

< o p e n - s o u r c e >

OpenXava

< j 2 e e - d e v e l o p m e n t >

V E R S I O N 2 . 0 . 2

R E F E R E N C E G U I D E

Contents

0 Preface.....	5
0.1 What is new in OpenXava 2.0.2?.....	5
0.2 What is new in OpenXava 2.0.1?.....	5
0.3 What is new in OpenXava 2.0?.....	6
1 Overview.....	7
1.1 Presentation.....	7
1.2 Business component.....	7
1.3 Controllers.....	7
1.4 Application.....	8
1.5 Project structure.....	8
1.6 Conclusion.....	9
2 My first OpenXava project.....	10
2.1 Create a new project.....	10
2.2 Configure database.....	10
2.3 Your first component	10
2.4 The application.....	12
2.5 The table.....	13
2.6 Executing your application.....	13
2.7 Automating the tests.....	14
2.8 The labels.....	16
2.9 Conclusion.....	17
3 Model.....	18
3.1 Java implementation.....	18
3.2 Business Component.....	18
3.3 Entity and aggregates.....	19
3.4 Entity.....	19
3.5 Bean (1).....	21
3.6 EJB (2).....	23
3.7 Implements (3).....	25
3.8 Property (4).....	27
3.8.1 Stereotype.....	28
3.8.2 IMAGES_GALLERY stereotype (new in v2.0).....	29
3.8.3 Valid values.....	30
3.8.4 Calculator.....	30
3.8.5 Default value calculator.....	35
3.8.6 Validator.....	37
3.9 Reference (5).....	38
3.9.1 Default value calculator in references.....	40
3.10 Collection (6).....	41
3.11 Method (7).....	45
3.12 Finder (8).....	48
3.13 Postcreate calculator (9).....	50
3.14 Postmodify calculator (11).....	51
3.15 Postload and preremove calculator (10, 12).....	53
3.16 Validator (13).....	53

3.17 Remove validator (14).....	55
3.18 Aggregate.....	56
3.18.1 Reference to aggregate.....	57
3.18.2 Collection of aggregates.....	58
4 View.....	60
4.1 Layout.....	61
4.2 Property view.....	66
4.2.1 Label format.....	66
4.2.2 Value change event.....	67
4.2.3 Actions of property.....	67
4.3 Reference view.....	69
4.3.1 Choose view.....	70
4.3.2 Customizing frame.....	72
4.3.3 Custom search action.....	72
4.3.4 Custom creation action.....	73
4.3.5 Descriptions list (combos).....	74
4.4 Collection view.....	76
4.4.1 Custom edit/view action.....	79
4.4.2 Custom list actions.....	80
4.4.3 Custom detail actions.....	81
4.4.4 Refining collection view default behavior (new in v2.0.2).....	83
4.5 View property.....	84
4.6 Editors configuration.....	85
4.7 Multiple values editors	88
4.8 Custom editors and stereotypes for displaying combos.....	90
4.9 View without model.....	92
5 Tabular data.....	94
5.1 Initial properties and emphasize rows.....	95
5.2 Filters and base condition.....	96
5.3 Pure SQL select.....	99
5.4 Default order.....	99
6 Object/relational mapping.....	100
6.1 Entity mapping.....	100
6.2 Property mapping.....	101
6.3 Reference mapping.....	103
6.4 Multiple property mapping.....	105
6.5 Reference to aggregate mapping.....	107
6.6 Aggregate used in collection mapping.....	108
6.7 Converters by default.....	110
6.8 Object/relational philosophy.....	112
7 Controllers.....	113
7.1 Environment variables and session objects.....	113
7.2 The controller and its actions.....	114
7.3 Controllers inheritance.....	118
7.4 List mode actions.....	118
7.5 Overwriting default search.....	121
7.6 Initialize a module with an action.....	123

7.7 Calling another module.....	124
7.8 Changing the module of current view.....	127
7.9 Go to a JSP page.....	127
7.10 Generating a custom report with JasperReports.....	128
7.11 Uploading and processing a file from client (multipart form).....	130
7.12 All action types.....	132
8 Application.....	134
8.1 Typical module example.....	135
8.2 Only detail module.....	136
8.3 Only list module.....	136
8.4 Documentation module.....	136
9 Aspects.....	138
9.1 Introduction to AOP.....	138
9.2 Aspects definition.....	138
9.3 AccessTracking: A practical application of aspects.....	140
9.3.1 The aspect definition.....	140
9.3.2 Setup AccessTracking.....	141
10 Miscellaneous.....	143
10.1 Many-to-many relationships.....	143
10.2 Programming with Hibernate.....	145
10.3 Custom JSP view and OpenXava taglibs.....	145
10.3.1 Example.....	145
10.3.2 xava:editor.....	147
10.3.3 xava:action, xava:link, xava:image, xava:button.....	147

This guide tries to be a complete reference to OpenXava from the application developer viewpoint. It's centered specially in XML file syntax. It's full of XML and Java code examples.

This document does not try to be an introduction to OpenXava but a complete reference guide, although chapter 1 and 2 are introductory. For the first steps it's better to use the tutorial (that you can find on the OpenXava web site or in the OpenXava distribution). On the other hand, the definitive reference of OpenXava is the OpenXavaTest project, that contains all possibilities offered by OpenXava.

This guide supposes that you use Eclipse as IDE, although OpenXava does not use the Eclipse resources and can be used without any problem in other IDE including a simple editor plus command line. Also we assume that your Eclipse workspace is the *workspace* subfolder of the OpenXava base directory. If you follow the tutorial instructions everything will work well.

Although this is a reference guide it is not a bad idea to read it sequentially at least the first time, to understand the possibilities of OpenXava.

This guide does neither include an API definition nor configuration or philosophical issues. You can find information about these things at <http://www.openxava.org>

All suggestions (including grammatical ones) are welcome. You can send any comment about this guide to javierpaniza@gestion400.com.

0.1 What is new in OpenXava 2.0.2?

- New stereotype `ZEROS_FILLED/RELLENADO_CON_CEROS`: Section 3.8.1
- `IdentityCalculator` for database *identity* automatic id generation: Section 3.8.5
- New action type `IPropertyAction` that can be used for an action associated to a property (in the view), this action will receive the name of the property: Section 4.2.3
- The `as-aggregate` attribute for `collection-view` elements allows that an collection of entities behaves as a collection of aggregates: Section 4.4
- Now you can add and remove programmatically detail and list actions to a collection view: Section 4.4.3
- It's possible refining collection view default behavior using `new new-action`, `save-action`, `hide-detail-action` and `remove-action` action in `collection-view`: section 4.4.4

0.2 What is new in OpenXava 2.0.1?

- `SequenceCalculator` for database *sequence* automatic id generation: Section 3.8.5
- Arbitrary actions are allowed in `<reference-view/>`: Section 4.3
- Keystrokes support for the actions: Section 7.2
- `<xava:editor/>` supports qualified properties: Section 10.3

0.3 What is new in OpenXava 2.0?

- Model layer now is generated in POJO (Plain Old Java Object) format.
- Persistence is managed by Hibernate, this allows you deploy OpenXava applications in a simple Tomcat (or whatever servlet container you wish).
- `IMAGE_GALLERY` stereotype: Section 3.8.2
- `IModelCalculator` added, as better alternative for `IEntityCalculator`: Section, 3.8.4
- Nested sections are allowed in views: Section 4.1
- It's possible to access environment variables inside a filter: Section 5.2
- `cmp-type` allowed for reference mapping: Section 6.3
- It's possible to hide and to show actions in the User Interface: Section 7.12
- New Chapter 10 Miscellaneous: with explanations for many-to-many relationship, using of Hibernate inside OpenXava, JSP custom views and OpenXava taglibs.

1.1 Presentation

OpenXava is a framework to develop J2EE applications quickly and easily.

The name OpenXava comes from **Open** source, **XML** and **Java**. And the underlying philosophy is to define with XML and to program with Java, the more definition (that is more XML) and less programming (that is less Java) the better.

The main goal is to make the more typical things in a business application easily, while you still have the necessary flexibility to develop the most advances features as you like.

Below you can see some basic concepts of OpenXava.

1.2 Business component

The fundamental pieces to create applications in OpenXava are the business components. In OpenXava context a business component is a XML file that contains all needed information about a business concept that allows you to create applications on it. For example, all needed information that the system has to know about the concept of invoice is defined in the file *Invoice.xml*. In a business component you can define:

- The data structure.
- Validations, calculations and in general all logic associated with the business concept.
- The possibles views, i. e. the configuration of all possible user interfaces for this component.
- The possibilities for the tabular data presentation. This is used in list mode (data navigation), reports, export to excel, etc.
- Object-relational mapping, this includes information about database tables and how to convert it to the objects of your Java application

This splitting way is good for work groups, and allows to develop generic business component for using in different projects.

1.3 Controllers

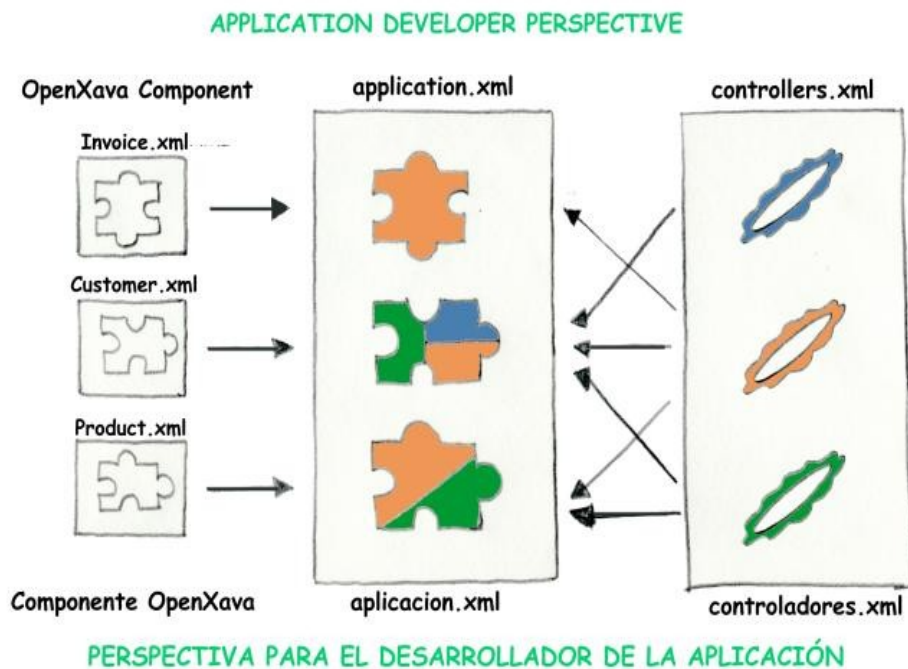
The business component does not define the things that user can do in the application; this is defined in controllers. The controllers are specified in the file *xava/controllers.xml* of your project; in addition OpenXava has a set of predefined controllers in *OpenXava/xava/controllers.xml*.

A controller is a set of actions. An action is a button or link that a user can click.

The controllers are separated from the business components because one controller can be assigned to several business components. For example, a controller to make CRUD operations, to print in PDF format or to export to plain files, etc. can be used and reused for components like invoices, customers, suppliers, etc.

1.4 Application

A OpenXava application is a set of modules. A module joins a business component with one or more controllers. Visually is:



Each module of the application is what the end user uses, and generally it is configured as a portlet within a portal.

1.5 Project structure

A typical OpenXava project usually contains the these folders:

- `[root]`: In the root you can find *build.xml* (with the Ant task) and the configuration files that it uses (usually one for customer).
- `src[source folder]`: Contains your Java source code.
- `components`: XML files with the definitions of your business components.
- `xava`: XML files to configure your OpenXava application. The main ones are *application.xml* and *controllers.xml*.
- `I18n`: Resource files with labels and messages in several languages.
- `gen-src[source folder]`: Code generated by XDoclet. Only needed if you generate EJB.
- `gen-src-xava[source folder]`: Code generated by OpenXava.
- `properties[source folder]`: Property files to configure your application.
- `build`: XML files needed in a J2EE application, some are generated and other can be edited manually.
- `filtered-files[source folder]`: It's for internal use of build tasks. Must be a source code folder. (*new in v2.0*)
- `data`: Useful to hold the scripts to create the tables of your application, if needed.

- `web`: Web content. Usually JSP files, lib and classes. Most of the content is generated automatically, but you can put here your own JSPs or other custom web resources.

1.6 Conclusion

This chapter introduce the most basic concepts of OpenXava to you. The rest of this guide will go into the details.

2.1 Create a new project

First open your Eclipse and make its workspace the one that comes with the OpenXava distribution (*openxava.zip*). To create a new project you have to edit the file *CreateNewProject.xml* in the *OpenXavaTemplate* project in this way:

```
<property name="project" value="Management" />
<property name="package" value="management" />
<property name="component" value="Warehouse" />
<property name="module" value="Warehouses" />
<property name="datasource" value="ManagementDS" />
```

Now execute *CreateNewProject.xml* using Ant. You can do it with *Right Button on CreateNewProject.xml > Run as > Ant Build*

Using the appropriate Eclipse Wizard create a new Java Project named *Management*.

And now you have a new project ready to start working, but before continuing you need to configure the database.

2.2 Configure database

OpenXava generates a J2EE application intended to be deployed in a J2EE application server (since v2.0 OpenXava applications also run in a simple servlet container, as Tomcat). In OpenXava you only need to indicate the Data Source JNDI and then configure the data source in your application server. Configuring a data source in an application server is out of the scope of this guide, nevertheless you have below detailed instructions to configure a database in order to run this first project using Tomcat as application server and Hypersonic as database.

With Tomcat stopped edit the file *context.xml* in Tomcat's directory *conf*. In this file add the next entry:

```
<Resource name="jdbc/ManagementDS" auth="Container" type="javax.sql.DataSource"
    maxActive="20" maxIdle="5" maxWait="10000"
    username="sa" password="" driverClassName="org.hsqldb.jdbcDriver"
    url="jdbc:hsqldb:file:../data/management-db" />
```

The main thing here is the JNDI name, this is the only thing referenced from OpenXava, in this case *ManagementDS*, also you must choose the name of database, here *management-db*, that references to a physical file where the data (in reality a SQL script) are.

2.3 Your first component

Creating an OpenXava component is easy: The definition of each component is a XML file with

simple syntax. In order to begin you have to edit *Warehouse.xml*, that you already have because it was created when project was created. Edit it in this way:

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

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

<component name="Warehouse">
    <entity>
        <property name="zoneNumber" key="true"
            size="3" required="true" type="int"/>
        <property name="number" key="true"
            size="3" required="true" type="int"/>
        <property name="name" type="String"
            size="40" required="true"/>
    </entity>

    <entity-mapping table="MANAGEMENT@separator@WAREHOUSES">
        <property-mapping
            property="zoneNumber" column="ZONE"/>
        <property-mapping
            property="number" column="NUMBER"/>
        <property-mapping
            property="name" column="NAME"/>
    </entity-mapping>
</component>
```

In this definition you can see two parts clearly different, the first one is `entity`; `entity` is used to define the main model for this component, the information here is used to create Java classes and others resources to work with the Warehouse concept. The generated code can use POJO (Simple Java Classes) + Hibernate (*new in v2.0*) or EJB technology (with EntityBeans CMP2). In this part you do not only define the data structure but also the associated business logic.

In entity you define properties, let's see how:

```
<property
    name="zoneNumber"           (1)
    key="true"                  (2)
    size="3"                    (3)
    required="true"             (4)
    type="int"                  (5)
/>
```

This is its meaning:

(1)`name`: It's the property name in the generated Java code and also serves as identifier inside

OpenXava files.

- (2)**key**: Indicates if this property is part of the key. The key is a unique identifier of the object and usually matches with the primary key of a database table.
- (3)**size**: Length of data. It's optional, but useful to display better user interfaces.
- (4)**required**: Indicates if it's required to validate the existence of data for this property just before creation or modification.
- (5)**type**: The type of the property. All valid types for a Java property are applicable here, including integrated types, JDK classes and custom classes.

The possibilities of the `property` element go far from what is shown here, you can see a more complete explanation in chapter 3.

The second part of *Warehouse.xml* is about the mapping, where you associate your component with a table in the database; the syntax is obvious. The use of `@separator@` in table name allows you to have an application that runs against a database with collection or schema support or not, you only need to edit *build.xml* and set the correct value to `separator`, `'.'` or `'_'`.

Now you are ready to generate code. To do this you have to execute the ant target *generateCode*. The more practical way to execute an ant target in Eclipse is creating it in *Externals Tools*. Then you have just to choose the option in the menu, when you want to re-execute the target. It's very important to configure the target for **refreshing** the *Management* project **after executing** it. In your *build.xml* the most frequently used ant targets are preconfigured already.

After generating code you can execute *Build*, and verify that there are no errors.

With your first component made you can define your OpenXava application.

2.4 The application

In OpenXava an application is the final product that a user will use. An application is made of a set of modules. A module is the union of a component (what data and logic) and a set of controllers (what actions). As usual in OpenXava an application is defined using a XML file, in this case *xava/application.xml*. If you have well done the step of the creation of the project, you should have already the application file. Anyway, let's examine it:

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

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

<application name="Management">

    <module name="Warehouses">
        <model name="Warehouse"/>
        <controller name="Typical"/>
    </module>

</application>
```

In this case you have only one module defined, `Warehouses`, this module name will be used in the URL of internet browser to execute it, or will be the portlet name if you deploy your application in a portal. You have defined as model `Warehouse`, the component that you have defined before, and as controller `Typical`, this predefined controller allows basic CRUD operations (Create, Read, Update and Delete), moreover allows to generate PDF reports and to export to excel. From a visual viewpoint we can say that a `controller` defines the buttons displayed to the user and `model` defines the data, although is a simplified explanation.

2.5 The table

Before testing the application you have to create the table in database.

Create the file *management-db.script*, in the *data* folder of Tomcat, and put:

```
CREATE SCHEMA PUBLIC AUTHORIZATION DBA

CREATE MEMORY TABLE MANAGEMENT_WAREHOUSES(ZONE INTEGER NOT NULL,NUMBER INTEGER NOT
NULL,NAME VARCHAR(40),PRIMARY KEY(ZONE,NUMBER))

CREATE USER SA PASSWORD ""

GRANT DBA TO SA

SET WRITE_DELAY 20
```

Also you have to create the file *management-db.properties* in the *data* folder of Tomcat with this content:

```
#HSQL Database Engine
hsqldb.script_format=0
runtime.gc_interval=0
sql.enforce_strict_size=false
hsqldb.cache_size_scale=8
readonly=false
hsqldb.nio_data_file=true
hsqldb.cache_scale=14
version=1.8.0
hsqldb.default_table_type=memory
hsqldb.cache_file_scale=1
hsqldb.log_size=200
modified=yes
hsqldb.cache_version=1.7.0
hsqldb.original_version=1.8.0
hsqldb.compatible_version=1.8.0
```

Start Tomcat and now everything is ready.

2.6 Executing your application

After your hard work it is time to see the fruit of your sweat. Let's go.

- Execute the ant target *deployWar*.

- Open an internet browser and go to <http://localhost:8080/Management/xava/module.jsp?application=Management&module=Warehouses>

And now you can play with your module and see its behavior.

Also you can deploy your module as a JSR-168 portlet, in this way:

- Execute ant target *deployPortlets*.
- Open an internet browser and go to <http://localhost:8080/>

Now you can login as 'admin' and test your module.

2.7 Automating the tests

Although it seems that the most natural way to test an application is to open a browser and use it like a final user; in fact it is more productive automating the tests, in this way as your system grows, you have it tied and you avoid to break it when you advance.

OpenXava uses a test system based on JUnit and HttpUnit. The OpenXava JUnit tests simulate the behavior of a real user with a browser. This way you can replicate exactly the same tests that you can do directly with an internet browser. The advantage of this approach is that you can test easily all layers of your program from user interface to database.

If you test the module manually you usually create a new record, search it, modify and finally delete it. Let's do this automatically:

First you must create a package for the test classes, `org.openxava.management.tests`, and then add the `WarehousesTest` class to it, with next code:

```
package org.openxava.management.tests;

import org.openxava.tests.*;

/**
 * @author Javier Paniza
 */

public class WarehousesTest extends ModuleTestBase {

    public WarehousesTest(String testName) {
        super(testName, "Management", "Warehouses"); // (1)
    }

    public void testCreateReadUpdateDelete() throws Exception {
        // Create
        execute("CRUD.new"); // (2)
        setValue("zoneNumber", "1"); // (3)
        setValue("number", "7");
    }
}
```

```

        setValue("name", "JUNIT Warehouse");
        execute("CRUD.save");
        assertNoErrors(); // (4)
        setValue("zoneNumber", ""); // (5)
        setValue("number", "");
        setValue("name", "");

        // Read
        setValue("zoneNumber", "1");
        setValue("number", "7");
        execute("CRUD.search");
        setValue("zoneNumber", "1");
        setValue("number", "7");
        setValue("name", "JUNIT Warehouse");

        // Update
        setValue("name", "JUNIT Warehouse MODIFIED");
        execute("CRUD.save");
        assertNoErrors();
        setValue("zoneNumber", "");
        setValue("number", "");
        setValue("name", "");

        // Verify if modified
        setValue("zoneNumber", "1");
        setValue("number", "7");
        execute("CRUD.search");
        setValue("zoneNumber", "1");
        setValue("number", "7");
        setValue("name", "JUNIT Warehouse MODIFIED");

        // Delete
        execute("CRUD.delete");
        assertMessage("Warehouse deleted successfully"); // (6)
    }
}

```

You can learn from this example:

- (1)Constructor: In the constructor you indicate the application and module name.
- (2)execute: Allows to simulate a button or link click. As an argument you send the action name;

you can view the action names in *OpenXava/xava/controllers.xml* (the predefined controllers) and *Management/xava/controllers.xml* (the customized ones). Also if you move the mouse over the link your browser will show you the JavaScript code with the OpenXava action to execute. That is `execute("CRUD.new")` is like click in 'new' button in the user interface.

- (3)`setValue`: Assigns a value to a form control. That is, `setValue("name", "Pepe")` has the same effect than typing in the field 'name' the text "Pepe". The values are always alphanumeric because they are assigned to a HTML form.
- (4)`assertNoErrors`: Verify that there are no errors. In the user interface errors are red messages showed to user and added by the application logic.
- (5)`assertValue`: Verify if the value in the form field is the expected one.
- (6)`assertMessage`: Verify if the application has shown the indicated informative message.

You can see that is very easy to test that a module works; writing this code can take 5 minutes, but at end you will save hours of work, because from now on you can test your module just in 1 second, and because when you break the `Warehouses` module (maybe touch in another part of your application) your test warns you just in time.

For more details have a look at the JavaDoc API of `org.openxava.tests.ModuleTestBase` and examine the examples in `org.openxava.test.tests` of *OpenXavaTest*.

By default the test runs against the module in alone (non portal) mode (that is deployed with `deployWar`). But if you want it's possible to test against the portlet version (that is deployed with `deployPortlets`). You only need to edit the file `properties/xava-junit.properties` and write:

```
jetspeed2.url=openxava
#jetspeed2.username=demo
#jetspeed2.password=demo
```

The `username` and `password` are optional, if not specified the test does not login to the portal (this is useful if you assign your module to the 'guest' user).

2.8 The labels

Now everything works well, but a little detail remains yet. The labels showed to user are not appropriate (for example, `zoneNumber`). You can assign a label to each property with the attribute `label`, but this is not a good solution. The ideal way is to write a file with all labels, thus you can translate your product to another language with no problems.

To define the labels you only have to create a file called *Management-labels_en.properties* in *i18n* folder. Edit that file and add:

```
Warehouse=Warehouse
zoneNumber=Zone number
```

You do not have to put all properties, because the more common cases (number, name, description and a big etc) is already included with OpenXava in English, Spanish, French, German, Indonesian and Catalan.

If you wish the version in an other language (Spanish for example), you only need to copy and paste with the appropriate suffix. For example, you can have a *Management-labels_es.properties* with the

next content:

```
Warehouse=Almacén  
zoneNumber=Código de zona
```

If you want to know more about how to define labels of your OpenXava elements please look in *OpenXavaTest/xava/i18n*.

2.9 Conclusion

In this chapter you have seen how to create a new project, and you had a first taste some OpenXava features. Of course, OpenXava offers more possibilities, and in the rest of the book you will see them in more detail.

The model layer in an object oriented application contains the business logic, that is the structure of the data and all calculations, validations and processes associated to this data.

OpenXava is a model oriented framework where the model is the most important, and the rest (e.g. user interface) depends on it.

The way to define the model in OpenXava is using XML and a few of Java. OpenXava generates a complete Java implementation of your model from your definition.

3.1 Java implementation

Currently OpenXava generates code for the next 3 alternatives:

1. Plain Java Classes (the so-called POJOs) for the model using Hibernate for persistence.
2. Classic EJB2 EntityBeans for the model and persistence.
3. POJOs + Hibernate inside an EJB container.

The option 1 is the default one and the best for the most cases. The option 2 is for supporting all OpenXava applications written until now using EJB. The option 3 can be useful in some circumstances. You can see how to configure this in *OpenXavaTest/properties/xava.properties*.

3.2 Business Component

As you have seen the basic unit to create an OpenXava application is the business component. A business component is defined using a XML file. The structure of a business component in OpenXava is:

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

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

<component name="ComponentName">

    <!-- Model -->
    <entity>...</entity>
    <aggregate name="...">...</aggregate>
    <aggregate name="...">...</aggregate>
    ...

    <!-- View -->
    <view>...</view>
    <view name="...">...</view>
```

```

<view name="...">...</view>

...

<!-- Tabular data -->
<tab>...</tab>
<tab name="...">...</tab>
<tab name="...">...</tab>
...

<!-- Object relational mapping -->
<entity-mapping table="...">...</entity-mapping>
<aggregate-mapping aggregate="..." table="...">...</aggregate-mapping>
<aggregate-mapping aggregate="..." table="...">...</aggregate-mapping>
...

</component>

```

The first part of the component, the part of entity and aggregates, is used to define the model. In this chapter you will learn the complete syntax of this part.

3.3 Entity and aggregates

The definition for entity and aggregate are practically identical. The entity is the main object that represents the business concept, while aggregates are additional object needed to define the business concept but cannot have its own life. For example, when you define an `Invoice` component, the heading data of invoice are in entity, while for invoice lines you can create an aggregate called `InvoiceDetail`; the life cycle of an invoice line is attached to the invoice, that is an invoice line without invoice has no meaning, and sharing an invoice line by various invoices is not possible, hence you will model `InvoiceDetail` as aggregate.

Formally, the relationship between A and B is aggregation, and B can be modeled as an aggregate when:

- You can say that A *has* an B.
- If A is deleted then its B is deleted too.
- B is not shared.

Sometimes the same concept can be modeled as aggregate or as entity in another component. For example, the address concept. If the address is shared by various persons then you must use a reference to entity, while if each person has his own address maybe an aggregate is a good option.

3.4 Entity

The syntax of entity is:

```

<entity>
    <bean ... />

```

(1)

```

    <ejb ... /> (2)
    <implements .../> ... (3)
    <property .../> ... (4)
    <reference .../> ... (5)
    <collection .../> ... (6)
    <method .../> ... (7)
    <finder .../> ... (8)
    <postcreate-calculator .../> ... (9)
    <postload-calculator .../> ... (10)
    <postmodify-calculator .../> ... (11)
    <preremove-calculator .../> ... (12)
    <validator .../> ... (13)
    <remove-validator .../> ... (14)
</entity>

```

- (1)bean (one, optional): Allows you to use an already existing JavaBean (a simple Java class, the so-called POJO). This applies if you use Hibernate as persistence engine. In this case the code generation for POJO and Hibernate mapping of this component will not be produced.
- (1)ejb (one, optional): Allows you to use an already existing EJB. This only applies if you use EJB CMP2 as persistence engine. In this case code generation for EJB code of this component will not be produced.
- (2)implements (several, optional): The generated code will implement this interface.
- (3)property (several, optional): The properties represent Java properties (with its *setters* and *getters*) in the generated code.
- (4)reference (several, optional): References to other models, you can reference to the entity of another component or an aggregate of itself.
- (5)collection (several, optional): Collection of references. In the generated code it is a property that returns a `java.util.Collection`.
- (6)method (several, optional): Creates a method in the generated code, in this case the method logic is in a calculator (`Icalculator`).
- (7)finder (several, optional): Used to create finder methods, in this case generates a EJB *finder*.
- (8)postcreate-calculator (several, optional): Logic to execute after making an object persistent. In Hibernate in a `PreInsertEvent`, in EJB2 in the `ejbPostCreate` method.
- (9)postload-calculator (several, optional): Logic to execute just after load the state of an object from persistent storage. In Hibernate in a `PostLoadEvent`, in EJB2 in the `ejbLoad` method.
- (10)postmodify-calculator (several, optional): Logic to execute after modifying a persistent object and before storing its state in persistent storage. In Hibernate in a `PreUpdateEvent`, in EJB2 in the `ejbStore` method.
- (11)preremove-calculator (several, optional): Logic to execute just before removing a persistent from persistent storage. In Hibernate in `PreDeleteEvent`, in EJB2 in the `ejbRemove` method.
- (12)validator (several, optional): Executes a validation at model level. This validator can receive

the value of various model properties. To validate a single property it is better to use a property level validator.

(13)`remove-validator` (several, optional): It's executed before removal, and can deny the object removing.

3.5 Bean (1)

With `<bean/>` you can specify that you want to use your own Java class.

For example:

```
<entity>
  <bean class="org.openxava.test.model.Family"/>
  ...
```

In this simple way you can write your own Java code instead of using OpenXava to generate it.

For our example you can write a `Family` class as follows:

```
package org.openxava.test.model;

import java.io.*;

/**
 * @author Javier Paniza
 */
public class Family implements Serializable {

    private String oid;
    private int number;
    private String description;

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

    public int getNumber() {
        return number;
    }
    public void setNumber(int number) {
        this.number = number;
    }
}
```

```

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

```

If you want a reference from the OpenXava generated code to your own handwritten code, then your Java class has to implement an interface (`IFamily` in this case) that extends `IModel` (see `org.openxava.test.Family` in *OpenXavaTest/src*).

Additionally you have to define the mapping using Hibernate:

```

<?xml version="1.0"?>

<!DOCTYPE hibernate-mapping
    SYSTEM "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="org.openxava.test.model">

    <class
        name="Family"
        table="XAVATEST@separator@FAMILY">

        <id name="oid" column="OID" access="field">
            <generator class="uuid"/>
        </id>

        <property name="number" column="NUMBER"/>
        <property name="description" column="DESCRIPTION"/>

    </class>

</hibernate-mapping>

```

You can put this file in the *hibernate* folder of your project. Moreover in this folder you have the *hibernate.cfg.xml* file that you have to edit in this way:

```

...
<session-factory>
    ...
    <mapping resource="Family.hbm.xml"/>
    ...

```

```
</session-factory>

...
```

In this easy way you can wrap your existing Java and Hibernate code with OpenXava. Of course, if you are creating a new system it is much better to rely on the OpenXava code generation.

3.6 EJB (2)

With `<ejb/>` you can specify that you want to use your own EJB (1.1 and 2.x version).

For example:

```
<entity>
  <ejb remote="org.openxava.test.ejb.Family"
        home="org.openxava.test.ejb.FamilyHome"
        primaryKey="org.openxava.test.ejb.FamilyKey"
        jndi="ejb/openxava.test/Family"/>
  ...
```

In this simple way you can write you own EJB code instead of using code that OpenXava generates.

You can write the EJB code from scratch (only for genuine men), if you are a normal programmer (hence lazy) probably you prefer to use wizards, or better yet XDoclet. If you choose to use XDoclet, then you can put your own XDoclet classes in the package `model` (or another package of your choice). This depends on the value of the `model.package` variable in *build.xml* in *src* folder of your project.; and your XDoclet code will be generated with the rest of OpenXava code.

For our example you can write a `FamilyBean` class in this way:

```
package org.openxava.test.ejb.xejb;

import java.util.*;
import javax.ejb.*;

import org.openxava.calculators.*;
import org.openxava.test.ejb.*;

/**
 * @ejb:bean name="Family" type="CMP" view-type="remote"
 *   jndi-name="OpenXavaTest/ejb/openxava.test/Family"
 * @ejb:interface extends="org.openxava.ejbx.EJBReplicable"
 * @ejb:data-object extends="java.lang.Object"
 * @ejb:home extends="javax.ejb.EJBHome"
 * @ejb:pk extends="java.lang.Object"
 *
 * @jboss:table-name "XAVATEST@separator@FAMILY"
 */
```

```

* @author Javier Paniza
*/
abstract public class FamilyBean
    extends org.openxava.ejbx.EJBReplicableBase // (1)
    implements javax.ejb.EntityBean {

    private UUIDCalculator oidCalculator = new UUIDCalculator();

    /**
     * @ejb:interface-method
     * @ejb:pk-field
     * @ejb:persistent-field
     *
     * @jboss:column-name "OID"
     */
    public abstract String getOid();
    public abstract void setOid(String nuevoOid);

    /**
     * @ejb:interface-method
     * @ejb:persistent-field
     *
     * @jboss:column-name "NUMBER"
     */
    public abstract int getNumber();
    /**
     * @ejb:interface-method
     */
    public abstract void setNumber(int newNumber);

    /**
     * @ejb:interface-method
     * @ejb:persistent-field
     *
     * @jboss:column-name "DESCRIPTION"
     */
    public abstract String getDescription();
    /**
     * @ejb:interface-method
     */
    public abstract void setDescription(String newDescription);

```



```

/**
 * @ejb:create-method
 */
public FamilyKey ejbCreate(Map properties) // (2)
    throws
        javax.ejb.CreateException,
        org.openxava.validators.ValidationException,
        java.rmi.RemoteException {
    executeSets(properties);
    try {
        setOid((String)oidCalculator.calculate());
    }
    catch (Exception ex) {
        ex.printStackTrace();
        throw new EJBException(
            "Impossible to create Family because:\n" +
            ex.getLocalizedMessage()
        );
    }
    return null;
}

public void ejbPostCreate(Map properties) throws javax.ejb.CreateException {
}
}

```

On writing your own EJB you must fulfill two little restrictions:

- (1) The class must extend from `org.openxava.ejbx.EJBReplicableBase`
- (2) It is required at least a `ejbCreate` (with its `ejbPostCreate`) that receives as argument a map and assign its values to the bean, as in the example.

Yes, yes, a little intrusive, but are not the EJB the intrusion culmination?

3.7 Implements (3)

With `<implements/>` you specify a Java interface that will be implemented by the generated code. Let's see it:

```

<entity>
    <implements interface="org.openxava.test.model.IWithName"/>
    ...
    <property name="name" type="String" required="true"/>
    ...

```

And you can write your Java interface in this way:

```

package org.openxava.test.model;

import java.rmi.*;

/**
 * @author Javier Paniza
 */
public interface IWithName {

    String getName() throws RemoteException;

}

```

Beware to make that generated code implements your interface. In this case you have a property named `name` that generates a method called `getName()` that implements the interface.

In your generated code you can find an `ICustomer` interface:

```

public interface ICustomer extends org.openxava.test.model.IWithName {

    ...

}

```

In the POJO generated code you can see:

```

public class Customer implements Serializable, org.openxava.test.model.ICustomer {

    ...

}

```

In the EJB generated code (if you generate it) you can see the remote interface:

```

public interface CustomerRemote extends

    org.openxava.ejbx.EJBReplicable,

    org.openxava.test.model.ICustomer

```

and the EJB bean class is affected too

```

abstract public class CustomerBean extends EJBReplicableBase

    implements

        org.openxava.test.model.ICustomer,

        EntityBean

```

This pithy feature makes the polymorphism a privileged guest of OpenXava.

As you can see OpenXava generates an interface for each component. It's good that in your code you use these interfaces instead of POJO classes or EJB remote interfaces. All code made in this way can be used with POJO and EJB version on same time, or allows you to migrate from a EJB to a POJO version with little effort. Although, if you are using POJOs exclusively you may use the POJOs classes directly and ignore the interfaces, as you wish.

3.8 Property (4)

An OpenXava property corresponds exactly to a Java property. It represents the state of an object that can be read and in some cases updated. The object does not have the obligation to store physically the property data, it only must return it when required.

The syntax to define a property is:

```
<property
  name="propertyName"           (1)
  label="label"                 (2)
  type="type"                   (3)
  stereotype="STEREOTYPE"      (4)
  size="size"                   (5)
  required="true|false"        (6)
  key="true|false"             (7)
  hidden="true|false"          (8)
>
  <valid-values .../>           (9)
  <calculator .../>            (10)
  <default-value-calculator .../> (11)
  <validator .../> ....        (12)
</property>
```

- (1)name (required): The property name in Java, therefore it must follow the Java convention for property names, like starting with lower-case. Using underline (_) is not advisable.
- (2)label (optional): Label showed to the final user. Is **much better** use the *i18n* files.
- (3)type (optional): It matches with a Java type. All types valid for a Java property are valid here, this include classes defined by you. You only need to provide a converter to allow saving in database and a editor to render as HTML; thus that things like `java.sql.Connection` or so can be a little complicated to manage as a property, but not impossible. It's optional, but only if you have specified `<bean/>` or `<ejb/>` or this property has a stereotype with a associated type.
- (4)stereotype(optional): Allows to specify an special behavior for some properties.
- (5)size (optional): Length in characters of property. Useful to generate user interfaces. If you do not specify the size, then a default value is assumed. This default value is associated to the stereotype or type and is obtained from *default-size.xml*.
- (6)required (optional): Indicates if this property is required. By default this is `true` for key properties without default value calculator on create and `false` in all other cases. On saving OpenXava verifies if the required properties are present. If this is not the case, then saving is not done and a validation error list is returned. The logic to determine if a property is present or not can be configured by creating a file called *validators.xml* in your project. You can see the syntax in *OpenXava/xava/validators.xml*.
- (7)key (optional): Indicates that this property is part of the key. At least one property (or reference) must be key. The combination of key properties (and key references) must be mapped to a group

of database columns that do not have duplicate values, typically the primary key.

- (8)`hidden` (optional): A hidden property has a meaning for the developer but not for the user. The hidden properties are excluded when the automatic user interface is generated. However at Java code level they are present and fully functional. Even if you put it explicitly into a view the property will be shown in the user interface.
- (9)`valid-values` (one, optional): To indicate that this property only can have a limited set of valid values.
- (10)`calculator` (one, optional): Implements the logic for a calculated property. A calculated property only has *getter* and is not stored in database.
- (11)`default-value-calculator` (one, optional): Implements the logic to calculate the default (initial) value for this property. A property with `default-value-calculator` has *setter* and it is persistent.
- (12)`validator` (several, optional): Implements the validation logic to execute on this property before modifying or creating the object that contains it.

3.8.1 Stereotype

A stereotype is the way to determine a specific behavior of a type. For example, a name, a comment, a description, etc. all correspond to the Java type `java.lang.String` but you surely wish validators, default sizes, visual editors, etc. different in each case and you need to tune finer; you can do this assigning a stereotype to each case. That is, you can have the next stereotypes NAME, MEMO or DESCRIPTION and assign them to your properties.

OpenXava comes with these generic stereotypes:

- DINERO, MONEY
- FOTO, PHOTO, IMAGEN, IMAGE
- TEXTO_GRANDE, MEMO, TEXT_AREA
- ETIQUETA, LABEL
- ETIQUETA_NEGRITA, BOLD_LABEL
- HORA, TIME
- FECHAHORA, DATETIME
- GALERIA_IMAGENES, IMAGES_GALLERY (setup instructions in 3.8.2) *new in v2.0*
- RELLENADO_CON_CEROS, ZEROS_FILLED *new in v2.0.2*

Now you will learn how to define your own stereotype. You will create one called PERSON_NAME to represent names of persons.

Edit (or create) the file *editors.xml* in your folder *xava*. And add:

```
<editor url="personNameEditor.jsp">
  <for-stereotype stereotype="PERSON_NAME"/>
</editor>
```

This way you define the editor to render for editing and displaying properties of stereotype

PERSON_NAME.

Also you can edit *stereotype-type-default.xml* and the line:

```
<for stereotype="PERSON_NAME" type="String"/>
```

Furthermore it is useful to indicate the default size; you can do this by editing *default-size.xml* of your project:

```
<for-stereotype name="PERSON_NAME" size="40"/>
```

Thus, if you do not put the size in a property of type PERSON_NAME a value of 40 is assumed.

Not so common is changing the validator for `required`, but if you wish to change it you can do it adding to *validators.xml* of your project the next definition:

```
<required-validator>
  <validator-class class="org.openxava.validators.NotBlankCharacterValidator"/>
  <for-stereotype stereotype="PERSON_NAME"/>
</required-validator>
```

Now everything is ready to define properties of stereotype PERSON_NAME:

```
<property name="name" stereotype="PERSON_NAME" required="true"/>
```

In this case a value of 40 is assumed as size, String as type and the `NotBlankCharacterValidator` validator is executed to verify if it is required.

3.8.2 IMAGES_GALLERY stereotype (new in v2.0)

If you want that a property of your component hold a gallery of images. You only have to declare your property with the IMAGES_GALLERY stereotype, in this way:

```
<property name="photos" stereotype="IMAGES_GALLERY"/>
```

Furthermore, in the mapping part you have to map your property to a table column suitable to store a String with a length of 32 characters (`VARCHAR(32)`).

And everything is done.

In order to support this stereotype you need to setup the system appropriately for your application.

First, create a table in your database to store the images:

```
CREATE TABLE IMAGES (
  ID VARCHAR(32) NOT NULL PRIMARY KEY,
  GALLERY VARCHAR(32) NOT NULL,
  IMAGE BLOB);

CREATE INDEX IMAGES01
  ON IMAGES (GALLERY);
```

The type of `IMAGE` column can be a more suitable one for your database to store `byte []` (for example `LONGVARBINARY`).

The name of the table is arbitrary. You need to specify the table name in your configuration file (a `.properties` file in the root of your OpenXava project). In this way:

```
images.table=IMAGES
```

And finally you need to define the mapping in your `hibernate/hibernate.cfg.xml` file, thus:

```
<hibernate-configuration>
    <session-factory>
        ...
        <mapping resource="GalleryImage.hbm.xml"/>
        ...
    </session-factory>
</hibernate-configuration>
```

After this you can use the `IMAGES_GALLERY` stereotype in all components of your application.

3.8.3 Valid values

The element `<valid-values/>` allows you to define a property that can hold one of the indicated values only. Something like a C (or Java 5) `enum`.

It's easy to use, let's see this example:

```
<property name="distance">
    <valid-values>
        <valid-value value="local"/>
        <valid-value value="national"/>
        <valid-value value="international"/>
    </valid-values>
</property>
```

The `distance` property only can take the following values: `local`, `national` or `international`, and as you have not put `required="true"` the blank value is allowed too. The type is not necessary, `int` is assumed.

At user interface level the current implementation uses a combo. The label for each value is obtained from the *il8n* files.

At Java generated code level creates a `distance` property of type `int` that can take the values 0 (no value), 1 (local), 2 (national) o 3 (international).

At database level the value is by default saved as an integer, but you can configure easily to use another type and work with no problem with legat databases. See more about this in chapter 6.

3.8.4 Calculator

A calculator implements the logic to execute when the *getter* method of a calculated property is

called. The calculated properties are read only (only have *getter*) and not persistent (they do not match with any column of database table).

A calculated property is defined in this way:

```
<property name="unitPriceInPesetas" type="java.math.BigDecimal" size="18">
    <calculator class="org.openxava.test.calculators.EurosToPesetasCalculator">
        <set property="euros" from="unitPrice"/>
    </calculator>
</property>
```

Now when you (or OpenXava to fill the user interface) call to `getUnitPriceInPesetas()` the system executes `EurosToPesetasCalculator` calculator, but before this it sets the value of the property `euros` of `EurosToPesetasCalculator` with the value obtained from `unitPrice` of the current object.

Seeing the calculator code may be instructive:

```
package org.openxava.test.calculators;

import java.math.*;
import org.openxava.calculators.*;

/**
 * @author Javier Paniza
 */
public class EurosToPesetasCalculator implements ICalculator { // (1)

    private BigDecimal euros;

    public Object calculate() throws Exception { // (2)
        if (euros == null) return null;
        return euros.multiply(new BigDecimal("166.386")).
            setScale(0, BigDecimal.ROUND_HALF_UP);
    }

    public BigDecimal getEuros() {
        return euros;
    }

    public void setEuros(BigDecimal euros) {
        this.euros = euros;
    }

}
```

You can notice two things, first (1) a calculator must implement

org.openxava.calculators.ICalculator, and (2) the method `calculate()` executes the logic to generate the value returned by the property.

According to the above definitions now you can use the generated code in this way:

```
Product product = ...
product.setUnitPrice(2);
BigDecimal result = product.getUnitPriceInPesetas();
```

And `result` will hold 332.772.

You can define a calculator without `set from` to define values for properties, as shown below:

```
<property name="detailsCount" type="int" size="3">
  <calculator class="org.openxava.test.calculators.DetailsCountCalculator">
    <set property="year"/>
    <set property="number"/>
  </calculator>
</property>
```

In this case the property `year` and `number` of `DetailsCountCalculator` calculator are filled from properties of same name from the current object.

Also it's possible to assign a constant value to a calculator property:

```
<property name="fullName" type="String">
  <calculator class="org.openxava.calculators.ConcatCalculator">
    <set property="string1" from="id"/>
    <set property="separator" value=" - "/>
    <set property="string2" from="name"/>
  </calculator>
</property>
```

In this case the property `separator` of `ConcatCalculator` has a constant value.

Another interesting feature of calculator is that you can access from it to the model object (entity or aggregate) that contains the property that is being calculated:

```
<property name="amountsSum" stereotype="MONEY">
  <calculator class="org.openxava.test.calculators.AmountsSumCalculator"/>
</property>
```

And the calculator:

```
package org.openxava.test.calculators;

import java.math.*;
import java.rmi.*;
import java.util.*;
```



```

import javax.rmi.*;

import org.openxava.calculators.*;
import org.openxava.test.ejb.*;

/**
 * @author Javier Paniza
 */

public class AmountsSumCalculator implements IModelCalculator { // (1)

    private IInvoice invoice;

    public Object calculate() throws Exception {
        Iterator itDetails = invoice.getDetails().iterator();
        BigDecimal result = new BigDecimal(0);
        while (itDetails.hasNext()) {
            IInvoiceDetail detail = (IInvoiceDetail) itDetails.next();
            result = result.add(detail.getAmount());
        }
        return result;
    }

    public void setModel(Object model) throws RemoteException { // (2)
        invoice = (IInvoice) model;
    }

}

```

This calculator implements `IModelCalculator` (1) (*new in v2.0*) and to do this it has a method `setModel` (2), this method is called before calling the `calculate()` method and thus allows access to the model object (in this case an invoice) that contains the property inside `calculate()`.

Within the code generated by OpenXava you can find an interface for each business concept that is implemented by the POJO class, the EJB remote interface and the EJB Bean class. That is for Invoice you have a `IInvoice` interface implemented by `Invoice` (POJO class), `InvoiceRemote` (EJB remote interface) and `InvoiceBean` (EJB bean class), this last two only if you generate EJB code. In the calculator of type `IModelCalculator` it is advisable to cast to this interface, because in this cases the same calculator works with POJOs, EJB remote interface and EJB bean class. If you are developing with a POJO only version (maybe the normal case) you can cast directly to the POJO class, in this case `Invoice`.

This calculator type is less reusable than that which receives simple properties, but sometimes are useful. Why is it less reusable? For example, if you use `IInvoice` to calculate a discount, this

calculator only could be applied to invoices, but if you uses a calculator that receives `amount` and `discountPercentage` as simple properties this last calculator could be applied to invoices, deliveries, orders, etc.

From a calculator you have direct access to JDBC connections, here is an example:

```
<property name="detailsCount" type="int" size="3">
    <calculator class="org.openxava.test.calculators.DetailsCountCalculator">
        <set property="year"/>
        <set property="number"/>
    </calculator>
</property>
```

And the calculator class:

```
package org.openxava.test.calculators;

import java.sql.*;

import org.openxava.calculators.*;
import org.openxava.util.*;

/**
 * @author Javier Paniza
 */
public class DetailsCountCalculator implements IJBCCalculator { // (1)

    private IConnectionProvider provider;
    private int year;
    private int number;

    public void setConnectionProvider(IConnectionProvider provider) { // (2)
        this.provider = provider;
    }

    public Object calculate() throws Exception {
        Connection con = provider.getConnection();
        try {
            PreparedStatement ps = con.prepareStatement(
                "select count(*) from XAVATEST_INVOICEDetail " +
                "where INVOICE_YEAR = ? and INVOICE_NUMBER = ?");

            ps.setInt(1, getYear());
            ps.setInt(2, getNumber());
            ResultSet rs = ps.executeQuery();
```

```

        rs.next();
        Integer result = new Integer(rs.getInt(1));
        ps.close();
        return result;
    }
    finally {
        con.close();
    }
}

public int getYear() {
    return year;
}

public int getNumber() {
    return number;
}

public void setYear(int year) {
    this.year = year;
}

public void setNumber(int number) {
    this.number = number;
}
}

```

To use JDBC your calculator must implement `IJBCCalculator` (1) and then it will receive a `IConnectionProvider` (2) that you can use within `calculate()`. Yes, the JDBC code is ugly and awkward, but sometime it can help to solve performance problems.

The calculators allow you to insert your custom logic in a system where all code is generated; and as you see it promotes the creation of reusable code because the calculators nature (simple and configurable) allows you to use them time after time to define calculated properties and methods. This philosophy, simple and configurable classes that can be plugged in several places is the cornerstone that sustains all OpenXava framework.

OpenXava comes with a set of predefined calculators, you can find them in `org.openxava.calculators`.

3.8.5 Default value calculator

With `<default-value-calculator/>` you can associate logic to a property, but in this case the property is readable, writable and persistent. This calculator is for calculating its initial value. For example:

```
<property name="year" type="int" key="true" size="4" required="true">
    <default-value-calculator
        class="org.openxava.calculators.CurrentYearCalculator"/>
</property>
```

In this case when the user tries to create a new `Invoice` (for example) he will find that the `year` field already has a value, that he can change if he wants to do.

You can indicate that the value will be calculated just before creating (inserting into database) an object for the first time; this is done this way:

```
<property name="oid" type="String" key="true" hidden="true">
    <default-value-calculator
        class="org.openxava.calculators.UUIDCalculator"
        on-create="true"/>
</property>
```

If you use `on-create="true"` then you will obtain that effect.

A typical use of the `on-create="true"` is for generating identifiers automatically. In the above example, an unique identifier of type `String` and 32 characters is generated. Also you can use other generation techniques, for example, a database *sequence* can be defined in this way:

```
<property name="id" key="true" type="int" hidden="true">
    <default-value-calculator
        class="org.openxava.calculators.SequenceCalculator" on-create="true">
        <set property="sequence" value="XAVATEST_SIZE_ID_SEQ"/>
    </default-value-calculator>
</property>
```

Or maybe you want to use an *identity* (auto increment) column as key:

```
<property name="id" key="true" type="int" hidden="true">
    <default-value-calculator
        class="org.openxava.calculators.IdentityCalculator" on-create="true"/>
</property>
```

`SequenceCalculator` (new in v2.0.1) and `IdentityCalculator` (new in v2.0.2) do not work with EJB2. They work with Hibernate and EJB3.

If you define a hidden key property with no default calculator with `on-create="true"` then it uses *identity*, *sequence* or *hilo* techniques automatically depending upon the capabilities of the underlying database. As this:

```
<property name="oid" type="int" hidden="true" key="true"/>
```

Also, this only works with Hibernate and EJB3, not in EJB2.

All others issues about `<default-value-calculator/>` are as in `<calculator/>`.

3.8.6 Validator

The validator execute validation logic on the value assigned to the property just before storing. A property may have several validators.

```
<property name="description" type="String" size="40" required="true">
    <validator class="org.openxava.test.validators.ExcludeStringValidator">
        <set property="string" value="MOTO"/>
    </validator>
    <validator class="org.openxava.test.validators.ExcludeStringValidator"
        only-on-create="true">
        <set property="string" value="COCHE"/>
    </validator>
</property>
```

The technique to configure the validator (with `<set/>`) is exactly the same than in calculators. With the attribute `only-on-create="true"` you can define that the validation will be executed only when the object is created, and not when it is modified.

The validator code is:

```
package org.openxava.test.validators;

import org.openxava.util.*;
import org.openxava.validators.*;

/**
 * @author Javier Paniza
 */

public class ExcludeStringValidator implements IPropertyValidator { // (1)

    private String string;

    public void validate(
        Messages errors,           // (2)
        Object value,              // (3)
        String objectName,         // (4)
        String propertyName)       // (5)
        throws Exception {
        if (value==null) return;
        if (value.toString().indexOf(getString()) >= 0) {
            errors.add("exclude_string", propertyName, objectName, getString());
        }
    }
}
```

```

    public String getString() {
        return string==null?"":string;
    }

    public void setString(String string) {
        this.string = string;
    }
}

```

A validator has to implement `IPropertyValidator` (1), this obliges to the calculator to have a `validate()` method where the validation of property is executed. The arguments of `validate()` method are:

- (2)`Messages errors`: A object of type `Messages` that represents a set of messages (like a smart collection) and where you can add the validation errors that you find.
- (3)`Object value`: The value to validate.
- (4)`String objectName`: Object name of the container of the property to validate. Useful to use in error messages.
- (5)`String propertyName`: Name of the property to validate. Useful to use in error messages.

As you can see when you find a validation error you have to add it (with `errors.add()`) by sending a message identifier and the arguments. If you want to obtain a significant message you need to add to your *i18n* file the next entry:

```
exclude_string={0} cannot contain {2} in {1}
```

If the identifier sent is not found in the resource file, this identifier is shown as is; but the recommended way is always to use identifiers of resource files.

The validation is successful if no messages are added and fails if messages are added. OpenXava collects all messages of all validators before saving and if there are messages, then it display them and does not save the object.

The package `org.openxava.validators` contains some common validators.

3.9 Reference (5)

A reference allows access from an entity or an aggregate to another entity or aggregate. A reference is translated to Java code as a property (with its *getter* and its *setter*) whose type is the referenced model Java type. For example a `Customer` can have a reference to his `Seller`, and that allows you to write code like this:

```

ICustomer customer = ...
customer.getSeller().getName();

```

to access to the name of the seller of that customer.

The syntax of reference is:

```
<reference
    name="name"                (1)
    label="label"              (2)
    model="model"              (3)
    required="true|false"      (4)
    key="true|false"           (5)
    role="role"                (6)
>
    <default-value-calculator .../> (7)
</reference>
```

- (1) `name` (optional, required if `model` is not specified): The name of reference in Java, hence must follow the rules to name members in Java, including start by lower-case. If you do not specify `name` the model name with the first letter in lower-case is assumed. Using underline (`_`) is not advisable.
- (2) `label` (optional): Label shown to the final user. It's **much better** use *camel case*.
- (3) `model` (optional, required if `name` is not specified): The model name to reference. It can be the name of another component, in which case it is a reference to entity, or the name of a aggregate of the current component. If you do not specify `model` the reference name with the first letter in upper-case is assumed.
- (4) `required` (optional): Indicates if the reference is required. When saving OpenXava verifies if the required references are present, if not the saving is aborted and a list of validation errors is returned.
- (5) `key` (optional): Indicates if the reference is part of the key. The combination of key properties and reference properties should map to a group of database columns with unique values, typically the primary key.
- (6) `role` (optional): Used only in references within collections. See below.
- (7) `default-value-calculator` (one, optional): Implements the logic for calculating the initial value of the reference. This calculator must return the key value, that can be a simple value (only if the key of referenced object is simple) or key object (a special object that wraps the key and is generated by OpenXava).

A little example of references use:

```
<reference model="Address" required="true"/> (1)
<reference name="seller"/> (2)
<reference name="alternateSeller" model="Seller"/> (3)
```

- (1) A reference to an aggregate called `Address`, the reference name will be `address`.
- (2) A reference to the entity of `Seller` component. The model is deduced from `name`.
- (3) A reference called `alternateSeller` to the entity of component `Seller`.

If you assume that this is in a component named `Customer`, you could write:

```

ICustomer customer = ...
Address address = customer.getAddress();
ISeller seller = customer.getSeller();
ISeller alternateSeller = customer.getAlternateSeller();

```

3.9.1 Default value calculator in references

In a reference `<default-value-calculator/>` works like in a property, only that it has to return the value of the reference key, and `on-create="true"` is not allowed.

For example, in the case of a reference with simple key, you can write:

```

<reference name="family" model="Family2" required="true">
    <default-value-calculator class="org.openxava.calculators.IntegerCalculator">
        <set property="value" value="2"/>
    </default-value-calculator>
</reference>

```

The `calculate()` method is:

```

public Object calculate() throws Exception {
    return new Integer(value);
}

```

As you can see a integer is returned, that is, the default value for family is 2.

In the case of composed key:

```

<reference name="warehouse" model="Warehouse">
    <default-value-calculator
        class="org.openxava.test.calculators.DefaultWarehouseCalculator"/>
</reference>

```

And the calculator code:

```

package org.openxava.test.calculators;

import org.openxava.calculators.*;
import org.openxava.test.ejb.*;

/**
 * @author Javier Paniza
 */
public class DefaultWarehouseCalculator implements ICalculator {

    public Object calculate() throws Exception {
        Warehouse key = new Warehouse();
    }
}

```



```

        key.setNumber(4);
        key.setZoneNumber(4);
        return key; // This works with POJO and EJB
        // return new WarehouseKey(new Integer(4), 4); // This only work with EJB
    }

}

```

Returns an object of type `Warehouse`, (or `WarehouseKey` if you use only EJB).

3.10 Collection (6)

With `<collection/>` you define a collection of references to entities or aggregates. This is translated to Java as a property of type `java.util.Collection`.

Here syntax for collection:

```

<collection
    name="name"                (1)
    label="label"              (2)
    minimum="N"                (3)
>
    <reference ... />          (4)
    <condition ... />         (5)
    <order ... />             (6)
    <calculator ... />        (7)
    <postremove-calculator ... /> (8)
</collection>

```

- (1) `name` (required): The collection name in Java, therefore it must follow the rules for name members in Java, including starting with lower-case. Using underline (`_`) is not advisable.
- (2) `label` (optional): Label shown to final user. Is **much better** to use *il8n* files.
- (3) `minimum` (optional): Minimum number of expected elements. This is validated just before saving.
- (4) `reference` (required): With the syntax you can see in the previous point.
- (5) `condition` (optional): Restricts the elements that appear in the collection.
- (6) `order` (optional): The elements in collections will be in the indicated order.
- (7) `calculator` (optional): Allows you to define your own logic to generate the collection. If you use this, then you cannot use neither `condition` nor `order`.
- (8) `postremove-calculator` (optional): Execute your custom logic just after an element is removed from collection.

Let's have a look at some example. First a simple one:

```

<collection name="deliveries">

```

```
<reference model="Delivery"/>
</collection>
```

If you have this within an `Invoice`, then you are defining a `deliveries` collection associated to that `Invoice`. The details to make the relationship are defined in the object/relational mapping (more about this in chapter 6).

Now you can write a code like this:

```
IInvoice invoice = ...
for (Iterator it = invoice.getDeliveries().iterator(); it.hasNext();) {
    IDelivery delivery = (IDelivery) it.next();
    delivery.doSomething();
}
```

To do something with all deliveries associated to an invoice.

Let's look at another example a little more complex, but still in `Invoice`:

```
<collection name="details" minimum="1">           (1)
    <reference model="InvoiceDetail"/>
    <order>${serviceType} desc</order>             (2)
    <postremove-calculator                         (3)
        class="org.openxava.test.calculators.DetailPostremoveCalculator"/>
</collection>
```

In this case you have a collection of aggregates, the details (or lines) of the invoice. The main difference between collection of entities and collection of aggregates is when you remove the main entity; in the case of a collection of aggregates its elements are deleted too. That is when you delete an invoice its details are deleted too.

- (1)The restriction `minimum="1"` requires at least one detail for the invoice to be valid.
- (2)With `order` you force that the `details` will be returned ordered by `serviceType`.
- (3)With `postremove-calculator` you indicate the logic to execute just after a invoice detail is removed. Let's look at the calculator code:

```
package org.openxava.test.calculators;

import java.rmi.*;

import org.openxava.calculators.*;
import org.openxava.test.ejb.*;

/**
 * @author Javier Paniza
 */
```

```

public class DetailPostremoveCalculator implements IModelCalculator {

    private IInvoice invoice;

    public Object calculate() throws Exception {
        invoice.setComment(invoice.getComment() + "DETAIL DELETED");
        return null;
    }

    public void setEntity(Object model) throws RemoteException {
        this.invoice = (IInvoice) model;
    }

}

```

As you see this is a conventional calculator as it is used in calculated properties. A thing to consider is that the calculator is applied to the container entity (in this case `Invoice`) and not to the collection element. That is, if your calculator implements `IModelCalculator` then it receives an `Invoice` and not an `InvoiceDetail`. This is consistent because it is executed after the detail is removed and the detail doesn't exist any more.

You have full freedom to define how the collection data is obtained, with `condition` you can overwrite the default condition generated by OpenXava:

```

<!-- Others carriers of same warehouse -->
<collection name="fellowCarriers">
    <reference model="Carrier"/>
    <condition>
        ${warehouse.zoneNumber} = ${this.warehouse.zoneNumber} AND
        ${warehouse.number} = ${this.warehouse.number} AND
        NOT (${number} = ${this.number})
    </condition>
</collection>

```

If you have this collection within `Carrier`, you can obtain with this collection all carriers of the same warehouse but not himself, that is the list of his fellow workers. As you see you can use `this` in the condition in order to reference the value of a property of current object.

If with this you have not enough, you can write the logic that returns the collection. The previous example can be written in the following way too:

```

<!--
The same that 'fellowCarriers' but implemented with a calculator
-->
<collection name="fellowCarriersCalculated">
    <reference model="Carrier"/>

```

```
<calculator class="org.openxava.test.calculators.FellowCarriersCalculator"/>
</collection>
```

And here the calculator code:

```
package org.openxava.test.calculators;

import java.rmi.*;

import org.openxava.calculators.*;
import org.openxava.test.ejb.*;

/**
 * @author Javier Paniza
 */
public class FellowCarriersCalculator implements IModelCalculator {

    private ICarrier carrier;

    public Object calculate() throws Exception {
        // Using Hibernate
        int warehouseZoneNumber = carrier.getWarehouse().getZoneNumber();
        int warehouseNumber = carrier.getWarehouse().getNumber();
        Session session = XHibernate.getSession();
        Query query = session.createQuery("from Carrier as o where " +
            "o.warehouse.zoneNumber = :warehouseZone AND " +
            "o.warehouse.number = :warehouseNumber AND " +
            "NOT (o.number = :number)");
        query.setInteger("warehouseZone", warehouseZoneNumber);
        query.setInteger("warehouseNumber", warehouseNumber);
        query.setInteger("number", carrier.getNumber());
        return query.list();

        /* Using EJB
        return CarrierUtil.getHome().findFellowCarriersOfCarrier(
            carrier.getWarehouseKey().getZoneNumber(),
            carrier.getWarehouseKey().get_Number(),
            new Integer(carrier.getNumber())
        );
        */
    }

    public void setModel(Object model) throws RemoteException {
```

```

        carrier = (ICarrier) model;
    }

}

```

As you see this is a conventional calculator. Obviously it must return a `java.util.Collection` whose elements are of type `ICarrier`.

The references in collections are bidirectional, this means that if in a `Seller` you have a `customers` collection, then in `Customer` you must have a reference to `Seller`. But if in `Customer` you have more than one reference to `Seller` (for example, `seller` and `alternateSeller`) OpenXava does not know which to choose, for this case you have the attribute `role` of reference. You can use it in this way:

```

<collection name="customers">
    <reference model="Customer" role="seller"/>
</collection>

```

To indicate that the reference `seller` and not `alternateSeller` will be used in this collection.

In the case of a collection of entity references you have to define the reference at the other side, but in the case of a collection of aggregate references this is not necessary, because in the aggregates a reference to this container is automatically generated.

3.11 Method (7)

With `<method/>` you can define a method that will be included in the generated code as a Java method.

The syntax for method is:

```

<method
    name="name"                (1)
    type="type"                (2)
    arguments="arguments"      (3)
    exceptions="exceptions"    (4)
>
    <calculator ... />         (5)
</method>

```

- (1)`name` (required): Name of the method in Java, therefore it must follow the Java rules to name members, like beginning with lower-case.
- (2)`type` (optional, by default `void`): Is the Java type that the method returns. All Java types valid as return type for a Java method are applicable here.
- (3)`arguments` (optional): Argument list of the method in Java format.
- (4)`exceptions` (optional): Exception list that can be thrown by this method, in Java format.
- (5)`calculator` (required): Implements the logic of the method.

Defining a method is easy:

```
<method name="increasePrice">
    <calculator class="org.openxava.test.calculators.IncreasePriceCalculator"/>
</method>
```

And the implementation depends on the logic that you want to program. In this case:

```
package org.openxava.test.calculators;

import java.math.*;
import java.rmi.*;

import org.openxava.calculators.*;
import org.openxava.test.model.*;

/**
 * @author Javier Paniza
 */
public class IncreasePriceCalculator implements IModelCalculator {

    private IProduct product;

    public Object calculate() throws Exception {
        product.setUnitPrice(        // (1)
            product.getUnitPrice().
                multiply(new BigDecimal("1.02")).setScale(2));
        return null;                // (2)
    }

    public void setModel(Object model) throws RemoteException {
        this.product = (IProduct) model;
    }
}
```

All applicable things for calculators in properties are applicable to methods too, with the next clarifications:

- (1) A calculator for a method has moral authority to change the state of the object.
- (2) If the return type of the method is `void` the calculator must return `null`.

Now you can use the method in the expected way:

```
IProduct product = ...
```

```
product.setUnitPrice(new BigDecimal("100"));
product.increasePrice();
BigDecimal newPrice = product.getUnitPrice();
```

And in `newPrice` you have 102.

Another example, now a little bit more complex:

```
<method name="getPrice" type="BigDecimal"
    arguments="String country, BigDecimal tariff"
    exceptions="ProductException, PriceException">
    <calculator class="org.openxava.test.calculators.ExportPriceCalculator">
        <set property="euros" from="unitPrice"/>
    </calculator>
</method>
```

In this case you can notice that in arguments and exceptions the Java format is used, since what you put there is inserted directly into the generated code.

The calculator:

```
package org.openxava.test.calculators;

import java.math.*;

import org.openxava.calculators.*;
import org.openxava.test.ejb.*;

/**
 * @author Javier Paniza
 */

public class ExportPriceCalculator implements ICalculator {

    private BigDecimal euros;
    private String country;
    private BigDecimal tariff;

    public Object calculate() throws Exception {
        if ("España".equals(country) || "Guatemala".equals(country)) {
            return euros.add(tariff);
        }
        else {
            throw new PriceException("Country not registered");
        }
    }
}
```

```

    }

    public BigDecimal getEuros() {
        return euros;
    }

    public void setEuros(BigDecimal decimal) {
        euros = decimal;
    }

    public BigDecimal getTariff() {
        return tariff;
    }

    public void setTariff(BigDecimal decimal) {
        tariff = decimal;
    }

    public String getCountry() {
        return country;
    }

    public void setCountry(String string) {
        country = string;
    }

}

```

Each argument is assigned to a property of the name in the calculator; that is, the value of the first argument, `country`, is assigned to `country` property, and the value of the second one, `tariff`, to the `tariff` property. Of course, you can configure values for others calculator properties with `<set/>` as usual in calculators.

And to use the method:

```

IProduct product = ...
BigDecimal price = product.getPrice("España", new BigDecimal("100")); // works
product.getPrice("El Puig", new BigDecimal("100")); // throws PriceException

```

Methods are the sauce of the objects, without them the object only would be a silly wrapper of data. When possible it is better to put the business logic in methods (model layer) instead of in actions (controller layer).

3.12 Finder (8)

A finder is a special method that allows you to find an object or a collection of objects that follow some criteria. In POJO + Hibernate version a finder method is a generated static method in the POJO class. In the EJB version a finder matches with a *finder* in *home*.

The syntax for finder is:


```

<finder
    name="name"                (1)
    arguments="arguments"      (2)
    collection="(true|false)"  (3)
>
    <condition ... />          (4)
    <order ... />              (5)
</finder>

```

- (1) **name** (required): Name of the finder method in Java, hence it must follow the Java rules for member naming, e.g. beginning with lower-case.
- (2) **arguments** (required): Argument list for the method in Java format. It is (most) advisable to use simple data types.
- (3) **collection** (optional, by default `false`): Indicates if the result will be a single object or a collection.
- (4) **condition** (optional): A condition with SQL/EJBQL syntax where you can use the names of properties inside `${}`.
- (5) **order** (optional): An order with SQL/EJBQL syntax where you can use the names of properties inside `${}`.

Some examples:

```

<finder name="byNumber" arguments="int number">
    <condition>${number} = {0}</condition>
</finder>

<finder name="byNameLike" arguments="String name" collection="true">
    <condition>${name} like {0}</condition>
    <order>${name} desc</order>
</finder>

<finder
    name="byNameLikeAndRelationWithSeller"
    arguments="String name, String relationWithSeller"
    collection="true">
    <condition>${name} like {0} and ${relationWithSeller} = {1}</condition>
    <order>${name} desc</order>
</finder>

<finder name="normalOnes" arguments="" collection="true">
    <condition>${type} = 1</condition>
</finder>

```

```

<finder name="steadyOnes" arguments="" collection="true">
    <condition>${type} = 2</condition>
</finder>

<finder name="all" arguments="" collection="true"/>

```

This generates a set of *finder* methods available from POJO class and EJB home. This methods can be used this way:

```

// POJO
ICustomer customer = Customer.findByNumber(8);
Collection javieres = Customer.findByNameLike("%JAVI%");

// EJB
ICustomer customer = CustomerUtil.getHome().findByNumber(8);
Collection javieres = CustomerUtil.getHome().findByNameLike("%JAVI%");

```

3.13 Postcreate calculator (9)

With `<postcreate-calculator/>` you can plug in your own logic to execute just after creating the object as persistent object.

Its syntax is:

```

<postcreate-calculator
    class="class">                (1)
    <set ... /> ...                (2)
</postcreate-calculator>

```

(1)class (required): Calculator class. This calculator must implement `ICalculator` or some of its children.

(2)set (several, optional): To set the value of the calculator properties before executing it.

A simple example is:

```

<postcreate-calculator
    class="org.openxava.test.calculators.DeliveryTypePostcreateCalculator">
    <set property="suffix" value="CREATED"/>
</postcreate-calculator>

```

And now the calculator class:

```

package org.openxava.test.calculators;

import java.rmi.*;

```

```

import org.openxava.calculators.*;
import org.openxava.test.ejb.*;

/**
 * @author Javier Paniza
 */
public class DeliveryTypePostcreateCalculator implements IModelCalculator {

    private IDeliveryType deliveryType;
    private String suffix;

    public Object calculate() throws Exception {
        deliveryType.setDescription(deliveryType.getDescription() + " " + suffix);
        return null;
    }

    public void setModel(Object model) throws RemoteException {
        deliveryType = (IDeliveryType) model;
    }

    public String getSuffix() {
        return suffix;
    }

    public void setSuffix(String suffix) {
        this.suffix = suffix;
    }

}

```

In this case each time that a `DeliveryType` is created, just after it, a suffix to description is added.

As you see, this is exactly the same as in others calculator (as calculated properties or method) but is executed just after creation.

3.14 Postmodify calculator (11)

With `<postmodify-calculator/>` you can plug in some logic to execute after the state of the object is changed and just before it is stored in the database, that is, just before executing UPDATE against database.

Its syntax is:

```

<postmodify-calculator
    class="class">                (1)
    <set ... /> ...                (2)

```

```
</postmodify-calculator>
```

(3)class (required): Calculator class. A calculator that implements `ICalculator` or some of its children.

(4)set (several, optional): To set the value of the calculator properties before execute it.

A simple example is:

```
<postmodify-calculator  
    class="org.openxava.test.calculators.DeliveryTypePostmodifyCalculator"/>
```

And now the calculator class:

```
package org.openxava.test.calculators;  
  
import java.rmi.*;  
  
import org.openxava.calculators.*;  
import org.openxava.test.ejb.*;  
  
/**  
 * @author Javier Paniza  
 */  
  
public class DeliveryTypePostmodifyCalculator implements IModelCalculator {  
  
    private IDeliveryType deliveryType;  
  
    public Object calculate() throws Exception {  
        deliveryType.setDescription(deliveryType.getDescription() + " MODIFIED");  
        return null;  
    }  
  
    public void setModel(Object model) throws RemoteException {  
        deliveryType = (IDeliveryType) model;  
    }  
  
}
```

In this case whenever that a `DeliveryType` is modified a suffix is added to its description.

As you see, this is exactly the same as in other calculators (as calculated properties or methods), but it is executed just after modifying.

3.15 Postload and preremove calculator (10, 12)

The syntax and behavior of postload and preremove calculators are the same of the postcreate and postmodify ones.

3.16 Validator (13)

This validator allows to define a validation at model level. When you need to make a validation on several properties at a time, and that validation does not correspond logically with any of them, then you can use this type of validation.

Its syntax is:

```
<validator
  class="class"                (1)
  name="name"                  (2)
  only-on-create="true|false" (3)
>
  <set ... /> ...              (4)
</validator>
```

- (1) `class` (optional, required if `name` is not specified): Class that implements the validation logic. It has to be of type `IValidator`.
- (2) `name` (optional, required if `class` is not specified): This name is a validator name from `xava/validators.xml` file of your project or of the OpenXava project.
- (3) `only-on-create` (optional): If `true` the validator is executed only when creating a new object, not when an existing object is modified. The default value is `false`.
- (4) `set` (several, optional): To set a value of the validator properties before executing it.

An example:

```
<validator class="org.openxava.test.validators.CheapProductValidator">
  <set property="limit" value="100"/>
  <set property="description"/>
  <set property="unitPrice"/>
</validator>
```

And the validator code:

```
package org.openxava.test.validators;

import java.math.*;

import org.openxava.util.*;
import org.openxava.validators.*;

/**
```

```

* @author Javier Paniza
*/
public class CheapProductValidator implements IValidator { // (1)

    private int limit;
    private BigDecimal unitPrice;
    private String description;

    public void validate(Messages errors) { // (2)
        if (getDescription().indexOf("CHEAP") >= 0
            getDescription().indexOf("BARATO") >= 0
            getDescription().indexOf("BARATA") >= 0) {
            if (getLimitBd().compareTo(getUnitPrice()) < 0) {
                errors.add("cheap_product", getLimitBd()); // (3)
            }
        }
    }

    public BigDecimal getUnitPrice() {
        return unitPrice;
    }

    public void setUnitPrice(BigDecimal decimal) {
        unitPrice = decimal;
    }

    public String getDescription() {
        return description==null?"":description;
    }

    public void setDescription(String string) {
        description = string;
    }

    public int getLimit() {
        return limit;
    }

    public void setLimit(int i) {
        limit = i;
    }
}

```

```

        private BigDecimal getLimitBd() {
            return new BigDecimal(limit);
        }
    }
}

```

This validator must implement `IValidator` (1), this forces you to write a `validate(Messages messages)` (2). In this method you add the error message ids (3) (whose texts are in the *il8n* files). And if the validation process (that is the execution of all validators) produces some error, then OpenXava does not save the object and displays the errors to the user.

In this case you see how `description` and `unitPrice` properties are used to validate, for that reason the validation is at model level and not at individual property level, because the scope of validation is more than one property.

3.17 Remove validator (14)

The `<remove-validator/>` is a level model validator too, but in this case it is executed just before removing an object, and it has the possibility to deny the deletion.

Its syntax is:

```

<remove-validator
    class="validator"           (1)
    name="name"                (2)
>
    <set ... /> ...            (3)
</remove-validator>

```

(1) `class` (optional, required if `name` is not specified): Class that implements the validation logic. Must implement `IRemoveValidator`.

(2) `name` (optional, required if `class` is not specified): This name is a validator name from the `xava/validators.xml` file of your project or from the OpenXava project.

(3) `set` (several, optional): To set the value of the validator properties before executing it.

An example can be:

```

<remove-validator
    class="org.openxava.test.validators.DeliveryTypeRemoveValidator"/>

```

And the validator:

```

package org.openxava.test.validators;

import java.util.*;

import org.openxava.test.ejb.*;
import org.openxava.util.*;

```

```

import org.openxava.validators.*;

/**
 * @author Javier Paniza
 */
public class DeliveryTypeRemoveValidator implements IRemoveValidator {           // (1)

    private IDeliveryType deliveryType;

    public void setEntity(Object entity) throws Exception {                       // (2)
        this.deliveryType = (IDeliveryType) entity;
    }

    public void validate(Messages errors) throws Exception {
        if (!deliveryType.getDeliveries().isEmpty()) {
            errors.add("not_remove_delivery_type_if_in_deliveries");           // (3)
        }
    }
}

```

As you see this validator must implement `IRemoveValidator` (1) this forces you to write a `setEntity()` (2) method that receives the object to remove. If validation error is added to the `Messages` object sent to `validate()` (3) the validation fails. If after executing all validations there are validation errors, then OpenXava does not remove the object and displays a list of validation messages to the user.

In this case it verifies if there are deliveries that use this delivery type before deleting it.

3.18 Aggregate

The aggregate syntax is:

```

<aggregate name="aggregate">           (1)
    <bean class="beanClass"/>           (2)
    <ejb ... />                         (3)
    <implements .../>
    <property .../> ...
    <reference .../> ...
    <collection .../> ...
    <method .../> ...
    <finder .../> ...
    <postcreate-calculator .../> ...
    <postmodify-calculator .../> ...
    <validator .../> ...

```



```
<remove-validator .../> ...  
</aggregate>
```

- (1)**name** (required): Each aggregate must have a unique name. The rules for this name are the same that for class names in Java, that is, to begin with upper-case and each new word starting with upper-case too.
- (2)**bean** (one, optional): Allows to specify a class written by you to implement the aggregate. The class has to be a JavaBean, that is a plain Java class with *getters* and *setters* for properties. Usually this is not used because it is much better that OpenXava generates the code for you.
- (3)**ejb** (one, optional): Allows to use existing EJB to implement an aggregate. This can be used only in the case of a collection of aggregates. Usually this is not used because it is much better that OpenXava generates the code for you.

An OpenXava component can have whichever aggregates you want. And you can reference it from the main entity or from another aggregate.

3.18.1 Reference to aggregate

The first example is an aggregate `Address` that is referenced from the main entity.

In the main entity you can write:

```
<reference name="address" model="Address" required="true"/>
```

And in the component level you define:

```
<aggregate name="Address">  
    <implements interface="org.openxava.test.ejb.IWithCity"/>           (1)  
    <property name="street" type="String" size="30" required="true"/>  
    <property name="zipCode" type="int" size="5" required="true"/>  
    <property name="city" type="String" size="20" required="true"/>  
    <reference name="state" required="true"/>                             (2)  
</aggregate>
```

As you see an aggregate can implement an interface (1) and contain references (2), among other things, in fact all thing that you can use in `<entity/>` are supported in an aggregate.

The resulting code can be used this way, for reading:

```
ICustomer customer = ...  
Address address = customer.getAddress();  
address.getStreet(); // to obtain the value
```

Or in this other way to set a new address:

```
// to set a new address  
Address address = new Address(); // it's a JavaBean, never an EJB  
address.setStreet("My street");
```

```
address.setZipCode(46001);
address.setCity("Valencia");
address.setState(state);
customer.setAddress(address);
```

In this case you have a simple reference (not collection), and the generated code is a simple `JavaBean`, whose life cycle is associated to its container object, that is, the `Address` is removed and created through the `Customer`. An `Address` never will have its own life and cannot be shared by other `Customer`.

3.18.2 Collection of aggregates

Now an example of a collection of aggregates. In the main entity (for example `Invoice`) you can write:

```
<collection name="details" minimum="1">
    <reference model="InvoiceDetail"/>
</collection>
```

And define the `InvoiceDetail` aggregate:

```
<aggregate name="InvoiceDetail">
    <property name="oid" type="String" key="true" hidden="true">
        <default-value-calculator
            class="org.openxava.test.calculators.InvoiceDetailOidCalculator"
            on-create="true"/>
    </property>
    <property name="serviceType">
        <valid-values>
            <valid-value value="special"/>
            <valid-value value="urgent"/>
        </valid-values>
    </property>
    <property name="quantity" type="int">
        size="4" required="true"/>
    <property name="unitPrice">
        stereotype="MONEY" required="true"/>
    <property name="amount">
        stereotype="MONEY">
            <calculator
                class="org.openxava.test.calculators.DetailAmountCalculator">
                    <set property="unitPrice"/>
                    <set property="quantity"/>
                </calculator>
            </property>
```

```

<reference model="Product" required="true"/>
<property name="deliveryDate" type="java.util.Date">
  <default-value-calculator
    class="org.openxava.calculators.CurrentDateCalculator"/>
</property>
<reference name="soldBy" model="Seller"/>
<property name="remarks" stereotype="MEMO"/>

<validator class="org.openxava.test.validators.InvoiceDetailValidator">
  <set property="invoice"/>
  <set property="oid"/>
  <set property="product"/>
  <set property="unitPrice"/>
</validator>

</aggregate>

```

As you see an aggregate is as complex as an entity, with calculators, validators, references and so on. In the case of an aggregate used in a collection a reference to the container is added automatically, that is, although you have not defined it, `InvoiceDetail` has a reference to `Invoice`.

In the generated code you can find an `Invoice` with a collection of `InvoiceDetail`. The difference between a collection of references and a collection of aggregates is that when you remove a `Invoice` its details are removed too (because they are aggregates). Also there are differences at user interface level (you can learn more on this in chapter 4).

OpenXava generates a default user interface from the model. In many simple cases this is enough, but sometimes it is necessary to model with precision the format of the user interface or view. In this chapter you will learn how to do this.

The syntax for view is:

```
<view
  name="name"                (1)
  label="label"              (2)
  model="model"              (3)
  members="members"         (4)
>
  <property ... /> ...        (5)
  <property-view ... /> ...   (6)
  <reference-view ... /> ...  (7)
  <collection-view ... /> ... (8)
  <members ... /> ...        (9)
</view>
```

- (1) `name` (optional): This name identifies the view, and can be used in other OpenXava places (for example in *application.xml*) or from another component. If the view has no name then the view is assumed as the default one, that is the natural form to display an object of this type.
- (2) `label` (optional): The label that is showed to the user, if needed, when the view is displayed. It's **much better** use the *i18n* files.
- (3) `model` (optional): If the view is for an aggregate of this component you need to specify here the name of that aggregate. If `model` is not specified then this view is for the main entity.
- (4) `members` (optional): List of members to show. By default it displays all members (excluding hidden ones) in the order in which are declared in the model. This attribute is mutually exclusive with the `members` element (that you will see below).
- (5) `property` (several, optional): Defines a property of the view, that is, information that can be displayed to the user and the programmer can work programmatically with it, but it is not a part of the model.
- (6) `property-view` (several, optional): Defines the format to display a property.
- (7) `reference-view` (several, optional): Defines the format to display a reference.
- (8) `collection-view` (several, optional): Defines the format to display a collection.
- (9) `members` (one, optional): Indicates the members to display and its layout in the user interface. Is mutually exclusive with the `members` attribute.

4.1 Layout

By default (if you do not use `<view/>`) all members are displayed in the order of the model, and one for each line.

For example, a model like this:

```
<entity>
  <property name="zoneNumber" key="true"
    size="3" required="true" type="int"/>
  <property name="officeNumber" key="true"
    size="3" required="true" type="int"/>
  <property name="number" key="true"
    size="3" required="true" type="int"/>
  <property name="name" type="String"
    size="40" required="true"/>
</entity>
```

Generates a view that looks like this:



The form displays four fields. The first three are labeled 'Zone', 'Office', and 'Number', each with a key icon and a numeric input box containing the value '1'. The fourth field is labeled 'Name' with a text icon and a text input box containing the value 'PEPE'.

You can choose the members to display and its order, with the `members` attribute:

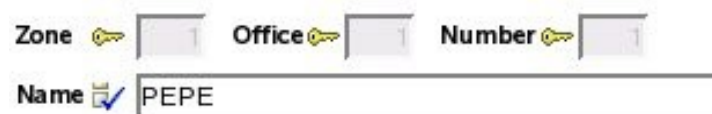
```
<view members="zoneNumber; officeNumber; number"/>
```

In this case `name` is not shown.

The members also can be specified using the `members` element, that is mutually exclusive with the `members` attribute, thus:

```
<view>
  <members>
    zoneNumber, officeNumber, number;
    name
  </members>
</view>
```

You can observe that the member names are separated by commas or by semicolon, this is used to indicate layout. With comma the member is placed just the following (at right), and with semicolon the next member is put below (in the next line). Hence the previous view is displayed in this way:



The form displays four fields. The first three are labeled 'Zone', 'Office', and 'Number', each with a key icon and a numeric input box containing the value '1'. The fourth field is labeled 'Name' with a text icon and a text input box containing the value 'PEPE'.

With groups you can lump a set of related properties and it has this visual effect:

```
<view>
  <members>
    <group name="id">
      zoneNumber, officeNumber, number
    </group>
    ; name
  </members>
</view>
```

In this case the result is:

The image shows a user interface for a form. At the top, there is a section titled 'Id' in a grey header. Below this header, three properties are displayed: 'Zone' with a key icon and a value of '1', 'Office' with a key icon and a value of '1', and 'Number' with a key icon and a value of '1'. These three properties are grouped together within a single frame. Below this frame, there is another property 'Name' with a checkmark icon and the value 'PEPE'.

You can see the three properties within the group are displayed inside a frame, and `name` is displayed outside this frame. The semicolon before `name` causes it to appear below, if not it appears at right.

You can put several groups in a view:

```
<group name="customer">
  type;
  name;
</group>
<group name="seller">
  seller;
  relationWithSeller;
</group>
```

In this case the groups are shown one next to the other:

The image shows a user interface with two side-by-side form sections. The left section is titled 'Customer' in a grey header. It contains two properties: 'Type' with a checkmark icon and a dropdown menu showing 'Normal', and 'Name' with a checkmark icon and an empty text field. The right section is titled 'Seller' in a grey header. It contains three properties: 'Number' with a key icon and an empty text field, 'Name' with a checkmark icon and an empty text field, and 'Relation with seller' with a text field containing the value 'GOOD'. There is also an 'Add' button next to the 'Number' field.

If you want one below the other then you must use a semicolon after the group name, like this:

```
<group name="customer">
  type;
  name;
```

```

</group>;
<group name="seller">
    seller;
    relationWithSeller;
</group>

```

In this case the view is shown this way:

Nested groups are allowed. This is a pretty feature that allows you to layout the elements of the user interface in a flexible and simple way. For example, you can define a view as this:

```

<members>
    invoice;
    <group name="deliveryData">
        type, number;
        date;
        description;
        shipment;
        <group name="transportData">
            distance; vehicle; transportMode; driverType;
        </group>
        <group name="deliveryByData">
            deliveredBy;
            carrier;
            employee;
        </group>
    </group>
</members>

```

And the result will be:

Invoice

Year

Number

Date

Year discount

€

Delivery data

Type

Number

Generate

Date

23/08/2005

Description

Shipment

Transport ata

Distance

Vehicle

Transport mode

Driver type

X

'Delivery by' data

Delivered by

Furthermore the members can be organized in sections. Let's see an example from the `Invoice` component:

```

<view>
  <members>
    year, number, date, paid;
    customerDiscount, customerTypeDiscount, yearDiscount;
    comment;

    <section name="customer">customer</section>
    <section name="details">details</section>
    <section name="amounts">amountsSum; vatPercentage; vat</section>
    <section name="deliveries">deliveries</section>
  </members>
</view>

```

The visual result is:

Year Number Date Paid ☐

Customer discount € Customer type discount € Year discount €

Comment

Seller Details Amounts Deliveries

Little code [Add](#)

Type

Name

Address

ViewProperty

Street Zip code State

The sections are rendered as tabs that the user can click to see the data contained in that section. You can observe how in the view you put members of all types (not only properties); thus, `customer` is a reference, `details` is a collection of aggregates and `deliveries` is a collection of entities.

Nested sections are allowed (*new in v2.0*). For example, you can define a view as this:

```
<view name="NestedSections">
  <members>
    year, number
    <section name="customer">customer</section>
    <section name="data">
      <section name="details">details</section>
      <section name="amounts">
        <section name="vat">vatPercentage; vat</section>
        <section name="amountsSum">amountsSum</section>
      </section>
    </section>
    <section name="deliveries">deliveries</section>
  </members>
</view>
```

In this case you will obtain a user interface like this:

Year Number

Customer Data Deliveries

Details Amounts

V.A.T. Amounts sum

VAT %

V.A.T. €

It's worth to notice that you have groups instead of frames and sections instead of tabs. Because OpenXava tries to maintain a high level of abstraction, that is, a group is a set of members semantically related, and the sections allow to split the data into parts. This is useful, if there is a big amount of data that cannot be displayed simultaneously. The fact that the group is displayed as frames or sections in a tabbed pane is only an implementation issue. For example, OpenXava (maybe in future) can choose to display sections (for example) with trees or so.

4.2 Property view

With `<property-view/>` you can refine the visual aspect and behavior of a property in a view:

It has this syntax:

```
<property-view
  property="propertyName"           (1)
  label="label"                     (2)
  read-only="true|false"           (3)
  label-format="NORMAL|SMALL|NO_LABEL" (4)
>
  <on-change ... />                 (5)
  <action ... /> ...                 (6)
</property-view>
```

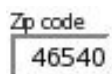
- (1) `property` (required): Usually the name of a model property, although it also can be the name of a property of the view itself.
- (2) `label` (optional): Modifies the label for this property in this view. To achieve this it is **much better** use the *i18n* files.
- (3) `read-only` (optional): If you set this property to `true` it never will be editable by the final user in this view. An alternative to this is to make the property editable or not editable programmatically using `org.openxava.view.View`.
- (4) `label-format` (optional): Format to display the label of this property.
- (5) `on-change` (one, optional): Action to execute when the value of this property changes.
- (6) `action` (several, optional): Actions (showed as links, buttons or images to the user) associated (visually) to this property and that the final user can execute.

4.2.1 Label format

A simple example of using label format:

```
<view model="Address">
  <property-view property="zipCode" label-format="SMALL"/>
</view>
```

In this case the zip code is displayed as:



The `NORMAL` format is the default style (with a normal label at the left) and the `NO_LABEL` simply does not display the label.

4.2.2 Value change event

If you wish to react to the event of a value change of a property you can write:

```
<property-view property="carrier.number">
    <on-change class="org.openxava.test.actions.OnChangeCarrierInDeliveryAction"/>
</property-view>
```

You can see how the property can be qualified, that is in this case your action listens to the change of carrier number (carrier is a reference).

The code to execute is:

```
package org.openxava.test.actions;

import org.openxava.actions.*;

/**
 * @author Javier Paniza
 */
public class OnChangeCarrierInDeliveryAction
    extends OnChangePropertyBaseAction { // (1)

    public void execute() throws Exception {
        if (getNewValue() == null) return; // (2)
        getView().setValue("remarks", "The carrier is " + getNewValue()); // (3)
        addMessage("carrier_changed");
    }
}
```

The action has to implement `IONChangePropertyAction` although it is more convenient to extend it from `OnChangePropertyBaseAction` (1). Within the action you can use `getNewValue()` (2) that provides the new value entered by user, and `getView()` (3) that allows you to access programmatically the view (change values, hide members, make them editable and so on).

4.2.3 Actions of property

You can also specify actions that the user can click directly:

```
<property-view property="number">
    <action action="Deliveries.generateNumber"/>
</property-view>
```

In this case instead of an action class you have to write the action identifier that is the controller

name and the action name. This action must be registered in *controllers.xml* in this way:

```
<controller name="Deliveries">
    ...
    <action name="generateNumber" hidden="true"
            class="org.openxava.test.actions.GenerateDeliveryNumberAction">
        <use-object name="xava_view"/>
    </action>
    ...
</controller>
```

The actions are displayed as a link or an image beside the property. Like this:

Number  [Generate](#)

The action link is present only when the property is editable, but if the property is read-only or calculated then it is always present.

The code of previous action is:

```
package org.openxava.test.actions;

import org.openxava.actions.*;

/**
 * @author Javier Paniza
 */
public class GenerateDeliveryNumberAction extends ViewBaseAction {

    public void execute() throws Exception {
        getView().setValue("number", new Integer(77));
    }

}
```

A simple but illustrative implementation. You can use any action defined in *controllers.xml* and its behavior is the normal for an OpenXava action. In the chapter 7 you will earn more details about actions.

Optionally you can make your action an *IPropertyAction* (*new in v2.0.2*) (this is available for actions used in `<property-view/>` only), thus the container view and the property name are injected in the action by OpenXava. The above action class could be rewritten in this way:

```
package org.openxava.test.actions;

import org.openxava.actions.*;
import org.openxava.view.*;
```

```

/**
 * @author Javier Paniza
 */
public class GenerateDeliveryNumberAction
    extends BaseAction
    implements IPropertyAction { // (1)
    private View view;
    private String property;

    public void execute() throws Exception {
        view.setValue(property, new Integer(77)); // (2)
    }

    public void setProperty(String property) { // (3)
        this.property = property;
    }

    public void setView(View view) { // (4)
        this.view = view;
    }
}

```

This action implements `IPropertyAction` (1), this required that the class implements `setProperty()` (3) and `setView()` (4), these values are injected in the action object before call to `execute()` method, where they can be used (2). In this case you does not need to inject `xava_view` object when defining the action in *controllers.xml*. The view injected by `setView()` (4) is the inner view that contains the property, for example, if the property is inside an aggregate the view is the view of that aggregate not the main view of the module. Thus, you can write more reusable actions.

4.3 Reference view

With `<reference-view/>` you can modify the format for displaying references.

Its syntax is:

```

<reference-view
    reference="reference"           (1)
    view="view"                   (2)
    read-only="true|false"        (3)
    frame="true|false"            (4)
    create="true|false"           (5)
    search="true|false"           (6)
>

<search-action ... />           (7)
<descriptions-list ... />      (8)

```

```

    <action ... /> ... (9) // new in v2.0.1
</reference-view>

```

- (1) **reference** (required): Name of the reference to refine its presentation.
- (2) **view** (optional): If you omit this attribute, then the default view of the referenced object is used. With this attribute you can indicate that it uses another view.
- (3) **read-only** (optional): If you set the value to `true` the reference never will be editable by final user in this view. An alternative is to make the property editable/uneditable programmatically using `org.openxava.view.View`.
- (4) **frame** (optional): If the reference is displayed inside a frame. The default value is `true`.
- (5) **create** (optional): If the final user can create new objects of the referenced type from here. The default value is `true`.
- (6) **search** (optional): If the user will have a link to make searches with a list, filters, etc. The default value is `true`.
- (7) **search-action** (one, optional): Allows you to specify your own action for searching.
- (8) **descriptions-list**: Display the data as a list of descriptions, typically as a combo. Useful when there are few elements of the referenced object.
- (9) **action** (several, optional): (*new in v2.0.1*) Actions (showed as links, buttons or images to the user) associated (visually) to this reference and that the final user can execute. Works as in `<property-view/>` case, look at section 4.2.3.

If you do not use `<reference-view/>` OpenXava draws a reference using the default view. For example, if you have a reference like this:

```

<entity>
...
    <reference name="family" model="Family" required="true"/>
...
</entity>

```

The user interface will look like this:

4.3.1 Choose view

The most simple customization is to specify the view of the referenced object that you want to use:

```

<reference-view reference="invoice" view="Simple"/>

```

In the `Invoice` component you must have a view named `Simple`:

```
<component name="Invoice">
...
  <view name="Simple">
    <members>
      year, number, date, yearDiscount;
    </members>
  </view>
...
</component>
```

Thus, instead of using the default view of `Invoice` (that shows all invoice data) OpenXava will use the next one:



Invoice							
Year	 2002	Number	 1	Date	 01/01/2002	Year discount	 200 €

4.3.2 Customizing frame

If you combine `frame="true"` with `group` you can group visually a property that is not a part of a reference with that reference, for example:

```
<reference-view reference="seller" frame="false"/>
<members>
...
  <group name="seller">
    seller;
    relationWithSeller;
  </group>
...
</members>
```

And the result:

Seller	
Number	 <input type="text"/>  Add
Name	 <input type="text"/>
Relation with seller	<input type="text" value="GOOD"/>

4.3.3 Custom search action

The final user can search a new value for the reference simply by keying the new code and leaving the editor the data of reference is obtained; for example, if the user keys “1” on the seller number field, then the name (and the other data) of the seller “1” will be automatically filled. Also the user can click in the lantern, in this case the user will go to a list where he can filter, order, etc, and mark the wished object.

To define your custom search logic you have to use `<search-action/>` in this way:

```
<reference-view reference="seller">
  <search-action action="MyReference.search"/>
</reference-view>
```

When the user clicks in the lantern your action is executed, which must be defined in *controllers.xml*.

```
<controller name="MyReference">
  <action name="search" hidden="true">
```



```

        class="org.openxava.test.actions.MySearchAction"
        image="images/search.gif">
        <use-object name="xava_view"/>
        <use-object name="xava_referenceSubview"/>
        <use-object name="xava_tab"/>
        <use-object name="xava_currentReferenceLabel"/>

    </action>
    ...
</controller>

```

The logic of your `MySearchAction` is up to you. You can, for example, refining the standard search action to filter the list for searching, as follows:

```

package org.openxava.test.actions;

import org.openxava.actions.*;

/**
 * @author Javier Paniza
 */

public class MySearchAction extends ReferenceSearchAction {

    public void execute() throws Exception {
        super.execute();    // The standard search behaviour
        getTab().setBaseCondition("${number} < 3"); // Adding a filter to the list
    }

}

```

You will learn more about actions in chapter 7.

4.3.4 Custom creation action

If you do not write `create="false"` the user will have a link to create a new object. By default when a user clicks on this link, a default view of the referenced object is displayed and the final user can type values and click a button to create it. If you want to define your custom actions (among them your `create` custom action) in the form used when creating a new object, you must have a controller named as component but with the suffix `Creation`. If OpenXava see this controller it uses it instead of the default one to allow creating a new object from a reference. For example, you can write in your *controllers.xml*:

```

<!--
Because its name is Warehouse2Creation (model name + Creation) it is used
by default for create from reference, instead of NewCreation.

```

Action 'new' is executed automatically.

-->

```
<controller name="Warehouse2Creation">
  <extends controller="NewCreation"/>
  <action name="new" hidden="true"
    class="org.openxava.test.actions.CreateNewWarehouseFromReferenceAction"
    image="images/new.gif"
    keystroke="F2">
    <use-object name="xava_view"/>
  </action>
</controller>
```

In this case when the user clicks on the 'create' link, the user is directed to the default view of Warehouse2 and the actions in Warehouse2Creation will be allowed.

If you have an action called 'new', it will be executed automatically before all. It can be used to initialize the view used to create a new object.

4.3.5 Descriptions list (combos)

With <descriptions-list/> you can instruct OpenXava to visualize references as a descriptions list (actually a combo). This can be useful, if there are only a few elements and these elements have a significant name or description.

The syntax is:

```
<descriptions-list
  description-property="property"           (1)
  description-properties="properties"       (2)
  depends="depends"                         (3)
  condition="condition"                    (4)
  order-by-key="true|false"                (5)
  order="order"                            (6)
  label-format="NORMAL|SMALL|NO_LABEL"     (7)
/>
```

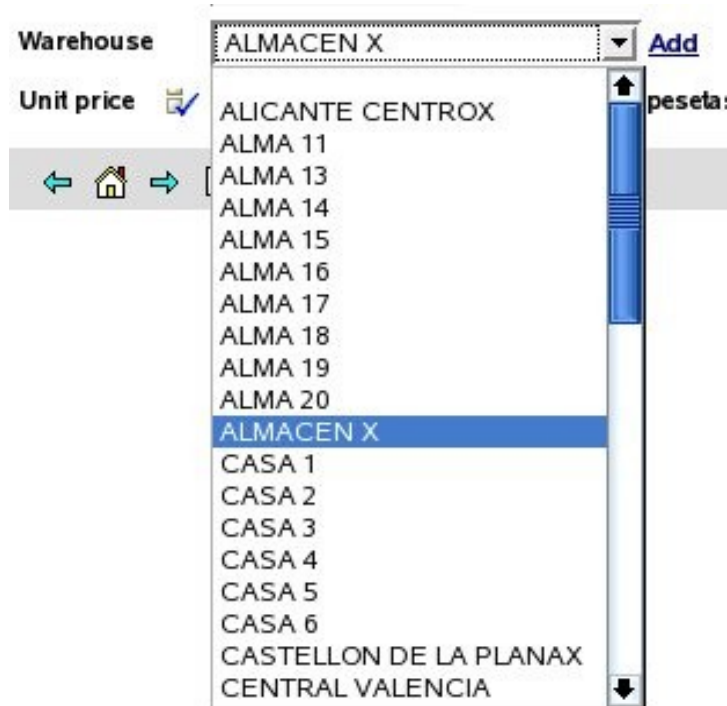
- (1)description-property (optional): The property to show in the list, if not specified, the property named description, descripcion, name or nombre is assumed. If the referenced object does not have a property called this way then it is required to specify a property name here.
- (2)description-properties (optional): As description-property (and excluding with it) but allows to set more than one property separated by commas. To the final user the values are concatenated.
- (3)depends (optional): It's used in together with condition. It can be achieve that the list content depends on another value displayed in the main view (if you simply type the name of the member) or in the same view (if you type this. before the name of the member).

- (4) `condition` (optional): Allows to specify a condition (with SQL style) to filter the values that are shown in the description list.
- (5) `order-by-key` (optional): By default the data is ordered by description, but if you set this property to `true` it will be ordered by key.
- (6) `order` (optional): Allows to specify an order (with SQL style) for the values that are shown in the description list.
- (7) `label-format` (optional): Format to display the label of the reference. See section 4.2.1.

The most simple usage is:

```
<reference-view reference="warehouse">
  <descriptions-list/>
</reference-view>
```

That displays a reference to warehouse in this way:



In this case it shows all warehouses, although in reality it uses the `base-condition` and the `filter` specified in the default `tab` of `Warehouse`. You will see more about tabs in chapter 5.

If you want, for example, to display a combo with the product families and when the user chooses a family, then another combo will be filled with the subfamilies of the chosen family. An implementation can look like this:

```
<reference-view reference="family">
  <descriptions-list order-by-key="true"/> (1)
</reference-view>

<reference-view reference="subfamily" create="false"> (2)
```

```

<descriptions-list
    description-property="description"           (3)
    depends="family"                           (4)
    condition="${family.number} = ?"/>         (5)
    order="${description} desc"/>              (6)

</reference-view>

```

Two combos are displayed one with all families loaded and the other one empty. When the user chooses a family, then the second combo is filled with all its subfamilies.

In the case of `Family` the property `description` of `Family` is shown, since the default property to show is 'description' or 'name'. The data is ordered by key and not by description (1). In the case of `Subfamily` (2) the link to create a new subfamily is not shown and the property to display is 'description' (in this case this maybe omitted).

With `depends` (4) you make that this combo depends on the reference `family`, when change `family` in the user interface, this descriptions list is filled applying the condition `condition` (5) and sending as argument (to set value to `?`) the new `family` value. And the entries are ordered descending by description (6).

In `condition` and `order` you put the property name inside a `${}` and the arguments as `?`. The comparator operators are the SQL operators.

You can specify several properties to be shown as description:

```

<reference-view reference="alternateSeller" read-only="true">
    <descriptions-list description-properties="level.description, name"/>
</reference-view>

```

In this case the concatenation of the `description` of `level` and the `name` is shown in the combo. Also you can see how it is possible to use qualified properties (`level.description`).

If you set `read-only="true"` in a reference as `descriptions-list`, then the description (in this case `level.description + name`) is displayed as a simple text property instead of using a combo.

4.4 Collection view

Suitable to refine the collection presentation. Here is its syntax:

```

<collection-view
    collection="collection"           (1)
    view="view"                      (2)
    read-only="true|false"           (3)
    edit-only="true|false"           (4)
    create-reference="true|false"     (5)
    as-aggregate="true|false"         (6) new in v2.0.2
>
    <list-properties ... />           (7)

```

```

    <edit-action ... />                (8)
    <view-action ... />                (9)
    <new-action ... />                 (10) new in v2.0.2
    <save-action ... />                (11) new in v2.0.2
    <hide-detail-action ... />         (12) new in v2.0.2
    <remove-action ... />              (13) new in v2.0.2
    <list-action ... /> ...            (14)
    <detail-action ... /> ...          (15)
</collection-view>

```

- (1)**collection** (required): The look of the collection with this name will be customized.
- (2)**view** (optional): The view of the referenced object (each collection element) which is used to display the detail. By default the default view is used.
- (3)**read-only** (optional): By default `false`; if you set it to `true`, then the final user only can view collection elements, he cannot add, delete or modify elements.
- (4)**edit-only** (optional): By default `false`; if you set it to `true`, then the final user can modify existing elements, but not add or remove collection elements.
- (5)**create-reference** (optional): By default `true`, if you set it to `false` then the final doesn't get the link to create new objects of the referenced object type. This only applies in the case of an entity references collection.
- (6)**as-aggregate** (optional): (*new in v2.0.2*) By default `false`. By default the collections of aggregates allow the users to create and to edit elements, while the collections of entities allow only to choose existing entities to add to (or remove from) the collection. If you put `as-aggregate` to `true` then the collection of entities behaves as a collection of aggregates, allowing to the user to add objects and editing them directly. It has no effect in case of collections of aggregates.
- (7)**list-properties** (one, optional): Properties to show in the list for visualization of the collection. You can qualify the properties. By default it shows all persistent properties of the referenced object (excluding references and calculated properties).
- (8)**edit-action** (one, optional): Allows you to define your custom action to begin the editing of a collection element. This is the action showed in each row of the collection, if the collection is editable.
- (9)**view-action** (one, optional): Allows you to define your custom action to view a collection element. This is the action showed in each row, if the collection is read only.
- (10)**new-action** (one, optional): (*new in v2.0.2*) Allows you to define your custom action to start adding a new element to the collection. This is the action executed on click in 'Add' link.
- (11)**save-action** (one, optional): (*new in v2.0.2*) Allows you to define your custom action to save the collection element. This is the action executed on click in 'Save detail' link.
- (12)**hide-detail-action** (one, optional): (*new in v2.0.2*) Allows you to define your custom action to hide the detail view. This is the action executed on click in 'Close' link.
- (13)**remove-action** (one, optional): (*new in v2.0.2*) Allows you to define your custom action to

remove the element from the collection. This is the action executed on click in 'Remove detail' link.

(14)`list-action` (several, optional): To add actions in list mode; usually actions which scope is the entire collection.

(15)`detail-action` (several, optional): To add actions in detail mode, usually actions which scope is the detail that is being edited.

If you do not use `<collection-view/>`, then the collection is displayed using the persistent properties in list mode and the default view to represent the detail; although in typical scenarios the properties of the list and the view for detail are specified:

```
<collection-view collection="customers" view="Simple">
  <list-properties>
    number, name, remarks, relationWithSeller, seller.level.description
  </list-properties>
</collection-view>
```

And the collection is displayed:

Customers					
	Number	Name	Remarks	Relation with seller	Description
Edit	1	Javi		BUENA	MANAGER
Edit	2	Juanillo			MANAGER
Add					

You see how you can put qualified properties into the properties list (as `seller.level.description`).

When the user clicks on 'Edit' or 'Add', then the view `Simple` of `Customer` will be rendered; for this you must have defined a view called `Simple` in the `Customer` component (the model of the collection elements).


If the view `Simple` of `Customer` is like this:


```
<view name="Simple" members="number; type; name; address"/>
```


On clicking in a detail the following will be shown:

Customers					
	Number	Name	Remarks	Relation with seller	Description
Edit	1	Javi		BUENA	MANAGER
Edit	2	Juanillo			MANAGER

Customer



Number 



Type  ☒ Steady

Name 

Address

ViewProperty

Street  Zip code 

City  State 

[Save detail](#) [Close](#) [Remove detail](#)

4.4.1 Custom edit/view action

You can refine easily the behavior when the 'Edit' link is clicked:

```
<collection-view collection="details">
  <edit-action action="Invoices.editDetail"/>
</collection-view>
```

You have to define `Invoices.editDetail` in *controllers.xml*:

```
<controller name="Invoices">
  ...
  <action name="editDetail" hidden="true"
    class="org.openxava.test.actions.EditInvoiceDetailAction">
    <use-object name="xava_view"/>
  </action>
  ...
</controller>
```

And finally write your action:

```
package org.openxava.test.actions;

import java.text.*;
```

```

import org.openxava.actions.*;

/**
 * @author Javier Paniza
 */
public class EditInvoiceDetailAction extends EditElementInCollectionAction { // (1)

    public void execute() throws Exception {
        super.execute();
        DateFormat df = new SimpleDateFormat("dd/MM/yyyy");
        getCollectionElementView().setValue( // (2)
            "remarks", "Edit at " + df.format(new java.util.Date()));
    }
}

```

In this case you only refine hence your action extends (1) `EditElementInCollectionAction`. In this case you only specify a default value for the `remarks` property. Note that to access the view that displays the detail you can use the method `getCollectionElementView()` (2).

The technique to refine the view action (the action for each row, if the collection is read only) is the same but using `<view-action/>` instead of `<edit-action/>`.

4.4.2 Custom list actions

Adding our custom list actions (actions that apply to entire collections) is easy:

```

<collection-view collection="fellowCarriers" view="Simple">
    <list-action action="Carriers.translateName"/>
</collection-view>

```

Now a new link is shown to user:

Fellow Carriers				
		Number	Name	Calculated Remarks
Edit	<input type="checkbox"/>	2	DOS	TR
Edit	<input type="checkbox"/>	3	THREE	TR
Edit	<input type="checkbox"/>	4	FOUR	TR
Add Translate name				

And also you see that there is a check box in each row.

Also you need to define the action in *controllers.xml*:

```

<controller name="Carriers">
    <action name="translateName"
        class="org.openxava.test.actions.TranslateCarrierNameAction">

```



```
</action>
</controller>
```

And the action code:

```
package org.openxava.test.actions;

import java.util.*;

import org.openxava.actions.*;
import org.openxava.test.model.*;

/**
 * @author Javier Paniza
 */
public class TranslateCarrierNameAction extends CollectionBaseAction { // (1)

    public void execute() throws Exception {
        Iterator it = getSelectedObjects().iterator(); // (2)
        while (it.hasNext()) {
            ICarrier carrier = (ICarrier) it.next();
            carrier.translate();
        }
    }
}
```

The action extends `CollectionBaseAction` (1), this way you can use methods as `getSelectedObjects()` (2) that returns a collection with the objects selected by the user. There are others useful methods, as `getObjects()` (all elements collection), `getMapValues()` (the collection values in map format) and `getMapsSelectedValues()` (the selected elements in map format).

As in the case of detail actions (see next section) you can use `getCollectionElementView()`.

4.4.3 Custom detail actions

Also you can add your custom actions to the detail view used for editing each element. These actions are applicable only to one element of collection. For example:

```
<collection-view collection="details">
    <detail-action action="Invoices.viewProduct"/>
</collection-view>
```

In this way the user has another link to click in the detail of the collection element:

[Save detail](#) [Close](#) [Remove detail](#) [View product](#)

You need to define the action in *controllers.xml*:

```
<controller name="Invoices">
    ...
    <action name="viewProduct" hidden="true"
        class="org.openxava.test.actions.ViewProductFromInvoiceDetailAction">
        <use-object name="xava_view"/>
        <use-object name="xavatest_invoiceValues"/>
    </action>
    ...
</controller>
```

And the code of your action:

```
package org.openxava.test.actions;

import java.util.*;
import javax.ejb.*;

import org.openxava.actions.*;

/**
 * @author Javier Paniza
 */
public class ViewProductFromInvoiceDetailAction
    extends CollectionElementViewBaseAction           // (1)
    implements INavigationAction {

    private Map invoiceValues;

    public void execute() throws Exception {
        try {
            setInvoiceValues(getView().getValues());
            Object number =
                getCollectionElementView().getValue("product.number"); // (2)
            Map key = new HashMap();
            key.put("number", number);
            getView().setModelName("Product");        // (3)
            getView().setValues(key);
            getView().findObject();
        }
    }
}
```

```

        getView().setKeyEditable(false);
        getView().setEditable(false);
    }
    catch (ObjectNotFoundException ex) {
        getView().clear();
        addError("object_not_found");
    }
    catch (Exception ex) {
        ex.printStackTrace();
        addError("system_error");
    }
}

public String[] getNextControllers() {
    return new String [] { "ProductFromInvoice" };
}

public String getCustomView() {
    return SAME_VIEW;
}

public Map getInvoiceValues() {
    return invoiceValues;
}

public void setInvoiceValues(Map map) {
    invoiceValues = map;
}
}

```

You can see that it extends `CollectionElementViewBaseAction` (1) thus it has available the view that displays the current element using `getCollectionElementView()` (2). Also you can get access to the main view using `getView()` (3). In chapter 7 you will see more details about writing actions.

Also, using the view returned by `getCollectionElementView()` you can add and remove programmatically detail and list actions (*new v2.0.2*) with `addDetailAction()`, `removeDetailAction()`, `addListAction()` and `removeListAction()`, see API doc for `org.openxava.view.View`.

4.4.4 Refining collection view default behavior (*new in v2.0.2*)

Using `<new-action/>`, `<save-action/>`, `<hide-detail-action/>` and `<remove-action/>` you can refine the default behavior of collection view. For example if you want to refine the behavior of save a detail action you can define your view in this way:

```
<collection-view collection="details">
    <save-action action="DeliveryDetails.save"/>
</collection-view>
```

You must have an action `DeliveryDetails.save` in your *controllers.xml*:

```
<controller name="DeliveryDetails">
    <action name="save"
        class="org.openxava.test.actions.SaveDeliveryDetailAction">
        <use-object name="xava_view"/>
    </action>
</controller>
```

And define your action class for saving:

```
package org.openxava.test.actions;

import org.openxava.actions.*;

/**
 *
 * @author Javier Paniza
 */

public class SaveDeliveryDetailAction extends SaveElementInCollectionAction { // (1)

    public void execute() throws Exception {
        super.execute();
        // Here your own code // (2)
    }

}
```

The more common case is extending the default behavior, for that you have to extend the original class for saving a collection detail (1), that is `SaveElementInCollection` action, then call to `super` from `execute()` method (2), and after it, writing your own code.

4.5 View property

With `<property/>` within `<view/>` you define a property that is not in the model but you want to show to the user. You can use it to provide UI controls to allow the user to manage his user interface.

An example:

```
<view>
```

```

    <property name="deliveredBy">
        <valid-values>
            <valid-value value="employee"/>
            <valid-value value="carrier"/>
        </valid-values>
        <default-value-calculator
            class="org.openxava.calculators.IntegerCalculator">
                <set property="value" value="0"/>
            </default-value-calculator>
        </property>

    <property-view property="deliveredBy">
        <on-change class="org.openxava.test.actions.OnChangeDeliveryByAction"/>
    </property-view>

    ...
</view>

```

You can see that the syntax is exactly the same as in the case of a property of a model; you can even use `<valid-values/>` and `<default-value-calculator/>`. After defining the property you can use it in the view as usual, for example with `on-change` or putting it in `members`.

4.6 Editors configuration

You see that the level of abstraction used to define views is high, you specify the properties to be shown and how to layout them, but not how to render them. To render properties OpenXava uses editors.

An editor indicates how to render a property. It consists of an XML definition put together with a JSP fragment.

To refine the behavior of the OpenXava editors or to add your custom editors you must create in the folder *xava* of you project a file called *editors.xml*. This file looks like this:

```

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

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

<editors>
    <editor .../> ...
</editors>

```

Simply it contains the definition of a group of editors, and an editor is defined like this:

```

<editor
    url="url" (1)
    format="true|false" (2)

```

```

    depends-stereotypes="stereotypes"                (3)
    depends-properties="properties"                  (4)
    frame="true|false"                               (5)
>
    <property ... /> ...                             (6)
    <formatter ... /> ...                           (7)
    <for-stereotype ... /> ...                       (8)
    <for-type ... /> ...                             (8)
    <for-model-property ... /> ...                  (8)
</editor>

```

- (1)url (required): URL of JSP fragment that implements editor.
- (2)format (optional): If `true`, then OpenXava has the responsibility of formatting the data from HTML to Java and vice versa; if `false`, then the responsibility of this is for the editor itself (generally getting the data from request and assigning it to `org.openxava.view.View` and vice versa). The default is `true`.
- (3)depends-stereotypes (optional): List of stereotypes (comma separated) which this editor depends on. If in the same view there are some editors for these stereotypes they throw a change value event if its values change.
- (4)depends-properties (optional): List of properties (comma separated) on which this editor depends. If in the same view there are some editors for these properties they throw a change value event if its values change.
- (5)frame (optional): If `true`, then the editor will be displayed inside a frame. The default is `false`. Useful for big editors (more than one line) that can be prettier this way.
- (6)property (several, optional): Set values in the editor; this way you can configure your editor and use it several times in different cases.
- (7)formatter (one, optional): Java class to define the conversion from Java to HTML and from HTML to Java.
- (8)for-stereotype or for-type or for-model-property (required one of these, but only one): Associates this editor with a stereotype, type or a concrete property of a model. The preference order is: first model property, then stereotype and finally type.

Let's see an example of an editor definition. This example is an editor that comes with OpenXava, but it is a good example to learn how to make your custom editors:

```

<editor url="textEditor.jsp">
    <for-type type="java.lang.String"/>
    <for-type type="java.math.BigDecimal"/>
    <for-type type="int"/>
    <for-type type="java.lang.Integer"/>
    <for-type type="long"/>
    <for-type type="java.lang.Long"/>
</editor>

```

Here a group of basic types is assigned to the editor *textEditor.jsp*. The JSP code of this editor is:

```
<%@ page import="org.openxava.model.meta.MetaProperty" %>

<%
String propertyKey = request.getParameter("propertyKey");           // (1)
MetaProperty p = (MetaProperty) request.getAttribute(propertyKey); // (2)
String fvalue = (String) request.getAttribute(propertyKey + ".fvalue"); // (3)
String align = p.isNumber()? "right": "left";                       // (4)
boolean editable="true".equals(request.getParameter("editable")); // (5)
String disabled=editable?"": "disabled";                           // (5)
String script = request.getParameter("script");                     // (6)
boolean label = org.openxava.util.XavaPreferences.getInstance().isReadOnlyAsLabel();
if (editable || !label) {                                           // (5)
%>
<input name="<%=propertyKey%>" class=editor                        <!-- (1) -->
    type="text"
    title="<%=p.getDescription(request)%>"
    align='<%=align%>'                                             <!-- (4) -->
    maxlength="<%=p.getSize()%>"
    size="<%=p.getSize()%>"
    value="<%=fvalue%>"                                           <!-- (3) -->
    <%=disabled%>                                                 <!-- (5) -->
    <%=script%>                                                    <!-- (6) -->

    />

<%
} else {
%>
<%=fvalue%>&nbsp;

<%
}
%>

<% if (!editable) { %>
    <input type="hidden" name="<%=propertyKey%>" value="<%=fvalue%>">
<% } %>
```

A JSP editor receives a set of parameters and has access to attributes that allows to configure it in order to work suitably with OpenXava. First you can see how it gets `propertyKey` (1) that is used as HTML id. From this id you can access to `MetaProperty` (2) (that contains meta information of the property to edit). The `fvalue` (3) attribute contains the value already formatted and ready to be displayed. Align (4) and editable (5) are obtained too. Also you need to obtain a JavaScript (6) fragment to put in the HTML editor.

Although creating an editor directly with JSP is easy it is not usual to do it. It's more common to

configure existing JSPs. For example, in your *xava/editors.xml* you can write:

```
<editor url="textEditor.jsp">
    <formatter class="org.openxava.formatters.UpperCaseFormatter"/>
    <for-type type="java.lang.String"/>
</editor>
```

In this way you are overwriting the OpenXava behavior for properties of String type, now all Strings are displayed and accepted in upper-cases. Let's see the code of the formatter:

```
package org.openxava.formatters;

import javax.servlet.http.*;

/**
 * @author Javier Paniza
 */

public class UpperCaseFormatter implements IFormatter { // (1)

    public String format(HttpServletRequest request, Object string) { // (2)
        return string==null?"":string.toString().toUpperCase();
    }

    public Object parse(HttpServletRequest request, String string) { // (3)
        return string==null?"":string.toString().toUpperCase();
    }

}
```

A formatter must implement `IFormatter` (1), this forces you to write a `format()` (2) method to convert the property value (that can be a Java object) to a string to be rendered in HTML; and a `parse()` (3) method to convert the string received from the submitted HTML form into an object suitable to be assigned to the property.

4.7 Multiple values editors

Defining an editor for editing multiple values is alike to define a single value editor. Let's see it.

For example if you want to define a stereotype `REGIONS` that allows the user to select more than one region for a single property. You may use the stereotype in this way:

```
<property name="regions" stereotype="REGIONS"/>
```

Then you need to add the next entry to your *stereotype-type-default.xml* file:

```
<for stereotype="REGIONS" type="String []"/>
```


And to define in your editor in your *editors.xml* file:

```
<editor url="regionsEditor.jsp"> (1)
    <property name="regionsCount" value="3"/> (2)
    <formatter class="org.openxava.formatters.MultipleValuesByPassFormatter"/> (3)
    <for-stereotype stereotype="REGIONS"/>
</editor>
```

regionsEditor.jsp (1) is the JSP file to render the editor. You can define properties that will be sent to the JSP as request parameters (2). And the formatter must implement IMultipleValuesFormatter, that is similar to IFormatter but it uses String [] instead of String. In this case we are using a generic formatter that simply do a bypass.

The last is to write your JSP editor:

```
<%@ page import="java.util.Collection" %>
<%@ page import="java.util.Collections" %>
<%@ page import="java.util.Arrays" %>
<%@ page import="org.openxava.util.Labels" %>

<jsp:useBean id="style" class="org.openxava.web.style.Style" scope="request"/>

<%
String propertyKey = request.getParameter("propertyKey");
String [] fvalues = (String []) request.getAttribute(propertyKey + ".fvalue"); // (1)
boolean editable="true".equals(request.getParameter("editable"));
String disabled=editable?"":"disabled";
String script = request.getParameter("script");
boolean label = org.openxava.util.XavaPreferences.getInstance().isReadOnlyAsLabel();
if (editable || !label) {
    String sregionsCount = request.getParameter("regionsCount");
    int regionsCount = sregionsCount == null?5:Integer.parseInt(sregionsCount);
    Collection regions = fvalues==null?Collections.EMPTY_LIST:Arrays.asList(fvalues);
%>
<select name="<%=propertyKey%>" multiple="multiple"
    class=<%=style.getEditor()%>
    <%=disabled%>
    <%=script%>>
    <%
    for (int i=1; i<regionsCount+1; i++) {
        String selected = regions.contains(Integer.toString(i))?"selected":"";
    %>
    <option
        value="<%=i%>" <%=selected%>>
```

```

        <%=Labels.get("regions." + i, request.getLocale())%>
    </option>
    <%
    }
    %>
</select>
<%
}
else {
    for (int i=0; i<fvalues.length; i++) {
    %>
    <%=Labels.get("regions." + fvalues[i], request.getLocale())%>
    <%
        }
    }
    %>

    <%
    if (!editable) {
        for (int i=0; i<fvalues.length; i++) {
    %>
        <input type="hidden" name="<%=propertyKey%>" value="<%=fvalues[i]%>">

    <%
        }
    }
    %>
}
    %>

```

As you see it is like defining a single value editor, the main difference is that the formatted value (1) is an array of strings (`String []`) instead of a simple string (`String`).

4.8 Custom editors and stereotypes for displaying combos

You can have simple properties displayed as combos and fill the combos with data from the database.

Let's see this.

You define the properties like this in your component:

```

<entity>
    ...
    <property name="familyNumber" stereotype="FAMILY" required="true"/>
    <property name="subfamilyNumber" stereotype="SUBFAMILY" required="true"/>
    ...

```

```
</entity>
```

And in your *editors.xml* put:

```
<editor url="descriptionsEditor.jsp"> (1)
    <property name="model" value="Family"/> (2)
    <property name="keyProperty" value="number"/> (3)
    <property name="descriptionProperty" value="description"/> (4)
    <property name="orderByKey" value="true"/> (5)
    <property name="readOnlyAsLabel" value="true"/> (6)
    <for-stereotype stereotype="FAMILY"/> (11)
</editor>

<!-- It is possible to specify dependencies from stereotypes or properties -->
<editor url="descriptionsEditor.jsp" (1)
    depends-stereotypes="FAMILY"> (12)
<!--
<editor url="descriptionsEditor.jsp" depends-properties="familyNumber"> (13)
-->
    <property name="model" value="Subfamily"/> (2)
    <property name="keyProperty" value="number"/> (3)
    <property name="descriptionProperties" value="number, description"/> (4)
    <property name="condition" value="${familyNumber} = ?"/> (7)
    <property name="parameterValuesStereotypes" value="FAMILY"/> (8)
    <!--
    <property name="parameterValuesProperties" value="familyNumber"/> (9)
    -->
    <property name="descriptionsFormatter" (10)
        value="org.openxava.test.formatters.FamilyDescriptionsFormatter"/>
    <for-stereotype stereotype="SUBFAMILY"/> (11)
</editor>
```

When you show a view with this two properties (`familyNumber` and `subfamilyNumber`) OpenXava displays a combo for each property, the family combo is filled with all families and the subfamily combo is empty; and when the user chooses a family, then the subfamily combo is filled with all the subfamilies of the chosen family.

In order to do that you need to assign to stereotypes (FAMILY and SUBFAMILY in this case(11)) the *descriptionsEditor.jsp* (1) editor and you configure it by assigning values to its properties. Some properties that you can set in this editor are:

- (2)model: Model to obtain data from. It can be the name of a component (e.g. Invoice) or the name of an aggregate used in a collection (Invoice.InvoiceDetail).
- (3)keyProperty or keyProperties: Key property or list of key properties; this is used to obtain the value to assign to the current property. It is not required that they are the key properties of the model, although this is the typical case.

- (4)`descriptionProperty` or `descriptionProperties`: Property or list of properties to show in combo.
- (5)`orderByKey`: If it has to be ordered by the key, by default it is ordered by description. You can also use `order` with an order clause in SQL style if you need it.
- (6)`readOnlyAsLabel`: When it is read only, then it is rendered as label. The default is `false`.
- (7)`condition`: Condition to limit the data to be displayed. Has SQL format, but you can use the property names with `${}`, even qualified properties are supported. You can put arguments with `?`. This last case is when this property depends on other ones and only obtain data when these other properties change.
- (8)`parameterValuesStereotypes`: List of stereotypes from which properties depend. It's used to fill the condition arguments and has to match with `depends-stereotypes` attribute (12).
- (9)`parameterValuesProperties`: List of properties from which properties depends. It's used to fill the condition arguments and has to match with `depends-properties` attribute (13).
- (10)`descriptionsFormatter`: Formatter for the descriptions displayed in a combo. It must implement `IFormatter`.

Following this example you can learn how to create your own stereotypes that display a simple property in combo format and with dynamic data. Nevertheless, in most cases it is more convenient to use references displayed as `descriptions-list`; but you always can choose.

4.9 View without model

In OpenXava it is not possible to have a view without model. Thus if you want to draw an arbitrary user interface, you need to create a component and map the component to a inexistent table (while you do not try to read or save everything will be fine) and define your view from it.

An example can be:

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

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

<!--
    Example of OpenXava component not persistent.

    This can be used, for example, to display a dialog,
    or any other graphical interface.

    Of course, if we try to save or read from db
    with this component it will crash.

    At moment it is needed to specify the mapping part.
-->
```

```

<component name="FilterBySubfamily">

    <entity>
        <property name="oid" key="true" type="String" hidden="true"/>
        <reference name="subfamily" model="Subfamily2" required="true"/>
    </entity>

    <view name="Family1">
        <reference-view reference="subfamily" create="false">
            <descriptions-list condition="${family.number} = 1"/>
        </reference-view>
    </view>

    <view name="Family2">
        <reference-view reference="subfamily" create="false">
            <descriptions-list condition="${family.number} = 2"/>
        </reference-view>
    </view>

    <view name="WithSubfamilyForm">
        <reference-view reference="subfamily" search="false"/>
    </view>

    <entity-mapping table="XAVATEST@separator@MOCKTABLE">
        <property-mapping property="oid" column="OID"/>
        <reference-mapping reference="subfamily">
            <reference-mapping-detail
                column="SUBFAMILY"
                referenced-model-property="number"/>
        </reference-mapping>
    </entity-mapping>

</component>

```

This way you can design a dialog that can be useful, for example, to print a report of families or products filtered by subfamily. The MOCKTABLE table does not exist.

With this simple trick you can use OpenXava as a simple and flexible generator for user interfaces although the displayed data won't be stored.

Tabular data is data that is displayed in table format. If you create a conventional OpenXava module, then the user can manage the component data with a list like this:

		Zone	Warehouse number	Name
Filter	=		=	starts
Detail	<input type="checkbox"/>	1	1	CENTRAL VALENCIA
Detail	<input type="checkbox"/>	1	2	VALENCIA SURETE
Detail	<input type="checkbox"/>	1	3	VALENCIA NORTE
Detail	<input type="checkbox"/>	2	1	CASTELLON DE LA PLANAX
Detail	<input type="checkbox"/>	3	1	ALICANTE CENTROX
Detail	<input type="checkbox"/>	4	2	ALMA 2
Detail	<input type="checkbox"/>	4	3	ALMA 3
Detail	<input type="checkbox"/>	4	4	ALMA 4
Detail	<input type="checkbox"/>	4	5	ALMA 5
Detail	<input type="checkbox"/>	4	6	ALMA 6

1 2 3 4 5 ▶ There are 49 objets in list

This list allows user to:

- Filter by any columns or a combination of them.
- Order by any column with a single click.
- Display data by pages, and therefore the user can work efficiently with millions of records.
- Customize the list: add, remove and change the column order (with the little pencil in the left top corner). This customizations are remembered by user.
- Generic actions to process the objects in the list: generate PDF reports, export to Excel or remove the selected objects.

The default list is enough for many cases, moreover the user can customize it. Nevertheless, sometimes it is convenient to modify the list behavior. For this you have the element `<tab/>` within the component definition.

The syntax of `tab` is:

```
<tab
  name="name"           (1)
>
  <filter ... />        (2)
  <row-style ... /> ...  (3)
  <properties ... />    (4)
  <base-condition ... /> (5)
```

```
<default-order ... />      (6)
</tab>
```

- (1)**name** (optional): You can define several tabs in a component, and set a name for each one. This name is used to indicate the tab that you want to use (usually in *application.xml*).
- (2)**filter** (one, optional): Allows to define programmatically some logic to apply to the values entered by user when he filters the list data.
- (3)**row-style** (several, optional): A simple way to specify a different visual style for some rows. Normally to emphasize rows that fulfill certain condition.
- (4)**properties** (one, optional): The list of properties to show initially. Can be qualified (that is you can specify `referenceName.propertyName` at any depth level).
- (5)**base-condition** (one, optional): Condition to be fulfilled by the displayed data. It's added to the user condition if needed.
- (6)**default-order** (one, optional): To specify the initial order for data.

5.1 Initial properties and emphasize rows

The most simple customization is to indicate the properties to show initially:

```
<tab>
  <row-style style="highlight" property="type" value="steady"/>
  <properties>
    name, type, seller.name, address.city, seller.level.description
  </properties>
</tab>
```

These properties are shown the first time the module is executed, after that the user will have the option to change the properties to display. Also you see how you can use qualified properties (properties of references) in any level.

In this case you can see also how to indicate a `<row-style/>`; you are saying that the object which property `type` has the value `steady` will use the style `highlight`. The style has to be defined in the CSS style-sheet. The `highlight` style are already defined in OpenXava, but you can define more.

The visual effect of above is:

	Name	Type	Seller	City	Seller level
Filter	starts ▾	▾	starts ▾	starts ▾	starts ▾
Detail	<input type="checkbox"/> Javi	Steady	MANUEL CHAVARRI	EL PUIG	MANAGER
Detail	<input type="checkbox"/> Juanillo	Normal	MANUEL CHAVARRI	VALENCIA	MANAGER
Detail	<input type="checkbox"/> Carmelo	Normal		EL PUIG	
Detail	<input type="checkbox"/> Cuatrero	Normal	JUANVI LLAVADOR	VALENCIA	MANAGER

1 There are 4 objets in list

5.2 Filters and base condition

An common technique is to combine a filter with a base condition:

```
<tab name="Current">
    <filter class="org.openxava.test.filters.CurrentYearFilter"/>
    <properties>
        year, number, amountsSum, vat, detailsCount, paid, customer.name
    </properties>
    <base-condition>${year} = ?</base-condition>
</tab>
```

The condition has to have SQL syntax, you can use ? for arguments and the property names inside \${}. In this case a filter is used to set the value of the argument. The filter code is:

```
package org.openxava.test.filters;

import java.util.*;

import org.openxava.filters.*;

/**
 * @author Javier Paniza
 */

public class CurrentYearFilter implements IFilter { // (1)

    public Object filter(Object o) throws FilterException { // (2)
        Calendar cal = Calendar.getInstance();
        cal.setTime(new java.util.Date());
        Integer year = new Integer(cal.get(Calendar.YEAR));
        Object [] r = null;
        if (o == null) { // (3)
            r = new Object[1];
            r[0] = year;
        }
        else if (o instanceof Object []) { // (4)
            Object [] a = (Object []) o;
            r = new Object[a.length + 1];
            r[0] = year;
            for (int i = 0; i < a.length; i++) {
                r[i+1]=a[i];
            }
        }
    }
}
```



```

        else {
            r = new Object[2];
            r[0] = year;
            r[1] = o;
        }

        return r;
    }
}

```

A filter gets the arguments of user type for filtering in lists and for processing, it returns the value that is sent to OpenXava to execute the query. As you see it must implement `IFilter` (1), this force it to have a method named `filter` (2) that receives a object with the value of arguments and returns the filtered value that will be used as query argument. These arguments can be null (3), if the user does not type values, a simple object (5), if the user types a single value or an object array (4), if the user types several values. The filter must consider all cases. The filter of this example adds the current year as first argument, and this value is used for filling the arguments in the `base-condition` of tab.

To sum up, the tab that you see above only shows the invoices of the current year.

Another case:

```

<tab name="DefaultYear">
    <filter class="org.openxava.test.filters.DefaultYearFilter"/>
    <properties>
        year, number, customer.number,
        customer.name, amountsSum, vat, detailsCount, paid, importance
    </properties>
    <base-condition>${year} = ?</base-condition>
</tab>

```

In this case the filter is:

```

package org.openxava.test.filters;

import java.util.*;

import org.openxava.filters.*;

/**
 * @author Javier Paniza
 */

public class DefaultYearFilter extends BaseContextFilter {

```

```

public Object filter(Object o) throws FilterException {
    if (o == null) {
        return new Object [] { getDefaultYear() };           // (2)
    }
    if (o instanceof Object []) {
        List c = new ArrayList(Arrays.asList((Object []) o));
        c.add(0, getDefaultYear());                         // (2)
        return c.toArray();
    }
    else {
        return new Object [] { getDefaultYear(), o };       // (2)
    }
}

private Integer getDefaultYear() throws FilterException {
    try {
        return getInteger("xavatest_defaultYear");         // (3)
    }
    catch (Exception ex) {
        ex.printStackTrace();
        throw new FilterException(
            "Impossible to obtain default year associated with the session");
    }
}
}

```

This filter extends `BaseContextFilter`, this allow you to access to the session objects of OpenXava. You can see how it uses a method `getDefaultYear()` (2) that call to `getInteger()` (3) which (as `getString()`, `getLong()` or the more generic `get()`) that allows you to access to value of the session object `xavatest_defaultYear`. This object is defined in *controllers.xml* this way:

```
<object name="xavatest_defaultYear" class="java.lang.Integer" value="1999"/>
```

The actions can modify it and its life is the user session life but it's private for each module. This issue is treated in more detail in chapter 7.

This is a good technique for data shown in list mode to depend on the user or the configuration that he has chosen.

Also it's possible to access environment variables inside a filter (*new in v2.0*) of type `BaseContextFilter`, using `getEnvironment()` method, just in this way:

```
new Integer(getEnvironment().getValue("XAVATEST_DEFAULT_YEAR"));
```

For learning more about environment variables see the *Chapter 7 Controllers*.

5.3 Pure SQL select

You can write the complete select statement to obtain the tab data:

```
<tab name="CompleteSelect">
  <properties>number, description, family</properties>
  <base-condition>
    select ${number}, ${description}, XAVATEST@separator@FAMILY.DESCRPTION
      from   XAVATEST@separator@SUBFAMILY, XAVATEST@separator@FAMILY
      where  XAVATEST@separator@SUBFAMILY.FAMILY =
            XAVATEST@separator@FAMILY.NUMBER
  </base-condition>
</tab>
```

Use it only in extreme cases. Normally it is not necessary, and if you use this technique the user cannot customize his list.

5.4 Default order

Finally, setting a default order is very easy:

```
<tab name="Simple">
  <properties>year, number, date</properties>
  <default-order>${year} desc, ${number} desc</default-order>
</tab>
```

This specified the initial order and the user can choose any other order by clicking in the heading of a column.

Object relational mapping allows you to declare in which tables and columns of your database the component data will be stored.

If ORM is familiar to you: The OpenXava mapping is used to generate the code and XML files needed to object/relational mapping. Actually the code is generated for:

- Hibernate 3.1.
- EntityBeans CMP 2 of JBoss 3.2.x y 4.0.x.
- EntityBeans CMP 2 of Websphere 5, 5.1y 6.

If Object/relational tools are not familiar to you: Object/relational tools allow you to work with objects instead of tables and columns, and to generate automatically the SQL code to read and update the database.

OpenXava generates a set of Java classes that represent the model layer of your application (the business concepts with its data and its behavior). You can work directly with these objects, and you do not need direct access to the SQL database. Of course you have to define precisely how to map your classes to your tables, and this work is done in the mapping part.

6.1 Entity mapping

The syntax to map the main entity is:

```
<entity-mapping table="table">           (1)
    <property-mapping ... /> ...         (2)
    <reference-mapping ... /> ...        (3)
    <multiple-property-mapping ... /> ... (4)
</entity-mapping>
```

(1)table (required): Maps this table to the main entity of component.

(2)property-mapping (several, optional): Maps a property to a column in the database table.

(3)reference-mapping (several, optional): Maps a reference to one or more columns in the database table.

(4)multiple-property-mapping (several, optional): Maps a property to several columns in database table.

A plain example can be:

```
<entity-mapping table="XAVATEST@separator@DELIVERYTYPE">
    <property-mapping property="number" column="NUMBER"/>
    <property-mapping property="description" column="DESCRIPTION" />
```

```
</entity-mapping>
```

More easier impossible.

You see how the table name is qualified (with collection/schema name included). Also you see that the separator is @separator@ instead of a dot (.), this is useful because you can define the separator value in your *build.xml* and thus the same application can run against databases with or without support for collections or schemes.

6.2 Property mapping

The syntax to map a property is:

```
<property-mapping
    property="property"           (1)
    column="column"              (2)
    cmp-type="type">            (3)
    <converter ... />           (4)
</property-mapping>
```

(1)property (required): Name of a property defined in the model part.

(2)column (required): Name of a table column.

(3)cmp-type (optional): Java type of the attribute used internally in your object to store the property value. This allows you to use Java types more closer to the database, without contaminating your Java model. It is used with a converter.

(4)converter (one, optional): Implements your custom logic to convert from Java to DB format and vice versa.

For now, you have seen simple examples for mapping a property to a column. A more advanced case is using a converter. A converter is used when the Java type and the DB type don't match, in this case a converter is a good idea. For example, imagine that in database the zip code is VARCHAR while in Java you want to use an int. A Java int is not directly assignable to a VARCHAR column in database, but you can use a converter to transform that int to String. Let's see it:

```
<property-mapping property="address_zipCode" column="ZIPCODE" cmp-type="String">
    <converter class="org.openxava.converters.IntegerStringConverter"/>
</property-mapping>
```

cmp-type indicates to which type the converter has to convert to and it is the type of the internal attribute in the generated class code. It must be a type close (assignable directly from JDBC) to the column type in database.

The converter code is:

```
package org.openxava.converters;

/**
```

```

* In java an int and in database a String.
*
* @author Javier Paniza
*/
public class IntegerStringConverter implements IConverter {           // (1)

    private final static Integer ZERO = new Integer(0);

    public Object toDB(Object o) throws ConversionException {         // (2)
        return o==null?"0":o.toString();
    }

    public Object toJava(Object o) throws ConversionException {       // (3)
        if (o == null) return ZERO;
        if (!(o instanceof String)) {
            throw new ConversionException("conversion_java_string_expected");
        }
        try {
            return new Integer((String) o);
        }
        catch (Exception ex) {
            ex.printStackTrace();
            throw new ConversionException("conversion_error");
        }
    }
}

```

A converter must implement `IConverter` (1), this forces it to have a `toDB()` (2) method, that receives the object of the type used in Java (in this case an `Integer`) and returns its representation using a type closer to the database (in this case `String` hence assignable to a `VARCHAR` column). The `toJava()` method has the opposite goal; it gets the object in database format and it must return an object of the type used in Java.

If there are any problem you can throw a `ConversionException`.

You see that this converter is in `org.openxava.converters`, i. e., it is a generic converter that comes with the `OpenXava` distribution. Another quite useful generic converter is `ValidValuesLetterConverter`. That one allows to map properties of type `valid-values`. For example, if you have a property like this:

```

<entity>
...
    <property name="distance">
        <valid-values>

```

```

        <valid-value value="local"/>
        <valid-value value="national"/>
        <valid-value value="international"/>
    </valid-values>
</property>
...
</entity>

```

`valid-values` generates a Java property of type `int` in which 0 is used to indicate the empty value, 1 is 'local', 2 is 'national' and 3 is 'international'. But what happens, if in the database a single letter ('L', 'N' or 'I') is stored? In this case you can use a mapping like this:

```

<property-mapping property="distance" column="DISTANCE" cmp-type="String">
    <converter class="org.openxava.converters.ValidValuesLetterConverter">
        <set property="letters" value="LNI"/>
    </converter>
</property-mapping>

```

As you put 'LNI' as a value to `letters`, the converter matches the 'L' to 1, the 'N' to 2 and the 'I' to 3. You also see how converters are configurable using its properties and this makes the converters more reusable (as calculators, validators, etc).

6.3 Reference mapping

The syntax to map a reference is:

```

<reference-mapping
    reference="reference"                                (1)
>
    <reference-mapping-detail ... /> ...                (2)
</reference-mapping>

```

(1)reference (required): The reference to map.

(2)reference-mapping-detail (several, required): Maps a table column to a property of the reference key. If the key of the referenced object is multiple, then you will have several reference-mapping-detail.

Making a reference mapping is easy. For example, if you have a reference like this:

```

<entity>
    ...
    <reference name="invoice" model="Invoice"/>
    ...
</entity>

```

You can map it this way:

```

<entity-mapping table="XAVATEST@separator@DELIVERY">
  <reference-mapping reference="invoice">
    <reference-mapping-detail
      column="INVOICE_YEAR"
      referenced-model-property="year"/>
    <reference-mapping-detail
      column="INVOICE_NUMBER"
      referenced-model-property="number"/>
  </reference-mapping>
  ...
</entity-mapping>

```

INVOICE_YEAR and INVOICE_NUMBER are columns of the DELIVERY table that allows accessing to its invoice, that is it's the foreign key although declaring it as a foreign key in database is not required. You must map this columns to the key properties in Invoice, like this:

```

<entity>
  <property name="year" type="int" key="true" size="4" required="true">
    <default-value-calculator
      class="org.openxava.calculators.CurrentYearCalculator"/>
  </property>
  <property name="number" type="int" key="true" size="6" required="true"/>
  ...

```

If you have a reference to a model which key itself includes references, you can define it in this way:

```

<reference-mapping reference="delivery">
  <reference-mapping-detail
    column="DELIVERY_INVOICE_YEAR"
    referenced-model-property="invoice.year"/>
  <reference-mapping-detail
    column="DELIVERY_INVOICE_NUMBER"
    referenced-model-property="invoice.number"/>
  <reference-mapping-detail
    column="DELIVERY_TYPE"
    referenced-model-property="type.number"/>
  <reference-mapping-detail
    column="DELIVERY_NUMBER"
    referenced-model-property="number"/>
</reference-mapping>

```

As you see, to indicate the properties of referenced models you can qualify them.

Also it's possible to use converters in a reference mapping:


```

<reference-mapping reference="drivingLicence">
  <reference-mapping-detail
    column="DRIVINGLICENCE_TYPE"
    referenced-model-property="type"
    cmp-type="String"> (1) <!-- In this case this line can be omitted -->
    <converter class="org.openxava.converters.NotNullStringConverter"/> (2)
  </reference-mapping-detail>
  <reference-mapping-detail
    column="DRIVINGLICENCE_LEVEL"
    referenced-model-property="level"/>
  </reference-mapping>

```

You can use the converter just like in a simple property (2). The difference in the reference case is that if you do not define a converter, then a default converter is not used. This is because applying in an indiscriminate way converters on keys can produce problems in some circumstances. You can use `cmp-type` (1) (*new in v2.0*) to indicate the Java type of the attribute used internally in your object to store the value. This allows you to use Java types closer to the database; `cmp-type` is not need if the database type is compatible with Java type.

6.4 Multiple property mapping

With `<multiple-property-mapping/>` you can map several table columns to a single Java property. This is useful if you have properties of custom class that have itself several attributes to store. Also it is used when you have to deal with legate database schemes.

The syntax for this type of mapping is:

```

<multiple-property-mapping
  property="property" (1)
>
  <converter ... /> (2)
  <cmp-field ... /> ... (3)
</multiple-property-mapping>

```

(1)`property` (required): Name of the property to map.

(2)`converter` (one, required): Implements the logic to convert from Java to the database and vice versa. Must implement `IMultipleConverter`.

(3)`cmp-field` (several, required): Maps each column in the database with a property of a converter.

A typical example is the generic converter `Date3Converter`, that allows to store in the database 3 columns and in Java a single property of type `java.util.Date`.

```

<multiple-property-mapping property="deliveryDate">
  <converter class="org.openxava.converters.Date3Converter"/>
  <cmp-field converter-property="day" column="DAYDELIVERY" cmp-type="int"/>
  <cmp-field converter-property="month" column="MONTHDELIVERY" cmp-type="int"/>

```

```
<cmp-field converter-property="year" column="YEAREDELIVERY" cmp-type="int"/>
</multiple-property-mapping>
```

DAYDELIVERY, MONTHDELIVERY and YEAREDELIVERY are 3 columns in database that store the delivery date, and day, month and year are properties of Date3Converter. And here Date3Converter:

```
package org.openxava.converters;

import java.util.*;

import org.openxava.util.*;

/**
 * In java a <tt>java.util.Date</tt> and in database 3 columns of
 * integer type. <p>
 *
 * @author Javier Paniza
 */
public class Date3Converter implements IMultipleConverter {           // (1)

    private int day;
    private int month;
    private int year;

    public Object toJava() throws ConversionException {              // (2)
        return Dates.create(day, month, year);
    }

    public void toDB(Object javaObject) throws ConversionException { // (3)
        if (javaObject == null) {
            setDay(0);
            setMonth(0);
            setYear(0);
            return;
        }
        if (!(javaObject instanceof java.util.Date)) {
            throw new ConversionException("conversion_db_utildate_expected");
        }
        java.util.Date date = (java.util.Date) javaObject;
        Calendar cal = Calendar.getInstance();
        cal.setTime(date);
        setDay(cal.get(Calendar.DAY_OF_MONTH));
    }
}
```

```

        setMonth(cal.get(Calendar.MONTH) + 1);
        setYear(cal.get(Calendar.YEAR));
    }

    public int getYear() {
        return year;
    }

    public int getDay() {
        return day;
    }

    public int getMonth() {
        return month;
    }

    public void setYear(int i) {
        year = i;
    }

    public void setDay(int i) {
        day = i;
    }

    public void setMonth(int i) {
        month = i;
    }
}

```

This converter must implement `IMultipleConverter` (1). This forces it to have a `toJava()` (2) method that must return a Java object from its property values (in this case `year`, `month` and `day`). The returned object is the mapped property (in this case `deliveryDate`). The calculator must have the method `toDB()` (3) too; this method receives the value of the property (a delivery date) and has to split it and to put the result in the converter properties (`year`, `month` and `day`).

6.5 Reference to aggregate mapping

A reference to an aggregate contains data that in the relational model are stored in the same table as the main entity. For example, if you have an aggregate `Address` associated to a `Customer`, the address data is stored in the same data table as the customer data. How can you map this case with OpenXava?

Let's see. In the model you can have:

```

<entity>
    ...
    <reference name="address" model="Address" required="true"/>
    ...
</entity>

<aggregate name="Address">
    <implements interface="org.openxava.test.ejb.IWithCity"/>
    <property name="street" type="String" size="30" required="true"/>
    <property name="zipCode" type="int" size="5" required="true"/>
    <property name="city" type="String" size="20" required="true"/>
    <reference name="state" required="true"/>
</aggregate>

```

Simply a reference to an aggregate. And for mapping it you can do:

```

<entity-mapping table="XAVATEST@separator@CUSTOMER">
    ...
    <property-mapping property="address_street" column="STREET"/>
    <property-mapping property="address_zipCode" column="ZIPCODE" cmp-type="String">
        <converter class="org.openxava.converters.IntegerStringConverter"/>
    </property-mapping>
    <property-mapping property="address_city" column="CITY"/>
    <reference-mapping reference="address_state">
        <reference-mapping-detail column="STATE" referenced-model-property="id"/>
    </reference-mapping>
</entity-mapping>

```

You see how the aggregate members are mapped within the entity mapping that contains it. The only thing that you have to do is using the name of the reference as a prefix with an underline (in this case `address_`). You can observe that in the case of aggregates you can map references, properties and that you can use converters in the usual way.

6.6 Aggregate used in collection mapping

In case that you have a collection of aggregates, for example the invoice details, obviously the detail data is stored in a different table as the heading data. In this case the aggregate must have its own mapping. Let's see the example:

Here the model part of `Invoice`:

```

<entity>
    ...
    <collection name="details" minimum="1">
        <reference model="InvoiceDetail"/>
    </collection>

```

```

        </collection>
        ...
    </entity>

    <aggregate name="InvoiceDetail">
        <property name="oid" type="String" key="true" hidden="true">
            <default-value-calculator
                class="org.openxava.test.calculators.InvoiceDetailOidCalculator"
                on-create="true"/>
        </property>
        <property name="serviceType">
            <valid-values>
                <valid-value value="special"/>
                <valid-value value="urgent"/>
            </valid-values>
        </property>
        <property name="quantity" type="int" size="4" required="true"/>
        <property name="unitPrice" stereotype="MONEY" required="true"/>
        <property name="amount" stereotype="MONEY">
            <calculator class="org.openxava.test.calculators.DetailAmountCalculator">
                <set property="unitPrice"/>
                <set property="quantity"/>
            </calculator>
        </property>
        <reference model="Product" required="true"/>
        <property name="deliveryDate" type="java.util.Date">
            <default-value-calculator
                class="org.openxava.calculators.CurrentDateCalculator"/>
        </property>
        <reference name="soldBy" model="Seller"/>
        <property name="remarks" stereotype="MEMO"/>
    </aggregate>

```

You can see a collection of InvoiceDetail which is an aggregate. InvoiceDetail has to be mapped this way:

```

<aggregate-mapping aggregate="InvoiceDetail" table="XAVATEST@separator@INVOICEDetail">
    <reference-mapping reference="invoice">          (1)
        <reference-mapping-detail
            column="INVOICE_YEAR"
            referenced-model-property="year"/>
        <reference-mapping-detail
            column="INVOICE_NUMBER"

```

```

        referenced-model-property="number" />
    </reference-mapping>
    <property-mapping property="oid" column="OID" />
    <property-mapping property="serviceType" column="SERVICETYPE" />
    <property-mapping property="unitPrice" column="UNITPRICE" />
    <property-mapping property="quantity" column="QUANTITY" />
    <reference-mapping reference="product">
        <reference-mapping-detail
            column="PRODUCT_NUMBER"
            referenced-model-property="number" />
    </reference-mapping>
    <multiple-property-mapping property="deliveryDate">
        <converter class="org.openxava.converters.Date3Converter" />
        <cmp-field
            converter-property="day" column="DAYDELIVERY" cmp-type="int" />
        <cmp-field
            converter-property="month" column="MONTHDELIVERY" cmp-type="int" />
        <cmp-field
            converter-property="year" column="YEARDELIVERY" cmp-type="int" />
    </multiple-property-mapping>
    <reference-mapping reference="soldBy">
        <reference-mapping-detail
            column="SOLDBY_NUMBER"
            referenced-model-property="number" />
    </reference-mapping>
    <property-mapping property="remarks" column="REMARKS" />
</aggregate-mapping>

```

The aggregate mapping must be below of the main entity mapping. A component must have as many aggregate mappings as aggregates used in collections. The aggregate mapping has the same possibilities than entity mapping, with the exception that it's required to map a reference to the container object although maybe this reference is not defined in model. That is, although you do not define a reference to `Invoice` in `InvoiceDetail` OpenXava adds it automatically and you must map it (1).

6.7 Converters by default

You have seen how to declare a converter in a property mapping. But what happens, if you do not declare a converter? In reality in OpenXava all properties (except the key properties) have a converter by default. The default converters are defined in `OpenXava/xava/default-converters.xml`, that has a content like this:

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

```

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

<!--
In your project use the name 'converters.xml' or 'convertsores.xml'
-->

<converters>

    <for-type type="java.lang.String"
        converter-class="org.openxava.converters.TrimStringConverter"
        cmp-type="java.lang.String"/>

    <for-type type="int"
        converter-class="org.openxava.converters.IntegerNumberConverter"
        cmp-type="java.lang.Integer"/>

    <for-type type="java.lang.Integer"
        converter-class="org.openxava.converters.IntegerNumberConverter"
        cmp-type="java.lang.Integer"/>

    <for-type type="boolean"
        converter-class="org.openxava.converters.Boolean01Converter"
        cmp-type="java.lang.Integer"/>

    <for-type type="java.lang.Boolean"
        converter-class="org.openxava.converters.Boolean01Converter"
        cmp-type="java.lang.Integer"/>

    <for-type type="long"
        converter-class="org.openxava.converters.LongNumberConverter"
        cmp-type="java.lang.Long"/>

    <for-type type="java.lang.Long"
        converter-class="org.openxava.converters.LongNumberConverter"
        cmp-type="java.lang.Long"/>

    <for-type type="java.math.BigDecimal"
        converter-class="org.openxava.converters.BigDecimalNumberConverter"
        cmp-type="java.math.BigDecimal"/>

    <for-type type="java.util.Date"
        converter-class="org.openxava.converters.DateUtilSQLConverter"

```

```
cmp-type="java.sql.Date" />

</converters>
```

If you use a property of a type that is not defined here, by default OpenXava will assign the converter `NoConversionConverter`, a silly converter that don't perform anything.

In the case of key properties and references no converter are assigned; applying a converter to key properties can be problematic in certain circumstances, but even if you want to perform a conversion you can declare a converter explicitly in your mapping.

If you wish to modify the behavior of default converters in your application, you do not modify the OpenXava file, but you create your own *converters.xml* file in the folder *xava* of your project. You can assign a converter by default to a stereotype (using `<for-stereotype/>`).

6.8 Object/relational philosophy

OpenXava has been born and has been developed in an environment when it was necessary to work with legacy database without changing its structure. The result is that OpenXava:

- Provides great flexibility when mapping with legacy database.
- Does not provide some features natural for OOT and that requires to change database scheme, as inheritance support or polymorphic queries.

Another cool feature of OpenXava mapping is that applications are 100% portables from JBoss CMP2 to Websphere CMP2 without writing a single line of code. Furthermore, the portability between Hibernate and EJB2 version of an application is very high, the mapping and all automatic controllers are 100% portable, obviously the custom EJB2 or Hibernate code is not so portable.

The controllers are used for defining actions (buttons, links, images) that final user can click. The controllers are defined in the *controllers.xml* file that has to be in the *xava* directory of your project.

The actions are not defined in components because there are a lot of generic actions that can be applied to any component.

In *OpenXava/xava* you have a *controllers.xml* that contains a group of generic controllers that can be used in your applications.

The *controllers.xml* file contains an element of type `<controllers/>` with the syntax:

```
<controllers>
  <env-var ... /> ...      (1)
  <object ... /> ...      (2)
  <controller ... /> ...  (3)
</controllers>
```

- (1) `env-var` (several, optional): Variable that contains configuration information. This variable can be accessed from the actions and filters, and its value can be overwritten in each module.
- (2) `object` (several, optional): Defines Java object with session scope; that is objects that are created for an user and exist during his session.
- (3) `controller` (several, required): A controller is a group of actions.

7.1 Environment variables and session objects

Defining environment variables and session objects is very easy, you can see the defined ones in *OpenXava/xava/controllers.xml*:

```
<env-var name="XAVA_SEARCH_ACTION" value="CRUD.searchByViewKey"/>
<env-var name="XAVA_LIST_ACTION" value="List.viewDetail"/>

<object name="xava_view" class="org.openxava.view.View"/>
<object name="xava_referenceSubview" class="org.openxava.view.View"/>
<object name="xava_tab" class="org.openxava.tab.Tab"/>
<object name="xava_mainTab" class="org.openxava.tab.Tab"/>
<object name="xava_row" class="java.lang.Integer" value="0"/>
<object name="xava_language" class="org.openxava.session.Language"/>
<object name="xava_newImageProperty" class="java.lang.String"/>
<object name="xava_currentReferenceLabel" class="java.lang.String"/>
<object name="xava_activeSection" class="java.lang.Integer" value="0"/>
```

```

<object name="xava_previousControllers" class="java.util.Stack"/>
<object name="xava_previousViews" class="java.util.Stack"/>

```

You see a simple syntax, `name` and `value` for each environment variable and `name`, `class` and `value` for the session objects. Regarding name style you can use as prefix the name of you application, since these are the variables and objects of the OpenXava core the prefix is `XAVA_` and `xava_`. Also about naming, the environment variables are in uppercase and the objects in lowercase.

These objects and variables are used by OpenXava in order to work, although it is quite normal that you use some of these from your actions. If you want to create your own variables and objects you can do it in your *controllers.xml* in the *xava* directory of your project.

7.2 The controller and its actions

The syntax of controller is:

```

<controller
    name="name"                (1)
>
    <extends ... /> ...        (2)
    <action ... /> ...         (3)
</controller>

```

(0)`name` (required): Name of the controller.

(1)`extends` (several, optional): Allows to use multiple inheritance, to do this the controller inherits all actions from other controller(s).

(2)`action` (several, required): Implements the logic to execute when the final user clicks a button or link.

The controllers consist of actions, and actions are the main things. Here is its syntax:

```

<action
    name="name"                (1)
    label="label"              (2)
    description="description"   (3)
    mode="detail|list|ALL"      (4)
    image="image"               (5)
    class="class"               (6)
    hidden="true|false"         (7)
    on-init="true|false"        (8)
    by-default="never|if-possible|almost-always|always" (9)
    takes-long="true|false"     (10)
    confirm="true|false"        (11)
    keystroke="keystroke"       (12) new in v2.0.1
>
    <set ... /> ...             (13)

```

```
<use-object ... /> ...  
</action>
```

(14)

- (1)**name** (required): Action name that must be unique within its controller, but it can be repeated in other controllers. When you reference an action always use the format `ControllerName.actionName`.
- (2)**label** (optional): Button label or link text. It's **much better** to use *i18n* files.
- (3)**description** (optional): Description text of the action. It's **much better** to use *i18n* files.
- (4)**mode** (optional): Indicates in which mode the action has to be visible. The default value is `ALL`, that means that this action is always visible.
- (5)**image** (optional): URL of the image associated with this action. In the current implementation if you specify an image, it is shown to user in link format.
- (6)**class** (optional): Implements the logic to execute. Must implement `IAction` interface.
- (7)**hidden** (optional): A hidden action is not shown in the button bar, although it can be used in all other places, for example to associate it to an event, as action of a property, in collections, etc. The default is `false`.
- (8)**on-init** (optional): If you set this property to `true`, then the action will be executed automatically on initiating the module. The default is `false`.
- (9)**by-default** (optional): Indicates the weight of this action on choosing the action to execute as the default one. The default action is executed when the user presses ENTER. The default is `never`.
- (10)**takes-long** (optional): If you set it to `true`, then you are indicating that this action takes long time in executing (minutes or hours). In the current implementation OpenXava shows a progress bar. The default is `false`.
- (11)**confirm** (optional): If you set it to `true`, then before executing the action a dialog is shown to the user to ask if he is sure to execute it. The default is `false`.
- (12)**keystroke** (optional): Defines a keystroke that the user can press for executing this action. The possible values are the same as for `javax.swing.KeyStroke`. Examples: “*control A*”, “*alt x*”, “*F7*”. (*new in v2.0.1*)
- (13)**set** (several, optional): Sets a value of action properties. Thus the same action class can be configured in different ways and it can be used in several controllers.
- (14)**use-object** (several, optional): Assigns a session object to an action property just before executing the action. After the execution the property value is put back into the context again (update the session object, thus you can update even immutable objects).

Actions are short life objects, when a user clicks a button, then the action object is created, configured (with `set` and `use-object`) and executed. After that the session objects are updated, and finally the action object is discarded.

A plain controller might look like this:

```
<controller name="Remarks">
```

```

<action name="hideRemarks"
        class="org.openxava.test.actions.HideShowPropertyAction">
    <set property="property" value="remarks" />
    <set property="hide" value="true" />
    <use-object name="xava_view"/>
</action>
<action name="showRemarks" mode="detail"
        class="org.openxava.test.actions.HideShowPropertyAction">
    <set property="property" value="remarks" />
    <set property="hide" value="false" />
    <use-object name="xava_view"/>
</action>
<action name="setRemarks" mode="detail"
        class="org.openxava.test.actions.SetPropertyValueAction">
    <set property="property" value="remarks" />
    <set property="value" value="Hell in your eyes" />
    <use-object name="xava_view"/>
</action>
</controller>

```

Now you can include this controller into the module that you want; this is made by editing in *xava/application.xml* the module in which you can use these actions:

```

<module name="Deliveries">
    <model name="Delivery"/>
    <controller name="Typical"/>
    <controller name="Remarks"/>
</module>

```

Thus you have in your module the actions of `Typical` (CRUD and printing) plus these defined by you in the controller named `Remarks`. The button bar will have this aspect:



You can write code for `hideRemarks` like this:

```

package org.openxava.test.actions;

import org.openxava.actions.*;

/**

```

```

* @author Javier Paniza
*/

public class HideShowPropertyAction extends ViewBaseAction { // (1)

    private boolean hide;
    private String property;

    public void execute() throws Exception { // (2)
        getView().setHidden(property, hide); // (3)
    }

    public boolean isHide() {
        return hide;
    }

    public void setHide(boolean b) {
        hide = b;
    }

    public String getProperty() {
        return property;
    }

    public void setProperty(String string) {
        property = string;
    }

}

```

An action must implement `IAction`, but usually it extends from a base class that implements this interface. The base action more basic is `BaseAction` that implements most of the method of `IAction` except `execute()`. In this case you use `ViewBaseAction` as base class. `ViewBaseAction` has the property `view` of type `View`. This joined to the next declaration in action...

```
<use-object name="xava_view"/>
```

...allows to manage the view (the user interface) from an action using `view`.

The `<use-object />` gets the session object `xava_view` and assigns it to the property `view` (removing the prefix `xava_`, in general removes the prefix `myapplication_` before assigning object to property) of your action just before calling `execute()`.

Now inside the `execute()` method you can use `getView()` as you want (3), in this case for hiding a property. You can see all `View` possibilities in the JavaDoc of `org.openxava.view.View`.

With...

```
<set property="property" value="remarks" />
<set property="hide" value="true" />
```

you can set constant values to the properties of your action.

7.3 Controllers inheritance

You can create a controller that inherits all actions from one or more controllers. An example of this is the generic controller called `Typical`, this controller is in `OpenXava/xava/controllers.xml`:

```
<controller name="Typical">
  <extends controller="Print"/>
  <extends controller="CRUD"/>
</controller>
```

When you assign the controller `Typical` to a module this module will have available all actions of `Print` controller (to generate PDF reports and export to Excel) and `CRUD` controller (to Create, Read, Update and Delete)

You can use inheritance to refine the way a standard controller works, e. g. like this:

```
<controller name="Families">
  <extends controller="Typical"/>
  <action name="new" image="images/new.gif"
    class="org.openxava.test.actions.CreateNewFamilyAction">
    <use-object name="xava_view"/>
  </action>
</controller>
```

As you see the name of your action `new` matches with an action in `Typical` controller (in reality in `CRUD` controller from which extends `Typical`). In this case the original action is ignored and your action is used. Thus you can put your own logic to execute when a final user clicks the 'new' link.

7.4 List mode actions

You can write actions that apply to several objects. These actions are usually are shown in list mode only and normally have effects on the objects chosen by user only.

An example can be:

```
<action name="deleteSelected" mode="list" (1)
  confirm="true" (2)
  class="org.openxava.actions.DeleteSelectedAction">
    <use-object name="xava_tab"/> (3)
  </action>
```

You set `mode="list"` in order to show it only in list mode (1), and you use the session object `xava_tab` that allows you to access the data displayed in the list (3). Since this action deletes

records you require that the user must confirm explicitly before the action is executed (2).

The action source code:

```
package org.openxava.actions;

import java.util.*;

import org.openxava.model.*;
import org.openxava.tab.*;
import org.openxava.validators.*;

/**
 * @author Javier Paniza
 */

public class DeleteSelectedAction extends BaseAction implements IModelAction { // (1)

    private Tab tab; // (2)
    private String model;

    public void execute() throws Exception {
        int [] selectedOnes = tab.getSelected(); // (3)
        if (selectedOnes != null) {
            for (int i = 0; i < selectedOnes.length; i++) {
                Map clave = (Map)
                    getTab().getTableModel().getObjectAt(selectedOnes[i]);
                try {
                    MapFacade.remove(model, clave); // (4)
                }
                catch (ValidationException ex) {
                    addError("no_delete_row", new Integer(i), clave); // (5)
                    addErrors(ex.getErrors());
                }
                catch (Exception ex) {
                    addError("no_delete_row", new Integer(i), clave);
                }
            }
            getTab().deselectAll(); // (6)
            resetDescriptionsCache(); // (7)
        }
    }
}
```

```

    public Tab getTab() {                                     // (2)
        return tab;
    }

    public void setTab(Tab web) {                             // (2)
        tab = web;
    }

    public void setModel(String modelName) {                 // (8)
        this.model = modelName;
    }

}

```

This action is a standard action of OpenXava, but it allows you to see the things that you can do within an action in list mode. You can observe (1) how the action extends from `BaseAction` and implements `IModelAction`. Since it extends from `BaseAction` it has a group of utilities and you don't need to implement all methods of `IAction`; and as it implements `IModelAction` this action has a method called `setModel()` (8) that receives the model name (the name of OpenXava component) before executing it.

You have a property `tab` of type `org.openxava.tab.Tab` (2), and this is joined to the next definition in your action...

```
<use-object name="xava_tab"/>
```

... allows you to manage the list of displayed objects. For example, with `tab.getSelected()` (3) you obtain the indexes of selected rows, with `getTab().getTableModel()` a table model to access to data, and with `getTab().deselectAll()` you deselect the rows. You can take a look of `org.openxava.tab.Tab` JavaDoc for more details on its possibilities.

Something very interesting you can see in this example is the use of `MapFacade` (2). `MapFacade` allows you to access the data model using Java maps (`java.util.Map`). This is useful, if you get data from `Tab` or `View` in `Map` format and you want to update the model (and therefore the database) with it, or vice versa. All generic classes of OpenXava use `MapFacade` to manage the model and you also can use `MapFacade`. As general design tip: working with maps is useful in the case of generic logic, but if you need to program specific things it is better to use directly the object of model layer (the POJOs or EJBs generated by OpenXava). For more details have a look at the JavaDoc of `org.openxava.model.MapFacade`.

You see here how to display messages to the user with `addError()`. The `addError()` method receives the id of an entry in your *i18n* files and the argument to send to the message. The added messages are displayed to the user as errors. If you want to add warning or informative messages you can use `addMessage()` whose behavior is like `addError()`. The *i18n* files that hold errors and messages must be called *MyProject-messages.properties* and the language suffix (`_en`, `_ca`, `_es`, `_it`, etc). You can see the examples in *OpenXavaTest/xava/i18n*. All not caught exceptions produces a generic error messages, except if the not caught exception is of the type `ValidationException`. In this case the message exception is displayed.

The `resetDescriptionsCache()` (7) method deletes all cache entries used by OpenXava to display descriptions list (combos). It's a good idea to call it whenever data is updated.

You can see more possibilities in `org.openxava.actions.BaseAction` JavaDoc.

7.5 Overwriting default search

When a module is shown in list mode and the user clicks to display a detail, then OpenXava searches the corresponding object and displays it in detail. Now, if in detail mode the user fills the key fields and clicks on search (the binoculars), it also does the same. And when the user navigates by the records clicking the next or previous buttons then it does the same search. How can you customize this search? Let's see that:

You only need to define the module in *xava/application.xml* this way:

```
<module name="Deliveries">
    <env-var name="XAVA_SEARCH_ACTION" value="Deliveries.search"/>
    <model name="Delivery"/>
    <controller name="Typical"/>
    <controller name="Remarks"/>
    <controller name="Deliveries"/>
</module>
```

You see how it is necessary to define an environment variable named `XAVA_SEARCH_ACTION` that contains the action that you want to use for searching. This action is defined in *xava/controllers.xml*:

```
<controller name="Deliveries">
    <action name="search" mode="detail"
        by-default="if-possible" hidden="true"
        class="org.openxava.test.actions.SearchDeliveryAction"
        keystroke="F8">
        <use-object name="xava_view"/>
    </action>
    ...
</controller>
```

And its code:

```
package org.openxava.test.actions;

import java.util.*;

import org.openxava.actions.*;
import org.openxava.util.*;

/**
```

```

* @author Javier Paniza
*/

public class SearchDeliveryAction extends SearchByKeyAction {           // (1)

    public void execute() throws Exception {

        super.execute();                                                // (2)
        if (!Is.emptyString(getView().getValueString("employee"))) {
            getView().setValue("deliveredBy", new Integer(1));
            getView().setHidden("carrier", true);
            getView().setHidden("employee", false);
        }
        else {
            Map carrier = (Map) getView().getValue("carrier");
            if (!(carrier == null || carrier.isEmpty())) {
                getView().setValue("deliveredBy", new Integer(2));
                getView().setHidden("carrier", false);
                getView().setHidden("employee", true);
            }
            else {
                getView().setHidden("carrier", true);
                getView().setHidden("employee", true);
            }
        }
    }
}

```

In this action you have to search the database (or through EJB, JDO or Hibernate) and fill the view. Most times it is better that it extends `SearchByKeyAction` (1) and within `execute()` write a `super.execute()` (2).

OpenXava comes with 2 predefined search actions:

- `CRUD.searchByKey`: This is the default one. It does a search using the key values in the view, it executes no event.
- `CRUD.searchExecutingOnChange`: Works as `searchByKey` but it executes the on-change actions after search data.

If you want that the on-change actions will be executed on search then you must define your module this way:

```

<module name="Products3ChangeActionsOnSearch">
    <env-var name="XAVA_SEARCH_ACTION" value="CRUD.searchExecutingOnChange"/>

```

```

<model name="Product3"/>
<view name="WithDescriptionsList"/>
<controller name="Typical"/>
<controller name="Products3"/>
<mode-controller name="Void"/>
</module>

```

As you see, simply by setting the value of the `XAVA_SEARCH_ACTION` environment variable.

7.6 Initialize a module with an action

By setting `on-create="true"` when you define an action, you configure that this action will be executed automatically when the module is executed for the first time. This is a chance to initialize the module. Let's see an example. In your *controllers.xml* you write:

```

<controller name="Invoices2002">
    <action name="init" on-init="true" hidden="true"
        class="org.openxava.test.actions.InitDefaultYearTo2002Action">
        <use-object name="xavatest_defaultYear"/>
        <use-object name="xava_tab"/>
    </action>
    ...
</controller>

```

And in your action:

```

package org.openxava.test.actions;

import org.openxava.actions.*;
import org.openxava.tab.*;

/**
 * @author Javier Paniza
 */

public class InitDefaultYearTo2002Action extends BaseAction {

    private int defaultYear;
    private Tab tab;

    public void execute() throws Exception {
        setDefaultYear(2002); // (1)
        tab.setTitleVisible(true); // (2)
        tab.setTitleArgument(new Integer(2002)); // (3)
    }
}

```

```

    }

    public int getDefaultYear() {
        return defaultYear;
    }

    public void setDefaultYear(int i) {
        defaultYear = i;
    }

    public Tab getTab() {
        return tab;
    }

    public void setTab(Tab tab) {
        this.tab = tab;
    }
}

```

In this action you set the default year to 2002 (1), you make the title list visible (2) and you assign a value as an argument to that title (3). The list title is defined in the *i18n* files, usually it's used for reports, but you can show it in list mode too.

7.7 Calling another module

Sometimes it's convenient to call programmatically one module from another one. For example, imagine that you want to show a list of customers and when the user clicks on one customer, then a list of its invoices is displayed and the user can choose an invoice to edit. One way to obtain this effect is to have a module with only list mode and when the user clicks on a detail, the user is directed to an invoices module that shows only the invoices of the chosen customer. Let's see it. First you need to define the module in *application.xml* this way:

```

<module name="InvoicesFromCustomers">
    <env-var name="XAVA_LIST_ACTION" value="Invoices.listOfCustomer"/>      (1)
    <model name="Customer"/>
    <controller name="Print"/>
    <controller name="ListOnly"/>                                           (2)
    <mode-controller name="Void"/>                                         (3)
</module>

```

In this module only the list is shown (without detail part), for this you set the mode controller to `Void` (3) thus 'detail' and 'list' links are not displayed; and also you add a controller called `ListOnly` (2) in order to show the list mode, and only the list mode (if you only set the mode controller to `Void` the detail, and only the detail is displayed). Moreover you declare the variable

XAVA_LIST_ACTION to define your custom action. When the user clicks the link in each row, then your own action will be executed. You must declare this action in *controllers.xml*:

```
<controller name="Invoices">
    <action name="listOfCustomer" hidden="true"
        class="org.openxava.test.actions.ListCustomerInvoicesAction">
        <use-object name="xava_tab"/>
    </action>
    ...
</controller>
```

And the action code:

```
package org.openxava.test.actions;

import java.util.*;

import org.openxava.actions.*;
import org.openxava.controller.*;
import org.openxava.tab.*;

/**
 * @author Javier Paniza
 */
public class ListCustomerInvoicesAction extends BaseAction
    implements IChangeModuleAction, // (1)
               IModuleContextAction { // (2)

    private int row; // (3)
    private Tab tab;
    private ModuleContext context;

    public void execute() throws Exception {
        Map customerKey = (Map) tab.getTableModel().getObjectAt(row); // (4)
        int customerNumber = ((Integer) customerKey.get("number")).intValue();
        Tab invoiceTab = (Tab)
            context.get("OpenXavaTest", getNextModule(), "xava_tab"); // (5)
        invoiceTab.setBaseCondition("${customer.number} = "+customerNumber); // (6)
    }

    public int getRow() { // (3)
        return row;
    }
}
```

```

    public void setRow(int row) {                                     // (3)
        this.row = row;
    }

    public Tab getTab() {
        return tab;
    }
    public void setTab(Tab tab) {
        this.tab = tab;
    }

    public String getNextModule() {                                  // (7)
        return "CustomerInvoices";
    }

    public void setContext(ModuleContext context) {                  // (8)
        this.context = context;
    }

    public boolean hasReinitNextModule() {                           // (9)
        return true;
    }
}

```

In order to change to another module the action implements `ICheckModuleAction` (1) thus forces the action to have a method called `getNextModule()` (7). This will indicate to which module OpenXava will switch after executing this action. The method `hasReinitNextModule()` (9) indicates, whether you want that the target module has re-initiated on changing to it.

On the other hand this action implements `IModuleContextAction` (2) too and therefore it receives an object of type `ModuleContext` with the method `setContext()` (8). `ModuleContext` allows you to access the session objects of others modules. This is useful to configure the target module before changing to it.

Another detail is that the action specified in `XAVA_LIST_ACTION` must have a property named `row` (3); before executing the action this property is filled with the row number that user has clicked.

If you keep in mind the above details it is easy to understand the action:

- Gets the key of the object associated to the clicked row (4), to do this it uses the `tab` of the current module.
- Accesses to the `tab` of the target module using `context` (5).
- Sets the base condition of the `tab` of target module using the key obtained from current `tab`.

7.8 Changing the module of current view

As an alternative to change the module you can choose changing the model of the current view. This is easy, you only need to use the APIs available in `View`. An example:

```
public void execute() throws Exception {
    try {
        setInvoiceValues(getView().getValues()); // (1)
        Object number = getCollectionElementView().getValue("product.number");
        Map key = new HashMap();
        key.put("number", number);
        getView().setModelName("Product"); // (2)
        getView().setValues(key); // (3)
        getView().findObject(); // (4)
        getView().setKeyEditable(false);
        getView().setEditable(false);
    }
    catch (ObjectNotFoundException ex) {
        getView().clear();
        addError("object_not_found");
    }
    catch (Exception ex) {
        ex.printStackTrace();
        addError("system_error");
    }
}
```

This is an extract of an action that allows to visualize an object of another type. First you need to memorize the current displayed data (1), to restore it on returning. After this, you change the model of `view` (2), this is the important part. Finally you fill the key values (3) and use `findObject()` (4) to load all data in the view.

When you use this technique you have to keep in mind that each module has only one `xava_view` object active at a time, thus if you wish to go back you have the responsibility of restoring the original model in the view and restoring the original data.

7.9 Go to a JSP page

The automatic view generator of OpenXava is good for most cases, but it can be required to display a JSP page hand-written by you. You can do this with an action like this:

```
package org.openxava.test.actions;

import org.openxava.actions.*;

/**
```

```

* @author Javier Paniza
*/

public class MySearchAction extends BaseAction implements INavigationAction { // (1)

    public void execute() throws Exception {
    }

    public String[] getNextControllers() { // (2)
        return new String [] { "MyReference" } ;
    }

    public String getCustomView() { // (3)
        return "doYouWishSearch.jsp";
    }

    public void setKeyProperty(String s) {
    }

}

```

In order to go to a custom view (in this case a JSP page) your action has to implement `INavigationAction` (`ICustomViewAction` is enough). This way you can indicate with `getNextControllers()` (2) the next controllers to use and with `getCustomView()` (3) the JSP page to display (3).

7.10 Generating a custom report with JasperReports

OpenXava allows the final user to generate their own reports from the list model. The user can perform filtering, ordering, adding/removing fields, changing the positions of the fields and then generate a PDF report of the list.

But in all non-trivial business application you need to create programatically your own reports. You can do that easily by using *JasperReports* and then by integrating the reports into your OpenXava application with the action `JasperReportBaseAction`.

In the first place you need to design your report with *JasperReports*, you can use *iReport* an excellent designer for *JasperReports*.

Then you can write your action to print the report in this way:

```

package org.openxava.test.actions;

import java.util.*;

import net.sf.jasperreports.engine.*;
import net.sf.jasperreports.engine.data.*;

```



```

import org.openxava.actions.*;
import org.openxava.model.*;
import org.openxava.test.model.*;
import org.openxava.util.*;
import org.openxava.validators.*;

/**
 * Report of products of the selected subfamily. <p>
 *
 * Uses JasperReports. <br>
 *
 * @author Javier Paniza
 */
public class FamilyProductsReportAction extends JasperReportBaseAction {    // (1)

    private ISubfamily2 subfamily;

    public Map getParameters() throws Exception {    // (2)
        Messages errors =
            MapFacade.validate("FilterBySubfamily", getView().getValues());
        if (errors.contains()) throw new ValidationException(errors);    // (3)
        Map parameters = new HashMap();
        parameters.put("family", getSubfamily().getFamily().getDescription());

        parameters.put("subfamily", getSubfamily().getDescription());
        return parameters;
    }

    protected JRDataSource getDataSource() throws Exception {    // (4)
        return new JRBeanCollectionDataSource(getSubfamily().getProductsValues());
    }

    protected String getJRXML() {    // (5)
        return "Products.jrxml";
    }

    private ISubfamily2 getSubfamily() throws Exception {
        if (subfamily == null) {
            int subfamilyNumber = getView().getValueInt("subfamily.number");
            // Using Hibernate, the usual case
            subfamily = (ISubfamily2)

```

```

        XHibernate.getSession().get(
            Subfamily2.class, new Integer(subfamilyNumber));

        // Using EJB
        //subfamily =      Subfamily2Util.getHome().
        //      findByPrimaryKey(new Subfamily2Key(subfamilyNumber));
    }
    return subfamily;
}
}

```

Your action has to extend `JasperReportBaseAction` (1) and it has to overwrite the next three method:

- `getParameters()` (2): A `Map` with the parameters to send to the report, in this case we validate the input data (using `MapFacade.validate()`) before (3).
- `getDataSource()` (4): A `JRDataSource` with data to print. In this case it is a collection of `JavaBeans` obtained calling a method of the model object. If you use EJB be careful and do not loop over an EJB collection inside this method, as in this case is better only one EJB call to obtain all data.
- `getJRXML()` (5): The XML with the *JasperReports* design, this file must to be in the classpath. You may have a source code folder called *reports* in your project to hold these files.

By default the report is displayed in a popup window, but if you wish the report in the current window, then you can overwrite the method `inNewWindow()`.

You can find more examples of `JasperReport` actions in the *OpenXavaTest* project, as `InvoiceReportAction` for printing an Invoice.

7.11 Uploading and processing a file from client (multipart form)

This feature allows you to process in your OpenXava application a binary file (or several) provided by the client. This is implemented in a HTTP/HTML context using HTML multipart forms, although the OpenXava code is technologically neutral, hence your action will be portable to another environment with no recoding.

In order to upload a file the first step is creating an action to direct to a form where the user can choose his file. This action must implements `ILoadFileAction` in this way:

```

public class ChangeImageAction extends BaseAction implements ILoadFileAction { // (1)
    ...
    public void execute() throws Exception { // (2)
    }

    public String[] getNextControllers() { // (3)
        return new String [] { "LoadImage" };
    }
}

```

```

    public String getCustomView() {                                // (4)
        return "xava/editors/changeImage";
    }

    public boolean isLoadFile() {                                  // (5)
        return true;
    }

    ...
}

```

An `ILoadFileAction` (1) action is also an `INavigationAction` action that allows you to navigate to another controller (3) and to a custom view (4). The new controller (3) usually will have an action of type `IProcessLoadedFileAction`. The method `isLoadFile()` (5) returns `true` in case that you want to navigate to the form to upload the file, you can use the logic in `execute()` (2) to determine this value. The custom view is (4) a JSP with your own form to upload the file.

An example of a JSP for a custom view is:

```

<%@ include file="../imports.jsp"%>

<jsp:useBean id="style" class="org.openxava.web.style.Style" scope="request"/>

<table>
<th align='left' class=<%=style.getLabel()%>>
<fmt:message key="enter_new_image"/>
</th>
<td>
<input name = "newImage" class=<%=style.getEditor()%> type="file" size='60' />
</td>
</table>

```

As you see, the HTML form is not specified, because the OpenXava module already has the form.

The last piece is the action for processing the uploaded files:

```

public class LoadImageAction extends BaseAction
    implements INavigationAction, IProcessLoadedFileAction {        // (1)

    private List fileItems;
    private View view;
    private String newImageProperty;

    public void execute() throws Exception {

```

```

        Iterator i = getFileItems().iterator(); // (2)
        while (i.hasNext()) {
            FileItem fi = (FileItem)i.next(); // (3)
            String fileName = fi.getName();
            if (!Is.emptyString(fileName)) {
                getView().setValue(getNewImageProperty(), fi.get()); // (4)
            }
        }
    }

    public String[] getNextControllers() {
        return DEFAULT_CONTROLLERS;
    }

    public String getCustomView() {
        return DEFAULT_VIEW;
    }

    public List getFileItems() {
        return fileItems;
    }

    public void setFileItems(List fileItems) { // (5)
        this.fileItems = fileItems;
    }

    ...
}

```

The action implements `IProcessLoadedFileAction` (1), thus the action must have a method `setFileItem()` (5) to receive the list of uploaded files. This list can be processed in `execute()` (2). The elements of the collection are of type `org.apache.commons.fileupload.FileItem` (4)(from fileupload project of apache commons). Only calling to `get()` (4) in the file item you will access to the content of the uploaded file.

7.12 All action types

You have seen until now that the behavior of your actions depends on which interfaces they implement. Next the available interfaces for actions are enumerated:

- `IAction`: Basic interface to be implemented by all actions.
- `IChainAction`: Allows you to chain actions, that is when the execution of the action finishes, then the next action is executed immediately.
- `IChangeControllersAction`: To change the controller (the actions) available to user.
- `IChangeModeAction`: To change the mode, from list to detail or vice versa.

- `ICheckModuleAction`: To change the module.
- `ICustomViewAction`: To use as view your custom JSP.
- `IForwardAction`: Redirects to a Servlet or JSP page. It is not like `ICustomViewAction`; `ICustomViewAction` puts your JSP inside the user interface generated by OpenXava (that can be inside a portal), while `IForwardAction` redirects completely to the specified URI.
- `IHideActionAction`, `IHideActionsAction`: Allows to hide an action or a group of actions in the User Interface. (*new in v2.0*)
- `IJDBCAction`: Allows to use JDBC in an action directly. It receives an `IConnectionProvider`. Works like an `IJBCCalculator` (see chapter 3).
- `ILoadFileAction`: Navigates to a view that allows the final user to load a file.
- `IModelAction`: An action that receives the model name.
- `IModuleContextAction`: Gets a `ModuleContext` in order to access the session objects of other modules.
- `INavigationAction`: Extends from `ICheckControllersAction` and `ICustomViewAction`.
- `IONChangePropertyAction`: This interface must be implemented by the actions that react to the value change event in the user interface.
- `IProcessLoadedFileAction`: Processes a list of files uploaded from client to server.
- `IRemoteAction`: Useful when you use EJBs. Well used it can be a good substitute for a `SessionBean`.
- `IRequestAction`: Receives a servlet request. This type of actions links your application to the Servlet/JSP technology, hence it is better avoiding it. But sometimes a little bit of flexibility is needed.
- `IShowActionAction`, `IShowActionsAction`: Allows to show an action or a group of actions previously hidden in an `IHideAction(s)Action`. (*new in v2.0*)

If you wish to learn more about actions the best thing you can do is to have a look at the JavaDoc API of the package `org.openxava.actions` and to try out the examples of OpenXavaTest project.

An application is the software that the final user can use. For now you have seen how to define the pieces that make up an application (mainly the components and the actions), now you will learn how to assemble these pieces in order to create applications.

The definition of an OpenXava application is given in the file *application.xml* that can be found in the *xava* directory of your project.

The file syntax is:

```
<application
  name="name"                (1)
  label="label"              (2)
>

  <module ... /> ...        (3)
</application>
```

(1)name (required): Name of the application.

(2)label (optional): It's **much better** to use *i18n* files.

(3)module (several, required): Each module executable by final user.

In short, an application is a set of modules. Let's see how define a module:

```
<module
  name="name"                (1)
  folder="folder"            (2)
  label="label"              (3)
  description="description"   (4)
>
  <env-var ... /> ...        (5)
  <model ... />              (6)
  <view ... />               (7)
  <web-view ... />           (8)
  <tab ... />                (9)
  <controller ... /> ...     (10)
  <mode-controller ... />    (11)
  <doc ... />                (12)
</module>
```

(1)name (required): Unique identifier of the module within this application.

- (2)`folder` (optional): Folder in which the module will reside. It's a tip to classify the modules. For the moment it's used to generate a folder structure for JetSpeed2 but its use can be amplified in future. You can use `/` or `.` to indicate subfolders (for example, “invoicing/reports” or “invoicing.reports”).
- (3)`label` (optional): Short name to be shown to the user. It's **much better** to use *i18n* files.
- (4)`description` (optional): Long description to be shown to the user. It's **much better** to use *i18n* files.
- (5)`env-var` (several, optional): Allows you to define variables with a value that can be accessed by actions. Thus you can have actions configurable by module.
- (6)`model` (one, optional): Name of the component used in this module. If you leave it blank, then it is required to set the value to `web-view`.
- (7)`view` (one, optional): The view used to display the detail. If you leave it blank, then the default view will be used.
- (8)`web-view` (one, optional): Allows you to define a JSP page to be used as a view.
- (9)`tab` (one, optional): The tab used in list mode. If you do not specify it, then the default tab will be used.
- (10)`controller` (several, optional): Controllers with the available actions used initially.
- (11)`mode-controller` (one, optional): Allows to define the behavior to switch from detail to list mode and vice versa, as well as to define a module without detail and view (with no modes).
- (12)`doc` (one, optional): It's mutually exclusive with all other elements. It allows you to define a module that contains documentation only and no logic. Useful for generating informational portlets for your application.

8.1 Typical module example

Defining a simple module can be like this:

```
<application name="Management">
  <module name="Warehouses" folder="warehousing">
    <model name="Warehouse"/>
    <controller name="Typical"/>
    <controller name="Warehouses"/>
  </module>
  ...
</application>
```

In this case you have a module that allows the user to create, to delete, to update, to search, to generate PDF reports and to export to Excel the warehouses data (thanks to `Typical` controller) and also to execute actions only for warehouses (thank to `Warehouses` controller). In the case the system generates a module structure (as in JetSpeed2 case) this module will be in folder “warehousing”.

In order to execute this module you need to open your browser and go to:

<http://localhost:8080/Management/xava/module.jsp?application=Management&module=Warehouse>

Also a portlet is generated to allow you to deploy the module as a JSR-168 portlet in a Java portal.

8.2 Only detail module

A module with only detail mode, without list, is defined this way:

```
<module name="InvoicesNoList">
  <model name="Invoice"/>
  <controller name="Typical"/>
  <mode-controller name="Void"/>          (1)
</module>
```

Void (1) mode controller is for removing the “detail – list” links; in this case by default the module uses detail mode only.

8.3 Only list module

A module with only list mode, without detail, is defined this way:

```
<module name="InvoicesOnlyList">
  <model name="Invoice"/>
  <controller name="Typical"/>
  <controller name="ListOnly"/>          (1)
  <mode-controller name="Void"/>          (2)
</module>
```

Void (2) mode controller is for removing the “detail – list” links. Furthermore on defining ListOnly (1) as controller the module changes to list mode on init, so this is an only list module.

8.4 Documentation module

A documentation module can only display a HTML document. It's easy to define:

```
<module name="Description">
  <doc url="doc/description" languages="en,es"/>
</module>
```

This module shows the document *web/doc/description_en.html* or *web/doc/description_es.html* depending on the browser language. If the browser language is not English nor Spanish then it assumes English (the first specified language). If you do not specify the language, then the document *web/doc/description.html* is shown.

This is useful for informational portlets. This type of module has no effect outside a portal environment.

The syntax for *application.xml* is not difficult. You can see more examples in

OpenXavaTest/xava/application.xml.

9.1 Introduction to AOP

AOP (Aspect Oriented Programming) introduces a new way for reusing code. In fact *aspects* complement some shortcomings in traditional Object Oriented Programming.

Which problems does AOP resolve? Sometime you have a functionality that is common to a group of classes but using inheritance is not practical (in Java we only have single inheritance) or not ethical (because there isn't a *is-a* relationship). Moreover the system may be already written, or maybe you need to include or not this functionality on demand. AOP is an easy way to resolve these problems.

What is an aspect? An aspect is a bunch of code that can be scattered as you wish in your application.

The Java language has a complete AOP support by means of the AspectJ project.

OpenXava adds some support for the *aspects* concept since version 1.2.1. At the moment the support is small and OpenXava is still far away from an AOP framework, but the support of aspects in OpenXava is useful.

9.2 Aspects definition

The *aspects.xml* file inside the *xava* folder of your project is used to define aspects.

The file syntax is:

```
<aspects>
  <aspect ... /> ...           (1)
  <apply ... /> ...           (2)
</aspects>
```

(1)`aspect` (several, optional): To define aspects.

(2)`apply` (several, optional): To apply the defined aspects to the selected models.

With `aspect` (1) you can define an aspect (that is a group of features) with a name, and using `apply` (2) you achieve that a set of models (entities or aggregates) will have these features automatically.

Let's see the aspect syntax:

```
<aspect
  name="name"                 (1)
>
  <postcreate-calculator .../> ... (2)
  <postload-calculator .../> ...   (3)
```

```

    <postmodify-calculator .../> ...      (4)
    <preremove-calculator .../> ...      (5)
</aspect>

```

- (1)name (required): Name for this aspect. It must be unique.
- (2)postcreate-calculator (several, optional): All model with this aspect will have this postcreate-calculator implicitly.
- (3)postload-calculator (several, optional): All model with this aspect will have this postload-calculator implicitly.
- (4)postmodify-calculator (several, optional): All model with this aspect will have this postmodify-calculator implicitly.
- (5)preremove-calculator (several, optional): All model with this aspect will have this preremove-calculator implicitly.

Furthermore, you need to assign the defined aspects to your models. The syntax to do that is:

```

<apply
    aspect="aspect"                (1)
    for-models="models"            (2)
    except-for-models="models"     (3)
/>

```

- (1)aspect (required): The name of the aspect that you want to apply.
- (2)for-models (optional): A comma separated list of models to which the aspect is applied to. It's mutually exclusive with except-for-models attribute.
- (3)except-for-models (optional): A comma separated list of models to be excluded when apply this aspect. In this case the aspect applies to all models excepts the indicated ones. It's mutually exclusive with for-models attribute.

If you use neither `for-models` nor `except-for-models`, then the aspect will apply to all models in the application. Models are the names of components (for its entities) or aggregates.

A simple example may be:

```

<aspect name="MyAspect">
    <postcreate-calculator
        class="com.mycompany.myapplication.calculators.MyCalculator"/>
</aspect>
<apply aspect="MyAspect" />

```

Whenever a new object is created (saved in database for the first time), then the logic of `MyCalculator` is executed. And this for all models.

At the moment only these few calculators are supported. We expect to extend the power of *aspects* for OpenXava in the future. Anyway the existing calculators offer interesting possibilities. Let's see an example in the next section.

9.3 AccessTracking: A practical application of aspects

The current OpenXava distribution includes the *AccessTracking* project. This project defines an aspect that allows you to track all access to the data in your application. Actually, this project allows your application to comply the Spanish Data Protection Law (Ley de Protección de Datos) including high level security data. Although it's generic enough to be useful in a broad variety of scenarios.

9.3.1 The aspect definition

You can find the aspect definition in *AccessTracking/xava/aspects.xml*:

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

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

<!-- AccessTracking -->

<aspects>

    <aspect name="AccessTracking">
        <postcreate-calculator
            class="org.openxava.tracking.AccessTrackingCalculator">
                <set property="accessType" value="Create"/>
            </postcreate-calculator>
        <postload-calculator
            class="org.openxava.tracking.AccessTrackingCalculator">
                <set property="accessType" value="Read"/>
            </postload-calculator>
        <postmodify-calculator
            class="org.openxava.tracking.AccessTrackingCalculator">
                <set property="accessType" value="Update"/>
            </postmodify-calculator>
        <preremove-calculator
            class="org.openxava.tracking.AccessTrackingCalculator">
                <set property="accessType" value="Delete"/>
            </preremove-calculator>
        </aspect>
    </aspects>
```

When you apply this aspect to your components, then the code of *AccessTrackingCalculator* is executed each time a object is created, loaded, modified or removed. *AccessTrackingCalculator* writes a record into a database table with information about the access.

In order to apply this aspect you need to write your *aspects.xml* like this:

```

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

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

<aspects>

    <apply aspect="AccessTracking" for-models="Warehouse, Invoice"/>

</aspects>

```

In this way this aspect is applied to `Warehouse` and `Invoice`. All access to these entities will be record in a database table.

9.3.2 Setup AccessTracking

If you want to use the `AccessTracking` aspect in your project you have to follow the next setup steps:

- Add *AccessTracking* as referenced project.
- Create the table in your database to store the tracking of accesses. You can find the CREATE TABLES in *AccessTracking/data/access-tracking-db.script* file.
- You have to include the `hibernate.dialect` property in your configuration files. You can see examples of this in *OpenXavaTest/jboss-hypersonic.properties* and other *OpenXavaTest/xxx.properties* files.
- Inside the *AccessTracking* project you need to select a configuration (editing *build.xml*) and regenerate hibernate code (using the ant target `generateHibernate`) for *AccessTracking* project.
- Edit the file of your project *build/ejb/META-INF/MANIFEST.MF* to add the next jars into the classpath: `./lib/tracking.jar ./lib/ehcache.jar ./lib/antlr.jar ./lib/asm.jar ./lib/cglib.jar ./lib/hibernate3.jar ./lib/dom4j.jar`. (This step isn't needed if you use only POJOs, not EJB CMP2, *new in v2.0*)

Also you need to modify the target `createEJBJars` (only if you are using EJB2 CMP) and `deployWar` of your *build.xml* in this way:

```

<target name="createEJBJars">    <!-- 'createEJBJars' only if you use EJB2 CMP -->
    ...
    <ant antfile="../AccessTracking/build.xml" target="createEJBTracker"/>
</target>
<target name="deployWar">
    <ant antfile="../AccessTracking/build.xml" target="createTracker"/>
    ...
</target>

```

After these steps, you have to apply the aspect in your application. Create a file in your project *xava/aspects.xml*:

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

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

<aspects>

    <apply aspect="AccessTracking"/>

</aspects>
```

Now you only have to deploy the war for your project. (*new in v2.0*)

In the case that you are using EJB2 CMP you have to regenerate the code, deploying EJB and deploying war for your project.

All access are recorded in a table with the name TRACKING.ACCESS. If you want you can deploy the module web or the portlet of *AccessTracking* project in order to have a web application to browse the accesses.

For more details you can have a look at the *OpenXavaTest* project.

10.1 Many-to-many relationships

In OpenXava there is no direct concept of a many-to-many relationship, only collections are available. Nevertheless modeling a many-to-many relationship in OpenXava is easy. You only need to define collections in both sides of the relationship.

For example, if you have customers and states, and a customer can work in several states, and, obviously, in a state several customers can work, then you have a many-to-many (using relational nomenclature) relationship. Suppose that you have a table CUSTOMER (without reference to state), a table STATE (without reference to customer) and a table CUSTOMER_STATE (to link both tables). Then you can model this case in this way:

```
<component name="Customer">
  <entity>
    ...
    <collection name="states"> (1)
      <reference model="CustomerState"/>
    </collection>
    ...
  </entity>

  <aggregate name="CustomerState"> (2)
    <reference name="customer" key="true"/> (3)
    <reference name="state" key="true"/> (4)
  </aggregate>
  ...
</component>
```

You define in `Customer` a collection of aggregates (1), each aggregate (`CustomerState`) (2) contains a reference to a `State` (4), and, of course, a reference of its container entity (`Customer`) (3).

Then you map this collection in the usual way:

```
<component name="Customer">
  ...
  <aggregate-mapping aggregate="CustomerState" table="CUSTOMER_STATE">
    <reference-mapping reference="customer">
      <reference-mapping-detail
        column="CUSTOMER" (1)
      </reference-mapping-detail>
    </reference-mapping>
  </aggregate-mapping>
</component>
```

```

        referenced-model-property="number" />
    </reference-mapping>
    <reference-mapping reference="state">
        <reference-mapping-detail
            column="STATE" (2)
            referenced-model-property="id" />
        </reference-mapping>
    </aggregate-mapping>
</component>

```

CustomerState is mapped to CUSTOMER_STATE, a table that only contains two columns, one to link to CUSTOMER (1) and other to link to STATE (2).

At model and mapping level all is right, but the User Interface generated by default by OpenXava is somewhat cumbersome in this case. Although with the next refinements to the view part your many-to-many collection will be just fine:

```

<component name="Customer">
    ...
    <view>
        ...
        <collection-view collection="states">
            <list-properties>state.id, state.name</list-properties> (1)
        </collection-view>

        <members>
            ...
            states
            ...
        </members>

    </view>

    <view model="CustomerState">
        <reference-view reference="state" frame="false" /> (2)
    </view>
    ...
</component>

```

In this view you can see how we define explicitly (1) the properties to be shown in list of the collection states. This is needed because we have to show the properties of the State, not the CustomerState ones. Additionally, we define that reference to State in CustomerState view to be showed without frames (2), this is to avoid two ugly nested frames.

By this easy way you can define a collection to map a many-to-many relationship in the database. If

you want a bidirectional relationship only need to create a `customers` collection in `State` entity, this collection may be of the aggregate type `StateCustomer` and must be mapped to the table `CUSTOMER_STATE`. All in analogy to the example here.

10.2 Programming with Hibernate

You can use Hibernate APIs in any part of an OpenXava application, that is, inside calculators, validators, actions, filters, etc.

In order to facilitate the use of Hibernate OpenXava provides the `XHibernate` class. For example, if you wish to store an object in a database using the Hibernate API, the normal way would be:

```
Session session = HibernateUtil.getSessionFactory().getCurrentSession();
session.beginTransaction();
Customer customer = ... ;
session.save(customer);
session.getTransaction().commit();
session.close();
```

But, inside OpenXava and using `XHibernate` class you can write this:

```
Customer customer = ... ;
XHibernate.getSession().save(customer);
```

No more.

The first time that you call to `XHibernate.getSession()` a session is created and assigned to the current thread and a transaction is created too; the next time that you call it, the same Hibernate session is used. At the end of the complete cycle of action execution, OpenXava commits automatically the transaction and closes the session. Moreover, `XHibernate.getSession()` works well inside and outside of a CMT environment.

You can optionally commit the transaction in any moment calling to `XHibernate.commit()`, if after this you use `XHibernate.getSession()` an new session and a new transaction are created for you.

You can learn more seeing the API doc of `org.openxava.hibernate.XHibernate` class.

10.3 Custom JSP view and OpenXava taglibs

Obviously the better way to create user interfaces is using the view section of components as explained in chapter 4. But, in extreme cases perhaps you have to define your view using JSP. OpenXava allows you to do it. And in order to help you to do it, you can use some JSP taglibs provided by OpenXava. Let's see an example.

10.3.1 Example

First you have to define in your module that you want to use your own JSP, in `application.xml`:

```
<module name="SellersJSP" folder="invoicing.variations">
  <model name="Seller"/>
```

```
<view name="ForCustomJSP"/> (1)
<web-view url="custom-jsp/seller.jsp"/> (2)
<controller name="Typical"/>
</module>
```

If you use `web-view` (2) on defining your module, OpenXava uses your JSP to render the detail, instead of generating the view automatically. Optionally you can define an OpenXava view using `view` (1), this view is used to know the events to throw and the properties to populate, if not it is specified the default view of the component is used; although it's advisable to create an explicit OpenXava view for your JSP custom view, in this way you can control the events, the properties to populate, the focus order, etc explicitly. You can put your JSP inside *web/custom-jsp* folder of your project, and it can be as this one:

```
<%@ include file="../../xava/imports.jsp"%>

<table>
<tr>
    <td>Number: </td>
    <td>
        <xava:editor property="number"/>
    </td>
</tr>
<tr>
    <td>Name: </td>
    <td>
        <xava:editor property="name"/>
    </td>
</tr>
<tr>
    <td>Level: </td>
    <td>
        <xava:editor property="level.id"/>
        <xava:editor property="level.description"/>
    </td>
</tr>
</table>
```

You are free to create your JSP file as you like, but it can be useful to use OpenXava taglibs, in this case, for example the `<xava:editor/>` taglib is used, this renders an editor suitable for the indicated property, furthermore add the needed javascript to throw the events. If you use `<xava:editor/>`, you can manage the displayed data using `xava_view` (of `org.openxava.view.View` type) object, therefore all standard OpenXava controllers (including CRUD) work.

You can notice that qualified properties are allowed (as `level.id` or `level.description`) (*new v2.0.1*), furthermore when you fill `level.id`, `level.description` is populated with the corresponding value. Yes, all the behaviour of an OpenXava view is available inside your JSP if you use the OpenXava taglibs.

Let's see the OpenXava taglibs.

10.3.2 xava:editor

The `<xava:editor/>` tag allows you to render an editor (a HTML control) for your property, in the same way that OpenXava does it when it generates the user interface automatically.

```
<xava:editor
  property="propertyName"           (1)
  editable="true|false"             (2) new in v2.0.1
  throwPropertyChanged="true|false" (3) new in v2.0.1
/>
```

- (1) `property` (required): It's the property of the model associated with the current module
- (2) `editable` (optional): *New in v2.0.1*. Forces to this editor to be editable, otherwise the appropriate default value is assumed.
- (3) `throwPropertyChanged` (optional): *New in v2.0.1*. Forces to this editor to throws property changed event, otherwise the appropriate default value is assumed.

This tag generates the needed JavaScript in order to allow your view to work in the same way as an automatic one. The qualified properties (properties of references) are supported (*new in v2.0.1*).

10.3.3 xava:action, xava:link, xava:image, xava:button

The `<xava:action/>` tag allows you to render an action (a button or a image that the user can click).

```
<xava:action action="controller.action" argv="argv"/>
```

The `action` attribute indicates the action to execute, and the `argv` attribute (optional) allows you to put values to some properties of the action before execute it. One example:

```
<xava:action action="CRUD.save" argv="resetAfter=true"/>
```

When the user clicks on it, then it executes the action `CRUD.save`, before it puts `true` to the `resetAfter` property of the action.

The action is rendered as an image, if it has an image associated. Otherwise it is rendered as a button. If you want to determine the render style, then you can use directly the next taglibs: `<xava:button/>`, `<xava:image/>` or `<xava:link/>` similars to `<xava:action/>`.