
VERSION 2.0

# REFERENCE GUIDE

# Contents

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 o 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 first steps it's better to use the tutorial (that you can find in the OpenXava web site or openxava.zip file). On the other hand, the definitive reference of OpenXava is the OpenXavaTest project, that contains all possibilities offered by OpenXava.

This guide suppose that you use Eclipse as IDE, although OpenXava does not use Eclipse resources and can be used without problem in other IDE including a simple editor plus command line. Also we assumed that your Eclipse point to *workspace* that comes with openxava.zip. If you have followed the tutorial instructions everything would have to 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 not 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?

- 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, page 28
- `IModelCalculator` added, as better alternative for `IEntityCalculator`: Section, 3.8.4, page 32
- Nested sections are allowed in views: Section 4.1, page 63
- It's possible to access to environment variables inside a filter: Section 5.2, page 93
- `cmp-type` allowed for reference mapping: Section 6.3, page 100
- It's possible to hide and to show actions from User Interface: Section 7.12, page 127 and 128
- 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 application quickly and easily.

The name OpenXava come from **Open** source, **X**ML and **Java**. And the underlaying 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 needed flexibility to develop the most advances features as you like.

Below you can see some basic concept 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. That is to say, all needed information that the system has to know about the invoice concept 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, that is, 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 you Java application

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

## 1.3 Controllers

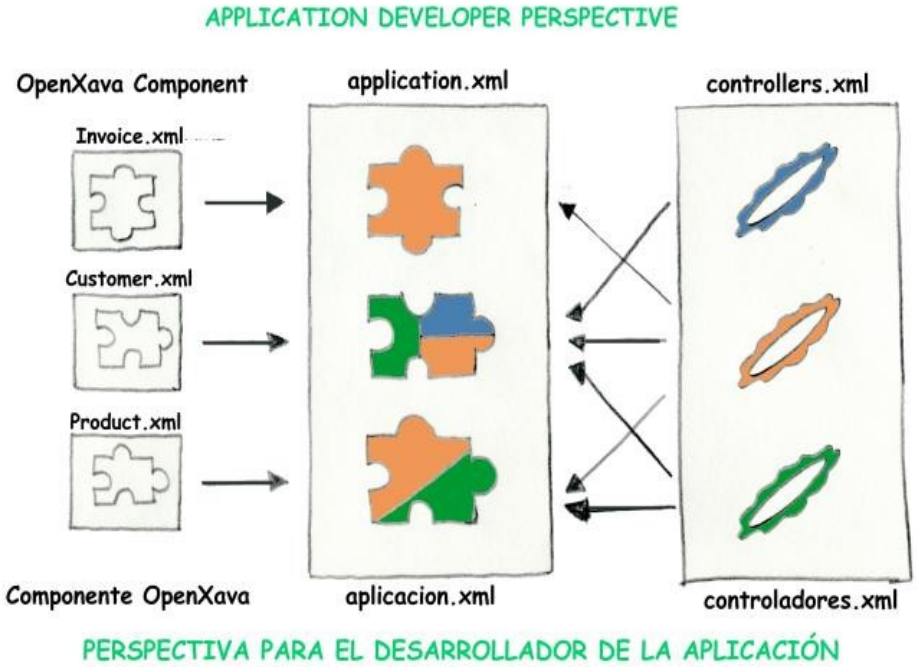
The business component does not define the things that user can do in the application; this is define in controllers. The controllers are in the file *xava/controllers.xml* of your project; in addition OpenXava has a set of predefined controllers in *OpenXava/xava/controllers.xml*.

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

The controllers are separated from business component because an unique controller can be assigned to several business component. For example, a controller to make CRUD operations, print in PDF format or export to plain files, etc. can be used and reused for 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 is configured as a portlet within a portal.

## 1.5 Project structure

A typical OpenXava project usually contains the next folders:

- `[root]`: In the root you can found *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 definitions of your business component.
- `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]`: Properties 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 you 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 another custom web resources.

## 1.6 Conclusion

This chapter introduces you a little concepts of OpenXava, the rest of this guide will extend all this information in detail.

## 2.1 Create a new project

First open your Eclipse and make its workspace the one that comes with OpenXava distribution (*openxava.zip*). To create a new project you have to edit the file *CreateNewProject.xml* in *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 thought to deploy 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. Configure a data source in a application server is out of the scope of this guide, nevertheless you have below concretes instructions to configure 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 directory *conf*. In this file you must 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

Create 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 2 parts clearly different, the firs 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 define only 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 match with primary key of 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 create or modify.

(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 property go far from what is shown here, you can see a more complete explanation in chapter 3.

On the other hand you have the mapping, where you associate you component with a table in the database; the syntax is obvious. The use of `@separator@` in table name allow you to have a application that run against 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 a ant target in Eclipse is creating it in *Externals Tools*. In this way when you need to re-execute only have to choose the option in menu. It's very important configure the target for refresh the *Management* project after executing it. In your *build.xml* you have already the most used ant targets.

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

With you 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. A 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 create 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 a 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 an Delete), moreover allows to generate PDF report and export to excel. From a visual viewpoint we can say that a `controller` defines the buttons showed to user and `model` defines the data, although is a very simple 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 as first line:

```
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 is time to see the fruit of your sweat. Let's go.

- Execute 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; the fact is that 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 in JUnit and HttpUnit. The OpenXava JUnit tests simulate the behavior of a real user with a browser, in this way you can replicate in an exact way the same test that you can do directly with a 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 usually you 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)
        assertValue("zoneNumber", "");              // (5)
        assertValue("number", "");
        assertValue("name", "");

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

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

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

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

    }

}
```

You can learn from this example:

(1)Constructor: In 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 that type in the field 'name' the text "Pepe". The values are always alphanumeric because are assigned to a HTML form.

(4) `assertNoErrors`: Verify that there are no errors. In 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 show the indicated informative message.

You can see that is very easy to test that a module works; write this code can take 5 minutes, but at end you will save hours of work, because from now 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) you test warn you just in time.

For more detail you can see 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 wish it's possible testing against 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 portal (this is useful if you assign your module to the 'guest' user).

## 2.8 The labels

Now everything works well, but remains a little detail yet. The labels showed to user are not appropriates (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 commons case (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 other language (Spanish for example) only need to copy an 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 define labels of your OpenXava elements please look in *OpenXavaTest/xava/i18n*.

## 2.9 Conclusion

In this chapter you have see how to create a new project, and you had a first taste to 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 support all OpenXava applications written until now using EJB. The option 3 can be useful in some circumstances. You can see how configure this in *OpenXavaTest/properties/xava.properties*.

## 3.2 Business Component

As you have seen the basic unit to create a OpenXava applications 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 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`; you can notice the life cycle of an invoice line is attached to the invoice, that is an invoice line without invoice have no meaning, and sharing an invoice line by various invoices is not possible, hence you will model `InvoiceDetail` as aggregate.

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 apply if you use Hibernate as persistence engine. In this case 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 apply 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* an *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 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 `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 is better a property level validator.

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

### 3.5 Bean (1)

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

For example:

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

In this simple way you can write you own Java code instead of left 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 reference from the OpenXava generated code to you own handwritten code you need that you Java class implements 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 wrapping you existing Java and Hibernate code with OpenXava. Of

course, if you are creating a new system is far better to left to OpenXava the code generation of all this stuff.

### 3.6 EJB (2)

With `<ejb/>` you can specify that you wish 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, you can put your own XDoclet classes in the package `model` (or another package of you choice, this depends on the value of `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 watch 2 little restrictions:

(1)The class must extend from `org.openxava.ejbx.EJBReplicableBase`

(2)It is required at least a `ejbCreate` (with its `ejbPostCreate`) that receive 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 you 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 well implements the interface.

In your generated code you can find a `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.

You can see as 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 work using POJOs exclusively you may choice using POJOs classes direclty ignoring the interfaces, as you wish.

### 3.8 Property (4)

A OpenXava property corresponds exactly to a Java property. It represents the state of an object that can be read and in some cases update. The object does not have the obligation to store physically the property data, 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 norm for property names, like starting with lower-case. Using underline (_) is not advisable.

(2)`label` (optional): Label showed to final user. Is **far better** use the *i18n* files.

(3)`type` (optional): It match with a Java type. All types valid for a Java property are valid here, this include class 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 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 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 presents, if not saving is not produced and a validation error list is returned. The logic to determine if a property is present or not can be configured creating a file called *validators.xml* in your project. You can see the syntax in *OpenXava/xava/validators.xml*.

(7)`key` (optional): Indicates if 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 developer but not for user. The hidden properties are excluded when automatic user interface is generated, however at Java code level are present and fully functional, even if you put it explicitly in a view the property will be shown in a user interface.

(9)`valid-values` (one, optional): To indicate 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 within 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 an stereotype to each case. That is, you can have the next sterotypes NAME, MEMO or DESCRIPTION and assign them to your properties.

OpenXava comes with the next 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*

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 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 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 assumes 40 as `size`, `String` as `type` and execute the `NotBlankCharacterValidator` validator 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 need to declare you property with the IMAGES_GALLERY stereotype , in this way:

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

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

And everything is done.

But, in order that your application supports this stereotype you need to setup your system.

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 other more suitable in your database to store `byte []` (for example `LONGVARBINARY`) .

The name of the table may be whatever your want. 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 to define a property that only can hold one of the indicated values. 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 *i18n* 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 by default saves a integer, but you can configure easily to use another type and work with no problem with legate 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, set value to property `euros` of `EurosToPesetasCalculator` with the value obtained from `unitPrice` of 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` on 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` have 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 have a method `setModel` (2), this method is called before calling `calculate()` method and thus allows access to the model object (in this case an invoice) that contains the property inside `calculate()`.

Within code generated by OpenXava you can find a 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` 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 choice cast directly to the POJO class, in this case `Invoice`.

Obviously this calculator type is less reusable than that which receives simple properties, but sometimes are useful.

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 IJDBCCalculator {       // (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 you calculator must implement `IJDBCCalculator` (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 can help to solve performance problems.

The calculators allow you to insert you custom logic in a system where all code is generated; and as you see 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 plug 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 the `year` field already has a value, that he can change if he want.

You can indicate that the value will be calculated just before creating (inserting in database) an object for the first time; this is doing 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.

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 property just before storing. A property may has 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 that 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 oblige to 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 wish obtain significant message you need to add to you *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 successfully 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 displaying them and does not save object.

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

## 3.9 Reference (5)

A reference allows access from a entity or 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 final user. It's **far better** use *i18n*.

(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 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. On 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 collection. 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 aggregated 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 `<defaut-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 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 **far better** to use *i18n* files.

(3)`minimum` (optional): Minimum number of expected elements. This is validate just before saving.

(4)`reference` (required): With the syntax you can see in the previous point.

(5)`condition` (optional): Restricts the elements that appears 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 you custom logic just after an element is removed from collection.

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

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

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

Now you can write a code as 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 a invoice.

Let's look 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 collection of aggregate its elements are deleted too. That is when you delete a invoice its details are deleted too.

(1)The restriction `minimum="1"` requires at least a detail for the invoice to be valid.

(2)With `order` you force that `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 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 that 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 you calculator implements `IModelCalculator` then it receive a `Invoice` an not a `InvoiceDetail`. This is logic because is executed after the detail is removed and it no longer exists.

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, 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 collection of entity references you have to define the reference in the other side, but in the case of collection of aggregate reference 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 starts 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): Arguments list of method in Java format.

(4)`exceptions` (optional): Exception list that can be throw by this method, in Java format.

(5)`calculator` (required): Implements the logic of this method.

Defining a method is easy:

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

And implementing it depends on the logic that you want 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 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 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 it is inserted directly in 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 value 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 silly wrapper of data. When possible 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 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 match 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 finder method in Java, hence must follow the Java rules for member naming, including start with lower-case.

(2)`arguments` (required): Arguments list for the method in Java format. The most advisable is 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 generate 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 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 value to 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 to others calculators (as calculated properties or method) but is executed just after creation.

### 3.14 Postmodify calculator (11)

With `<postmodify-calculator/>` you can plug some logic to execute after the state of the object is changed and just before it is stored in 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 value to 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 to others calculators (as calculated properties or method) but 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 and postmodify ones.

### 3.16 Validator (13)

This validator allows define a validation at model level. When you need make a validation on several properties at 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 value to 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", getLimiteBd());   // (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 force you to write a `validate(Messages messages)` (2). In this method you add the error message ids (3) (whose texts are in the *i18n* files). And if the validation process (that is the execution of all validators) produces some error OpenXava does not save and display the errors to 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 is executed just before

removing an object, and has the possibility to veto the deleting.

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 *xava/validators.xml* file of your project or of the OpenXava project.

(3)`set` (several, optional): To set value to 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 force you to write a `setEntity()` (2) method that receives the object to remove. If validation error is added to `Messages` object sent to `validate()` (3) the validation fails. If after executing all validation there are validation error OpenXava does not remove the object and display a list of validation messages to 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 start 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 is far 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 is far 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 reference from the main entity.

In the main entity you can write:

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

And a 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 implements a interface (1) and contains 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 a 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 so 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 a `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 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 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 identify 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 object of this type.

(2)`label` (optional): Label that is showed to user, if needed, when the view is displayed. It's **far 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 excluding 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 user and you can work programmatically with it, but it is not a part of the model.

(6)`property-view` (several, optional): Defines the format to display certain property.

(7)`reference-view` (several, optional): Defines the format to display certain reference.

(8)`collection-view` (several, optional): Defines the format to display certain collection.

(9)`members` (one, optional): Indicates the members to display and its layout in the user interface. Is excluding with `members` attribute.

### 4.1 Layout

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


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 excluding with `members` attribute, thus:

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

You can observe 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:

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>
```

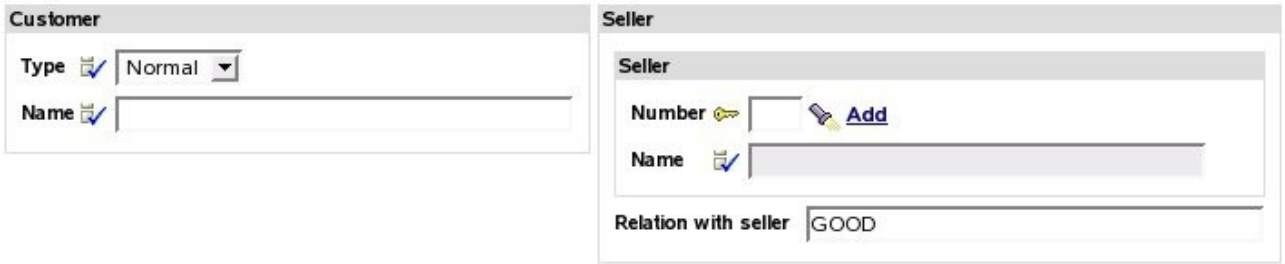
In this case the result is:


You can see the 3 properties within group are displayed inside a frame, and `name` is displayed outside this frame. The semicolon before name is for 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:

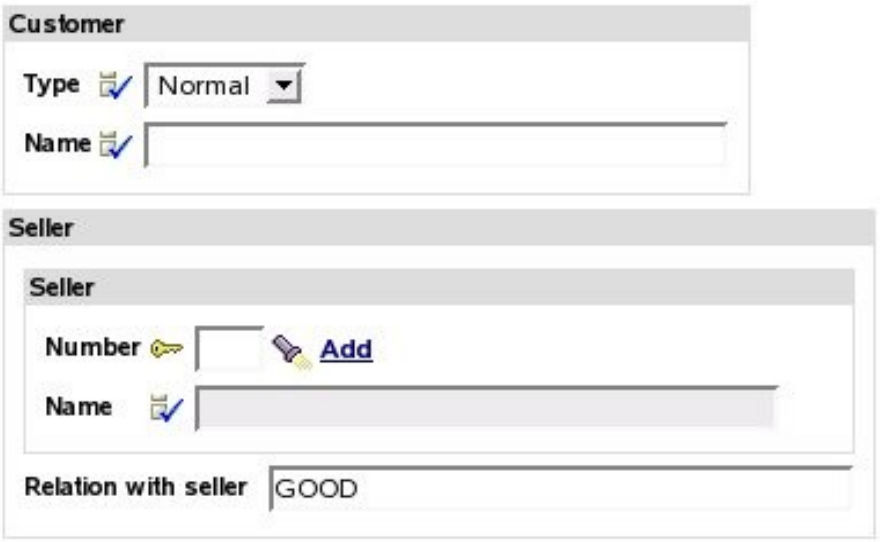



If you want one below the other then you must to use semicolon after group, like this:

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

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

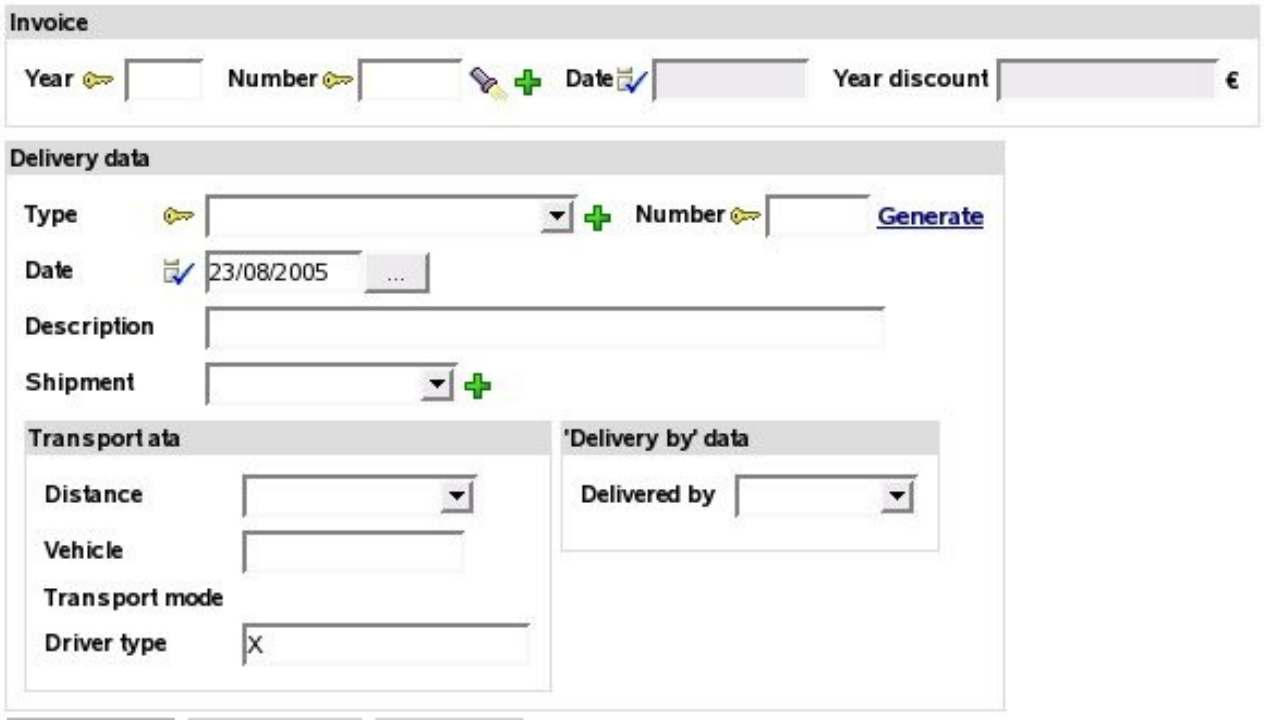
In this case the view is shown this way:




Nested groups are allowed. This is a pretty feature that allows you 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>
```

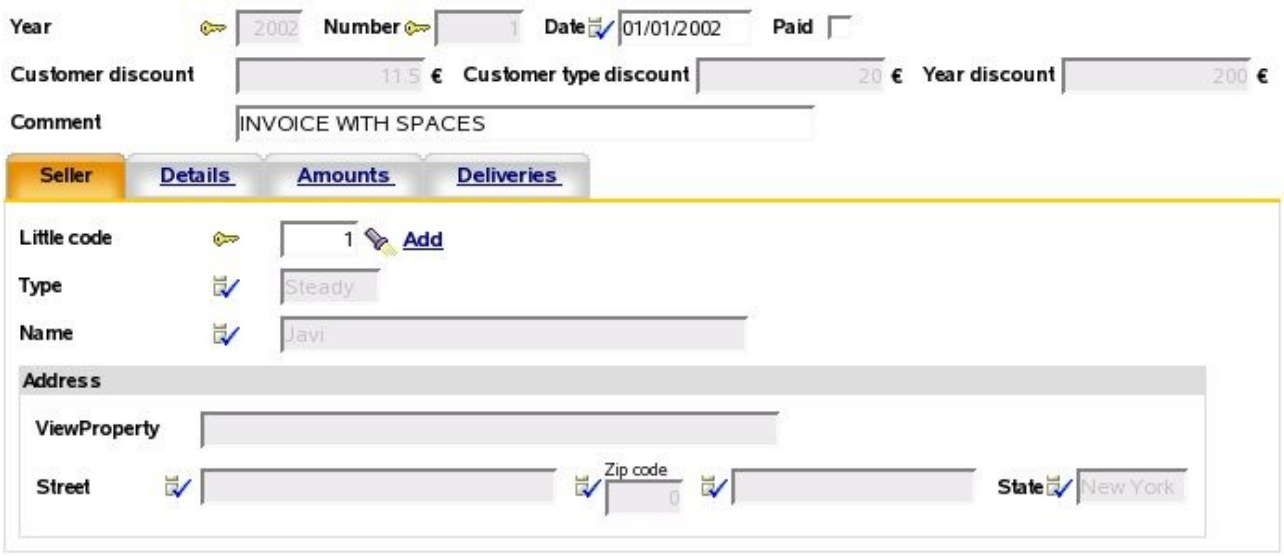
And the result will be:




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

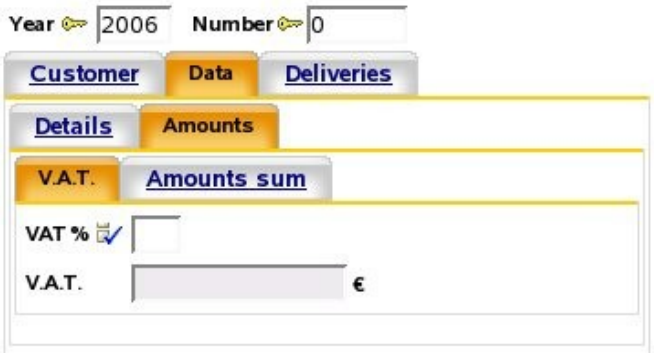



The sections are rendered as tabs that user can click on to see the data contained in that section. You can observe how in the view you put members of all types (not only properties); thus, `customer` is a reference, `details` is a collection of aggregates and `deliveries` is a collection of entities.

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 an user interface like this:



It's worth 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 member semantically related, and the sections allows divide all data to display in parts when there are a big amount of data in possibly cannot be displayed at once. 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 displaying 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): Modify the label for this property in this view. To do this is **far 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 no 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.

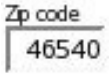
(6)`action` (several, optional): Actions (showed as links, buttons or images to the user) associated (visually) to this property and that the final user can execute.

## 4.2.1 Label format

An 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 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 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 listen 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 is more convenient to make that extended 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 to the view (change values, hide member, make them editables 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 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 image beside property. Like this:


The action link is present only when the property is editable, but if the property is read-only or calculated then 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));
     }

}
```

An implementation simple but illustrative. 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.

## 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)
</reference-view>
```

(1)`reference` (required): Name of reference to refine its presentation.

(2)`view` (optional): If you omit this attribute uses the default view to display the referenced object, with this attribute you can indicate that it uses another view.

(3)`read-only` (optional): If you set the value to `true` this reference never will be editable by final user in this view. An alternative to this is making 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.

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 be:

### 4.3.1 Choose view

The most simple customization is specifying the view of the referenced object that you wish to use:

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

For this 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>
```

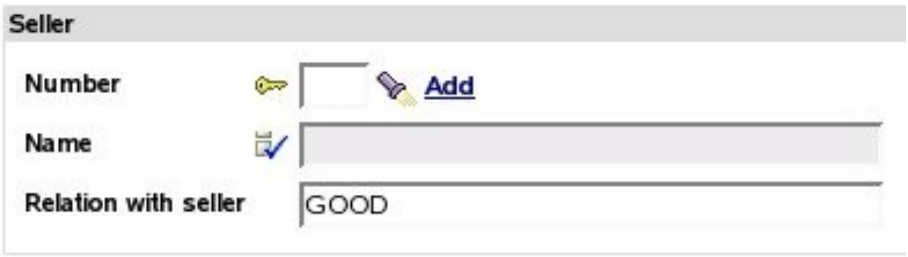
Thus, instead of using the default view of Invoice (that shows all invoice data) OpenXava will use the next one:


### 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:



### 4.3.3 Custom search action

The final user can search a new value for reference simply by typing the new code and leaving the editor the data of reference is obtained. 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 need to use `<search-action/>`, in this way:

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

Now 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 your. 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 earn 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 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 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 'create' link, the user goes 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 for visualizing references as descriptions list (actually a combo). This can be useful when there are 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 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 is required to specify here a property name.

(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 union of `condition` to do that the list contents depends on another values 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 is shown in the description list.

(5) `order-by-key` (optional): By default the data are ordered by descriptions, but if you set this property to `true` will be shown ordered by key.

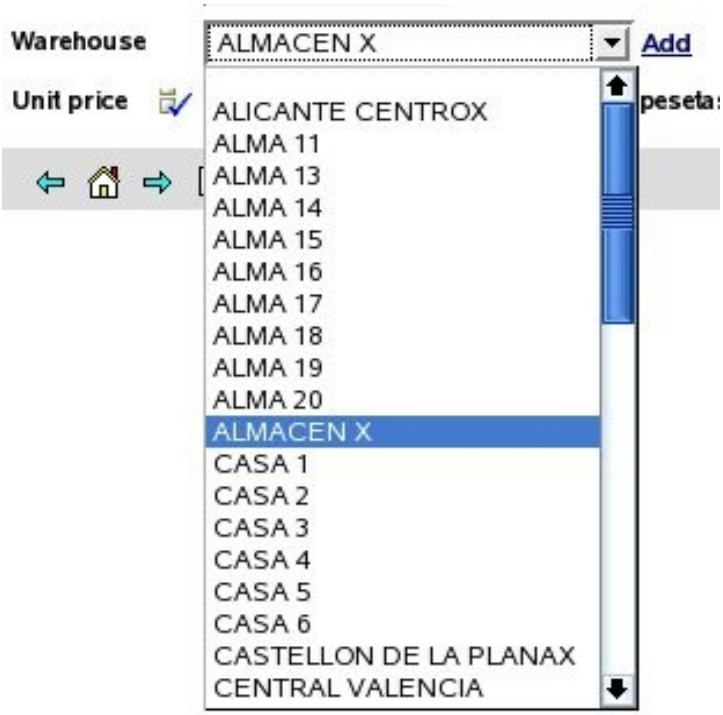
(6) `order` (optional): Allows to specify an order (with SQL style) for the values that is shown in the description list.

(7) `label-format` (optional): Format to display the label of this reference. See section 4.2.1.

The most simple use is:

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

That display a reference to warehouse in this way:




In this case it shows all warehouses, although in reality uses the `base-condition` and `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 choose a family another combo will be filled with the subfamilies of the chosen family, in this case you can do something as 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 empty, and when the user choose a family 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 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)
>
     <list-properties ... />          (6)
     <edit-action ... />              (7)
     <view-action ... />              (8)
     <list-action ... /> ...          (9)
     <detail-action ... /> ...       (10)
</collection-view>
```

(1)`collection` (required): Collection which you want to customize its look.

(2)`view` (optional): The view of the referenced object (each collection element) to use to display the detail. By default uses the default view.

(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 user cannot has the link that allows him to create new objects of the referenced object type. This only apply in the case of entity references collection.

(6)`list-properties` (one, optional): Properties to show in the list on visualizing the collection.

You can qualify the properties. By default it shows all persistent properties of referenced object (excluding references and calculated properties).

(7) `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 collection when the collection is editable.

(8) `view-action` (one, optional): Allows you to define your custom action to view a collection element. This is the action showed in each row when the collection is read only.

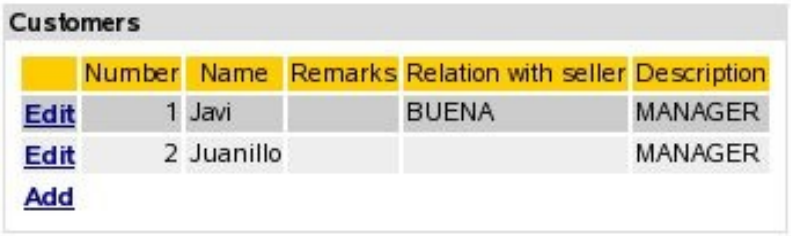
(9) `list-action` (several, optional): To add actions in list mode; usually actions which scope is the entire collection.

(10) `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/>` the collection is displayed using the persistent properties in list mode and the default view to represent the detail; although typically the properties of 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:



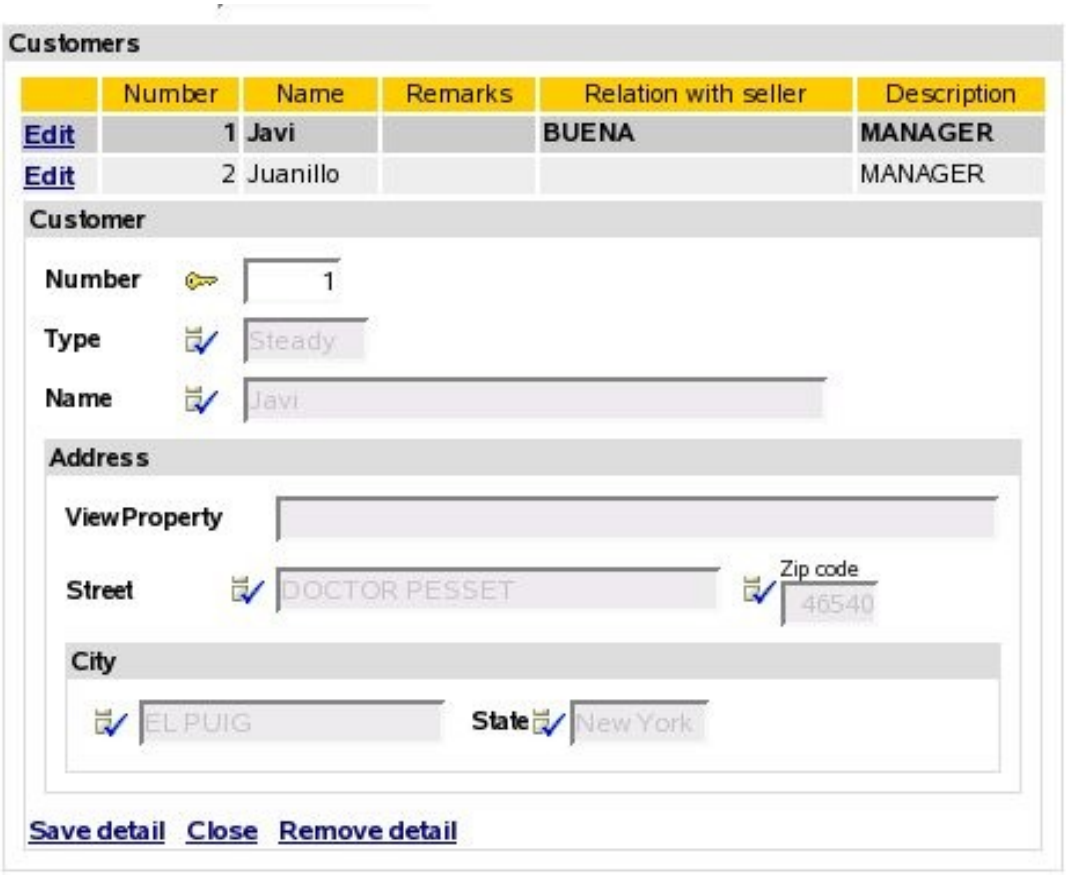

You see how you can put qualified properties in the properties list (as `seller.level.description`).

When the user clicks on 'Edit' or 'Add' the view `Simple` of `Customer` will be shown; for this you must have 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:




### 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 put a default value to the `remarks` property. Note that to access to the view that displays the detail you can use the method `getCollectionElementView()` (2).

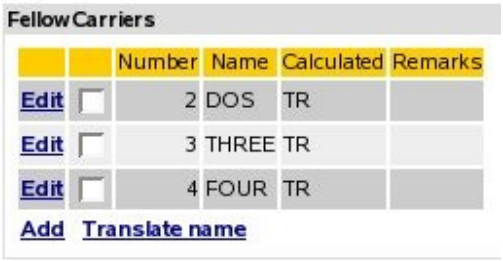
The technique to refine the view action (the action for each row when the collection is read only) is the same but using `<view-action/>` instead of `<edit-action/>`.

### 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:




And also you see how 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).

### 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:

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 access to the main view using `getView()` (3). In chapter 7 you will see more details about writing actions.

## 4.5 View property

With `<property/>` within `<view/>` you define a property that is not in the model but you want to show to 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 how the syntax is exactly the same that in case of a property of 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 puting it in `members`.

### 4.6 Editors configuration

You see how 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. Consists of a XML definition put together with a JSP fragment.

To refine the behavior of the OpenXava editors or add your custom editors you must create in the folder *xava* of you project a file called *editors.xml*. This file is 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 define 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` is OpenXava that has the responsibility of formatting the data from HTML to Java and vice versa, if `false` 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). By 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) in 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 is `true` the editor will be displayed inside a frame. By default is `false`. Useful for big editors (more than one line) that can be prettier this way.

(6)`property` (several, optional): Set values to 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 editor definition. This example is a editor that comes with OpenXava, but 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 are assigned to 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%> 
<%
}
%>
<% 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 formated 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 is not usual to do it. It's more habitual 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 force you to write a `format()` (2) method to convert the property value (that can be a Java object) to a string to render in HTML; and a `parse()` (3) method to convert the string received from the HTML form submit in a object suitable to assign to 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 wish to define a stereotype REGIONS that allows to user 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 is like defining a single value editor, the main difference is that formatted value (1) is 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 its combos with data from 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 and you *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 choose a family 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 aggregated used in a collection (`Invoice.InvoiceDetail`).

(3)`keyProperty` o `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 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 order by key, by default is ordered by description. You can also to use `order` with an order with SQL style if you need it.

(6) `readOnlyAsLabel`: When is read only is rendered as label. By default is `false`.

(7) `condition`: Condition to limit the data to display it. 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 another ones and only obtain data when these others properties change.

(8) `parameterValuesStereotypes`: List of stereotypes from which properties depends. 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) `formateadorDescripciones`: Formatter for the descriptions displayed in combo. It must implement `IFormatter`.

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

### *4.9 View without model*

In OpenXava 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:

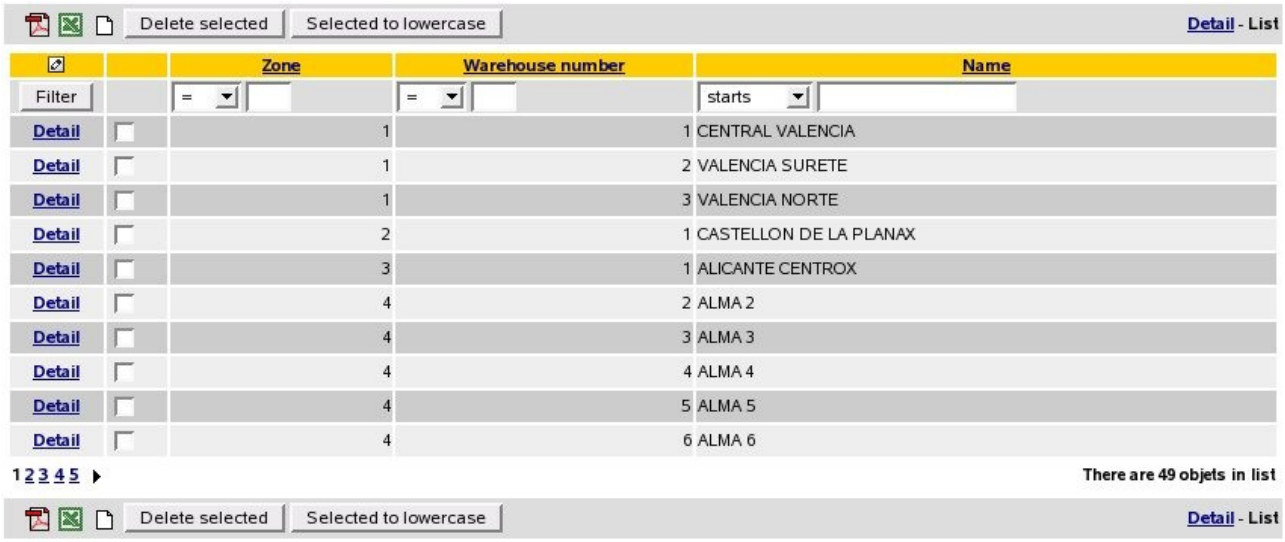
```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!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 are what are displayed in table format. When you create a conventional OpenXava module the user can manage the component data with a list like this:




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 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, sometime 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 at begin. Can be qualified.

(5)`base-condition` (one, optional): Condition to apply always to 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 at begin:

```
<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 | | Javi | Steady | MANUEL CHAVARRI | EL PUIG | MANAGER |
| Detail | | Juanillo | Normal | MANUEL CHAVARRI | VALENCIA | MANAGER |
| Detail | | Carmelo | Normal | | EL PUIG | |
| Detail | | Cuatrero | Normal | JUANVI LLAVADOR | VALENCIA | MANAGER |

1 There are 4 objets in list

### 5.2 Filters and base condition

An habitual 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 the SQL syntax, you can use `?` for arguments and the property names inside `${}`. In this case a filter is used to set value in 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 {                                                // (5)
                 r = new Object[2];
                 r[0] = year;
```

```
                r[1] = o;
            }

            return r;
      }

}
```

A filter gets the arguments of user type for filtering in list and process, 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. This arguments can be null (3), if the user does not type values, a simple object (5), if the user type a single value or a object array (4), if the user type 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 argument 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>
```

Int 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 {                    // (1)

      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 to 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 learn more about environment variables see the *Chapter 7 Controllers*.

### 5.3 Pure SQL select

You have the option to write the complete select to obtain the tab data:

```
<tab name="CompleteSelect">
     <properties>number, description, family</properties>
     <base-condition>
            select ${number}, ${description}, XAVATEST@separator@FAMILY.DESCRIPTION
                   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 order is only at beginning, the user can choose any other order only clicking in the heading of a column.

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

If ORM are 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 allows you to work with objects instead of tables and columns, and to generate automatically the needed 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 this object, and you does not need direct access to the SQL database, but for this you must define with precision 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 database table.

(3)`reference-mapping` (several, optional): Maps a reference to one or more columns in 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 attribute used internally in your object to store the property value. This allows you to use Java type more closer to the database, without getting dirty 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 does not 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 and is the type of the internal attribute in the generated class code, must to 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 force it to have a `toDB()` (2) method, that receives the object of the type used in Java (in this case a `Integer`) and return its representation using a type closer to the database (in this case `String` hence assignable to VARCHAR column). The `toJava()` method have the opposite goal, 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 how this converter is in `org.openxava.converters`, that is, is a generic converter that comes with OpenXava distribution. Another generic converter quite useful is `ValidValuesLetterConverter`, that 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 indicates empty value, 1 is 'local', 2 is 'national' and 3 is 'international'. But, what happens if in database is stored a single letter ('L', 'N' or 'I')? 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 make 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): Map a table column to a property of the reference key. If the key of the referenced object is multiple 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 model 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 does not define a converter 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 attribute used internally in your object to store the value. This allows you to use Java type more 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, for example, if you have properties of custom class that have itself several attributes to store. Also 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 database and vice versa. Must implement `IMultipleConverter`.

(3)`campo-cmp` (several, required): Maps each column in database with a property of 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="YEARDELIVERY" cmp-type="int"/>
```

```
</multiple-property-mapping>
```

DAYDELIVERY, MONTHDELIVERY and YEARDELIVERY 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 force 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 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 that the main entity. For example, if you have an aggregate `Address` associated to a `Customer`, the address data is stored in the same data that 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 need is using as a prefix the name of the reference with a nunderline (in this case `address_`). You can observe how in the case of aggregates you can map references, properties and 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 details data are stored in a different table than 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>
```

```
    ...
</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 main entity mapping. A component must have so many aggregate mapping 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 does 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 when 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 'conversores.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 any.

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 do not modify the OpenXava file, but create you 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 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 exists 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 object. 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 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 inherit all actions from other controller(s).

(2)`action` (several, required): Implements the logic to execute when the final user click in a button or link.

The controllers are made by 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)
>
    <set ... /> ...                                            (12)
    <use-object ... /> ...                                     (13)
```

```
</action>
```

(1)`name` (required): Action name that must be unique within its controller, but can be repeated in other controllers. When you reference an action always use `ControllerName.actionName` format.

(2)`label` (optional): Button label or link text. It's **far better** to use *i18n* files.

(3)`description` (optional): Description text of the action. It's **far better** to use *i18n* files.

(4)`mode` (optional): Indicates in what mode the action has to be visible. The default value is `ALL`, that it 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): An 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 property, in collections, etc. By default is `false`.

(8)`on-init` (optional): If you set this property to `true` this action will be executed automatically on initiating the module. By 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. By default is `never`.

(10)`takes-long` (optional): If you set it to `true` you are indicating that this action takes long time in executing (minutes or hours), in the current implementation OpenXava shows a progress bar. By 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. By default is `false`.

(12)`set` (several, optional): Sets value to action properties. Thus the same action class can be configured in different ways and it can be used in several controllers.

(13)`use-object` (several, optional): Assigns a session object to an action property just before execute action, and after execution gets the property value and put it in the context again (update the session object, thus you can update even immutable objects).

Actions are short life objects, when a user click a button the action object is created, configured (with `set` and `use-object`), executed, the session objects are updated, and finally the action object is discarded.

An plain controller can be:

```
<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 in the module that you wish; this is made 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 you make 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) of your action just before calling `execute()`.

Now inside `execute()` method you can use `getView()` as you wish (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 that works a standard controller, 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 click in 'new' link.

### 7.4 List mode actions

You can write actions that apply to several objects. These actions usually only are shown in list mode and normally have effect only on the objects chosen by user.

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 to data displayed in list (3). Since this action deletes records you make that user must to confirm before execute it (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 allows you to see the things 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 needn't to implements 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 joined to the next definition in you 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 to data model using Java maps (`java.util.Map`), this is practical when 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 class of OpenXava use `MapFacade` to manage the model and you also can use `MapFacade`; but, as general design tip, working with maps is useful in case of generic logic, but if you need to program specific things is better to use directly the object of model layer (the POJOs or EJBs generated by OpenXava). For more details you can see the JavaDoc of `org.openxava.model.MapFacade`.

You learn here how to display messages to 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 sufix (_en, _ca, _es, _it, etc). You can see the examples in *OpenXavaTest/xava/i18n*. All not cached exceptions produces a generic error messages, except if the not cached exception is the type ValidationException, in this case the message exception is displayed.

The `resetDescriptionsCache()` (7) method delete all cache used by OpenXava to display descriptions list (combos). It's a good idea to call it whenever data are 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 display it in detail. Now, if in the 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 a 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 SearchByViewKeyAction {      // (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 in database (or through EJB, JDO or Hibernate) and fill the view. Most times is better that it extends `SearchByViewKeyAction` (1) and within `execute()` write a `super.execute()` (2).

OpenXava comes with 2 predefined search actions:

- `CRUD.searchByViewKey`: This is the default one. It does a search using the key values in view, it executes no event.
- `CRUD.searchExecutingOnChange`: Works as `searchByViewKey` 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 you 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 putting value to the `XAVA_SEARCH_ACTION` environment variable.

### 7.6 Initialize a module with an action

Just by setting `on-create="true"` when you define an action you that this action to 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 argument to that title (3). The list title is defined in *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 to a module from another. For example, imagine that you want to show a customers list and when the user clicks in one 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 in detail goes to an invoices module filtered by customer and filtering by 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, now when the user clicks in the link in each row, 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 `IChangeModuleAction` (1) thus forces the action to have a method called `getNextModule()` (7) that will indicate to which module must change the OpenXava after executing this action, and `hasReinitNextModule()` (9) to indicate if 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 to session object of others modules, it's 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 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 can be interesting to display to user 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) you do that your action implements `INavigationAction` (`ICustomViewAction` is enough) and 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 final user to generate their own reports from list model. The user can filtering, ordering, add/remove field, changing the positions of the fields and then to generate a PDF report.

But in all no trivial business application you need to create programatically your own reports. You can do that easily using *JasperReports* and integrating the report in your OpenXava application with the action `JasperReportBaseAction`.

In the first place you need to design you report with *JasperReports*, you can use *iReport* an excellent designer for *JasperReports*.

Then you can write you 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;
      }

}
```

You only need that you action extends `JasperReportBaseAction` (1) and overwrite the next 3 method:

- `getParameters()` (2): A `Map` with the parameters to send to the report, in this case we do data input validation (using `MapFacade.validate()`) before (3).
- `getDataSource()` (4): A `JRDataSource` with data to print. In this case is a collection of JavaBeans obtained calling a method of the model object. If you uses EJB be careful and do not make 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 you project for hold these files.

By default the report is displayed in a popup window, but if you wish the report in the current window you can overwrite the method `inNewWindow()`.

You can find more examples of JasperReport actions in *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 environments with no recoding.

In order to upload a file the first step is creating an action for go 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 navigate to another controllers (3) and to 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 JSP for 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 execute the next action immediately.
- `IChangeControllersAction`: To change the controller (the actions) available to user.
- `IChangeModeAction`: To change the mode, from list to detail or vice versa.
- `IChangeModuleAction`: 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 User Interface. *(new in v2.0)*
- `IJDBCAction`: Allows to use direct JDBC in an action. It receives an `IConnectionProvider`. Works like an `IJDBCCalculator` (see chapter 3).
- `ILoadFileAction`: Navigates to a view that allows final user to load a file.
- `IModelAction`: An action that receives the model name.
- `IModuleContextAction`: Gets a `ModuleContext` in order to access to session objects of other modules.
- `INavigationAction`: Extends from `IChangeControllersAction` 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 can be a good substitute for a SessionBean.
- `IRequestAction`: Receives a servlet `request`. This type of actions link our application to the Servlets/JSP technology, hence is better avoiding it. But sometimes a little 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 ideal way is looking the JavaDoc API of package `org.openxava.actions` and seeing 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 assembly these pieces in order to create applications.

The definition of an OpenXava application is in *application.xml* file that can be found in the *xava* directory of your project.

The file syntax is:e

```
<application
      name="name"                    (1)
      label="label"                  (2)
>

      <module ... /> ...             (3)
</application>
```

(1)`name` (required): Name of the application.

(2)`label` (optional): It's **far 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 in this application.

(2)`folder` (optional): Folder in which 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 show to user. It's **far better** to use *i18n* files.

(4)`description` (optional): Long description to show to user. It's **far 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 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 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 does not specify it the default tab will be used.

(10)`controller` (several, optional): Controllers with the available actions to user on begining.

(11)`mode-controller` (one, optional): Allows to define the behavior to go from detail to list and vice versa, as well as to define a module without detail and view (with no modes).

(12)`doc` (one, optional): It's exclusive with all other elements. It allows you to define a module that contains only documentation, no logic. Useful for generating informational portlets for you 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 to user create, delete, update, search, generate PDF reports and export to Excel the warehouses data (thanks to `Typical` controller) and also executing 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/Gestion/xava/modulo.jsp?application=Management&module=Warehouse

Also a portlet is generated to allow you deploying 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 remove 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 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 browser language is not English nor Spanish then it assumes English (the first specified language). If you do not specify 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. Really complements some lacks in traditional Object Oriented Programming.

What problem resolves AOP? 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) nor ethical (because there isn't *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 AspectJ project.

OpenXava add some support of the *aspects* concept since version 1.2.1. At the moment the support is little and OpenXava is still far 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 defined 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 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 for applying the aspect. It's exclusive with `except-for-models` attribute.

(3)`except-for-models` (optional): A comma separated list of models to excluded when apply this aspect. In this case the aspect apply to all models excepts the indicated ones. It's exclusive with `for-models` attribute.

If you do not use `for-models` nor `except-for-models` then the aspect will apply to all models in the application. Models are 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"/>
```

In this simple way you can do that when a new object is created (saving it in database by first time) the logic of `MyCalculator` is executed. And this for all models.

As you see at the moment only these few calculators are supported. We expect to extend the power of *aspects* for OpenXava in the future. Anyways these calculators now offers 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 accesses to the data in your application. Actually, this project allows your applications to follow 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 circumstances.

## 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, the code of `AccessTrackingCalculator` is executed each time the a object is created, loaded, modified or removed. `AccessTrackingCalculator` writes a record in 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 accesses 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 for *AccessTracking* project.
- Edit the file of your project *build/ejb/META-INF/MANIFEST.MF* to add the next jars in 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 need deploying war for your project. *(new in v2.0)*

In the case that you are using EJB2 CMP you need regenerating code, deploying EJB and deploying war for your project.

The accesses are recorded in a table with the name TRACKING.ACCESS. If you wish 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 does not exist the direct concept of many-to-many relationship, instead in OpenXava only exist collections. Even so model a many-to-many relationship in OpenXava is easy. You only need to define collections in both sides of relationship.

For example, if you have customers and states, and an customer can work in several states, and, obviously, in a state can work several customers, you have a case of many-to-many (using relational nomenclature) relationship. Supposing 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)
```

```
                        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 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 show 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 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 exactly as 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 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 uses. At the end of the complete cycle of action execution, OpenXava commits automatically the transaction and close 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 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 need 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 use your own JSP, in `application.xml`:

```
<module name="DeliveryTypesJSP" folder="invoicing/variations">
    <model name="DeliveryType"/>
    <web-view url="custom-jsp/deliveryType.jsp"/>              (1)
```

```
    <controller name="Typical"/>
</module>
```

If you use `web-view` (1) on defining your module, OpenXava uses your JSP to render the detail, instead of generating automatically the view. You can put your JSP inside *web/custom-jsp* folder of you 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>Description: </td><td>

    <xava:editor property="description"/>

    </td>
</tr>
</table>
```

Your are free to create you 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 render an editor suitable for the indicated property, furthermore add the needed javascript for throw events. If you uses `<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) works.

Let's see the OpenXava taglibs.

### 10.3.2 xava:editor

The `<xava:editor/>` tag allows you render an editor (a HTML control) for your property, in the same way that OpenXava does it when it generates automatically the user interface.

```
<xava:editor property="propertyName"/>
```

The `property` attribute is a property of the model associated to the current module. This tag generates the needed JavaScript in order to allows your view work in the same way of an automatic one.

### 10.3.3 xava:action, xava:link, xava:image, xava:button

The `<xava:action/>` tag allows you render an action (a button or image that 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 user click on it executes the action `CRUD.save`, before it put `true` to the its `resetAfter` property of the action.

The action is render as an image is the it has an image associated and as a button if it has no image. If you wish to determine the render style you can use directly the next taglibs: `<xava:button/>`, `<xava:image/>` or `<xava:link/>` similars to `<xava:action/>`.