Refining the standard behavior

lesson 7

I hope you are very happy with the current code of your Invoicing application. It's really simple, you basically have entities, simple classes that model your problem. All the business logic is attached to those entities, and OpenXava generates an application with a decent behavior around them.

However, man shall not live by business logic alone. A good behavior is desirable too. Inevitably, you will face that either you, or your user, crave behavior deviating from, or adding to, the standard OpenXava behavior for certain parts of your application. To make sure your application is comfortable and intuitive for the end user refinement of the standard behavior is needed.

Application behavior is defined by the controllers. A controller is simply a collection of actions, and an action is what executes when a user clicks a link or button. You can define your own controllers and actions and associate them to modules or entities, and in that way refine how OpenXava behaves.

In this lesson we'll refine the standard controllers and actions in order to customize the behavior of your Invoicing application.

7.1 Custom actions

By default, an OpenXava module allows you to manage your entity in a way powerful and adequate enough to perform the most common tasks: adding, modifying, removing, searching, generating PDF reports and exporting to Excel. These default actions are contained in the `Typical` controller. You can refine or enhance the behavior of your module by defining your own controller. This section will teach you how to define your own controller and write custom actions.

7.1.1 Typical controller

By default the `Invoice` module uses the actions from `Typical` controller. This controller is defined in the *default-controllers.xml* located in the *OpenXava/xava* folder of your workspace. A controller definition is an XML fragment with a relation of actions. OpenXava applies the `Typical` controller by default to all modules. You can see its definition in listing 7.1.

Listing 7.1 Definition of the "Typical" controller in default-controllers.xml

```
<controller name="Typical">                  <!-- "Typical" inherits all actions -->
    <extends controller="Navigation"/>       <!-- from "Navigation", -->
    <extends controller="CRUD"/>             <!-- from "CRUD"  -->
    <extends controller="Print"/>            <!-- and from "Print" controllers -->
</controller>
```

Here you can see how a controller can be defined from other controllers. This

is controllers inheritance. In this case the `Typical` controller gets all the actions from the `Navigation`, `Print` and `CRUD` controllers. `Navigation` contains the action to navigate for object in detail mode. `Print` contains the actions for printing PDF reports as well as exporting to Excel. `CRUD` is composed of the actions for creating, reading, updating and deleting. An extract of the `CRUD` controllers is shown in listing 7.2.

Listing 7.2 Definition of the "CRUD" controller in default-controllers.xml

```
<controller name="CRUD">

    <action name="new"
       class="org.openxava.actions.NewAction"
       image="images/new.gif"
       keystroke="Control N">
       <!--
       name="new" : Name to reference the action from other parts of the application
       class="org.openxava.actions.NewAction": Class with action logic
       image="images/new.gif": Image to show for this action
       keystroke="Control N": Keys that the user can press to execute this action
       -->

       <set property="restoreModel"
          value="true"/>  <!-- The restoreModel property of the action class will
                              be set to true before it's executed -->
    </action>


    <action name="save"
       mode="detail"
       by-default="if-possible"
       class="org.openxava.actions.SaveAction"
       image="images/save.gif"
       keystroke="Control S"/>
       <!--
       mode="detail": This action will be shown only in detail mode
       by-default="if-possible": This action will be executed when the user press ENTER
       -->


    <action name="delete" mode="detail"
       confirm="true"
       class="org.openxava.actions.DeleteAction"
       image="images/delete.gif"
       keystroke="Control D"/>
       <!-- confirm="true": Ask the user for confirmation before executing the action -->

    ...

</controller>
```

Here you see how actions are defined. Basically it amounts to linking a name with a class holding the logic to execute. Moreover, it defines an image and a keystroke, and using `<set />` you can configure your action class.

The actions are shown by default in both list and detail mode, although you

can, by means of `mode` attribute, specify that the action should be shown only in "list" or "detail" mode.

7.1.2 Refining the controller for a module

To start of, we will refine the delete action of the `Invoice` module. Our objective is to modify the delete procedure so that when a user clicks on the delete button the invoice will not be removed but instead simply marked as removed. This way, we can recover the deleted invoices if needed.

Figure 7.1 shows the actions from `Typical`. We want all of these actions in our `Invoice` module, except that we're going to write our own logic for the delete action.

In order to define your own controller for `Invoice` you have to create a file named *controllers.xml* in the *xava* folder of your project. Then add the code in listing 7.3 to this file.

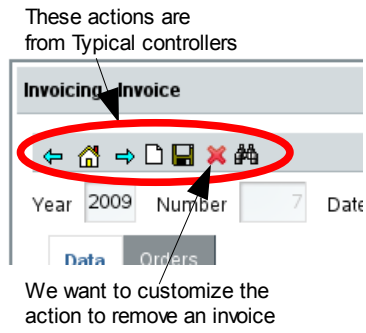

Figure 7.1 Typical actions

Listing 7.3 The controllers.xml file with Order controller definition

```
<?xml version = "1.0" encoding = "ISO-8859-1"?>

<!DOCTYPE controllers SYSTEM "dtds/controllers.dtd">

<controllers>

    <controller name="Invoice">  <!-- The same name as the entity -->

       <extends controller="Typical"/>  <!-- It has all the actions from Typical -->

       <!-- Typical already has a 'delete' action, by using the same name we override it -->
       <action name="delete"
          mode="detail" confirm="true"
          class="org.openxava.invoicing.actions.DeleteInvoiceAction"
          image="images/delete.gif"
          keystroke="Control D"/>

    </controller>

</controllers>
```

In order to define a controller for your entity, you have to create a controller with the same name as the entity, i.e., if a controller named "Invoice" exists, this controller will be used instead of `Typical` when you run the `Invoice` module.

The `Invoice` controller extends `Typical`, thus all actions from `Typical` are available for your `Invoice` module. Any action you define in your `Invoice` controller will be a button available for the user to click. But since we named our

action “delete”, which is shared with an action of the `Typical` controller, we override the action defined in `Typical`. In other words, only our new delete action will be present in the user interface.

7.1.3 Writing your own action

Let's write a first version of our delete action. See listing 7.4.

Listing 7.4 First mock version of DeleteInvoiceAction

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

import org.openxava.actions.*;  // For using ViewBaseAction

public class DeleteInvoiceAction
    extends ViewBaseAction {  // ViewBaseAction has getView(), addMessage(), etc

    public void execute() throws Exception {  // The logic of the action
       addMessage(  // Add a message to show to the user
          "Don't worry! I have cleared only the screen");
       getView().clear();  // getView() returns the xava_view object
                           // clear() clears the data in user interace
    }

}
```

An action is a simple class. It has an `execute()` method with the logic to perform when users click on the corresponding button or link. An action must implement `org.openxava.actions.IAction` interface, though usually it is more practical to extend `BaseAction`, `ViewBaseAction` or any other base action from the `org.openxava.actions` package.

`ViewBaseAction` has a `view` property that you can use from inside `execute()` with `getView()`. This object of type `org.openxava.view.View` allows you manage the user interface. Above we clear the displayed data using `getView().clear()`.

We also use `addMessage()`. All the messages added by `addMessage()` will be shown to the user at the end of action execution. You can either add the message to show or an id of an entry in *i18n/Invoicing-messages_en.properties*.

Figure 7.2 demonstrates the behavior of the `Invoice` module after adding your custom delete action. This is of course mock behavior, so let us now add the real functionality.

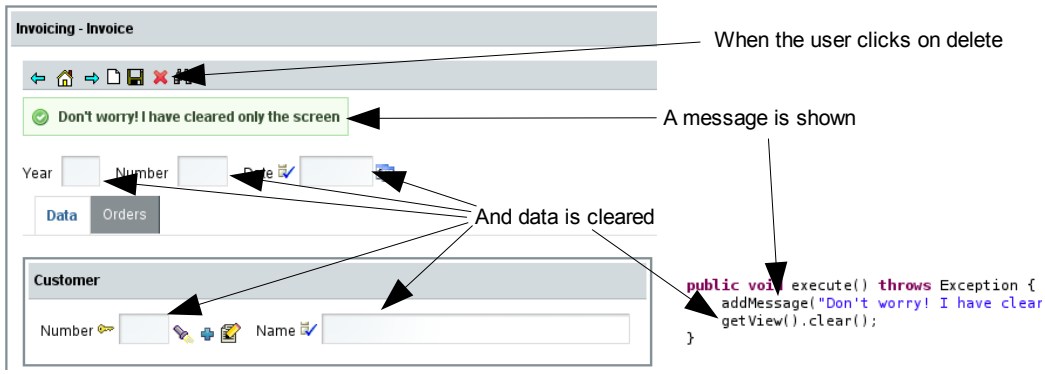

Figure 7.2 Behavior of the custom delete action

In order to mark the current invoice as deleted without actually deleting it, we need to add a new property to `Invoice`. Let's call it `deleted`. You can see it in listing 7.5.

Listing 7.5 New 'deleted' property in Invoice

```
public class Invoice extends CommercialDocument {

    @Hidden  // It will not be shown by default in views and tabs
    private boolean deleted;

    public boolean isDeleted() {
       return deleted;
    }

    public void setDeleted(boolean deleted) {
       this.deleted = deleted;
    }

    ...
}
```

As you see, it's a plain boolean property. The only new detail is the use of the `@Hidden` annotation. It means that when a default view or a tabular list is generated the `deleted` property will not be shown; although if you explicitly put it in `@View(members=)` or `@Tab(properties=)` it will be shown. Use this annotation for properties intended for internal use of the developer but makes no sense to the end user.

Now update the database schema. Then execute the SQL statement in listing 7.6 against the database. You can use the Database perspective in Eclipse to do it (see lesson 1).

Listing 7.6 Set to false the 'deleted' column of CommercialDocument table

```
update CommercialDocument
set deleted = false
```

Remember that the `Invoice` data is stored in the CommercialDocument table.

We are now ready to write the real code for your action. See it in listing 7.7.

Listing 7.7 The implementation of execute() method of DeleteInvoiceAction

```
public void execute() throws Exception {
    Invoice invoice = XPersistence.getManager().find(
       Invoice.class,
       getView().getValue("oid"));   // We read the id from the view
    invoice.setDeleted(true);   // We modify the entity state
    addMessage("object_deleted", "Invoice");   // The "deleted" message
    getView().clear();   // The view is cleared
}
```

The visual effect is the same, a message is shown and the view cleared, but in this case we actually do some work. We find the `Invoice` entity associated with the current view and modify its `deleted` property. That is all you have to do, because OpenXava automatically commits the JPA transaction. This means that you can read any object and modify its state from an action, and when the action finishes the modified values will be stored in the database.

But we left a loose end here. If the user clicks on the delete button when no object is selected then the find instruction fails and a somewhat technical and unintelligible message is shown to our helpless user. You can refine it with a simple "if ". Note the small modification of the `execute()` method in listing 7.8.

Listing 7.8 Refinement for the case of no object currently in view

```
public void execute() throws Exception {
    if (getView().getValue("oid") == null) {   // Is there an object in view?
       addError("no_delete_not_exists");
       return;
    }
    ...
}
```

If the `oid` property has no value in the view then the view is not displaying an existing object in the database, so there is nothing to remove. In this case, we just show an error message. You do not need to add "no_delete_not_exists" to your messages file, because it exists in the message files included with OpenXava. Have a look at *OpenXava/i18n/Messages_en.properties* to see the built-in OpenXava messages.

Now that you know how to write your own custom actions, it's time to learn how to write generic code.

7.2 Generic actions

The above code for your `DeleteInvoiceAction` reflects the typical way of

writing actions. It is concrete code directly accessing and manipulating your entities.

But sometimes you have action logic potentially to be used and reused across your application, even across all your applications. In this case, you can use techniques to produce reusable code, converting your custom actions into generic actions.

Let's learn these techniques to write generic action code.

7.2.1 MapFacade for generic code

Imagine that you want to use your `DeleteInvoiceAction` for `Order` entities too. Moreover, imagine that you want to use it for all the entities of the application having a `deleted` property. That is, you want an action to mark as deleted instead of actually removing from the database any entity, not just invoices. In this case, the current code for your action is not enough. You need a more generic code.

You can develop more generic actions using an OpenXava class named `MapFacade`. `MapFacade` (from `org.openxava.model` package) allows you to manage the state of your entities using maps, which is very practical since `View` works with maps. Furthermore, maps are more dynamic than objects, and hence more suitable for generic code.

Let's rewrite our delete action. First, we'll rename it from `DeleteInvoiceAction` (an action to delete `Invoice` objects) to `InvoicingDeleteAction` (the delete action of the Invoicing application). This implies that you must alter the class name entry for the action in *controllers.xml*, as shown in listing 7.9.

Listing 7.9 Class changed to InvoicingDeleteAction in controllers.xml

```
<action name="delete" mode="detail" confirm="true"
    class="org.openxava.invoicing.actions.DeleteInvoiceAction"
    class="org.openxava.invoicing.actions.InvoicingDeleteAction"
    image="images/delete.gif"
    keystroke="Control D"/>
```

Now, rename your `DeleteInvoiceAction` to `InvoicingDeleteAction` and rewrite its code leaving it as in list 10.10.

Listing 7.10 InvoicingDeleteAction with generic code using MapFacade

```
public class InvoicingDeleteAction extends ViewBaseAction {

    public void execute() throws Exception {
       if (getView()
          .getKeyValuesWithValue()   // Instead of getValue("oid") we use
```

```
            .isEmpty())                    // the more generic getKeyValuesWithValue()
        {
            addError("no_delete_not_exists");
            return;
        }
        Map values = new HashMap();   // The values to modify in the entity
        values.put("deleted", true);   // We set true to deleted property
        MapFacade.setValues(   // Modifies the values of the indicated entity
            getModelName(),   // A method from ViewBaseAction
            getView().getKeyValues(),   // The key of the entity to modify
            values);   // The values to change
        resetDescriptionsCache();   // Clears the caches for combos
        addMessage("object_deleted", getModelName());
        getView().clear();
        getView().setEditable(false);   // The view is left as not editable
    }

}
```

This action executes the same logic as the previous one, but it contains no reference to the `Invoice` entity. This action is generic and you can use it with `Order`, `Author` or any other entity as long they have a `deleted` property. The trick lies in `MapFacade`. It allows you to modify an entity from maps. These maps can be obtained directly from the view (using `getView().getKeyValues()` for example) or created generically as in the case with `values` above.

Additionally you can see two small improvements over the old version. First, we call `resetDescriptionsCache()`, a method from `BaseAction`. This method clears the cache used for combos. When you modify an entity, this is needed if you want the combos to reflect the change instantaneously. Second, we call `getView().setEditable(false)`. This disables the editors of the view, to prevent the user from filling in data in the view. In order to create a new entity the user must click the "new" button.

Now your action is ready to use with any other entity. We could copy and paste the `Invoice` controller as `Order` controller in *controllers.xml*. In this way, our new generic delete logic would be used for `Order` too. Wait a minute! Did I say "copy and paste"? We don't want to rot in hell, right? So we'll use a more automatic way to apply our new action to all modules. Let us find out how in the next section.

7.2.2 Changing the default controller for all modules

If you use the `InvoicingDeleteAction` just for `Invoice` then defining it in the `Invoice` controller in *controllers.xml* is a good way to go. But, we improved this action with the sole purpose making it reusable, so let's reuse it. For this we will assign a controller to all modules at once.

The first step is to change the name of the controller from `Invoice` to `Invoicing`. Listing 7.11 shows the renamed controller in *controllers.xml*.

Listing 7.11 Invoicing controller is the Invoice controller renamed

```
<controller name="Invoicing">
<controller name="Invoice">

    <extends controller="Typical"/>

    <action name="delete" mode="detail" confirm="true"
       class=
       "org.openxava.invoicing.actions.InvoicingDeleteAction"
       image="images/delete.gif"
       keystroke="Control D">
       <use-object name="xava_view"/>
    </action>

</controller>
```

As you already know, when you use the name of an entity, like `Invoice`, as controller name, that controller will be used by default for the module of that entity. Therefore, if we change the name of the controller, this controller will not be used for the entity. Indeed, the `Invoicing` controller is not used by any module since there is no entity called `Invoicing`.

We want to make the `Invoicing` controller the default controller for all the modules in our application. To achieve this we need to modify the *application.xml* file located in the *xava* folder of your application. Leave it as the one in listing 7.12.

Listing 7.12 The application.xml file with Invoicing as the default module

```
<?xml version = "1.0" encoding = "ISO-8859-1"?>

<!DOCTYPE application SYSTEM "dtds/application.dtd">

<application name="Invoicing">

    <!--
    A default module for each entity is assumed with the
    controllers on <default-module/>
    -->

    <default-module>
       <controller name="Invoicing"/>
    </default-module>

</application>
```

In this simple way all modules of your application will use `Invoicing` instead of `Typical` as the default controller. Try to execute your `Invoice` module and see how the `InvoicingDeleteAction` is executed on deletion of an element.

You can try the `Order` module too, but it will not work because it has no

deleted property. We could add the deleted property to Order and it would work with our new controller, but instead of "copying and pasting" the deleted property to all our entities we are going to use a better technique. Let's see it in the next section.

7.2.3 Come back to the model a little

Now your task is to add the deleted property to all the entities of your application, in order to make InvoicingDeleteAction work. This is a good occasion to use inheritance to put this shared code in the same place, instead of using the infamous "copy & paste".

First remove the deleted property from Invoice as shows listing 7.13.

Listing 7.13 The 'deleted' property is removed from Invoice

```
public class Invoice extends CommercialDocument {
    ...
    @Hidden
    private boolean deleted;

    public boolean isDeleted() {
       return deleted;
    }

    public void setDeleted(boolean deleted) {
       this.deleted = deleted;
    }
    ...
}
```

And now create a new mapped superclass named Deletable. Listing 7.14. shows its code.

Listing 7.14 Deletable mapped superclass with the common deleted property

```
@MappedSuperclass
public class Deletable extends Identifiable {

    @Hidden
    private boolean deleted;

    public boolean isDeleted() {
       return deleted;
    }

    public void setDeleted(boolean deleted) {
       this.deleted = deleted;
    }

}
```

Deletable is a mapped superclass. Remember, a mapped superclass is not an entity, but a class with properties, methods and mapping annotations to be used as

a superclass for entities. `Deletable` extends from `Identifiable`, so any entity that extends `Deletable` will have the `oid` and `deleted` properties.

Now you can convert any of your current entities to deletable, you only have to change `Identifiable` to `Deletable` as superclass. Listing 7.15 shows how this is done for `CommercialDocument`.

Listing 7.15 CommercialDocument now extends from Deletable

```
abstract public class CommercialDocument extends Deletable {
abstract public class CommercialDocument extends Identifiable {
...
}
```

Given that `Invoice` and `Order` are subclasses of `ComercialDocument`, now you can use the `Invocing` controller with the `InvoicingDeleteAction` against them.

A subtle detail remains. The `Order` entity has a `@PreRemove` method to do a validation on remove. This validation can prevent the removal. We can maintain this validation for our custom deletion by just overwriting the `setDeleted()` method for `Order`, as shown in listing 7.16.

Listing 7.16 Calling explicitly the @PreRemove method from setDeleted()

```
public class Order extends CommercialDocument {

    ...

    @PreRemove
    private void validateOnRemove() {   // Now this method is not executed automatically
       if (invoice != null) {           // since a real deleletion is not done
          throw new IllegalStateException(
             XavaResources.getString(
                "cannot_delete_order_with_invoice"));
       }
    }

    public void setDeleted(boolean deleted) {
       if (deleted) validateOnRemove();  // We call the validation explicitly
       super.setDeleted(deleted);
    }

}
```

With this change the validation works just as in the real deletion case, thus we preserve untouched the original behavior.

7.2.4 Metadata for more generic code

With your current code `Invoice` and `Order` work fine. Though if you try to remove an entity in any other module, you'll get an ugly error message. Figure 7.3 shows what happens when you try to delete a `Customer`.

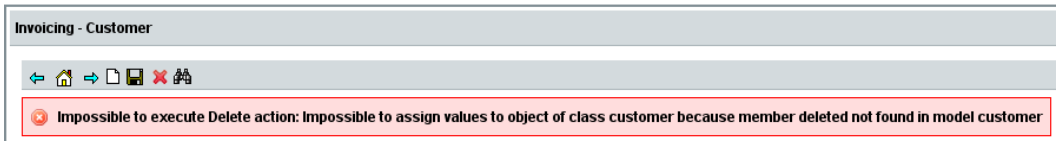

Figure 7.3 Error message trying delete an entity with no deleted property

If your entity has no `deleted` property, the delete action fails. Using `Deletable` as base class you can add the `deleted` property to all your entities. However, you might want to have entities marked as deleted (so `Deletable`) and entities to be actually removed from the database. We want the action to work in all cases.

OpenXava stores metadata for all entities, and you can access this metadata from your code. This allows you, among other things, to figure out if an entity has a `deleted` property.

Listing 7.17 shows how to modify the action to find out if the entity has a `deleted` property, if not, the remove process is not performed.

Listing 7.17 Using metadata to find out if a 'deleted' property exists

```
public void execute() throws Exception {
    if (getView().getKeyValuesWithValue().isEmpty()) {
       addError("no_delete_not_exists");
       return;
    }

    if (!getView().getMetaModel()  // Metadata about the current entity
       .containsMetaProperty("deleted"))  // Is there a deleted property?
    {
       addMessage(  // For now, only shows a message if 'deleted' is not present
          "Not deleted, it has no deleted property");
       return;
    }

    Map values = new HashMap();
    values.put("deleted", true);
    MapFacade.setValues(
       getModelName(),
       getView().getKeyValues(),
       values);
    resetDescriptionsCache();
    addMessage("object_deleted", getModelName());
    getView().clear();
    getView().setEditable(false);
}
```

The key thing here is `getView().getMetaModel()` that returns a `MetaModel` object from `org.openxava.model.meta` package. This object contains metadata about the entity currently displayed in the view. You can ask for the properties, references, collections, methods and other entity metadata. Consult the `MetaModel` API to learn more. In this case we ask if the `deleted` property exists.

For now we only show a message. Let's improve on this to actually delete the entity.

7.2.5 Chaining actions

Our objective is to remove the entity in the usual way if it lacks the `deleted` property. Our first option is to write the removal logic ourself, a rather simple task. Nonetheless, it is a much better idea to use the standard removal logic of OpenXava, since we don't need to write any deletion logic and we use a more refined and tested piece of code.

For this OpenXava provides the useful feature of chaining actions. This means that after your action another action will be executed. This is as easy as implementing the `IChainAction` interface in your class. Listing 7.18 shows `InvoicingDeleteAction` modified to chain to the standard OpenXava delete action.

Listing 7.18 InvoicingDeleteAction chains to standard OpenXava delete action

```
public class InvoicingDeleteAction extends ViewBaseAction
    implements IChainAction {  // It chains to another action,
                               // returned by getNextAction() method
    private String nextAction = null;   // To store the next action to execute

    public void execute() throws Exception {
       if (getView().getKeyValuesWithValue().isEmpty()) {
          addError("no_delete_not_exists");
          return;
       }

       if (!getView().getMetaModel()
          .containsMetaProperty("deleted"))
       {
          nextAction = "CRUD.delete";   // "CRUD.delete" will be
          return;                       // executed after this action finishes
       }

       Map values = new HashMap();
       values.put("deleted", true);
       MapFacade.setValues(
          getModelName(),
          getView().getKeyValues(),
          values);
       resetDescriptionsCache();
       addMessage("object_deleted", getModelName());
       getView().clear();
       getView().setEditable(false);
    }

    public String getNextAction()   // Required because of IChainAction
       throws Exception
    {
       return nextAction;   // If null no action will be chained
    }
```

```
}
```

Simple. We only return "CRUD.delete" in `getNextAction()` if we want the default delete action of OpenXava to be executed. Thus, we write our custom removal logic (in this case marking a property as true) for some cases, and we "by-pass" to the standard logic for all other cases.

Now you can use your `InvoicingDeleteAction` with any entity. If the entity has a `deleted` property it will be marked as deleted, otherwise it will be actually deleted from the database.

This example shows you how to use `IChainAction` to refine the standard OpenXava logic. Another way to do so is by means of inheritance. Let's see how in the next section.

7.2.6 Refining the default search action

Now, your `InvoiceDeleteAction` works fine, though it is useless. It is useless to mark objects as deleted if the rest of the application is not aware of that. In other words, we have to modify other parts of the application to treat the marking-as-deleted object as if they do not exist.

The most obvious place to start is the search action. If you delete an invoice and then search for it, it should not be found. Figure 7.4 shows how searching works in OpenXava.

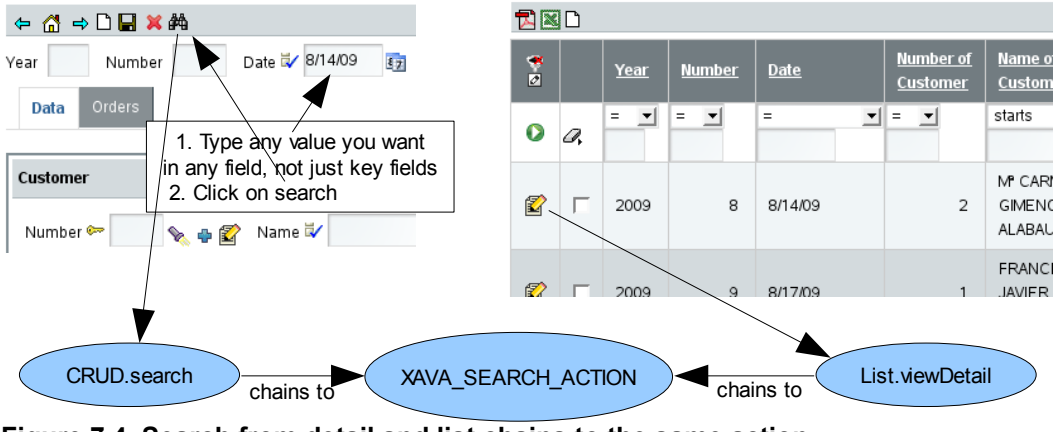

Figure 7.4 Search from detail and list chains to the same action

The first thing to observe in figure 7.4 is that searching in detail mode is more flexible than it seems. The user can enter any value in any field, or set of fields, and then click the search button. The first matching object will get loaded in the

view.

Now you might think: "Well, I can refine the `CRUD.search` action just in the same was as I refined `CRUD.delete`". Yes, you can, and it will work; when the user clicks on the search action of detail mode your code will be executed. However, there is a subtle detail here. The search logic is not called only from the detail view, but from several other places in an OpenXava module. For example, when the user chooses a detail from list mode, the `List.viewDetail` action takes the key from the row, puts it in the detail view, and then executes the search action.

We put the search logic for a module in one action, and all actions that need to search will chain to this action. Just as shown in the above figure 7.4.

This gets clearer when you see the code for the standard `CRUD.search` action, `org.openxava.actions.SearchAction` whose code is displayed in listing 7.19.

Listing 7.19 Standard action for searching in detail mode (CRUD.seach)

```
public class SearchAction extends BaseAction
    implements IChainAction {  // It chains to another action

    public void execute() throws Exception {  // Do nothing
    }

    public String getNextAction() throws Exception {
       return getEnvironment()  // To access an environment variables
          .getValue("XAVA_SEARCH_ACTION");
    }

}
```

As you can see, the standard search action for detail mode does nothing except redirect to another action. This other action is defined in an environment variable called XAVA_SEARCH_ACTION, and can be read using `getEnvironment()`. Therefore, if you want to refine the OpenXava search logic, the best way to do it is to define your action as XAVA_SEARCH_ACTION. So, let's do it this way.

To set the environment variable, edit the *controllers.xml* file in the *xava* folder of your project, and add the `<env-var />` line from listing 7.20 at the beginning.

Listing 7.20 Environment variable definition in controllers.xml

```
...
<controllers>

    <!-- To define the global value for an environment variable -->
    <env-var
       name="XAVA_SEARCH_ACTION"
       value="Invoicing.searchExcludingDeleted"/>

    <controller name="Invoicing">
...
```

This way the value of the XAVA_SEARCH_ACTION environment variable in any module will be "Invoicing.searchExcludingDeleted", and hence, the search logic for all modules will be in this action.

Next, we define this action in *controllers.xml*, shown in listing 7.21.

Listing 7.21 Definition of the search action for Invoicing

```
<controller name="Invoicing">

    ...

    <action name="searchExcludingDeleted"
       hidden="true"
       class="org.openxava.invoicing.actions.SearchExcludingDeletedAction"/>
       <!-- hidden="true": Thus the action will not be shown in button bar -->

</controller>
```

And now it's time to write the implementation class. In this case we only want to refine the search logic so that the search is done the regular way with the exception that it is leaving out entities where `deleted` is set to true. To do this refinement we are going to use inheritance. Listing 7.22 shows the action code.

Listing 7.22 Action with search logic to exclude the objects marked as deleted

```
public class SearchExcludingDeletedAction
    extends SearchByViewKeyAction {  // The standard OpenXava action to search

    private boolean isDeletable() {  // To see if this entity has a deleted property
       return getView().getMetaModel()
          .containsMetaProperty("deleted");
    }

    protected Map getValuesFromView()   // Gets the values displayed in this view
       throws Exception                 // These values are used as keys for the search
    {
       if (!isDeletable()) {  // If not deletable we run the standard logic
          return super.getValuesFromView();
       }
       Map values = super.getValuesFromView();
       values.put("deleted", false);  // We populate deleted property with false
       return values;
    }

    protected Map getMemberNames()  // The members to be read from the entity
       throws Exception
    {
       if (!isDeletable()) {  // If not deletable we run the standard logic
          return super.getMemberNames();
       }
       Map members = super.getMemberNames();
       members.put("deleted", null);    // We want to get the deleted property from entity,
       return members;                  // though it might not be in the view
    }
```

```
    protected void setValuesToView(Map values)  // Assigns the values from the
       throws Exception                         // entity to the view
    {
       if (isDeletable() &&  // If it has an deleted property and
          (Boolean) values.get("deleted")) {  // it is true
          throw new ObjectNotFoundException();  // The same exception OpenXava
       }                                        // throws when the object is not found
       else {
          super.setValuesToView(values);  // Otherwise we run the standard logic
       }
    }

}
```

The standard logic to search resides in the `SearchByViewKeyAction` class. Basically, what this action does is that it reads the values from the view and searches for an object. If the id property is present this is used, otherwise all values are used, and the first object that matches the condition is returned. We want to use this same algorithm changing only some details about the `deleted` property. So instead of overwriting the `execute()` method, that contains the search logic, we overwrite three protected methods, that are called from `execute()` and contain some parts suitable to be refined.

After these changes you can try your application, and you will find that when you search, a removed invoice or order will never be shown. Even if you choose a deleted invoice or order from list mode an error will be produced and you will not see the data in detail mode.

You have seen how defining a XAVA_SEARCH_ACTION environment variable in *controllers.xml* you establish the search logic in a global way for all your modules at once. If you want to define a specific search action for a particular module just define the environment variable in the module definition in *application.xml*, as demonstrated in listing 7.23.

Listing 7.23 Environment variable definition at module level in application.xml

```
<module name="Product">
    <!-- To give a local value to the environment variable for this module -->
    <env-var
       name="XAVA_SEARCH_ACTION"
       value="Product.searchByNumber"/>
    <model name="Product"/>
    <controller name="Product"/>
    <controller name="Invoicing"/>
</module>
```

In this way for the module `Product` the XAVA_SEARCH_ACTION environment variable will be "Product.searchByNumber". This way, the environment variables are local to the modules. You can define a default value in *controllers.xml,* but you always have the option to overwrite the value for

particular modules. The environment variables are a powerful way to configure your application in an declarative way.

We don't want a special way to search for `Product`, so don't add this module definition to your *application.xml*. The code here is only to illustrate how to use `<env-var />` in modules.

7.3 List mode

We have almost completed the job. When the user removes an entity with a `deleted` property the entity is marked as deleted instead of actually being removed from the database. And if the user attempts to search for a marked-as-deleted entity he cannot view it in detail mode. However, the user can still see such entities in list mode though, and even worse, if he deletes entities when in list mode, they are actually removed from the database. Let's fix these remaining details.

7.3.1 Filtering tabular data

Only entities where the `deleted` property is false should be shown in list mode. This is very easy to achieve using the `@Tab` annotation. This annotation allows you to define the way the tabular data (the data shown in list mode) is displayed, moreover it allows you to define a condition. So, adding this annotation to the entities with `deleted` property is sufficient to achieve our goal, just as shown in listing 7.24.

Listing 7.24 Condition to exclude the marked as deleted entities using @Tab

```
@Tab(baseCondition = "deleted = false")
public class Invoice extends CommercialDocument { … }

@Tab(baseCondition = "deleted = false")
public class Order extends CommercialDocument { … }
```

And in this simple way the list mode will not show the marked-as-deleted entities.

7.3.2 List actions

The only remaining detail is that when removing entities from list mode, they should be marked as deleted if applicable. We are going to refine the standard `CRUD.deleteSelected` action in the same way we used for `CRUD.delete`.

First, we overwrite the `deleteSelected` and `deleteRow` actions for our application. Add the action definition in listing 7.25 to your `Invoicing` controller defined in *controllers.xml*.

Listing 7.25 Definition of our custom deleteSelected action in controllers.xml

```
<controller name="Invoicing">
    <extends controller="Typical"/>
    ...
    <action name="deleteSelected" mode="list" confirm="true"
       class="org.openxava.invoicing.actions.InvoicingDeleteSelectedAction"
       keystroke="Control D"/>

    <action name="deleteRow" mode="NONE" confirm="true"
       class="org.openxava.invoicing.actions.InvoicingDeleteSelectedAction"
       image="images/delete.gif"
       in-each-row="true"/>

</controller>
```

The standard actions to remove entities from list mode are `deleteSelected` (to delete the checked rows) and `deleteRow` (the action displayed in each row). These actions are defined in the `CRUD` controller. `Typical` extends `CRUD`, and `Invoicing` extends `Typical`; so `Invoicing` controller includes these actions by default. Given we have defined these actions with the same name, our actions overwrite the standard ones. That is, from now on the logic for removing selected rows in list mode is in the `InvoicingDeleteSelectedAction` class. Note how the logic for both actions is in the same Java class. Listing 7.26 shows its code.

Listing 7.26 Action with the custom logic to remove entities from list mode

```
public class InvoicingDeleteSelectedAction
    extends TabBaseAction  // To work with tabular data (list) by means of getTab()
    implements IChainAction {  // It chains to another action,
                               // returned by getNextAction() method
    private String nextAction = null;  // To store the next action to execute

    public void execute() throws Exception {
       if (!getMetaModel().containsMetaProperty("deleted")) {
          nextAction="CRUD.deleteSelected";   // "CRUD.deleteSelected"  will be
                                              // executed after this action is finished
          return;
       }
       markSelectedEntitiesAsDeleted();    // The logic to mark the selected rows
    }                                      // as deleted objects

    private MetaModel getMetaModel() {
       return MetaModel.get(getTab().getModelName());
    }

    public String getNextAction()  // Required because of IChainAction
       throws Exception
    {
       return nextAction;  // If null no action will be chained
    }

    private void markSelectedEntitiesAsDeleted() throws Exception { … }

}
```

You can see that this action is very similar to `InvoicingDeleteAction`. If the selected entities don't have the `deleted` property it chains to the standard action, otherwise it executes its own logic to delete the entities. Usually the actions for list mode extend `TabBaseAction`, thus you can use `getTab()` to obtain the `Tab` objects associated to the list. A `Tab` (from `org.openxava.tab`) allows you to manage the tabular data. For example in the `getMetaModel()` method we retrieve the model name from the `Tab` in order to obtain the corresponding `MetaModel`.

If the entity has a `deleted` property then our custom delete logic is executed. This logic is in the `markSelectedEntitiesAsDeleted()` method you can see in listing 7.27.

Listing 7.27 Logic to mark as deleted the entities from list mode

```
private void markSelectedEntitiesAsDeleted() throws Exception {
    Map values = new HashMap();  // Values to assign to each entity to be marked
    values.put("deleted", true);  // Just set deleted to true
    for (int row: getSelected()) {  // Loop over all selected rows
       Map key = (Map) getTab().getTableModel().getObjectAt(row);
       try {
          MapFacade.setValues(  // Each entity is modified
             getTab().getModelName(),
             key,
             values);
       }
       catch (ValidationException ex) {  // If there is a ValidationException..
          addError("no_delete_row", row + 1, key);
          addErrors(ex.getErrors());  // ...we show the messages
       }
       catch (Exception ex) {  // If any other exception is thrown, a generic
          addError("no_delete_row", row + 1, key);  // message is added
       }
    }
    getTab().deselectAll();  // After removing we deselect the rows
    resetDescriptionsCache();  // And reset the cache for combos for this user
}
```

As you see the logic is a plain loop over all selected rows, in each iteration we set the `deleted` property to true using `MapFacade.setValues()`. Exceptions are caught inside the loop, so if there is a problem deleting one of the entities, this does not affect the removal of the other entities. Furthermore, we have added a small refinement for the `ValidationException` case, adding the validation errors (`ex.getErrors()`) to the errors to be shown to the user.

Here we also deselect all rows by means of `getTab().deselectAll()`, because we are removing rows. If we don't remove the selection, it would be moved after the action execution.

We call `resetDescriptionsCache()` to update all combos in the current user session and remove all deleted entities. The combos (references marked with `@DescriptionsList`) use the `@Tab` of the referenced entity to filter the data. In

our case all the combos for invoices and orders use the "`deleted = false`" condition of the `@Tab`. So, it's needed to reset the combos cache.

Now you have thoroughly improved the way your application deletes the entities. Nevertheless, there still remains a few interesting things we can do with this.

7.4 Reusing actions code

Fantastic! Now your application marks the invoices and orders as deleted instead of actually removing them. One advantage of this approach is that in any given moment, the user can restore an invoice or order deleted by mistake. For this feature to be useful, you have to provide a tool for restoring the deleted entities. Therefore, we are going to create an `Invoice` trash and an `Order` trash to bring deleted documents back to life.

7.4.1 Using properties to create reusable actions

The trash we want is exhibited in figure 7.5. It is a list of invoices or orders where the user can choose several of them and click the 'Restore' button, or alternatively, for a single entity, click the 'Restore' link on the row of the item to restore.

The logic of this restore action is to set the `deleted` property of the selected entities to false. Obviously, this is the same logic we used for deleting, but setting `deleted` to false instead of true. Since our conscience does not allow us to copy and paste, we're going to reuse the present code. Just adding a `restore` property to `InvoicingDeleteSelectedAction` is enough to enable recovery of deleted entities.

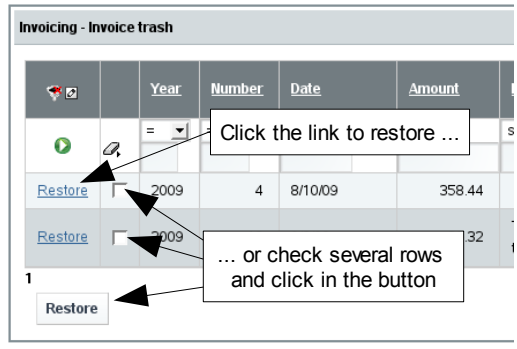

Figure 7.5 Invoice trash behavior

Listing 7.28 shows the required code for adding the `restore` property to the action.

Listing 7.28 New restore property in InvoiceDeleteSelectedAction

```
public class InvoicingDeleteSelectedAction ... {

    ...
    private boolean restore;  // A new restore property...
```

```
    public boolean isRestore() {  // ...with its getter
       return restore;
    }

    public void setRestore(boolean restore) {  // ...and setter
       this.restore = restore;
    }


    private void markSelectedEntitiesAsDeleted()
       throws Exception
    {
       Map values = new HashMap();
       values.put("deleted", true);  // Instead of a hardcoded true, we use
       values.put("deleted", !isRestore());  // the restore property value

       ...
    }

    ...
}
```

As you can see, we only added a `restore` property, and then use its complement as the new value for the `deleted` property in the entity. If `restore` is false, (the default), `deleted` is set to true, meaning that we perform a deletion. But if `restore` is true the action will save the `deleted` property as false, making the invoice, order or whatever entity available again in the application.

To make this action available you have to define it in *controllers.xml*, as in listing 7.29.

Listing 7.29 Restore action definition in controllers.xml

```
<controller name="Trash">
    <action name="restore" mode="list"
       class="org.openxava.invoicing.actions.InvoicingDeleteSelectedAction">
       <set property="restore" value="true"/>  <!-- Initialize the restore property to -->
    </action>                                  <!-- true before calling the execute() method of the action -->
</controller>
```

From now on you can utilize the `Trash.restore` action when you need an action for restoring. You're reusing the same code to delete and to restore, thanks to `<set />` element of `<action />` that allows you to configure the properties of your action.

Let's use this new restore action in the new trash modules.

7.4.2 Custom modules

As you already know, OpenXava generates a default module for each entity in your application. Still, you always have the option of defining the modules by hand, either by refining the behavior of a certain entity module, or by defining

completely new functionality associated to the entity. In this case we're going to create two new modules, `InvoiceTrash` and `OrderTrash`, to restore deleted documents. In these modules we will use the `Trash` controller. Listing 7.30 shows the module definitions in the `application.xml` file.

Listing 7.30 The InvoiceTrash and OrderTrash definitions in application.xml

```
<application name="Invoicing">

    <default-module>
       <controller name="Invoicing"/>
    </default-module>

    <module name="InvoiceTrash">
       <env-var name="XAVA_LIST_ACTION"
          value="Trash.restore"/>  <!-- The action to be shown in each row -->
       <model name="Invoice"/>
       <tab name="Deleted"/>  <!-- To show only the deleted entities -->
       <controller name="Trash"/>  <!-- With only one action: restore -->
       <mode-controller name="ListOnly"/>  <!-- Only list mode -->
    </module>

    <module name="OrderTrash">
       <env-var name="XAVA_LIST_ACTION" value="Trash.restore"/>
       <model name="Order"/>
       <tab name="Deleted"/>
       <controller name="Trash"/>
       <mode-controller name="ListOnly"/>
    </module>

</application>
```

These modules work with `Invoice` and `Order`, but they are list only modules, thanks to the `ListOnly` controller used as `mode-controller`. Moreover, they define a specific action as a row action using the XAVA_LIST_ACTION environment variable. Figure 7.6 shows the `InvoiceTrash` module in action.

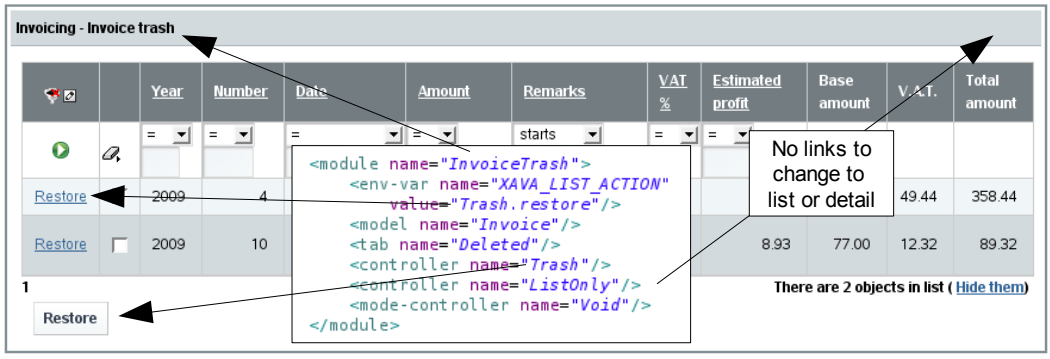

Figure 7.6 InvoiceTrash module only has list mode and a special row action

7.4.3 Several tabular data definitions by entity

Another important detail is that only deleted entities are shown in the list. This

is possible because we define a specific `@Tab` by name for the module. Listing 7.31 shows it again.

Listing 7.31 Detail about to choose the @Tab for a module

```
<module … >
    ...
    <tab name="Deleted"/>  <!-- 'Deleted' is a @Tab defined in the entity -->
    ...
</module>
```

Of course you must have a `@Tab` named "Deleted" in your `Order` and `Invoice` entities. Just as listing 7.32 shows.

Listing 7.32 The 'Deleted' tab definition in Invoice and Order

```
@Tabs({  // @Tabs is for defining several tabs for the same entity
    @Tab(baseCondition = "deleted = false"),  // No name, so the default tab
    @Tab(name="Deleted", baseCondition = "deleted = true")  // A named tab
})
public class Invoice extends CommercialDocument { … }

@Tabs({
    @Tab(baseCondition = "deleted = false"),
    @Tab(name="Deleted", baseCondition = "deleted = true")
})
public class Order extends CommercialDocument { … }
```

You see how `@Tabs` allows you to define several tabular data definitions for each entity. Thus, you use the `@Tab` with no name as default list for `Invoice` and `Order`, but you have a `@Tab` named 'Deleted' you can use for generating a list with only the deleted rows. In our context you use it for the trash modules.

7.4.4 Reusable obsession

We have done a good job! The `InvoicingDeleteSelectedAction` code can delete and restore entities, and we added the restore capacity with a very small amount of code and without copy & paste.

Now a lot of pernicious thoughts are swarming around your head. Surely you are thinking: "This action is no longer a mere delete action, but rather a delete and restore action", and then: "Wait a minute, it is actually an action that updates the `deleted` property of the current entity", followed by your next thought: "With very little improvement we can update any property of the entity".

Yes, you're right. You can easily create a more generic action, an `UpdatePropertyAction` for example; and use it to declare your `deleteSelected` and `restore` actions, just as shown in listing 7.33.

Listing 7.33 Using ultra generic action to update any property of any entity

```
<action name="deleteSelected" mode="list" confirm="true"
    class="org.openxava.invoicing.actions.UpdatePropertyAction"
```

```
    keystroke="Control D">
    <set property="property" value="deleted"/>
    <set property="value" value="true"/>
</action>

<action name="restore" mode="list"
    class="org.openxava.invoicing.actions.UpdatePropertyAction">
    <set property="property" value="deleted"/>
    <set property="value" value="false"/>
</action>
```

Although it seems like a good idea, we're not going to create this flexible `UpdatePropertyAction`. Because the more flexible your code is, the more sophisticated it is. And we don't want sophisticated code. We want simple code, and though simple code is impossible to achieve, we must make an effort to have the simplest possible code. The advice is: create reusable code only when it simplifies your application right now.

7.5 JUnit tests

We have refined the way your application deletes entities, moreover we added two custom modules, the trash modules. Before we move ahead, we must write the tests for the new features.

7.5.1 Testing the customized delete behavior

We do not have to write new tests for the deletion, because the current testing code already tests the delete functionality. Usually, when you change the implementation of some functionality but not their use from the user viewpoint, as in our current case, you do not need to add new tests.

Just run all the tests for your application, and adjust the needed details in order for them to work properly. In fact you only need to change "CRUD.delete" to "Invoicing.delete" and "CRUD.deleteSelected" to "Invoicing.deleteSelected" in a few tests. Listing 7.34 summarizes the changes you need to apply to your current test code.

Listing 7.34 Changing "CRUD.delete" by "Invoicing.delete" in tests

```
// In CustomerTest.java file
public class CustomerTest extends ModuleTestBase {
    ...
    public void testCreateReadUpdateDelete() throws Exception {
       ...
       // Delete
       execute("CRUD.delete");
       execute("Invoicing.delete");
       assertMessage("Customer deleted successfully");
    }
```

```
    ...
}

// In CommercialDocumentTest.java file
abstract public class CommercialDocumentTest extends ModuleTestBase {
    ...
    private void remove() throws Exception {
       execute("CRUD.delete");
       execute("Invoicing.delete");
       assertNoErrors();
    }
    ...
}

// In ProductTest.java file
public class ProductTest extends ModuleTestBase {
    ...
    public void testRemoveFromList() throws Exception {
       ...
       execute("CRUD.deleteSelected");
       execute("Invoicing.deleteSelected");
       ...
    }
    ...
}

// In OrderTest.java file
public class OrderTest extends CommercialDocumentTest {
    ...
    public void testSetInvoice() throws Exception {
       ...
       execute("CRUD.delete");
       execute("Invoicing.delete");
       ...
    }

}
```

After these changes all your tests will work properly, and this verifies that your refined delete actions preserve the original semantics. Only the implementation has changed.

7.5.2 Testing several modules in the same test method

You also have to test the new custom modules, `OrderTrash` and `InvoiceTrash`. Additionally, we will also verify that the removal logic works fine, and that the entities are only marked as deleted and not actually removed.

To test the `InvoiceTrash` module we will follow the steps below:

- Start in the `Invoice` module.
- Delete an invoice from detail mode and verify that it has been deleted.
- Delete an invoice from list mode and verify that it has been deleted.

- Go to the `InvoiceTrash` module.
- Verify that it contains the two deleted invoices.
- Restore them and verify that they disappear from the trash list.
- Come back to the `Invoice` module.
- Verify that the two restored invoices are in the list.

You can see that we start in the `Invoice` module. Moreover, you surely have realized that the logic to test the `Order` module is exactly the same. So instead of creating the new test classes, `OrderTrashTest` and `InvoiceTrash`, we'll just add the test in the already existing `CommercialDocumentTest`. Thus we reuse the same code for testing `OrderTrash`, `InvoiceTrash`, and the custom delete logic. This code is in the `testTrash()` method shown in listing 7.35.

Listing 7.35 The testTrash() method in CommercialDocumentTest

```
public void testTrash() throws Exception {
    assertListOnlyOnePage();  // Only one page in the list, that is less than 10 rows
    execute("List.orderBy", "property=number");  // We order the list by number

    // Deleting from detail mode
    int initialRowCount = getListRowCount();
    String year1 = getValueInList(0, 0);
    String number1 = getValueInList(0, 1);
    execute("Mode.detailAndFirst");
    execute("Invoicing.delete");
    execute("Mode.list");

    assertListRowCount(initialRowCount - 1);  // There is one row less
    assertDocumentNotInList(year1, number1);  // The deleted entity is not in the list

    // Deleting from list mode
    String year2 = getValueInList(0, 0);
    String number2 = getValueInList(0, 1);
    checkRow(0);
    execute("Invoicing.deleteSelected");
    assertListRowCount(initialRowCount - 2); // There are 2 fewer rows
    assertDocumentNotInList(year2, number2); // The other deleted entity is not
                                             // in the list now
    // Verifying deleted entities are in the trash module
    changeModule(model + "Trash");  // model can be 'Invoice' or 'Order'
    assertListOnlyOnePage();
    int initialTrashRowCount = getListRowCount();

    assertDocumentInList(year1, number1);  // We verify that the deleted entities
    assertDocumentInList(year2, number2);  // are in the trash module list
    // Restoring using row action
    int row1 = getDocumentRowInList(year1, number1);
    execute("Trash.restore", "row=" + row1);
    assertListRowCount(initialTrashRowCount - 1);  // 1 row less after restoring
    assertDocumentNotInList(year1, number1); // The restored entity is no longer shown
                                             // in the trash module list
    // Restoring by checking a row and using the bottom button
```

```
    int row2 = getDocumentRowInList(year2, number2);
    checkRow(row2);
    execute("Trash.restore");
    assertListRowCount(initialTrashRowCount - 2);   // 2 fewer rows
    assertDocumentNotInList(year2, number2);  // The restored entity is no longer shown
                                              // in the trash module list

    // Verifying entities are restored
    changeModule(model);
    assertListRowCount(initialRowCount);   // After restoring we have the original
    assertDocumentInList(year1, number1);  // rows again
    assertDocumentInList(year2, number2);
}
```

As you see the `testTrash()` follows the aforesaid steps. Note how using the `changeModule()` method from `ModuleTestBase` your test can change to another module. We use it to switch to the trash module, and then come back.

In our tests we're using a few auxiliary methods you have to add to `CommercialDocumentTest`. The first is `assertListOnlyOnePage()` to assert that the list is adequate for running this test. See listing 7.36.

Listing 7.36 Method in CommercialDocumentTest to verify the correct list state

```
private void assertListOnlyOnePage() throws Exception {
    assertListNotEmpty();  // This is from ModuleTestBase
    assertTrue("Must be less than 10 rows to run this test",
          getListRowCount() < 10);
}
```

We need to have less than 10 rows, because the `getListRowCount()` method only informs about rows that are displayed. If you have more than 10 rows (10 is the default row count per page) you cannot take advantage of `getListRowCount()`, it would always return 10.

The remaining methods are for verifying if some order or invoice is (or is not) in the list. See them in listing 7.37.

Listing 7.37 CommercialDocumentTest methods to assert if a document is in list

```
private void assertDocumentNotInList(String year, String number)
    throws Exception
{
    assertTrue(
       "Document " + year + "/" + number +" must not be in list",
       getDocumentRowInList(year, number) < 0);
}

private void assertDocumentInList(String year, String number)
    throws Exception
{
    assertTrue(
       "Document " + year + "/" + number + " must be in list",
       getDocumentRowInList(year, number) >= 0);
}
```

```
private int getDocumentRowInList(String year, String number)
    throws Exception
{
    int c = getListRowCount();
    for (int i=0; i<c; i++) {
       if (  year.equals(getValueInList(i, 0)) &&
          number.equals(getValueInList(i, 1)))
       {
          return i;
       }
    }
    return -1;
}
```

You can see in `getDocumentRowInList()` how to create a loop for searching after a specific value in a list.

Now you can run all the tests for your Invoicing application. Everything should be green.

7.6 Summary

The standard OpenXava behavior is only a starting point. Using the delete actions as excuse we have explored a couple of ways to refine the details of the application behavior. With the techniques in this lesson you can not only refine the delete logic, but fully alter the way your OpenXava application works. Thus, you have the possibility of tailoring the behavior of your application to fit your user expectations.

The default OpenXava behavior is pretty limited, only CRUD and reporting. If you want a valuable application for your user you need to add specific functionality to help your user solve his problems. We will do more of that in the next lesson.