Behavior & business logic

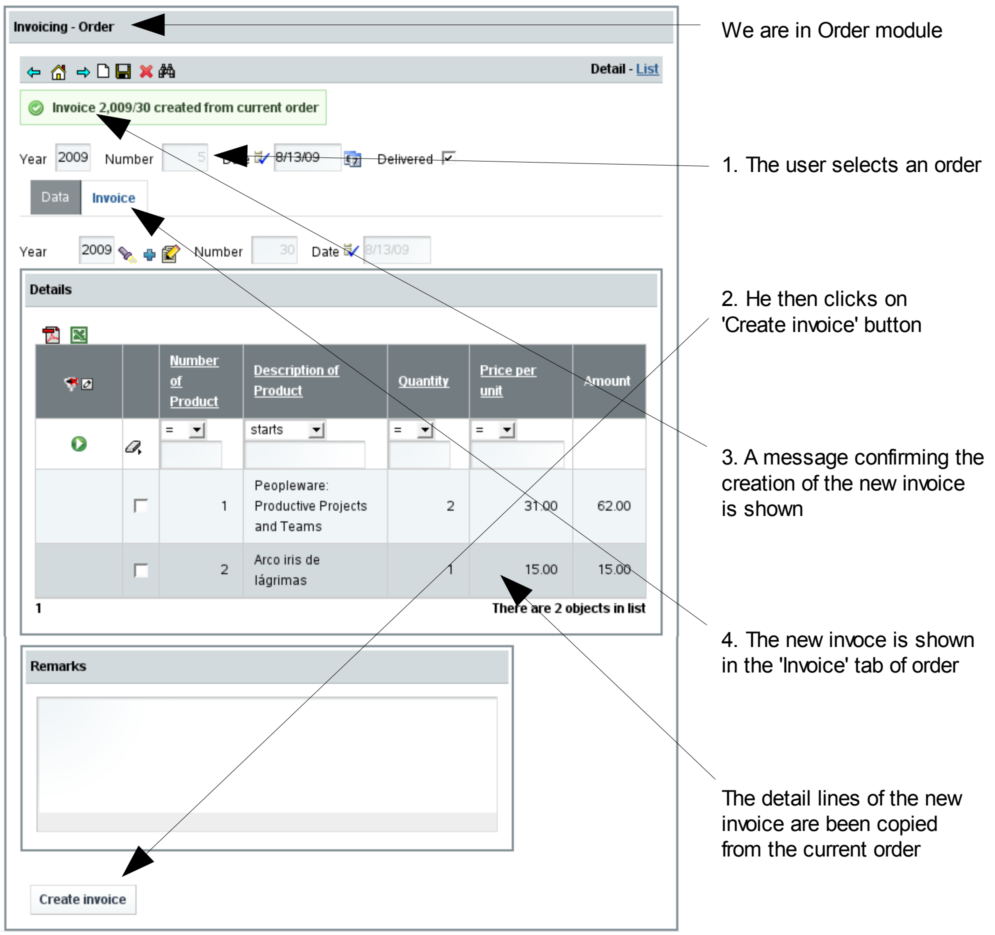
lesson 8

OpenXava is not just a CRUD framework, but a framework for developing full-fledged business applications. Until now we have learned how to create and enhance a data management application. We will now improve the application further by giving the user the possibility to perform specific business logic.

In this lesson we'll see how to add business logic to a model and call this logic from custom actions. In this way you can transform a database management application into a useful tool for the everyday work of your user.

8.1 Business logic in detail mode

We'll start with the simplest case: a button in the detail mode that executes some concrete logic. In this case we'll add a button for creating an invoice from an order. Figure 8.1 shows the desired behavior.



Invoicing - Order

Detail - List

Invoice 2,009/30 created from current order

Year 2009 Number 5 Date 8/13/09 Delivered ✓

Data | Invoice

Year 2009 Number 30 Date 8/13/09

Details

		Number of Product	Description of Product	Quantity	Price per unit	Amount
		=	starts	=	=	
	☐	1	Peopleware: Productive Projects and Teams	2	31.00	62.00
	☐	2	Arco iris de lágrimas	1	15.00	15.00

1 There are 2 objects in list

Remarks

Create invoice

We are in Order module

1. The user selects an order

2. He then clicks on 'Create invoice' button

3. A message confirming the creation of the new invoice is shown

4. The new invoce is shown in the 'Invoice' tab of order

The detail lines of the new invoice are been copied from the current order

Figure 8.1 Creating an invoice from an order using an action

Figure 8.1 shows how this new action takes the current order and creates an invoice from it. It just copies all the order data to the new invoice, including the detail lines. A message is shown and the 'Invoice' tab of the order will display the recently created invoice. Let's see how to implement this.

8.1.1 Creating an action for custom logic

As you already know the first step towards having a custom action in your module is defining a controller for that action. Let's edit *controllers.xml*, to add such a controller. Listing 8.1 shows the `Order` controller.

Listing 8.1 Controller Order in controllers.xml, with the createInvoice action

```
<controller name="Order">
    <extends controller="Invoicing"/>  <!-- In order to have the standard actions -->

    <action name="createInvoice" mode="detail"
       class="org.openxava.invoicing.actions.CreateInvoiceFromOrderAction"/>
       <!-- mode="detail" : Only in detail mode -->

</controller>
```

Since we follow the convention of giving the controller the same name as the entity and the module, you automatically have this new action available for `Order`. `Order` controller extends `Invoicing` controller. Remember that we created `Invoicing` controller in lesson 7. It is a refinement of the `Typical` controller.

Now we have to write the Java code for the action, see it in listing 8.2.

Listing 8.2 Code for the action to create an invoice from an order

```
package org.openxava.invoicing.actions;   // In 'actions' package

import org.openxava.actions.*;
import org.openxava.invoicing.model.*;
import org.openxava.jpa.*;

public class CreateInvoiceFromOrderAction
    extends ViewBaseAction {  // To use getView()

    public void execute() throws Exception {
       Order order = XPersistence.getManager().find( // We use JPA to obtain the
             Order.class,                           // Order entity displayed in the view (1)
             getView().getValue("oid"));
       order.createInvoice();  // The real work is delegated to the entity (2)
       getView().refresh();    // In order to see the created invoice in 'Invoice' tab (3)
       addMessage("invoice_created_from_order",   // Confirmation message (4)
          order.getInvoice());
    }

}
```

Really simple. We find the `Order` entity (1), call the `createInvoice()`

method (2), refresh the view (3) and display a message (4). Note how the action is a mere intermediary between the view (the user interface) and the model (the business logic).

Remember to add the message text to the *Invoicing-messages_en.properties* file in *i18n* folder. Listing 8.3 shows a possible text.

Listing 8.3 Confirmation message in Invoicing-messages_en.properties

```
invoice_created_from_order=Invoice {0} created from current order
```

However, just “as is” the message is not shown nicely, because we pass an `Invoice` object as argument. We need a `toString()` for `Invoice` and `Order` useful to the user. We'll overwrite the `toString()` of `CommercialDocument` (the parent of `Invoice` and `Order`) to achieve this. You can see this `toString()` method in listing 8.4.

Listing 8.4 Method toString() for CommercialDocument

```
abstract public class CommercialDocument extends Deletable {

    ...

    public String toString() {
       return year + "/" + number;
    }

}
```

Year and number are perfect to identify an invoice or order from the user perspective.

That's all for the action. Let's see the missing piece, the `createInvoice()` method of the `Order` entity.

8.1.2 *Writing the real business logic in the entity*

The business logic for creating the new `Invoice` is defined in the `Order` entity, not in the action. This is just the natural way to go. This is the natural way to go in accordance with the essential principle behind Object-Orientation where the objects are not just data, but data and logic. The most beautiful code is that whose objects contain the logic for managing their own data. If your entities are mere data containers (simple wrappers around database tables), and your actions contain all the logic for manipulating them, your code is a perversion of the original goal of Object-Orientation[11].

Apart from the spiritual reason, to put the logic for creating an `Invoice` inside the `Order` entity is a very pragmatic approach, because in this way we can use

11 Unfortunately many of the J2EE patterns and best practice are perversions of OO

this logic from other actions, batch processes, web services, etc.

Let's see the code. Listing 8.5 shows the `createInvoice()` method of the `Order` class.

Listing 8.5 Method createInvoice() in Order entity

```
public class Order extends CommercialDocument {
    ...
    public void createInvoice() throws Exception { // throws Exception is just
                                                   // to get simpler code for now
       Invoice invoice = new Invoice();  // Instantiates an Invoice (1)
       BeanUtils.copyProperties(invoice, this); // and copies the state (2)
                                                // from the current Order
       invoice.setOid(null);  // To let JPA know this entity does not exist yet
       invoice.setDate(new Date());  // The date for the new invoice is today
       invoice.setDetails(new ArrayList());  // Deletes the details collection
       XPersistence.getManager().persist(invoice);
       copyDetailsToInvoice(invoice);  // Fills the details collection
       this.invoice = invoice;  // Always after persist() (3)
    }
}
```

The logic consists of creating a new `Invoice` object (1), copying the data from the current `Order` to it (2) and assigning the resulting entity to the `invoice` reference in the current `Order` (3).

There are two subtle details here. First, you have to write `invoice.setOid(null)`, otherwise the new `Invoice` will get the same identity as the source `Order`. Moreover, JPA does not like to persist objects with the autogenerated id pre-filled. Second, you have to assign the new `Invoice` to the current `Order` (`this.invoice = invoice`) after your call to `persist(invoice)`, if not you get a error from JPA (something like “object references an unsaved transient instance”).

8.1.3 Write less code using Apache Commons BeanUtils

Note how we have used `BeanUtils.copyProperties()` to copy all properties from the current `Order` to the new `Invoice`. This method copies all properties with the same name from one object to another, even if the objects belong to different classes. This utility is from the Commons BeanUtils project from Apache. The jar for this utility, *commons-beanutils.jar*, is already included in your project.

Listing 8.6 shows how using `BeanUtils` you actually write less code.

Listing 8.6 BeansUtil.copyProperties() versus copying the properties manually

```
BeanUtils.copyProperties(invoice, this);
// Is the same as
invoice.setOid(getOid());
invoice.setYear(getYear());
invoice.setNumber(getNumber());
invoice.setDate(getDate());
invoice.setDeleted(isDeleted());
invoice.setCustomer(getCustomer());
invoice.setVatPercentage(getVatPercentage());
invoice.setAmount(getAmount());
invoice.setRemarks(getRemarks());
invoice.setDetails(getDetails());
```

However, the main advantage of using `BeanUtils` is not to save some typing, but that you have code more resilient to changes. Because, if you add, remove or rename some property of `ComercialDocument` (the parent of `Invoice` and `Order`) you don't need to change your code, while if you're copying the properties manually you must change the code manually.

8.1.4 Copying a collection from entity to entity

The new `Invoice` must have the same detail lines as those of the `Order`. Actually, not the same collection but a copy. We cannot just assign the collection as shown in listing 8.7.

Listing 8.7 Wrong way to copy a collection from entity to entity

```
invoice.setDetails(getDetails());  // This does not work
```

This does not work because a one-to-many collection cannot be assigned to two entities at same time, so we have to make a copy. Note how in the `createInvoice()` method (listing 8.4) we used `invoice.setDetails(new ArrayList())` to reset the collection. This is because `BeanUtils.copyProperties()` copied the `details` collection from `Order`. In fact, it copies anything with a setter and getter.

Listing 8.8 shows the `copyDetailsToInvoice()` method that copies the `details` collection from `Order` to `Invoice`.

Listing 8.8 Method copyDetailsToInvoice() in Order entity

```
private void copyDetailsToInvoice(Invoice invoice) throws Exception {
    for (Detail orderDetail: getDetails()) { // Iterates over the details of current order
       Detail invoiceDetail = (Detail)  // Clones the detail (1)
          BeanUtils.cloneBean(orderDetail);
       invoiceDetail.setOid(null);  // To be persisted as a new entity(2)
       invoiceDetail.setParent(invoice);  // The important point: set a new parent (3)
       XPersistence.getManager().persist(invoiceDetail); // (4)
    }
}
```

This is the simplest way to clone the collection, just clone each element (1) and assign a new parent to it (3). Furthermore, you have to remove its identity (2) and mark it as persistent (4).

To clone the bean we use `BeanUtils` again, in this case the `cloneBean()` method. This method creates a new instance of the same type as the argument, and copies all the properties of the source object to the newly created object.

8.1.5 Application exceptions

Remember the phrase: "The exception that proves the rule"? Rules, life and software are full of exceptions. And our `createInvoice()` method is not an exception. We have written the code to work in the most common cases. But, what happens if the order is not ready to be invoiced, or if there is some problem accessing the database? Obviously, in these cases we need to take different paths.

This is to say that the simple `throws Exception` we have written for `createInvoice()` method is not enough to ensure a robust behavior. Listing 8.9 shows an improved version of the method using exceptions.

Listing 8.9 The createInvoice() method treating exceptional cases

```
public void createInvoice()
    throws ValidationException  // An application exception (1)
{
    if (this.invoice != null) {  // If an invoice is already present we cannot create one
       throw new ValidationException(   // Allows an i18n id as argument
          "impossible_create_invoice_order_already_has_one");
    }
    if (!isDelivered()) {  // If the order is not delivered we cannot create the invoice
       throw new ValidationException(
          "impossible_create_invoice_order_is_not_delivered");
    }
    try {
       Invoice invoice = new Invoice();
       BeanUtils.copyProperties(invoice, this);
       invoice.setOid(null);
       invoice.setDate(new Date());
       invoice.setDetails(new ArrayList());
       XPersistence.getManager().persist(invoice);
       copyDetailsToInvoice(invoice);
       this.invoice = invoice;
    }
    catch (Exception ex) {  // Any unexpected exception (2)
       throw new SystemException(   // A runtime exception is thrown (3)
          "impossible_create_invoice", ex);
    }
}
```

Now we declare explicitly which application exceptions this method throws (1). An application exception is a checked exception that indicates a special but expected behavior of the method. An application exception is related to the method's business logic. You could create an application exception for every possible case, such as an `OrderAlreadyHasInvoiceException` and an `InvoiceNotDeliveredException`. This enables you to handle each case differently in the calling code. This is not needed in our case, so we simply use

ValidationException, a generic application exception included in OpenXava.

Additionally, we have to deal with unexpected problems (2). Unexpected problems can be system errors (database access, net or hardware problems) or programmer errors (NullPointerException, IndexOutOfBoundsException, etc). When we find any unexpected problem we throw a runtime exception. In this instance we have thrown SystemException, a runtime exception included in OpenXava for convenience, but you can throw any runtime exception you want.

You do not need to modify the action code. If your action does not catch the exceptions, OpenXava does it automatically. It displays the messages from the ValidationExceptions to the user, and, for the runtime exceptions, shows a generic error message, and rolls back the transaction.

In order to be complete, we have to add the messages used for the exceptions in the i18n files. Edit the *Invoicing-messages_en.properties* file from *Invoicing/i18n* folder adding the entries in listing 8.10.

Listing 8.10 Messages used by the exceptions

```
 impossible_create_invoice_order_already_has_one=Impossible to create invoice:
the order already has an invoice
 impossible_create_invoice_order_is_not_delivered=Impossible to create invoice:
the order is not delivered yet
 impossible_create_invoice=Impossible to create invoice
```

There is some debate in the developer community regarding the correct way of using exceptions in Java. The approach in this section is the classic way to work with exceptions in the J2EE world.

8.1.6 Validation from action

Usually the best place for validations is the model, i.e., the entities. However, sometimes it's necessary to put validation logic in the actions. For example, if you want to obtain the current state of the user interface, the validation must be done from the action.

In our case, if the user clicks on "Create invoice" when creating a new order, and this order is not yet saved, it will fail. It fails because it's impossible to create an invoice from an non-existent order. The user must first save the order.

Listing 8.11 shows the execute() method of CreateInvoiceFromOrderAction modified to validate that the currently displayed order is saved.

Listing 8.11 Validation from the action to ask for view state

```
 public void execute() throws Exception {
     Object oid = getView().getValue("oid");
```

```
      if (oid == null) {  // If oid is null the current order is not saved yet (1)
         addError(
            "impossible_create_invoice_order_not_exist");
         return;
      }
      MapFacade.setValues("Order",   // If the order exists we save it (2)
         getView().getKeyValues(), getView().getValues());
      Order order = getManager().find(
         Order.class, oid);
      order.createInvoice();
      getView().refresh();
      addMessage("invoice_created_from_order",
         order.getInvoice());
   }
```

The validation consists of verifying if the oid is null (1), in which case the user is entering a new order, but he did not save it yet. In this case a message is shown, and the creation of the invoice is aborted. If the order already exists we save the data from the user interface to the database using `MapFacade` (2). It's important to have the database synchronized with the view before calling the entity method to create the invoice. Imagine that the user marks the order as delivered and then clicks on "Create invoice". In this case he would get an error message stating "Order not delivered". This can be confusing, so saving the data automatically before calling any entity method is a good idea. Note how convenient a tool `MapFacade` is for moving data between the user interface and the model.

Here we also have a message to add to the i18n file. Edit the *Invoicing-messages_en.properties* file in the *Invoicing/i18n* folder adding the entry in listing 8.12.

Listing 8.12 Message used by the action validation

```
 impossible_create_invoice_order_not_exist=Impossible to create invoice: the
order does not exist yet
```

Validations tell the user that he has done something wrong. This is needed, of course, but better still is to create an application that helps the user to avoid any wrong doings. Let's see one way to do so in the next section.

8.1.7 *On change event to hide/show an action programmatically*

Our current code is robust enough to prevent user slips from breaking data. We will go one step further, preventing the user to slip at all. We're going to hide the action for creating a new invoice, if the order is not valid to be invoiced.

OpenXava allows to hide and show actions programmatically. It also allows the execution of an action when some property is changed by the user on the screen. We can use these two techniques to show the button only when the action is ready to be used.

Remember that an invoice can be generated from an order only if the order has been delivered and it does not yet have an invoice. So, we have to monitor the changes in the `invoice` reference and `delivered` property of the `Order` entity. We'll do that using the `@OnChange` annotation as shown in listing 8.13.

Listing 8.13 @OnChange added to invoice and delivered in Order

```
public class Order extends CommercialDocument {

    @ManyToOne
    @ReferenceView("NoCustomerNoOrders")
    @OnChange(ShowHideCreateInvoiceAction.class)
    private Invoice invoice;

    @OnChange(ShowHideCreateInvoiceAction.class)
    private boolean delivered;

    ...
}
```

With the above code when the user changes the value of `delivered` or `invoice` in the screen, the `ShowHideCreateInvoiceAction` will be executed. See the action code in listing 8.14.

Listing 8.14 Action to show/hide the createInvoice action dynamically

```
package org.openxava.invoicing.actions;  // In the 'actions' package

import org.openxava.actions.*;    // Needed to use OnChangePropertyAction,
                                  // IShowActionAction and IHideActionAction
public class ShowHideCreateInvoiceAction
    extends OnChangePropertyBaseAction  // Needed for @OnChange actions (1)
    implements IShowActionAction,   // To show an action
       IHideActionAction {   // To hide an action

    private boolean show;   // If true the 'Order.createInvoice' action will be shown

    public void execute() throws Exception {
       show = isOrderCreated()    // We set the value to 'show'. This value
          && isDelivered()        // will be used in the below methods:
          && !hasInvoice();       // getActionToShow() and getActionToHide() (2)
    }

    private boolean isOrderCreated() {
       return getView().getValue("oid") != null;   // We read the value from the view
    }

    private boolean isDelivered() {
       Boolean delivered = (Boolean)
          getView().getValue("delivered");   // We read the value from the view
       return delivered == null?false:delivered;
    }

    private boolean hasInvoice() {
       return getView().getValue("invoice.oid") != null;  // We read the value
    }                                                     // from the view
    public String getActionToShow() {  // Required because of IShowActionAction
       return show?"Order.createInvoice":"";   // The action to show (3)
    }

    public String getActionToHide() {  // Required because of IHideActionAction
       return !show?"Order.createInvoice":"";   // The action to hide (3)
    }

}
```

This is a conventional action with an `execute()` method, moreover it extends `OnChangePropertyBaseAction` (1). All the actions annotated with `@OnChange` must implement `IOnChangePropertyAction`, however it's easier to extend `OnChangePropertyBaseAction` which implements it. From this action you can use the `getNewValue()` and `getChangedProperty()`, although in this specific case we don't need them.

The `execute()` method sets the `show` field to true if the displayed order is saved, delivered, and does not already have an invoice (2). The `show` field is then used in the `getActionToShow()` and `getActionToHide()` methods. These methods indicate the qualified name of the action to hide or to show (3). Thus we hide or show the `Order.createInvoice` action, only showing it when it is applicable.

Now you can try the `Order` module. You will see how when you check the delivered checkbox, or choose an invoice, the action button is shown or hidden. Accordingly, when the user clicks on 'New' to create a new order the button for creating the invoice is hidden. However, if you choose to modify an already existing order, the button is always present, regardless if the prerequisites are fulfilled. This is because when an object is searched and displayed the `@OnChange` actions are not executed by default. We can change this with a little modification in `SearchExcludingDeleteAction`. See it in listing 8.15.

Listing 8.15 The search action extends from SearchExecutingOnChangeAction

```
public class SearchExcludingDeletedAction
    ~~extends SearchByViewKeyAction {~~
    extends SearchExecutingOnChangeAction {   // Use this as base class
```

The default search action, i.e., `SearchByViewKeyAction` does not execute the `@OnChange` actions, so we change our search action to extend from `SearchExecutingOnChangeAction`. `SearchExecutingOnChangeAction` behaves like `SearchByViewKeyAction` but executes the on-change events. This way, when the user selects an order, the `ShowHideCreateInvoiceAction` is executed.

A tiny detail remains to make all this perfect: when the user click on 'Create invoice', after the invoice has been created, the button should be hidden. It should not be possible to create the same invoice twice. We can implement this functionality by refining the `CreateInvoiceFromOrderAction`. Look at the listing 8.16.

Listing 8.16 CreateInvoiceFromOrderAction hides itself after execution

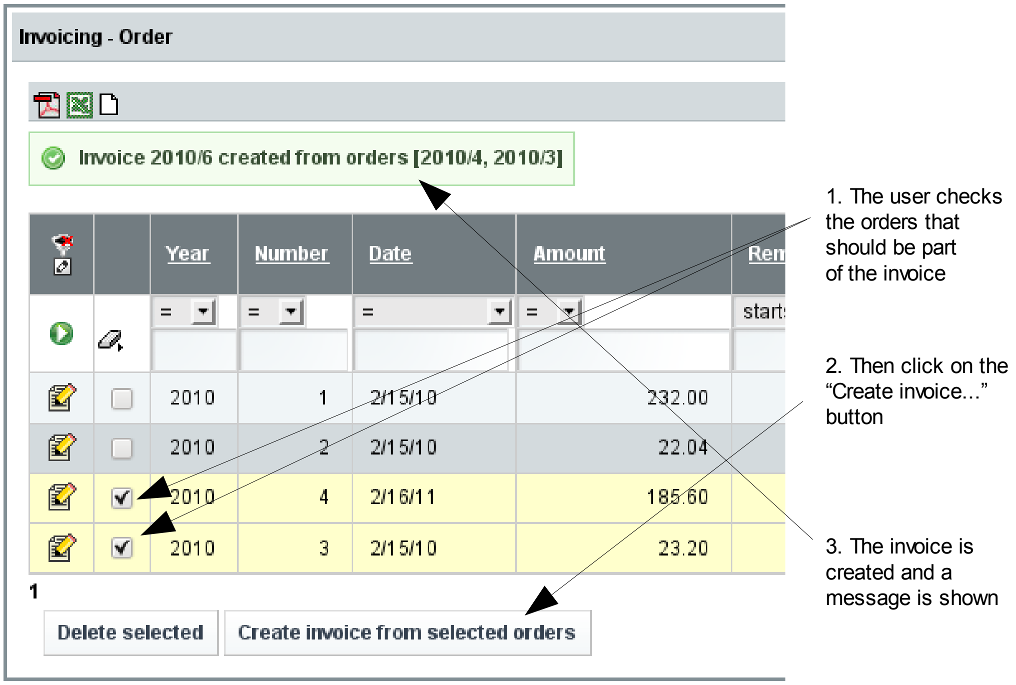
```java
public class CreateInvoiceFromOrderAction extends ViewBaseAction
    implements IHideActionAction   // To hide the action
{

    private boolean hideAction = false;   // To indicate if the action will be hidden

    public void execute() throws Exception {
       ...
       addMessage("invoice_created_from_order",
          order.getInvoice());
       hideAction = true;   // Everything worked fine, so we'll hide the action
    }

    public String getActionToHide() {   // The action to hide, in this case itself
       return hideAction?"Order.createInvoice":null;
    }

}
```

As you can see the action implements `IHideActionAction` in order to hide itself.

Showing and hiding actions is not a substitute for validation in the model. Validations are still necessary since the entities can be used from any other part of the application, not just from the CRUD module. However, the trick of hiding and showing actions improves the user experience.

8.2 Business logic from list mode

In lesson 4 (section 4.3) you learned how to create list actions. List actions are very useful tools that provides the user with the ability to perform some specific logic on multiple objects at the same time. In our case, we can add an action in list mode to create a new invoice automatically from several selected orders in the list. Figure 8.2 shows the way we want this action to work.


Invoicing - Order

Invoice 2010/6 created from orders [2010/4, 2010/3]

		Year	Number	Date	Amount	Rem
		=	=	=	=	starts
	☐	2010	1	2/15/10	232.00	
	☐	2010	2	2/15/10	22.04	
	✓	2010	4	2/16/11	185.60	
	✓	2010	3	2/15/10	23.20	

1
Delete selected
Create invoice from selected orders
1. The user checks the orders that should be part of the invoice
2. Then click on the "Create invoice..." button
3. The invoice is created and a message is shown

Figure 8.2 Creating an invoice from several orders using an list action

Figure 8.2 shows how this list action takes the selected orders and creates an invoice from them. It just copies the order data into the new invoice, adding the detail lines of all the orders in one unique invoice. Also a message is shown. Let's see how to code this behavior.

8.2.1 *List action with custom logic*

As you already know, the first step towards having a new custom action in your module is to add that action to a controller. So, let's edit *controllers.xml* adding a new action to the `Order` controller. Listing 8.17 shows the `Order` controller modified.

Listing 8.17 Controller Order with the createInvoiceFromSelectedOrders action

```
<controller name="Order">
    <extends controller="Invoicing"/>

    <action name="createInvoice" mode="detail"
       class=
    "org.openxava.invoicing.actions.CreateInvoiceFromOrderAction">
       <use-object name="xava_view"/>
    </action>

    <!-- The new action -->
    <action name="createInvoiceFromSelectedOrders"
       mode="list"
```

```
        class=
        "org.openxava.invoicing.actions.CreateInvoiceFromSelectedOrdersAction"
    />
    <!-- mode="list":  Only shown in list mode -->

</controller>
```

This is all that is needed to have this new action available for `Order` in list mode.

Now we have to write the Java code for the action. See it in listing 8.18.

Listing 8.18 Code for the list action to create an invoice from several orders

```
public class CreateInvoiceFromSelectedOrdersAction
    extends TabBaseAction   // Typical for list actions. It allows you to use getTab() (1)
{

    public void execute() throws Exception {
       Collection<Order> orders = getSelectedOrders();   // (2)
       Invoice invoice = Invoice.createFromOrders(orders); // (3)
       addMessage(  // (4)
          "invoice_created_from_orders", invoice, orders);
    }

    private Collection<Order> getSelectedOrders()   // (5)
       throws FinderException
    {
       Collection<Order> result = new ArrayList<Order>();
       for (Map key: getTab().getSelectedKeys()) {   // (6)
          Order order = (Order)
             MapFacade.findEntity("Order", key);   // (7)
          result.add(order);
       }
       return result;
    }

}
```

Really simple. We obtain the list of the checked orders in the list (2), call `createFromOrders()` static method (3) of `Invoice` and show a message (4). In this case we also put the real logic in the model class, not in the action. Since the logic applies to several orders and creates a new invoice the natural place to put it is a static method of `Invoice` class.

The `getSelectedOrders()` method (5) returns a collection containing the `Order` entities checked by the user in the list. This is easily achieved using `getTab()` (6), available from `TabBaseAction` (1), that returns an `org.openxava.tab.Tab` object. The `Tab` object allows you to manage the tabular data of the list. In this case we use `getSelectedKeys()` (6) that returns a collection with the keys of the selected rows. Since these keys are in `Map` format we use `MapFacade.findEntity()` (7) to convert them to `Order` entities.

As always, add the message text to the *Invoicing-messages_en.properties* file in *i18n* folder. Listing 8.19 shows a possible text.

Listing 8.19 Confirmation message in Invoicing-messages_en.properties

```
invoice_created_from_orders=Invoice {0} created from orders: {1}
```

That's all for the action. Let's see the missing piece, the `createFromOrders()` method of the `Invoice` class.

8.2.2 *Business logic in the model over several entities*

The business logic for creating a new `Invoice` from several `Order` entities is in the model layer, i.e., the entities, not in the action. We cannot put the method in `Order` class, because the process is done from several `Orders`, not just one. We cannot use an instance method in `Invoice` because the invoice does not exist yet, in fact we want to create it. Therefore, we are going to create a static factory method in the `Invoice` class for creating a new `Invoice` from several `Orders`. You can see this method in listing 8.20.

Listing 8.20 Method createFromOrders() in Invoice entity

```
public class Invoice extends CommercialDocument {

    ...

    public static Invoice createFromOrders(Collection<Order> orders)
       throws ValidationException
    {
       Invoice invoice = null;
       for (Order order: orders) {
          if (invoice == null) {  // The first order
             order.createInvoice();  // We reuse the logic for creating an invoice
                                     // from an order
             invoice = order.getInvoice();  // and use the created invoice
          }
          else {  // For the remainding orders the invoice is already created
             order.setInvoice(invoice);  // Assign the invoice
             order.copyDetailsToInvoice(invoice);  // Copies the lines
          }                                  // The copyDetailsToInvoice method is
       }                             // private in Order, so we need to change to public
       if (invoice == null) {  // If there are no orders
          throw new ValidationException(
          "impossible_create_invoice_orders_not_specified");
       }
       return invoice;
    }

}
```

We use the first `Order` to create the new `Invoice` using the already existing `createInvoice()` method from `Order`. Then we copy the lines from the remainding `Orders` to the new `Invoice`. Moreover, we set the new `Invoice` as the `Invoice` for the `Orders` of the collection.

If `invoice` is null at the end of the process it's because the `orders` collection is empty. In this case we throw a `ValidationException`. Since the action does not catch the exceptions, OpenXava shows the `ValidationException` message to the user. This is fine. If the user does not check any orders and he clicks on the button for creating an invoice, then this error message will be shown to him.

We use the `copyDetailsToInvoice()` method from `Order`. This method was declared as private, so we need to modify this and make it public to be able to use it from `Invoice`. See the change in listing 8.21.

Listing 8.21 Refinements in copyDetailsToInvoice() of Order

```
public class Order extends CommercialDocument {

    ...

    public ~~private~~  // public instead of private
       void copyDetailsToInvoice(Invoice invoice)
       ~~throws Exception~~  // throws Exception is removed. Now we'll throw
    {                        // a runtime exception instead
       try {  // We wrap all the code for the method with a try/catch
          for (Detail orderDetail: getDetails()) {
             Detail invoiceDetail = (Detail)
                BeanUtils.cloneBean(orderDetail);
             invoiceDetail.setOid(null);
             invoiceDetail.setParent(invoice);
             XPersistence.getManager()
                .persist(invoiceDetail);
          }
       }
       catch (Exception ex) {  // We convert every exception into
          throw new SystemException(  // a runtime exception
             "impossible_copy_details_to_invoice", ex);
       }
    }

}
```

In addition to changing 'private' to 'public' we convert any exception to a runtime exception by wrapping the method with a a try/catch clause. This way, we adhere to the aforesaid convention of using runtime exception for unexpected problems.

Remember to add the messages text to the *Invoicing-messages_en.properties* file in *i18n* folder. Listing 8.22 shows possible texts.

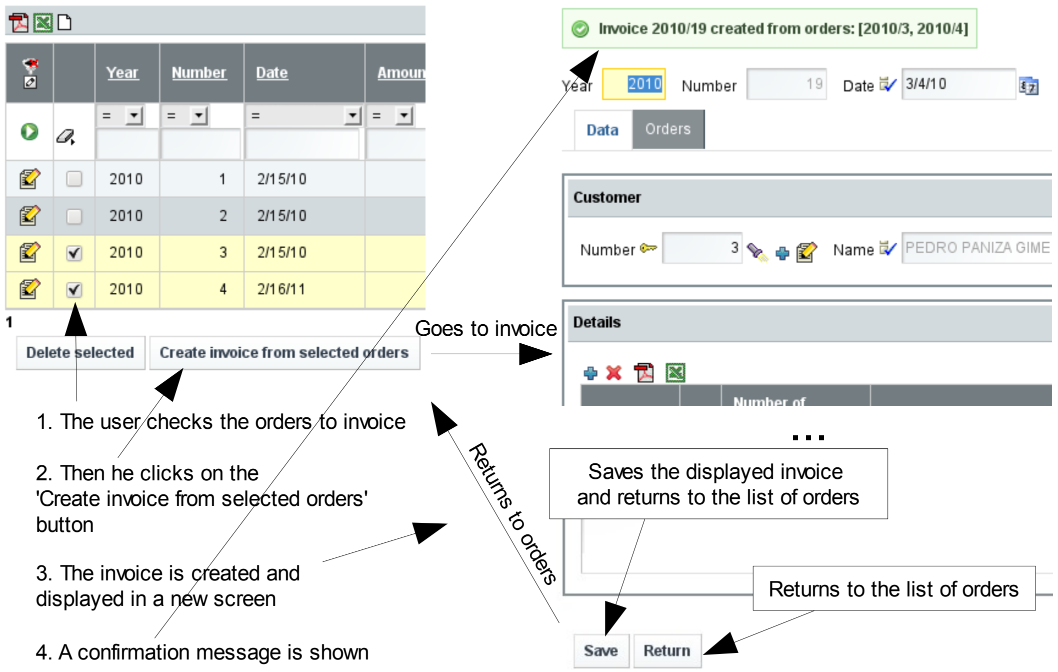
Listing 8.22 Validation error in Invoicing-messages_en.properties

```
impossible_create_invoice_orders_not_specified=Impossible to create invoice:
orders not specified
impossible_copy_details_to_invoice=Impossible to copy details from order to
invoice
```

These are not the only errors the user can get. All previously written validations for `Invoice` and `Order` still apply automatically. This ensures that the user has to choose orders from the same customer, that are delivered, that lacks an invoice, etc. Model validation prevents the user from creating invoices from the wrong orders.

8.3 Changing module

After creating an invoice from several orders, it would be practical for the user to see and possibly edit the newly created invoice. One way of achieving this is by creating a module solely for editing an invoice, with no list mode and without the typical CRUD actions. This way, we can move into this module upon invoice creation for inspection and editing. Figure 8.3 shows the desired behavior.



Year
Number
Date
Amoun
Delete selected
Create invoice from selected orders
Goes to invoice
Invoice 2010/19 created from orders: [2010/3, 2010/4]
Data
Orders
Customer
Name
PEDRO PANIZA GIME
Details
Number of
1. The user checks the orders to invoice
2. Then he clicks on the 'Create invoice from selected orders' button
3. The invoice is created and displayed in a new screen
4. A confirmation message is shown
Returns to orders
Saves the displayed invoice and returns to the list of orders
Returns to the list of orders
Save
Return

		Year	Number	Date	Amoun
	☐	2010	1	2/15/10	
	☐	2010	2	2/15/10	
	✓	2010	3	2/15/10	
	✓	2010	4	2/16/11	

Year 2010 Number 19 Date 3/4/10

Customer Number 3

Figure 8.3 Editing the invoice after creating it from several orders

Let's see how to implement this behavior.

8.3.1 Using IChangeModuleAction

The first step is to modify `CreateInvoiceFromSelectedOrdersAction` to change the module after its execution. Listing 8.23 shows the modification.

Listing 8.23 Modification in the action to change to a new module

```
public class CreateInvoiceFromSelectedOrdersAction
    extends TabBaseAction
    implements IChangeModuleAction  // To change to another module after execution
{

    public String getNextModule() {
       return "CurrentInvoiceEdition";  //  Module name as defined in application.xml
    }

    public boolean hasReinitNextModule() {
       return true;  // The module is initialized each time we change to it
    }

    ...

}
```

As you can see, it suffices to implement the `IChangeModuleAction`. This forces you to add the methods `getNextModule()`, to return the name of the module as defined in *application.xml*, and `hasReinitNextModule()`. We return true from `hasReinitNextModule()` because we will write an `on-init` action (action executed when the module is initialized) in the `CurrentInvoiceEdition` module and use this to load the correct invoice into the view on every initialization.

Obviously, this will not work until we have the `CurrentInvoiceEdition` module defined. We'll do this in the next section.

8.3.2 Detail only module

The purpose of the `CurrentInvoiceEdition` module is to display a single invoice, and present the option of editing it.

To define the module, edit the *application.xml* file and add the module definition of listing 8.24

Listing 8.24 Detail only module to edit an Invoice in application.xml

```
<module name="CurrentInvoiceEdition">
    <model name="Invoice"/>
    <controller name="CurrentInvoiceEdition"/>
    <mode-controller name="Void"/>  <!-- Thus the module only has detail mode -->
</module>
```

Since this module is for editing a particular `Invoice`, it doesn't have a list mode, but only a detail mode. We use `Void` as `mode-controller` to achieve this.

This module only allows the user to change the `Invoice` and save the changes or return to the calling module. This requires a new controller with actions to perform saving and returning. We call this controller `CurrentInvoiceEdition` and add it to *controllers.xml*, as shown in listing 8.25.

Listing 8.25 Controller for editing an invoice in controllers.xml

```
<controller name="CurrentInvoiceEdition">

    <action name="save"
       class="org.openxava.invoicing.actions.SaveInvoiceAction"
       keystroke="Control S"/>

    <action name="return"
       class="org.openxava.actions.ReturnPreviousModuleAction"/>

</controller>
```

The two actions of this controller represent the two buttons, 'Save' and 'Return' you saw in the previous figure 8.3.

8.3.3 Returning to the calling module

`SaveInvoiceAction` contains just a minor extension of the standard `SaveAction` of OpenXava. Listing 8.26 shows its code.

Listing 8.26 Action that saves the invoice and returns to the calling module

```
public class SaveInvoiceAction
    extends SaveAction   // Standard OpenXava action to save the view content
    implements IChangeModuleAction   // For module navigation
{

    public String getNextModule() {
       return PREVIOUS_MODULE;    // Returns to the calling module,
    }                             // that is the Order module in this case

    public boolean hasReinitNextModule() {
       return false;   // We don't want to initialize the Order module
    }

}
```

The action extends `SaveAction` without overwriting the `execute()` method which means that its behavior is exactly the same as that of the generic OpenXava `SaveAction`: to save the displayed data in the database. As our addition, we state that the action must return to the calling module, the `Order` module in our example, when it finishes.

In this way, when the user clicks on 'Save', the invoice data is saved and the application returns to the list of orders, ready to continue the creation of invoices from orders.

For returning to the calling module we must always use PREVIOUS_MODULE. Do not use the module name, just as shown in listing 8.27.

Listing 8.27 Never use the module name to return to the calling module

```
public String getNextModule() { return PREVIOUS_MODULE; }  // Good
public String getNextModule() { return "Order"; }  // Very BAD
```

If you use PREVIOUS_MODULE you get the advantage that you can call this module from any module in the application, and this will know what module to return to in each case. But even more important is the fact that OpenXava uses a stack of module calls in order to return, so if you call to the calling module you will produce a reentrance problem.

For the 'Return' button we use the `ReturnPreviousModuleAction`, an action included in OpenXava that simply returns to the previous module.

8.3.4 Global session object and on-init action

The current code is still incomplete. When the user generates the invoice the `CurrentInvoiceEdition` module is activated, but it is empty, no invoice is shown. We have to fill the view of the new module with the newly created invoice. Let's learn how to share data between modules.

One way of sharing data between modules is to declare a session object with a global scope. This is accomplished by adding an entry in *controllers.xml* as shown in listing 8.28.

Listing 8.28 A global scoped session object defined in controllers.xml

```
<controllers>

    ...

    <object name="invoicing_currentInvoiceKey"
       class="java.util.Map"
       scope="global"/>
       <!--
       name="invoicing_currentInvoiceKey": Name must be unique
       class="java.util.Map": The type of the object
       scope="global": Shared by all modules. The default value is "module"
       -->

    ...
```

A session object is an object associated to the user session, therefore it lives while the user session is alive, and each user has its own copy of the object. If you use `scope="global"` the object will be shared by all the modules, otherwise each module has its own copy of the object.

We declare the scope of the object as global because we want to use it to pass data from the `Order` module to `CurrentInvoiceEdition` module. The way to use such an object is by injecting it into an action by means of `@Inject`[12] annotation. Before calling the `execute()` method of the action, the `invoicing_currentInvoiceKey` object is injected into the `currentInvoiceKey` field of the action. The name of the field in the action is the name of the session object without the prefix (without `invoicing_` in this case), though you can inject the object in a field with another name if you use the `@Named` annotation. Listing 8.29 shows the `currentInvoiceKey` field with the `@Inject` annotation added to the action class.

Listing 8.29 Field currentInvoiceKey to be injected from the session object

```
...

import javax.inject.*;

public class CreateInvoiceFromSelectedOrdersAction ... {

    ...

    @Inject
    private Map currentInvoiceKey;   // A private field without getter and setter

    ...

}
```

The interesting thing about `@Inject` is that, in addition to injecting the object in the field before calling `execute()`, it extracts the value of the action field and puts it back into the session context after running the `execute()` method. In other words, if you modify the value of the `currentInvoiceKey` field from `CreateInvoiceFromSelectedOrdersAction` then the `invoicing_currentInvoiceKey` session object is modified too. Hence, we can use this action to give value to this session object. Listing 8.30 shows the modification in the action code.

Listing 8.30 Filling the currentInvoiceKey session object from the action

```
public class CreateInvoiceFromSelectedOrdersAction ... {

    ...

    public void execute() throws Exception {
       Collection<Order> orders = getSelectedOrders();
       Invoice invoice = Invoice.createFromOrders(orders);
       addMessage("invoice_created_from_orders",
          invoice, orders);
       currentInvoiceKey = toKey(invoice);   // Puts the key of the newly
    }                                // created invoice in the currentInvoiceKey field, it
                                     // also sets the invoicing_ currentInvoiceKey session object

    private Map toKey(Invoice invoice) {   // Extracts the key from the
```

12 The @javax.inject.Inject annotation is defined by the JSR-330 Java standard

```
        Map key = new HashMap();          // invoice in Map format
        key.put("oid",invoice.getOid());
        return key;
    }

    ...

}
```

After the creation of the invoice, we put the key of that invoice in the session object. Populating a session object is a breeze, you only have to set a value to the field declared with `@Inject`. In this case assigning value to `currentInvoiceKey` is sufficient to fill the corresponding `invoicing_currentInvoiceKey` object. Afterwards, you can use this object from other actions, and since its scope is global, from actions of other modules too.

We are going to create a new action in the `CurrentInvoiceEdition` module to load the values of the invoice created from the `Order` module with `CreateInvoiceFromSelectedOrdersAction`. Listing 8.31 shows the declaration of this `load` action in the *controllers.xml* file.

Listing 8.31 The load action declaration in controllers.xml with on-init=true

```
<controller name="CurrentInvoiceEdition">

    <action name="load"
       class=
       "org.openxava.invoicing.actions.LoadCurrentInvoiceAction"
       hidden="true"  on-init="true"/>
       <!--
       hidden="true": There is not a button or link in the UI for this action
       on-init="true": It is executed automatically when the module initiates
       -->

    ...

</controller>
```

We declare the action as `hidden=true`, thus it will not be visible, and so the user will not have the possibility to execute it. Moreover, we declare it with `on-init=true`, so it will be executed automatically when the module is initialized.

Remember, that when this module is called, the method `hasReinitNextModule()` returns true. This causes `CurrentInvoiceEdition` to be initialized every time it is called from the `Order` module which in turn means it will get its values loaded via the load action. Therefore, this `load` action is the ideal place to populate the view with the recently created invoice. Let's see its code in listing 8.32.

Listing 8.32 Action to load the last created invoice in the view

```
public class LoadCurrentInvoiceAction
    extends SearchByViewKeyAction {  // To populate the view from the key values

    @Inject
    private Map currentInvoiceKey;  // To get the value of the session object
                                    // invoicing_currentInvoiceKey, filled in the Order module
    public void execute() throws Exception {
       getView().setValues(currentInvoiceKey);  // Puts the key in the view
       super.execute();  // Populates the view from the key fields
    }

}
```

It extends from `SearchByViewKeyAction` which is the standard OpenXava action for searching. `SearchByViewKeyAction` takes the key fields in the view, does a search for the corresponding entity, and then fills the rest of the view fields from the entity. Therefore, we only have to fill the view with the key values before calling `super.execute()`.

By using `currentInvoiceKey`, we access the key values stored there by `CreateInvoiceFromSelectedOrdersAction`. Now you have learned how to use a session object to share data between actions, even between actions of different modules.

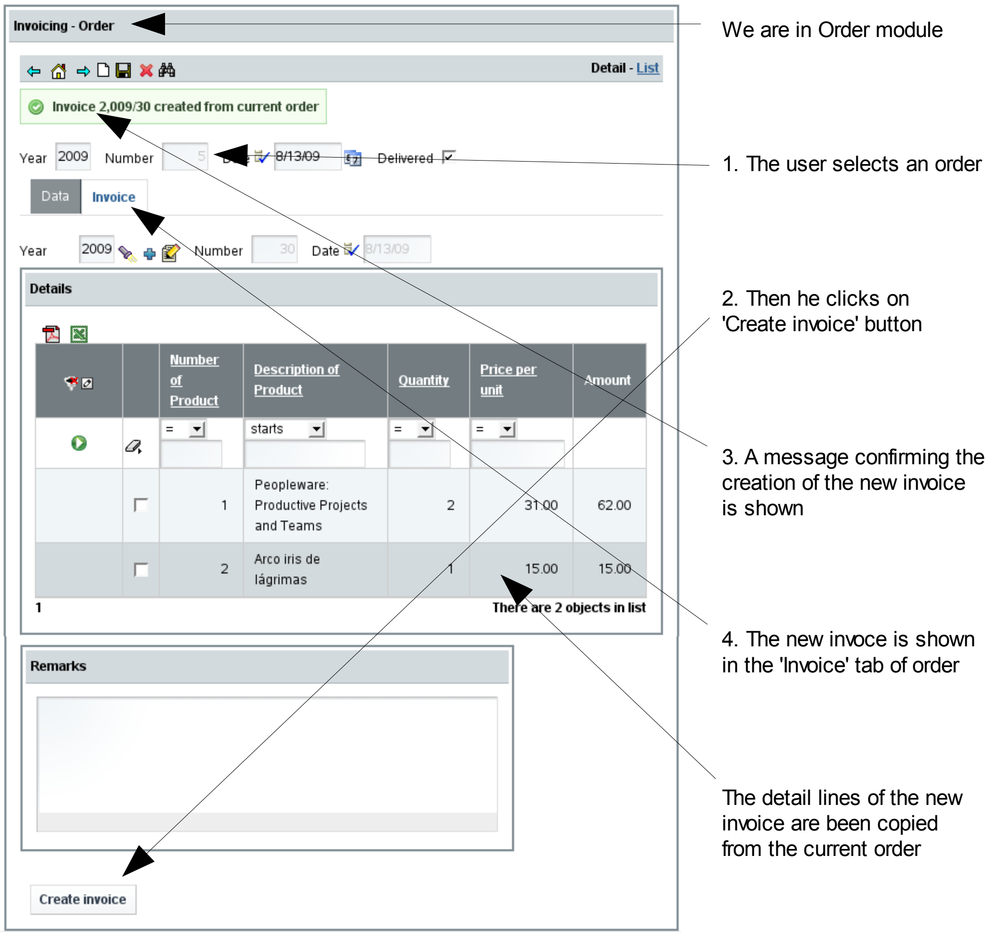
Our work is almost done. If you try out the `Order` module, choose several orders, and click on the 'Create invoice from selected orders' button you will be directed to the detail mode of the newly created invoice. Just as you saw in the previous figure 8.3.

8.4 JUnit tests

The code we have written in this lesson is not complete until we write the tests. Remember, all new code must have its corresponding test code. Let's write tests for the two new actions.

8.4.1 Testing the detail mode action

First we'll test the `Order.createInvoice` action, the action for creating an invoice from the displayed order in detail mode. We reprint here figure 8.1 of section 8.1 that shows how this process works.



Invoicing - Order

Detail - List

Invoice 2,009/30 created from current order

Year 2009 Number 5 Date 8/13/09 Delivered ✓

Data | Invoice

Year 2009 Number 30 Date 8/13/09

Details

		Number of Product	Description of Product	Quantity	Price per unit	Amount
		=	starts	=	=	
		1	Peopleware: Productive Projects and Teams	2	31.00	62.00
		2	Arco iris de lágrimas	1	15.00	15.00

1 There are 2 objects in list

Remarks

Create invoice

We are in Order module

1. The user selects an order

2. Then he clicks on 'Create invoice' button

3. A message confirming the creation of the new invoice is shown

4. The new invoce is shown in the 'Invoice' tab of order

The detail lines of the new invoice are been copied from the current order

Figure 8.1 (reprint) Creating an invoice from an order using an action

Now we're going to write a test to verify that it really works in this way. Add the `testCreateInvoiceFromOrder()` method of listing 8.33 to the `OrderTest` class.

Listing 8.33 The testCreateInvoiceFromOrder() method in OrderTest

```
public void testCreateInvoiceFromOrder() throws Exception {
    // Looking for the order
    searchOrderSusceptibleToBeInvoiced();  // Locates an order
    assertValue("delivered", "true");  // The order is delivered
    int orderDetailsCount = getCollectionRowCount("details");  // Takes note of the
                                                         // details count of the order
    execute("Sections.change", "activeSection=1");  // The section of the invoice
    assertValue("invoice.year", "");  // There is no invoice yet
    assertValue("invoice.number", "");  // in this order

    // Creating the invoice
    execute("Order.createInvoice");  // Executes the action under test (1)
    String invoiceYear = getValue("invoice.year");  // Verifies that now
    assertTrue("Invoice year must have value",     // there is an invoice in
        !Is.emptyString(invoiceYear));            // the invoice tab (2)
    String invoiceNumber = getValue("invoice.number");
    assertTrue("Invoice number must have value",
        !Is.emptyString(invoiceNumber));  // Is.emptyString()  is from org.openxava.util
    assertMessage("Invoice " + invoiceYear + "/" + invoiceNumber +
        " created from current order");  // The confirmation message (3)
    assertCollectionRowCount("invoice.details",  // The newly created invoice
        orderDetailsCount);                 // has the same details count as the order (4)

    // Restoring the order for running the test the next time
    setValue("invoice.year", "");
    assertValue("invoice.number", "");
    assertCollectionRowCount("invoice.details", 0);
    execute("CRUD.save");
    assertNoErrors();
}
```

As you can see, the test clicks the button for executing `Order.createInvoice` action (1), then verifies that an invoice has been created, is displayed in the invoice tab (2), and has the same number of detail lines as the current order (4). The test also verifies that the correct confirmation message is generated (3).

To run this test it's needed to choose an order suitable to be invoiced. This is done in the `searchOrderSusceptibleToBeInvoiced()` method that we are going to examine in the next section.

8.4.2 *Finding an entity for testing using list mode and JPA*

To select an order suitable for our test we'll use JPA to determine the year and number of that order, and then we'll use the list mode to select the order to be edited in detail mode. Listing 8.34 shows the methods to implement this.

Listing 8.34 Methods from OrderTest to search an order using JPA and list mode

```
private void searchOrderSusceptibleToBeInvoiced() throws Exception {
    searchOrderUsingList("o.delivered = true and o.invoice = null");  // Sends
}                       // the condition, in this case to search for a delivered order with no invoice

private void searchOrderUsingList(String condition) throws Exception {
    Order order = findOrder(condition);  // Finds the order with the condition using JPA
    String year = String.valueOf(order.getYear());
    String number = String.valueOf(order.getNumber());
    setConditionValues(new String [] { year, number });  // Fills the year and number
    execute("List.filter");  // and clicks the filter button of the list
    assertListRowCount(1);   // Only one row corresponding to the desired order
    execute("Mode.detailAndFirst");  // To see the order in detail mode
    assertValue("year", year);       // Verifies that the edited order
    assertValue("number", number);   // is the desired one

}

private Order findOrder(String condition) {
    Query query = XPersistence.getManager().createQuery(  // Creates a JPA query
        "from Order o where o.deleted = false and "  // from the condition. Note the
        + condition);                      // deleted = false for excluding deleted orders
    List orders = query.getResultList();
    if (orders.isEmpty()) {   // It's needed at least one order with the condition
        fail("To run this test you must have some order with " + condition);
    }
    return (Order) orders.get(0);
}
```

The `searchOrderSusceptibleToBeInvoiced()` method simply calls a more generic method, `searchOrderUsingList()`, to locate an entity from a condition. The `searchOrderUsingList()` method obtains an `Order` entity by means of `findOrder()`, then it uses the list to filter by year and number of this `Order` before finally going to detail mode. The `findOrder()` method uses plain JPA for searching.

Combining list mode and JPA can be a very useful technique in some cases. We will continue to use the methods `searchOrderUsingList()` and `findOrder()` in the remaining tests.

8.4.3 Testing hiding of the action

We refined the `Order` module in section 8.1.7 to show the action for creating an invoice only when the displayed order would be suitable to be invoiced. Listing 8.35 shows the test method for this.

Listing 8.35 Testing that the action is correctly hidden in OrderTest

```
public void testHidesCreateInvoiceFromOrderWhenNotApplicable()
    throws Exception
{
    searchOrderUsingList(
        "delivered = true and invoice <> null");   // If the order already has an invoice
    assertNoAction("Order.createInvoice");   // it cannot be invoiced again

    execute("Mode.list");

    searchOrderUsingList(
        "delivered = false and invoice = null");   // If the order is not delivered
    assertNoAction("Order.createInvoice");   // it cannot be invoiced

    execute("CRUD.new");   // If the order is not saved yet
    assertNoAction("Order.createInvoice");   // it cannot be invoiced
}
```

We test three cases when the button for creating an invoice should not be visible. Note the usage of `assertNoAction()` for asking if a link or button for an action is not present in the user interface. Here we are reusing the `searchOrderUsingList()` method developed in the previous section.

We have already implicitly tested that the button is present when applicable in our previous test, because `execute()` fails if the action is not in the user interface.

8.4.4 Testing the list mode action

Now we'll test the `Order.createInvoiceFromSelectedOrders` action, the action for creating an invoice from multiple selected orders in list mode. We repeat figure 8.3 of section 8.3 below, showing how this process works.

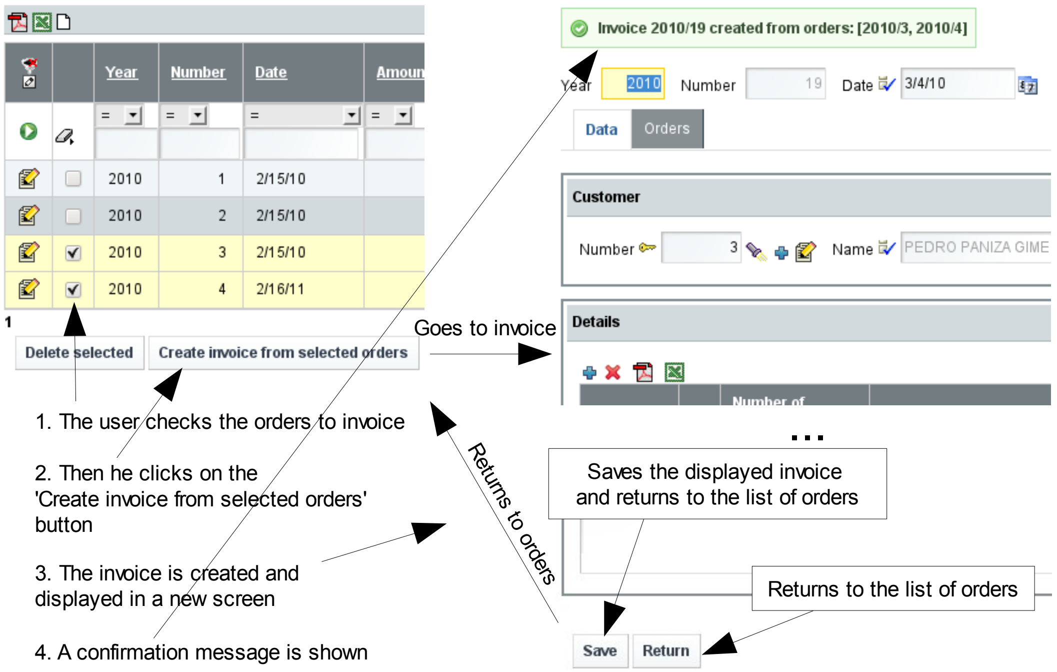

Year
Number
Date
Amoun
2010 1 2/15/10
2010 2 2/15/10
2010 3 2/15/10
2010 4 2/16/11
Delete selected
Create invoice from selected orders
Goes to invoice
1. The user checks the orders to invoice
2. Then he clicks on the 'Create invoice from selected orders' button
3. The invoice is created and displayed in a new screen
4. A confirmation message is shown
Invoice 2010/19 created from orders: [2010/3, 2010/4]
Year 2010
Number 19
Date 3/4/10
Data
Orders
Customer
Number 3
Name PEDRO PANIZA GIME
Details
Number of
...
Returns to orders
Saves the displayed invoice and returns to the list of orders
Returns to the list of orders
Save
Return

Figure 8.3 (reprint) Editing the invoice after creating it from several orders

Let's write a test to verify that it works in just this way. Add the `testCreateInvoiceFromSelectedOrders()` method of listing 8.36 to the `OrderTest` class.

Listing 8.36 The testCreateInvoiceSelectedOrders() method in OrderTest

```
public void testCreateInvoiceFromSelectedOrders() throws Exception {
    assertOrder(2010,  9, 2, 362);  // Order 2010/9 has 2 lines and 362 as base amount
    assertOrder(2010, 10, 1, 126);  // Order 2010/10 has 1 line and 126 as base amount

    execute("List.orderBy", "property=number");  // Sorts the list by number
    checkRow(   // Checks the row from the row number
       getDocumentRowInList("2010",  "9")  // Obtains the row from order year and number
    );   // So, this line checks the order 2010/9 in the list (1)
    checkRow(
       getDocumentRowInList("2010", "10")
    );   // Checks the order 2010/10 in the list (1)

    execute("Order.createInvoiceFromSelectedOrders");   // Executes the action we
```

```
                                                            // are currently testing (2)

    String invoiceYear = getValue("year");    // We are now in detail mode of the
    String invoiceNumber = getValue("number");   // newly  created invoice
    assertMessage("Invoice " + invoiceYear + "/" + invoiceNumber +
       " created from orders: [2010/9, 2010/10]");   // The confirmation message
    assertCollectionRowCount("details", 3);   // Asserts that the line count of the new
                                  // invoice equals the sum of lines from the source orders (3)
    assertValue("baseAmount", "488.00");   // Asserts that base amount of the new
                               // invoice equals the sum of the amounts of the source orders (4)
    execute("Sections.change", "activeSection=1"); // Changes to the orders
                                                  // tab of invoice
    assertCollectionRowCount("orders", 2);// The new invoice has 2 associated orders (5)
    assertValueInCollection("orders", 0, 0, "2010"); // and they should be the correct
    assertValueInCollection("orders", 0, 1, "9");          // ones
    assertValueInCollection("orders", 1, 0, "2010");
    assertValueInCollection("orders", 1, 1, "10");

    assertAction("CurrentInvoiceEdition.save");   // The Save (6)
    assertAction("CurrentInvoiceEdition.return");   // and Return buttons (6)

    checkRowCollection("orders", 0);   // We select the 2 orders
    checkRowCollection("orders", 1);
    execute("Collection.removeSelected",   // and remove them, in order to be able to
       "viewObject=xava_view_section1_orders");// repeat this test using the same orders
    assertNoErrors();

    execute("CurrentInvoiceEdition.return");   // Returns to the orders list (7)
    assertDocumentInList("2010",  "9");   // Asserts that we are really in orders list
    assertDocumentInList("2010", "10");
}
```

This test checks two orders (1) and clicks the 'Create invoice from selected orders' button (2). Then it verifies that a new invoice is created with the correct number of lines (3), base amount (4) and list of orders (5). Furthermore the test verifies that the 'Save' and 'Return' actions are available (6) and uses the latter for returning to the orders list (7).

We use `getDocumentRowInList()` and `assertDocumentInList()`, methods from `CommercialDocumentTest` base class. They were originally defined as private, therefore we must redefine them as protected to use them from `OrderTest`. Edit `CommercialDocumentTest` and make the changes in listing 8.37.

Listing 8.37 Change private to protected in 2 ComercialDocumentTest methods

```
protected ~~private~~ void assertDocumentInList(String year, String number) ...

protected ~~private~~ int getDocumentRowInList(String year, String number) ...
```

The only remaining detail is the `assertOrder()` method that we'll see in the next section.

8.4.5 Asserting test data

In section 3.5 (lesson 3) you learned how to use data existing in the database for your tests. Obviously, if your database is accidentally altered, your test, albeit correct, will not pass. So, asserting the database values before running a test that relies on them is a good practice. In our example we do this by calling `assertOrder()` at the beginning. The contents of `assertOrder()` are displayed in listing 8.38.

Listing 8.38 Method to verify the state of an already existing order

```java
private void assertOrder(
    int year, int number, int detailsCount, int baseAmount)
{
    Order order = findOrder("year = " + year + " and number=" + number);
    assertEquals("To run this test the order " +
       order + " must have " + detailsCount + " details",
       detailsCount, order.getDetails().size());
    assertTrue("To run this test the order " +
       order + " must have " + baseAmount + " as base amount",
       order.getBaseAmount().compareTo(new BigDecimal(baseAmount)) == 0);
}
```

This method finds an order and verifies its details count and base amount. Using this method has the advantage that if the required orders for the test are not in the database with the correct values you get a precise message. Thus, you will not waste time figuring out what is wrong. This is especially useful if the test is not performed by the original developer.

8.4.6 Testing exceptional cases

Given that the action for creating the invoice is hidden if the order is not ready to be invoiced, we cannot test the code from detail mode we wrote in section 8.1.5 for handling exceptional cases. In list mode however, the user still has the option of choosing any order for invoicing. Therefore, we will create the invoice for verifying the correct behavior in exceptional cases from list mode. Listing 8.39 shows the code for `OrderTest`.

Listing 8.39 Asserting exceptional case creating an invoice from order

```java
public void testCreateInvoiceFromOrderExceptions() throws Exception {
    assertCreateInvoiceFromOrderException( // Verifies that when the order already has (1)
       "delivered = true and invoice <> null",// an invoice the correct error is produced
       "Impossible to create invoice: the order already has an invoice"
    );

    assertCreateInvoiceFromOrderException(  // Verifies that when the order is not (2)
       "delivered = false and invoice = null",// delivered the correct error is produced
       "Impossible to create invoice: the order is not delivered yet"
    );
}
```

```
private void assertCreateInvoiceFromOrderException(
    String condition, String message) throws Exception
{
    Order order = findOrder(condition);   // Finds an order satisfying the condition (3)
    int row = getDocumentRowInList(   // and obtains the row number for that order (4)
       String.valueOf(order.getYear()),
       String.valueOf(order.getNumber())
    );
    checkRow(row);  // Checks the row (5)
    execute("Order.createInvoiceFromSelectedOrders");  // Tries to create the invoice(6)
    assertError(message);  // Is the expected message shown? (7)
    uncheckRow(row);  // Uncheck the row so we can call this method again
}
```

The test verifies that the message is the correct one when trying to create an invoice from an order that already has an invoice (1), and also from an order not delivered yet (2). To do these verifications it calls the method `assertCreateInvoiceFromOrderException()`. This method finds an `Order` entity using the condition (3), locates the row where the entity is displayed (4) and checks it (5). Afterward, the test executes the action (6) and asserts that the expected message is shown (7).

8.5 Summary

The salt of your application comes from the actions and entity methods. Thanks to them you can convert a simple data management application into a useful tool. In this lesson, for example, we provided the user with a way to automatically create invoices from orders.

You have learned how to create instance and static methods for business logic, and how to call them from actions in detail and list mode. Along the way you also saw how to hide and show actions, use exceptions, validating from actions, change to another module and how to test all this.

We still have many interesting things to learn, in the next lesson for example we are going to refine the behavior of references and collections.