("invoice.number", "");  // This order has no
    assertValue("invoice.year", "");  // invoice assigned yet
    execute("Reference.search",  // Clicks on the button to search the invoice, this
       "keyProperty=invoice.year");  // takes you to a list of invoices
    execute("List.orderBy", "property=number");  // To sort the list of invoices
    String year = getValueInList(0, "year");  // Stores the year and number of
    String number = getValueInList(0, "number");  // the first invoice in the list
    execute("ReferenceSearch.choose", "row=0");  // Chooses the first invoice
    assertValue("invoice.year", year);  // On return to order detail we verify
    assertValue("invoice.number", number);  // the invoice has been selected
}
```

In this case we rely on the fact that there is at least one order, and that the first order in the list has no invoice. Before running this test, if you have no orders, create one with no invoice, or if you already have orders, be sure that the first one has no invoice.

With this you have your tests ready. Just execute them, and you'll get the result as shown in figure 4.9. Note that the base test `CommercialDocumentTest` is not shown because it is abstract. And the `testCreate()` method of `CommercialDocumentTest` is executed for both `InvoiceTest` and `OrderTest`.

Not only have you adapted your testing code to the improved code for Invoicing, but you have also learned how to use inheritance in the test code as well.



```
Runs: 9/9    Errors: 0    Failures: 0
org.openxava.invoicing.tests.InvoiceTest [R
    testAddOrders (25,683 s)
    testCreate (22,636 s)
org.openxava.invoicing.tests.CategoryTest
org.openxava.invoicing.tests.CustomerTest
org.openxava.invoicing.tests.ProductTest [F
org.openxava.invoicing.tests.OrderTest [Ru
    testSetInvoice (4,470 s)
    testCreate (16,293 s)
org.openxava.invoicing.tests.AuthorTest [R
```

Figure 4.9 All test of Invoicing

4.5 Summary

This lesson has shown you some practical examples about how to use the inheritance of Java and JPA to simplify your code. We used the default JPA configuration for inheritance, though you can refine JPA behavior for inheritance using some JPA annotations like `@Inheritance`, `@DiscriminatorColumn`, `@DiscriminatorValue`, etc. To learn more about inheritance in JPA read the documents cited in appendix B.

OpenXava, at least up to release 4.0.x, only supports single table strategy of JPA, which means that all the data for a class hierarchy are stored in the same table.