



REFERENCE GUIDE

VERSION **3.0**

.....

Table of Contents

Chapter 1: Overview	4
Chapter 2: My first OpenXava project	7
Chapter 3: Model	15
Chapter 4: View	56
Chapter 5: Tabular data	96
Chapter 6: Object/relational mapping	103
Chapter 7: Controllers	114
Chapter 8: Application	138
Chapter 9: Customizing	144

.....

.....

Chapter 1: Overview

Presentation

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

The underlying philosophy is to define with Java annotations or XML and to program with Java, the more definition and less programming the better.

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

Below you can see some basic concepts of OpenXava.

Business component

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

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

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

Controllers

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

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

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

Application

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

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

Project structure

A typical OpenXava project usually contains the these folders:

- **[root]**: In the root you can find *build.xml* (with the Ant task).
- **src[source folder]**: Contains your Java source code.
- **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.
- **properties[source folder]**: Property files to configure your application.
- **data**: Useful to hold the scripts to create the tables of your application, if needed.
- **web**: Web content. Usually JSP files, lib and classes. Most of the content is generated automatically, but you can put here your own JSPs or other custom web resources.

.....

.....

Chapter 2: My first OX project

Create a new project

First open your Eclipse and make its workspace the one that comes with the OpenXava distribution.

Using the appropriate Eclipse Wizard create a new Java Project named *Management*. Now you have a new empty Java project in the workspace, the next step is to give it the correct shape for an OpenXava project.

Go to the project *OpenXavaTemplate* and edit the file *CreateNewProject.xml* in this way:

```
<property name="project" value="Management" />
```

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

Select the project *Management* and press F5 for refreshing.

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

Configure database

OpenXava generates a JavaEE/J2EE application intended to be deployed in a Java Application Server (since v2.0 OpenXava applications also run in a simple servlet container, as Tomcat). In OpenXava you only need to indicate the Data Source JNDI and then configure the data source in your application server. Configuring a data source in an application server is out of the scope of this guide, nevertheless you have below detailed instructions to configure a database in order to run this first project using the Tomcat included in OpenXava distribution as application server and Hypersonic as database. This Tomcat is in the folder *openxava-3.x/tomcat*.

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

```
<Resource name="jdbc/ManagementDS" auth="Container" type="javax.sql.DataSource"
maxActive="20" maxIdle="5" maxWait="10000"
username="sa" password="" driverClassName="org.hsqldb.jdbcDriver"
url="jdbc:hsqldb:hsq://localhost:1666"/>
```

The main thing here is the JNDI name, this is the only thing referenced from OpenXava, in this case *ManagementDS*. The *driverClassName* and *url* attributes depend on your particular database, in this case Hypersonic is used.

Your first business component

Creating an OpenXava business component is easy: The definition of each component is a Java class with annotations. In order to begin you have to create a class called *Warehouse*:

- Put on *src* folder and use the *Right Button* > *New* > *Package*
- Create a package named *org.openxava.management.model*
- Put on package *org.openxava.management.model* and use *Right Button* > *New* > *Class*
- Create a class named *Warehouse*

Now edit your new class and write this code:

```
package org.openxava.management.model;

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

@Entity
public class Warehouse {

    @Id @Column(length=3) @Required
    private int number;

    @Column(length=40) @Required
    private String name;

    public int getNumber() {
        return number;
    }

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

    public String getName() {
        return name;
    }
}
```



```

    }

    public void setName(String name) {
        this.name = name;
    }
}

```

This class contains (and it will be contain) all data needed by the application about the Warehouse business concept. At the moment we only have data structure, but in this class you can put the mapping against the database, the business logic, the visual presentation, the tabular data management, etc. Really this class is an *Entity* that follows the EJB3 standard. For defining a class as a entity you only need to use the *@Entity* annotation in the class declaration.

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

```

@Id                                // 1
@Column(length=3)                  // 2
@Required                          // 3
private int number;                // 4
private int getNumber() {          // 4
    return number;
}
private void setNumber(int number) { // 4
    this.number = number;
}
}

```

This is its meaning:

1. **@Id**: Indicates if this property is part of the key. The key is a unique identifier of the object and usually matches with the primary key of a database table.
2. **@Column(length=)**: Length of data. It's optional, but useful to display better user interfaces and generating database tables.
3. **@Required**: Indicates if it's required to validate the existence of data for this property just before creation or modification.
4. The property defined in the usual way for a Java class. All valid types for a Java property are applicable here, including integrated types, JDK classes and custom classes.

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

The table

Before testing the application you have to create the table in database. Follow these steps:

- Start your database server: From command line go to *openxava-3.x/tomcat/bin* folder and execute:
 - In Linux/Unix: *./start-hsqldb.sh management-db 1666*
 - In Windows: *start-hsqldb management-db 1666*
- Create the table:
 - Edit *Management/build.xml*. Search the ant target *updateSchema*.
 - Put a correct value for *schema.path*, in this case *"../OpenXavaTest/lib/hsqldb.jar"*.
 - Execute the ant target *updateSchema*.
- Start Tomcat and now everything is ready.

Executing your application

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

- Execute the ant target *deployWar*.
- Open an internet browser and go to <http://localhost:8080/Management/xava/module.jsp?application=Management&module=Warehouse>

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 *generatePortlets*.
- Take the file *Management.war* in the folder *openxava-3.x/workspace.dist/Management.dist* and deploy it into your Portal Server.

Automating the tests

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

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

of your program from user interface to database.

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

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

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

import org.openxava.tests.*;

/**
 * @author Javier Paniza
 */

public class WarehouseTest extends ModuleTestBase {

    public WarehouseTest(String testName) {
        super(testName, "Management", "Warehouse"); // 1
    }

    public void testCreateReadUpdateDelete() throws Exception {
        // Create
        execute("CRUD.new"); // 2
        setValue("number", "7"); // 3
        setValue("name", "JUNIT Warehouse");
        execute("CRUD.save");
        assertNoErrors(); // 4
        assertEquals("number", ""); // 5
        assertEquals("name", "");

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

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

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

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

```

}
}

```

You can learn from this example:

1. **Constructor:** In the constructor you indicate the application and module name.
2. **execute:** Allows to simulate a button or link click. As an argument you send the action name; you can view the action names in *OpenXava/xava/default-controllers.xml* (the predefined controllers) and *Management/xava/controllers.xml* (the customized ones). Also if you move the mouse over the link your browser will show you the JavaScript code with the OpenXava action to execute. That is `execute("CRUD.new")` is like click in 'new' button in the user interface.
3. **setValue:** Assigns a value to a form control. That is, `setValue("name", "Pepe")` has the same effect than typing in the field 'name' the text "Pepe". The values are always alphanumeric because they are assigned to a HTML form.
4. **assertNoErrors:** Verify that there are no errors. In the user interface errors are red messages showed to user and added by the application logic.
5. **assertValue:** Verify if the value in the form field is the expected one.
6. **assertMessage:** Verify if the application has shown the indicated informative message.

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

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

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

```
liferay.url=web/guest
```

This is for testing against Liferay. Also it's possible to test against a JetSpeed2 portal, look at *OpenXavaTest/properties/xava-junit.properties* to learn more.

The labels

Now everything works well, but a little detail remains yet. Maybe you want to define the labels to be show to the user. The 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 edit the file *Management-labels_en.properties* in *i18n* folder. Edit that file and add:

```
Warehouse=Warehouse
```

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

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

```
Warehouse=Almacén
```

The OpenXava default labels and messages are in *OpenXava/i18n/Labels.properties* and *OpenXava/i18n/Messages.properties*. If you want to override some of these resources you do not need to edit these files, instead, you can use the same key names in the resource files of your project, then your labels and messages will be used instead of the standard ones of OpenXava (*new in v2.0.3*). For example, if you want to change the standard message in list mode "*There are 23663 objects in list*" for other, you have to add to your *Management-messages_en.properties* this entry:

```
# list_count is in Messages_en.properties of OpenXava, this is an example
# of overriding a standard openxava message
list_count=There are {0} records in list
```

Now, your application will show "There are 23663 records in list" instead of the default OpenXava message "*There are 23663 objects in list*".

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

.....

.....

Chapter 3: Model

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 plain Java classes (although a XML version is also available). OpenXava provides generates a full featured application from your model definition.

Business Component

The basic unit to create an OpenXava application is the business component. A business component is defined using a Java class called *Entity*. This class is a regular EJB3 entity, or in other words, a POJO class with annotations that follows the Java Persistence API (JPA) standard.

JPA is the Java standard for persistence, that is, for objects that store its state in a database. If you know how to develop using POJOs with JPA, you already know how to develop OpenXava applications.

Using a simple Java class you can define a Business Component with:

- **Model:** Data structure, validations, calculations, etc.
- **View:** How the model can be shown to the user.
- **Tabular data:** How the data of the component is displayed in list mode (in tabular format).
- **Object/relational mapping:** How to store and retrieve the object state from database.

This chapter explains how to define the model part, that is, all about structure, validations, calculations, etc.

Entity

In order to define the model part you have to define a Java class with annotations. In addition to its own annotations, OpenXava supports annotations from JPA and Hibernate Validator. This Java class is an entity, that

is, a persistent class that represents a business concept.

In this chapter JPA is used to indicate that it's a standard Java Persistent API annotations, HV for indicating it's a Hibernate Validator annotation, and OX for indicating that is an annotation of OpenXava.

This is the syntax for a entity:

```
@Entity                // 1
@EntityValidator       // 2
@RemoveValidator      // 3
public class EntityName { // 4
    // Properties        // 5
    // References        // 6
    // Collections       // 7
    // Methods           // 8
    // Finders           // 9
    // Callback methods  // 10
}
```

1. **@Entity** (JPA, one, required): Indicates that this class is a JPA entity, in other words, its instances will be persistent objects.
2. **@EntityValidator** (OX, several, optional): Executes a validation at model level. This validator can receive the value of various model properties. To validate a single property it is better to use a property level validator.
3. **@RemoveValidator** (OX, several, optional): It's executed before removal, and can deny the object removing.
4. **Class declaration**: As a regular Java class. You can use *extends* and *implements*.
5. **Properties**: Regular Java properties. They represent the main state of the object.
6. **References**: References to other entities.
7. **Collections**: Collections of references to other entities.
8. **Methods**: Java methods with the business logic.
9. **Finders**: Finder methods are static method that do searches using JPA query facilities.
10. **Callback methods**: JPA callbacks methods for insert logic on creating, modifying, loading, removing, etc.

Properties

A property represents the state of an object that can be read and in some cases updated. The object does not have the obligation to store physically

the property data, it only must return it when required.
The syntax to define a property is:

```

@Stereotype // 1
@Column(length=) @Max @Length(max=) @Digits(integerDigits=) // 2
@Digits(fractionalDigits=) // 3
@Required @Min @Range(min=) @Length(min=) // 4
@Id // 5
@Hidden // 6
@SearchKey // 7
@Version // 8
@DefaultValueCalculator // 9
@PropertyValidator // 10
private type propertyName; // 11
public type getPropertyName() { ... } // 11
public void setPropertyName(type newValue) { ... } // 11

```

1. **@Stereotype** (OX, optional): Allows to specify a special behavior for some properties.
2. **@Column(length=)** (JPA), **@Max** (HV), **@Length(max=)** (HV), **@Digits(integerDigits=)** (HV, optional, usually you only use one of them): Length in characters of property, except in *@Max* case that is the max value. Useful to generate user interfaces. If you do not specify the size, then a default value is assumed. This default value is associated to the stereotype or type and is obtained from *default-size.xml*.
3. **@Digits(fractionalDigits=)** (HV, optional): Scale (size of decimal part) of property. Only applies to numeric properties. If you do not specify the scale, then a default value is assumed. This default value is associated to the stereotype or type and is obtained from *default-size.xml*.
4. **@Required** (OX), **@Min** (HV), **@Range(min=)** (HV), **@Length(min=)** (HV) (optional, usually you only use one of them): Indicates if this property is required. In the case of *@Min*, *@Range* and *@Length* you have to put a value greather than zero for *min* in order to assume that the property is required. By default a property is required for key properties hidden (*new in v2.1.3*) and false in all other cases. On saving OpenXava verifies if the required properties are present. If this is not the case, then saving is not done and a validation error list is returned. The logic to determine if a property is present or not can be configured by creating a file called *validators.xml* in your project. You can see the syntax in *OpenXava/xava/validators.xml*.

5. **@Id** (JPA, optional): Indicates that this property is part of the key. At least one property (or reference) must be key. The combination of key properties (and key references) must be mapped to a group of database columns that do not have duplicate values, typically the primary key.
6. **@Hidden** (OX, optional): A hidden property has a meaning for the developer but not for the user. The hidden properties are excluded when the automatic user interface is generated. However at Java code level they are present and fully functional. Even if you put it explicitly into a view the property will be shown in the user interface.
7. **@SearchKey** (OX, optional): The search key properties are used by the user as key for searching objects. They are editable in user interface of references allowing to the user type its value for searching. OpenXava uses the *@Id* properties for searching by default, and if the id properties are hidden then it uses the first property in the view. With *@SearchKey* you can choose explicitly the properties for searching.
8. **@Version** (JPA, optional): A version property is used for optimistic concurrency control. If you want control concurrency you only need to have a property marked as *@Version* in your entity. Only a single version property should be used per entity. The following types are supported for version properties: *int*, *Integer*, *short*, *Short*, *long*, *Long*, *Timestamp*. The version properties are considered hidden.
9. **@DefaultValueCalculator** (OX, one, optional): Implements the logic to calculate the default (initial) value for this property. A property with *@DefaultValueCalculator* has *setter* and it is persistent.
10. **@PropertyValidator** (OX, several, optional): Implements the validation logic to execute on this property before modifying or creating the object that contains it.
11. **Property declaration**: A regular Java property declaration with its getters and setters. You can create a calculated property using only a getter with no field nor setter. Any type legal for JPA is available, you only need to provide a Hibernate Type to allow saving in database and an OpenXava editor to render as HTML.

Stereotype

A stereotype (*@Stereotype*) is the way to determine a specific behavior of a type. For example, a name, a comment, a description, etc. all correspond to

the Java type `java.lang.String` but you surely wish validators, default sizes, visual editors, etc. different in each case and you need to tune finer; you can do this assigning a stereotype to each case. That is, you can have the next stereotypes `NAME`, `MEMO` or `DESCRIPTION` and assign them to your properties.

OpenXava comes with these generic stereotypes:

- `DINERO`, `MONEY`
- `FOTO`, `PHOTO`, `IMAGEN`, `IMAGE`
- `TEXTO_GRANDE`, `MEMO`, `TEXT_AREA`
- `ETIQUETA`, `LABEL`
- `ETIQUETA_NEGRITA`, `BOLD_LABEL`
- `HORA`, `TIME`
- `FECHAHORA`, `DATETIME`
- `GALERIA_IMAGENES`, `IMAGES_GALLERY` (setup instructions)
- `RELLENADO_CON_CEROS`, `ZEROS_FILLED`
- `TEXTO_HTML`, `HTML_TEXT` (text with editable format)
- `IMAGE_LABEL`, `ETIQUETA_IMAGEN` (image depending on property content)
- `EMAIL`
- `TELEFONO`, `TELEPHONE`
- `WEBURL`
- `IP`
- `ISBN`
- `TARJETA_CREDITO`, `CREDIT_CARD`
- `LISTA_EMAIL`, `EMAIL_LIST`

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`.

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

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

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

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

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

Now everything is ready to define properties of stereotype PERSON_NAME:

```
@Stereotype("PERSON_NAME")
private String name;
```

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

IMAGES_GALLERY stereotype

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

```
@Stereotype("IMAGES_GALLERY")
private String photos;
```

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

And everything is done.

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

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

```
CREATE TABLE IMAGES (
  ID VARCHAR(32) NOT NULL PRIMARY KEY,
  GALLERY VARCHAR(32) NOT NULL,
  IMAGE BLOB);
CREATE INDEX IMAGES01
ON IMAGES (GALLERY);
```

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

And finally you need to define the mapping in your *persistence/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.

Concurrency and version property

Concurrency is the ability of the application to allow several users to save data at same time without losing data. OpenXava uses the optimistic concurrency of JPA. Using optimistic concurrency the records are not locked allowing high concurrency without losing data integrity.

For example, if a user A read a record and then a user B read the same record, modify it and save the changes, when the user A try to save the record he receives an error, then he need to refresh the data and retry his modification.

For activating concurrency support for an entity you only need to declare a property using `@Version`, in this way:

```

@Version
private int version;

```

This property is for use of persistence engine, your application or your user must not use this property directly.

Enums

OpenXava supports Java 5 enums. An enum allows you to define a property that can hold one of the indicated values only .

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

```

private Distance distance;
public enum Distance { LOCAL, NATIONAL, INTERNATIONAL };

```

The `distance` property only can take the following values: `LOCAL`, `NATIONAL` or `INTERNATIONAL`, and as you have not put `@Required` no value (null) is allowed too.

At user interface level the current implementation uses a combo. The label

for each value is obtained from the *i18n* files.

At database level the value is by default saved as an integer (0 for LOCAL, 1 for NATIONAL, 2 for INTERNATIONAL and null for no value), but you can configure easily to use another type and work with no problem with legacy databases. See more about this in mapping chapter.

Calculated properties

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:

```
@Depends("unitPrice") // 1
@Max(9999999999L)      // 2
public BigDecimal getUnitPriceInPesetas() {
    if (unitPrice == null) return null;
    return unitPrice.multiply(new BigDecimal("166.386")).setScale(0, BigDecimal.ROUND_HALF_UP);
}
```

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

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

And *result* will hold 332.772.

When the property *unitPriceInPesetas* is displayed to the user it's not editable, and its editor has a length of 10, indicated using *@Max(9999999999L)* (2). Also, because of you use *@Depends("unitPrice")* (1) when the user will change the value of the *unitPrice* property in the user interface the *unitPriceInPesetas* property will be recalculated and its value will be refreshed to the user.

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

```
@Max(999)
public int getDetailsCount() {
    // An example of using JDBC
    Connection con = null;
    try {
        con = DataSourceConnectionProvider.getByComponent("Invoice").getConnection(); // 1
        String table = MetaModel.get("InvoiceDetail").getMapping().getTable();
        PreparedStatement ps = con.prepareStatement("select count(*) from " + table +
            " 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;
    }
    catch (Exception ex) {
        log.error("Problem calculating details count of an Invoice", ex);
        // You can throw any runtime exception here
        throw new SystemException(ex);
    }
    finally {
        try {
            con.close();
        }
        catch (Exception ex) {
        }
    }
}

```

Yes, the JDBC code is ugly and awkward, but sometimes it can help to solve performance problems. The *DataSourceConnectionProvider* class allows you to obtain a connection associated to the same data source that the indicated entity (*Invoice* in this case). This class is for your convenience, but you can access to a JDBC connection using JNDI or any other way you want. In fact, in a calculated property you can write any code that Java allows you.

Default value calculator

With *@DefaultValueCalculator* you can associate logic to a property, in this case the property is readable and writable. This calculator is for calculating its initial value. For example:

```

@DefaultValueCalculator(CurrentYearCalculator.class)
private int year;

```

In this case when the user tries to create a new Invoice (for example) he will find that the year field already has a value, that he can change it if he wants to do. The logic for generating this value is in the *CurrentYearCalculator* class, that it's:

```

package org.openxava.calculators;

import java.util.*;

```

```

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

    public Object calculate() throws Exception {
        Calendar cal = Calendar.getInstance();
        cal.setTime(new java.util.Date());
        return new Integer(cal.get(Calendar.YEAR));
    }
}

```

It's possible to customize the behaviour of a calculator setting the value of its properties, as following:

```

@DefaultValueCalculator(
    value=org.openxava.calculators.StringCalculator.class,
    properties={ @PropertyValue(name="string", value="GOOD") }
)
private String relationWithSeller;

```

In this case for calculating the default value OpenXava instances *StringCalculator* and then injects the value "GOOD" in the property *string* of *StringCalculator*, and finally it calls to the *calculate()* method in order to obtain the default value for *relationWithSeller*. As you see, the use of *@PropertyValue* annotation allows you create reusable calculators.

@PropertyValue allows to inject the value from other displayed properties, in this way:

```

@DefaultValueCalculator(
    value=org.openxava.test.calculators.CarrierRemarksCalculator.class,
    properties={
        @PropertyValue(name="drivingLicenceType", from="drivingLicence.type")
    }
)
private String remarks;

```

In this case before to execute the calculator OpenXava fills the *drivingLicenceType* property of *CarrierRemarksCalculator* with the value of the displayed property *type* from the reference *drivingLicence*. As you see the *from* attribute supports qualified properties (reference.property).

Also you can use *@PropertyValue* without *from* nor *value*:

```

@DefaultValueCalculator(value=DefaultProductPriceCalculator.class, properties={
    @PropertyValue(name="familyNumber")
})

```


In this case OpenXava takes the value of the displayed property *familyNumber* and inject it in the property *familyNumber* of the calculator; that is `@PropertyValue(name="familyNumber")` is equivalent to `@PropertyValue(name="familyNumber", from="familyNumber")`.

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

```
@DefaultValueCalculator(value=DetailsCountCalculator.class,
    properties= {
        @PropertyValue(name="year"),
        @PropertyValue(name="number"),
    }
)
private int detailsCount;
```

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 your calculator must implement *IJDBCCalculator* (1) and then it will receive an *IConnectionProvider* (2) that you can use within *calculate()*. OpenXava comes with a set of predefined calculators, you can find them in *org.openxava.calculators*.

Default values on create

You can indicate that the value will be calculated just before creating (inserting into database) an object for the first time. Usually for the key case you use the JPA standard. For example, if you want to use an *identity* (auto increment) column as key:

```

@Id @Hidden
@GeneratedValue(strategy=GenerationType.IDENTITY)
private Integer id;

```

You can use other generation techniques, for example, a database *sequence* can be defined in this JPA standard way:

```

@SequenceGenerator(name="SIZE_SEQ", sequenceName="SIZE_ID_SEQ", allocationSize=1 )
@Id @Hidden @GeneratedValue(strategy=GenerationType.SEQUENCE, generator="SIZE_SEQ")
private Integer id;

```

If you want to generate a unique identifier of type String and 32 characters, you can use a Hibernate extension of JPA:

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

Look at section 9.1.9 of JPA 1.0 specification (part of JSR-220) for learning more about *@GeneratedValue*.

If you want to use your own logic for generating the value on create, or you want a generated new value for a non-key property then you cannot use the JPA *@GeneratedValue*, although it's easy to solve these cases using JPA. You only need to add this code to your class:

```
@PrePersist
private void calculateCounter() {
    counter = new Long(System.currentTimeMillis()).intValue();
}
```

The JPA *@PrePersist* annotation does that this method will be executed before inserting the data the first time in database, in this method you can calculate the value for your key or non-key properties with your own logic.

Property validator

A *@PropertyValidator* executes validation logic on the value assigned to the property just before storing. A property may have several validators:

```
@PropertyValidators ({
    @PropertyValidator(value=ExcludeStringValidator.class, properties=
        @PropertyValue(name="string", value="MOTO")
    ),
    @PropertyValidator(value=ExcludeStringValidator.class, properties=
        @PropertyValue(name="string", value="COCHE"),
        onlyOnCreate=true
    )
})
private String description;
```

The technique to configure the validator (with *@PropertyValue*) is exactly the same than in calculators. With the attribute *onlyOnCreate="true"* you can define that the validation will be executed only when the object is created, and not when it is modified.

The validator code is:

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

```

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

/**
 * @author Javier Paniza
 */

public class ExcludeStringValidator implements IPropertyValidator { // 1

    private String string;

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

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

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

}

```

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

1. **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.
2. **Object value:** The value to validate.
3. **String objectName:** Object name of the container of the property to validate. Useful to use in error messages.
4. **String propertyName:** Name of the property to validate. Useful to use in error messages.

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

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

If the identifier sent is not found in the resource file, this identifier is shown as is; but the recommended way is always to use identifiers of resource files. The validation is successful if no messages are added and fails if messages are added. OpenXava collects all messages of all validators before saving and if there are messages, then it display them and does not save the object. The package *org.openxava.validators* contains some common validators.

Default validator (*new in v2.0.3*)

You can define a default validator for properties depending of its type or stereotype. In order to do it you have to use the file *xava/validators.xml* of your project to define in it the default validators.

For example, you can define in your *xava/validators.xml* the following:

```
<validators>
  <default-validator>
    <validator-class
      class="org.openxava.test.validators.PersonNameValidator">
      <for-stereotype stereotype="PERSON_NAME"/>
    </default-validator>
  </validators>
```

In this case you are associating the validator *PersonNameValidator* to the stereotype *PERSON_NAME*. Now if you define a property as the next one:

```
@Required @Stereotype("PERSON_NAME")
private String name;
```

This property will be validated using *PersonNameValidator* although the property itself does not define any validator. *PersonNameValidator* is applied to all properties with *PERSON_NAME* stereotype.

You can also assign a default validator to a type.

In *validators.xml* files you can also define the validators for determine if a required value is present (executed when you use *@Required*). Moreover you can assign names (alias) to validator classes.

You can learn more about validators examining *OpenXava/xava/validators.xml* and *OpenXavaTest/xava/validators.xml*.

References

A reference allows access from an entity to another entity. 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:

```
Customer customer = ...
customer.getSeller().getName();
```

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

The syntax of reference is:

```
@Required                // 1
@Id                      // 2
@DefaultValueCalculator  // 3
@ManyToOne(              // 4
    optional=false       // 1
)
private type referenceName; // 4
public type getReferenceName() { ... } // 4
public void setReferenceName(type newValue) { ... } // 4
```

1. **@ManyToOne(optional=false)** (JPA), **@Required** (OX) (optional, the JPA is the preferred one): Indicates if the reference is required. When saving OpenXava verifies if the required references are present, if not the saving is aborted and a list of validation errors is returned.
2. **@Id** (JPA, 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.
3. **@DefaultValueCalculator** (OX, 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).
4. **Reference declaration**: A regular Java reference declaration with its getters and setters. The reference is marked with **@ManyToOne (JPA)** and the type must be another entity.

A little example of references:

```
@ManyToOne
private Seller seller; // 1
public Seller getSeller() {
    return seller;
```

```

}
public void setSeller(Seller seller) {
    this.seller = seller;
}

@ManyToOne(fetch=FetchType.LAZY)
private Seller alternateSeller; // 2
public Seller getAlternateSeller() {
    return alternateSeller;
}
public void setAlternateSeller(Seller alternateSeller) {
    this.alternateSeller = alternateSeller;
}

```

1. A reference called *seller* to the entity of *Seller* entity.
2. A reference called *alternateSeller* to the entity *Seller*. In this case we use *fetch=FetchType.LAZY*, in this way the data is readed from database on demand. This is the more efficient approach, but it's not the JPA default, therefore it's advisable to **use always *fetch=FetchType.LAZY*** when declaring the references.

If you assume that this is in an entity named *Customer*, you could write:

```

Customer customer = ...
Seller seller = customer.getSeller();
Seller alternateSeller = customer.getAlternateSeller();

```

Default value calculator in references

In a reference *@DefaultValueCalculator* works like in a property, only that it has to return the value of the reference key.

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

```

@ManyToOne(optional=false, fetch=FetchType.LAZY) @JoinColumn(name="FAMILY")
@DefaultValueCalculator(value=IntegerCalculator.class, properties=
    @PropertyValue(name="value", value="2")
)
private Family family;

```

The *calculate()* method is:

```

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

```

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

In the case of composed key:

```
@ManyToOne(fetch=FetchType.LAZY)
@JoinColumns({
    @JoinColumn(name="ZONE", referencedColumnName="ZONE"),
    @JoinColumn(name="WAREHOUSE", referencedColumnName="NUMBER")
})
@DefaultValueCalculator(DefaultWarehouseCalculator.class)
private Warehouse warehouse;
```

And the calculator code:

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

import org.openxava.calculators.*;

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

    public Object calculate() throws Exception {
        WarehouseKey key = new WarehouseKey();
        key.setNumber(4);
        key.setZoneNumber(4);
        return key;
    }

}
```

Returns an object of type *WarehouseKey*.

Using references as key

You can use references as key, or as part of the key. You have to declare the reference as *@Id*, and use an id class, as following:

```
@Entity
@IdClass(AdditionalDetailKey.class)
public class AdditionalDetail {

    // JoinColumn is also specified in AdditionalDetailKey because
    // a bug in Hibernate, see http://opensource.atlassian.com/projects/hibernate/browse/ANN-361
    @Id @ManyToOne(fetch=FetchType.LAZY)
    @JoinColumn(name="SERVICE")
    private Service service;

    @Id @Hidden
```



```

private int counter;

...

}

```

Also, you need to write your key class:

```

public class AdditionalDetailKey implements java.io.Serializable {

    @ManyToOne(fetch=FetchType.LAZY)
    @JoinColumn(name="SERVICE")
    private Service service;

    @Hidden
    private int counter;

    // equals, hashCode, toString, getters and setters
    ...

}

```

You need to write the key class although the key would be only a reference with only a join column.

It's better to use this feature only when you are working against legacy databases, if you have control over the schema use an autogenerated id instead.

Collections

You can define a collection of references to entities. A collection is a Java property of type *java.util.Collection*.

Here syntax for collection:

```

@Size // 1
@Condition // 2
@OrderBy // 3
@X0rderBy // 4
@OneToMany/@ManyToMany // 5
private Collection collectionName; // 5
public Collection getCollectionName() { ... } // 5
public void setCollectionName(Collection newValue) { ... } // 5

```

1. **@Size** (HV, optional): Minimum (*min*) and/or maximum (*max*) number of expected elements. This is validated just before saving.
2. **@Condition** (OX, optional): Restricts the elements that appear in the collection.

3. **@OrderBy** (JPA, optional): The elements in collections will be in the indicated order.
4. **@XOrderBy** (OX, optional): The `@OrderBy` of JPA does not allow to use qualified properties (properties of references). `@XOrderBy` does allow it.
5. **Collection declaration**: A regular Java collection declaration with its getters and setters. The collection is marked with `@OneToMany (JPA)` or `@ManyToMany (JPA)` and the type must be another entity.

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

```
@OneToMany (mappedBy="invoice")
private Collection deliveries;
public Collection getDeliveries() {
    return deliveries;
}
public void setDeliveries(Collection deliveries) {
    this.deliveries = deliveries;
}
```

If you have this within an *Invoice*, then you are defining a *deliveries* collection associated to that *Invoice*. The details to make the relationship are defined in the object/relational mapping. You use `mappedBy="invoice"` to indicate that the reference *invoice* of *Delivery* is used to mapping this collection.

Now you can write a code like this:

```
Invoice invoice = ...
for (Delivery delivery: invoice.getDeliveries()) {
    delivery.doSomething();
}
```

To do something with all deliveries associated to an invoice.

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

```
@OneToMany (mappedBy="invoice", cascade=CascadeType.REMOVE) // 1
@OrderBy("serviceType desc") // 2
@org.hibernate.validator.Size(min=1) // 3
private Collection details;
```

1. Using REMOVE as cascade type produces that when the user removes an invoice its details are also removed.
2. With `@OrderBy` you force that the details will be returned ordered by *serviceType*.
3. The restriction `@Size(min=1)` requires at least one detail for the invoice to be valid.

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

```
@Condition(
    "${warehouse.zoneNumber} = ${this.warehouse.zoneNumber} AND " +
    "${warehouse.number} = ${this.warehouse.number} AND " +
    "NOT (${number} = ${this.number})"
)
public Collection getFellowCarriers() {
    return null;
}
```

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. `@Condition` only applied to the user interface generated by OpenXava, if you call directly to `getFellowCarriers()` it will be returns null.

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:

```
public Collection getFellowCarriers() {
    Query query = XPersistence.getManager().createQuery("from Carrier c where " +
        "c.warehouse.zoneNumber = :zone AND " +
        "c.warehouse.number = :warehouseNumber AND " +
        "NOT (c.number = :number) ");
    query.setParameter("zone", getWarehouse().getZoneNumber());
    query.setParameter("warehouseNumber", getWarehouse().getNumber());
    query.setParameter("number", getNumber());
    return query.getResultList();
}
```

As you see this is a conventional getter method. Obviously it must return a `java.util.Collection` whose elements are of type *Carrier*.

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

```
@OneToMany(mappedBy="seller")
private Collection customers;
```

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

The `@ManyToMany (JPA)` annotation allows to define a collection with many-to-many multiplicity. As following:

```
@Entity
public class Customer {
    ...
    @ManyToMany
    private Collection<State> states;
    ...
}
```

In this case a customer have a collection of states, but a state can be present in several customers.

Methods

Methods are defined in an OpenXava entity (really a JPA entity) as in a regular Java class. For example:

```
public void increasePrice() {
    setUnitPrice(getUnitPrice().multiply(new BigDecimal("1.02")).setScale(2));
}
```

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

Finders

A finder is a special static method that allows you to find an object or a collection of objects that follow some criteria.

Some examples:

```
public static Customer findByNumber(int number) throws NoResultException {
    Query query = XPersistence.getManager().createQuery(
        "from Customer as o where o.number = :number");
    query.setParameter("number", number);
    return (Customer) query.getSingleResult();
}

public static Collection findAll() {
    Query query = XPersistence.getManager().createQuery("from Customer as o");
    return query.getResultList();
}
```

```

public static Collection findByNameLike(String name) {
    Query query = XPersistence.getManager().createQuery(
        "from Customer as o where o.name like :name order by o.name desc");
    query.setParameter("name", name);
    return query.getResultList();
}

```

This methods can be used this way:

```

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

```

As you see, using finder methods creates a more readable code than using the verbose query API of JPA. But this is only a style recomendation, you can choose do not write finder methods and to use directly JPA queries.

Entity validator

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

Its syntax is:

```

@EntityValidator(
    value=class, // 1
    onlyOnCreate=(true|false), // 2
    properties={ @PropertyValue ... } // 3
)

```

1. **value** (required): Class that implements the validation logic. It has to be of type *IValidator*.
2. **onlyOnCreate** (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.
3. **properties** (several *@PropertyValue*, optional): To set a value of the validator properties before executing it.

An example:

```

@EntityValidator(value=org.openxava.test.validators.CheapProductValidator.class, properties= {
    @PropertyValue(name="limit", value="100"),
    @PropertyValue(name="description"),
    @PropertyValue(name="unitPrice")
}
)

```

```

))
public class Product {

```

And the validator code:

```

package org.openxava.test.validators;

import java.math.*;

/**
 * @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 (getLimiteBd().compareTo(getUnitPrice()) < 0) { // 3
                errors.add("cheap_product", getLimitBd());
            }
        }
    }

    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(Integer.toString(limit));
    }
}

```

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

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

You can define more than one validator for entity using *@EntityValidators*, as following:

```

@EntityValidators({
    @EntityValidator(value=org.openxava.test.validators.CheapProductValidator.class, properties= {
        @PropertyValue(name="limit", value="100"),
        @PropertyValue(name="description"),
        @PropertyValue(name="unitPrice")
    }),
    @EntityValidator(value=org.openxava.test.validators.ExpensiveProductValidator.class, properties= {
        @PropertyValue(name="limit", value="1000"),
        @PropertyValue(name="description"),
        @PropertyValue(name="unitPrice")
    }),
    @EntityValidator(value=org.openxava.test.validators.ForbiddenPriceValidator.class,
        properties= {
            @PropertyValue(name="forbiddenPrice", value="555"),
            @PropertyValue(name="unitPrice")
        },
        onlyOnCreate=true
    )
})
public class Product {

```

Remove validator

The *@RemoveValidator* is a level model validator too, but in this case it is executed just before removing an object, and it has the possibility to deny the deletion.

Its syntax is:

```
@RemoveValidator(
    value=class,           // 1
    properties={ @PropertyValue ... } // 2
)
```

1. **class** (required): Class that implements the validation logic. Must implement *IRemoveValidator*.
2. **properties** (several *@PropertyValue*, optional): To set the value of the validator properties before executing it.

An example can be:

```
@RemoveValidator(value=DeliveryTypeRemoveValidator.class,
    properties=@PropertyValue(name="number")
)
public class DeliveryType {
```

And the validator:

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

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

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

    private DeliveryType deliveryType;
    private int number; // We use this (instead of obtaining it from deliveryType)
                       // for testing @PropertyValue for simple properties

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

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

    public int getNumber() {
        return number;
    }

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



```
}

```

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

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

As in the case of *@EntityValidator* you can use several *@RemoveValidator* for entity using *@RemoveValidators* annotation.

JPA callback methods

With *@PrePersist* you can plug in your own logic to execute just before creating the object as persistent object.

As following:

```
@PrePersist
private void prePersist() {
    setDescription(getDescription() + " CREATED");
}

```

In this case each time that a *DeliveryType* is created a suffix to description is added.

As you see, this is exactly the same as in other methods but is automatically executed just before creation.

With *@PreUpdate* you can plug in some logic to execute after the state of the object is changed and just before it is stored in the database, that is, just before executing UPDATE against database.

As following:

```
@PreUpdate
private void preUpdate() {
    setDescription(getDescription() + " MODIFIED");
}

```

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

As you see, this is exactly the same as in other methods, but it is executed

just before modifying.

You can use all the JPA callback annotations: `@PrePersist`, `@PostPersist`, `@PreRemove`, `@PostRemove`, `@PreUpdate`, `@PostUpdate` and `@PostLoad`.

Embeddable classes

As stated in JPA specification:

"An entity may use other fine-grained classes to represent entity state. Instances of these classes, unlike entity instances themselves, do not have persistent identity. Instead, they exist only as embedded objects of the entity to which they belong. Such embedded objects belong strictly to their owning entity, and are not sharable across persistent entities."

The embeddable class syntax is:

```
@Embeddable                // 1
public class EmbeddableName { // 2
    // Properties            // 3
    // References             // 4
    // Methods                // 5
}
```

1. **@Embeddable** (JPA, one, required): Indicates that this class is a embeddable class of JPA, in other words, its instances will be part of persistent objects.
2. **Class declaration**: As a regular Java class. You can use *extends* and *implements*.
3. **Properties**: Regular Java properties.
4. **References**: References to entities. This is not supported by JPA 1.0 (EJB 3.0), but the Hibernate implementation support it.
5. **Methods**: Java methods with the business logic.

Embedded reference

This example is an embedded (annotated with `@Embedded`) `Address` that is referenced from the main entity.

In the main entity you can write:

```
@Embedded
private Address address;
```

And you have to define the *Address* class as embeddable:

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

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

/**
 *
 * @author Javier Paniza
 */
@Embeddable
public class Address implements IWithCity { // 1

    @Required @Column(length=30)
    private String street;

    @Required @Column(length=5)
    private int zipCode;

    @Required @Column(length=20)
    private String city;

    // ManyToOne inside an Embeddable is not supported by JPA 1.0 (see at 9.1.34),
    // but Hibernate implementation supports it.
    @ManyToOne(fetch=FetchType.LAZY, optional=false) @JoinColumn(name="STATE")
    private State state; // 2

    public String getCity() {
        return city;
    }

    public void setCity(String city) {
        this.city = city;
    }

    public String getStreet() {
        return street;
    }

    public void setStreet(String street) {
        this.street = street;
    }

    public int getZipCode() {
        return zipCode;
    }

    public void setZipCode(int zipCode) {
```

```

        this.zipCode = zipCode;
    }

    public State getState() {
        return state;
    }

    public void setState(State state) {
        this.state = state;
    }
}

```

As you see an embeddable class can implement an interface (1) and contain references (2), among other things, but it cannot have persistent collections nor to use JPA callback methods.

This code can be used this way, for reading:

```

Customer 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();
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*.

Embedded collections

Embedded collections are not supported by JPA 1.0. But you can simulate them using collections to entities with cascade type REMOVE or ALL. OpenXava manages these collections in a special way, as they were embedded collections.

Now an example of an embedded collection. In the main entity (for example *Invoice*) you can write:

```
@OneToMany(mappedBy="invoice", cascade=CascadeType.REMOVE)
private Collection details;
```

Note that you use *CascadeType.REMOVE*, and *InvoiceDetail* is an entity, not an embeddable class:

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

import java.math.*;

import javax.persistence.*;

import org.hibernate.annotations.Columns;
import org.hibernate.annotations.Type;
import org.hibernate.annotations.Parameter;
import org.hibernate.annotations.GenericGenerator;
import org.openxava.annotations.*;
import org.openxava.calculators.*;
import org.openxava.test.validators.*;

/**
 *
 * @author Javier Paniza
 */
@Entity
@EntityValidator(value=InvoiceDetailValidator.class,
    properties= {
        @PropertyValue(name="invoice"),
        @PropertyValue(name="oid"),
        @PropertyValue(name="product"),
        @PropertyValue(name="unitPrice")
    }
)
public class InvoiceDetail {

    @ManyToOne // Lazy fetching produces a fails on removing a detail from invoice
    private Invoice invoice;

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

    private ServiceType serviceType;
    public enum ServiceType { SPECIAL, URGENT }

    @Column(length=4) @Required
    private int quantity;

    @StoredProcedure("MONEY") @Required
    private BigDecimal unitPrice;

    @ManyToOne(fetch=FetchType.LAZY, optional=false)
```

```

private Product product;

@DefaultValueCalculator(CurrentDateCalculator.class)
private java.util.Date deliveryDate;

@ManyToOne(fetch=FetchType.LAZY)
private Seller soldBy;

@Stereotype("MEMO")
private String remarks;

@Stereotype("MONEY") @Depends("unitPrice, quantity")
public BigDecimal getAmount() {
    return getUnitPrice().multiply(new BigDecimal(getQuantity()));
}

public boolean isFree() {
    return getAmount().compareTo(new BigDecimal("0")) <= 0;
}

@PostRemove
private void postRemove() {
    invoice.setComment(invoice.getComment() + "DETAIL DELETED");
}

public String getOid() {
    return oid;
}

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

public ServiceType getServiceType() {
    return serviceType;
}

public void setServiceType(ServiceType serviceType) {
    this.serviceType = serviceType;
}

public int getQuantity() {
    return quantity;
}

public void setQuantity(int quantity) {
    this.quantity = quantity;
}

public BigDecimal getUnitPrice() {
    return unitPrice==null?BigDecimal.ZERO:unitPrice;
}

public void setUnitPrice(BigDecimal unitPrice) {
    this.unitPrice = unitPrice;
}

public Product getProduct() {
    return product;
}

public void setProduct(Product product) {

```

```

        this.product = product;
    }

    public java.util.Date getDeliveryDate() {
        return deliveryDate;
    }

    public void setDeliveryDate(java.util.Date deliveryDate) {
        this.deliveryDate = deliveryDate;
    }

    public Seller getSoldBy() {
        return soldBy;
    }

    public void setSoldBy(Seller soldBy) {
        this.soldBy = soldBy;
    }

    public String getRemarks() {
        return remarks;
    }

    public void setRemarks(String remarks) {
        this.remarks = remarks;
    }

    public Invoice getInvoice() {
        return invoice;
    }

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

```

As you see this is a complex entity, with calculators, validators, references and so on. Also you have to define a reference to the container class (*invoice*). In this case when an *Invoice* is removed all its details are removed too. Moreover there are differences at user interface level (you can learn more on view chapter).

Inheritance

OpenXava supports Java and JPA inheritance.

For example you can define a *@MappedSuperclass* in this way:

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

```

import javax.persistence.*;

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

/**
 * Base class for defining entities with a UUID oid. <p>
 *
 * @author Javier Paniza
 */
@MappedSuperclass
public class Identifiable {

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

    public String getOid() {
        return oid;
    }

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

}

```

You can define another *@MappedSuperclass* that extends from this one, for example:

```

package org.openxava.test.model;

import javax.persistence.*;

import org.openxava.annotations.*;

/**
 * Base class for entities with a 'name' property. <p>
 *
 * @author Javier Paniza
 */
@MappedSuperclass
public class Nameable extends Identifiable {

    @Column(length=50) @Required
    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {

```



```

        this.name = name;
    }
}

```

Now you can use *Identifiable* or *Nameable* for defining your entities, as following:

```

package org.openxava.test.model;

import javax.persistence.*;

/**
 *
 * @author Javier Paniza
 */
@Entity
@DiscriminatorColumn(name="TYPE")
@DiscriminatorValue("HUM")
@Table(name="PERSON")
@AttributeOverrides(
    @AttributeOverride(name="name", column=@Column(name="PNAME"))
)
public class Human extends Nameable {

    @Enumerated(EnumType.STRING)
    private Sex sex;
    public enum Sex { MALE, FEMALE };

    public Sex getSex() {
        return sex;
    }
    public void setSex(Sex sex) {
        this.sex = sex;
    }
}

```

And now, the real entity inheritance, an entity that extends other entity:

```

package org.openxava.test.model;

import javax.persistence.*;

/**
 *
 * @author Javier Paniza
 */
@Entity
@DiscriminatorValue("PRO")

```

```

public class Programmer extends Human {

    @Column(length=20)
    private String mainLanguage;

    public String getMainLanguage() {
        return mainLanguage;
    }

    public void setMainLanguage(String mainLanguage) {
        this.mainLanguage = mainLanguage;
    }

}

```

You can create an OpenXava module for *Human* and *Programmer* (not for *Identifiable* or *Nameable* directly). In the *Programmer* module the user can only access to programmers, in the other hand using *Human* module the user can access to *Human* and *Programmer* objects. Moreover when the user tries to view the detail of a *Programmer* from the *Human* module the *Programmer* view will be show. True polymorphism.

About mapping, `@AttributeOverrides` is supported, but, at the moment, only a single table per class hierarchy mapping strategy works.

Composite key

The preferred way for defining the key of an entity is a single autogenerated key (annotated with `@Id` and `@GeneratedValue`), but sometimes, for example when you go against legat database, you need to have an entity mapped to a table that uses several column as key. This case can be solved with JPA (therefore with OpenXava) in two ways, using `@IdClass` or using `@EmbeddedId`

Id class

In this case you use `@IdClass` in your entity to indicate a key class, and you mark the key properties as `@Id` in your entity:

```

package org.openxava.test.model;

import javax.persistence.*;

import org.openxava.annotations.*;

```

```

import org.openxava.jpa.*;

/**
 *
 * @author Javier Paniza
 */

@Entity
@IdClass(WarehouseKey.class)
public class Warehouse {

    @Id
    // Column is also specified in WarehouseKey because a bug in Hibernate, see
    // http://opensource.atlassian.com/projects/hibernate/browse/ANN-361
    @Column(length=3, name="ZONE")
    private int zoneNumber;

    @Id @Column(length=3)
    private int number;

    @Column(length=40) @Required
    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getNumber() {
        return number;
    }

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

    public int getZoneNumber() {
        return zoneNumber;
    }

    public void setZoneNumber(int zoneNumber) {
        this.zoneNumber = zoneNumber;
    }
}

```

You also need to declare your id class, a serializable regular class with all key properties from the entity:

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

import java.io.*;

import javax.persistence.*;

/**
 *
 * @author Javier Paniza
 */

public class WarehouseKey implements Serializable {

    @Column(name="ZONE")
    private int zoneNumber;
    private int number;

    @Override
    public boolean equals(Object obj) {
        if (obj == null) return false;
        return obj.toString().equals(this.toString());
    }

    @Override
    public int hashCode() {
        return toString().hashCode();
    }

    @Override
    public String toString() {
        return "WarehouseKey::" + zoneNumber + ":" + number;
    }

    public int getNumber() {
        return number;
    }

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

    public int getZoneNumber() {
        return zoneNumber;
    }

    public void setZoneNumber(int zoneNumber) {
        this.zoneNumber = zoneNumber;
    }

}
```

Embedded id

In this case you have a reference to a *@Embeddable* object marked as *@EmbeddedId*:

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

import javax.persistence.*;

import org.openxava.annotations.*;

/**
 *
 * @author Javier Paniza
 */

@Entity
public class Warehouse {

    @EmbeddedId
    private WarehouseKey key;

    @Column(length=40) @Required
    private String name;

    public WarehouseKey getKey() {
        return key;
    }

    public void setKey(WarehouseKey key) {
        this.key = key;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

}
```

And you key is an embeddable class that holds the key properties:

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

import javax.persistence.*;

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

```
@Embeddable
public class WarehouseKey implements java.io.Serializable {

    @Column(length=3, name="ZONE")
    private int zoneNumber;

    @Column(length=3)
    private int number;

    public int getNumber() {
        return number;
    }

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

    public int getZoneNumber() {
        return zoneNumber;
    }

    public void setZoneNumber(int zoneNumber) {
        this.zoneNumber = zoneNumber;
    }

}
```

.....

.....

Chapter 4: View

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

Layout

The `@View` annotation can be used in an entity or embeddable class in order to define the layout of its members in the user interface.

The syntax for `@View` is:

```
@View(  
    name="name",      // 1  
    members="members" // 2  
)  
public class MyEntity {
```

1. **name** (optional): This name identifies the view, and can be used in other OpenXava places (for example in *application.xml*) or from another entity. If the view has no name then the view is assumed as the default one, that is the natural form to display an object of this type.
2. **members** (optional): Indicates the members to display and its layout in the user interface. By default it displays all members (excluding hidden ones) in the order in which are declared in the model. Inside members you can use section and group elements for layout purposes; or action element for showing a link associated to a custom action inside your view.

You can define several views for an entity using the `@Views` annotation.

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

For example, a model like this:

```
@Entity  
@IdClass(ClerkKey.class)  
public class Clerk {
```



```

@Id @Required
@Column(length=3, name="ZONE")
private int zoneNumber;

@Id @Required
@Column(length=3, name="OFFICE")
private int officeNumber;

@Id @Required
@Column(length=3, name="NUMBER")
private int number;

@Required @Column(length=40)
private String name;

// Getters and setters
...
}

```

Generates a view that looks like this:

Zone		333
Office		1
Number		1
Name		PEPE

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

```

@Entity
@IdClass(ClerkKey.class)
@View(members="zoneNumber; officeNumber; number")
public class Clerk {

```

In this case *name* is not shown.

Also you can use *members* for tuning the layout:

```

@View(members=
    "zoneNumber, officeNumber, number;" +
    "name"
)

```

You can observe that the member names are separated by commas or by semicolon, this is used to indicate layout. With comma the member is placed just the following (at right), and with semicolon the next member is put

below (in the next line). Hence the previous view is displayed in this way:



Groups

With groups you can lump a set of related properties and it has visual effect. For defining a group you only need to put the name of the group and after it its members within square brackets. Just in this way:

```
@View(members=
    "id [ zoneNumber, officeNumber, number ]," +
    "name"
)
```

In this case the result is:



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

You can put several groups in a view:

```
@View(members=
    "general [" +
    "    number;" +
    "    type;" +
    "    name;" +
    "]" +
    "contact [" +
    "    telephone;" +
    "    email;" +
    "    website;" +
    "]"
)
```

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

General		Contact	
Code	1	Telephone	961112233
Type	Steady	eMail	pepe@mycompany.com
Name	Javi	Web site	http://www.openxava.org

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

```
@View(members=
  "general [" +
  "  number;" +
  "  type;" +
  "  name;" +
  "];" +
  "contact [" +
  "  telephone;" +
  "  email;" +
  "  website;" +
  "]"
)
```

In this case the view is shown this way:




General		Contact	
Code	1	Telephone	961112233
Type	Steady	eMail	pepe@mycompany.com
Name	Javi	Web site	http://www.openxava.org

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



```
@View(members=  
  "invoice;" +  
  "deliveryData [" +  
    "  type, number;" +  
    "  date;" +  
    "  description;" +  
    "  shipment;" +  
    "  transportData [" +  
      "    distance; vehicle; transportMode; driverType;" +  
    "  ]" +  
    "  deliveryByData [" +  
      "    deliveredBy;" +  
      "    carrier;" +  
      "    employee;" +  
    "  ]" +  
  "]"  
)
```


And the result will be:

Invoice




Year  Number  Date  Year discount €

Delivery data


Type  Number 

Date 

Description

Shipment   

Transport ata


Distance 

Vehicle

Transport mode

Driver type



Delivery by data

Delivered by 


Sometimes it's useful to layout members aligned by columns, like in a table. For example, the next view:

```
@View(name="Amounts", members=
  "year, number;" +
  "amounts [" +
    "customerDiscount, customerTypeDiscount, yearDiscount;" +
    "amountsSum, vatPercentage, vat;" +
  "]"
)
```

...will be displayed as following:

Year  Number 

Amounts

Customer discount	11.50	€	Customer type discount	20.00	€	Year discount	200.00	€
Amounts sum	2,500.00	€	VAT %  <input type="text" value="16"/>	V.A.T.	400.00			

This is ugly. It would be better to have all data aligned by columns. You can define the group in this way:

```
@View(name="Amounts", members=
  "year, number;" +
  "amounts [#" +
    "customerDiscount, customerTypeDiscount, yearDiscount;" +
    "amountsSum, vatPercentage, vat;" +
  "]"
)
```

Note that now you use [# instead of [. Now you obtain this result:

Year Number

Amounts			
Customer discount	11.50 €	Customer type discount	20.00 €
Amounts sum	2,500.00 €	VAT %	16
		V.A.T.	400.00 €

Now, thanks to the #, the members are aligned by columns. This feature is also available for the sections (see below).

Sections

Furthermore the members can be organized in sections. For defining a section you only need to put the name of the section and after it its members within braces. Let's see an example from the *Invoice* entity:

```
@View(members=
  "year, number, date, paid;" +
  "comment;" +
  "customer { customer }" +
  "details { details }" +
  "amounts { amountsSum; vatPercentage; vat }" +
  "deliveries { deliveries }"
)
```

The visual result is:

Year 2002 Number 1 Date 12/31/01 Paid

Comment

Customer Details Amounts Deliveries

Little code

Type

Name [Change name label](#)

Address

ViewProperty

Street Zip code

 State

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

Nested sections are allowed. For example, you can define a view as this:

```
@View(name="NestedSections", members=
  "year, number, date;" +
  "customer { customer }" +
  "data {" +
  "  details { details }" +
  "  amounts {" +
  "    vat { vatPercentage; vat }" +
  "    amountsSum { amountsSum }" +
  "  }" +
  "}" +
  "deliveries { deliveries }"
)
```

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

As in the groups case, the sections allow using # for aligning by columns, like this:

```
@View(name="AlignedAmountsInSection", members=
  "year, number;" +
  "customer { customer }" +
  "details { details }" +
  "amounts {#" +
    "customerDiscount, customerTypeDiscount, yearDiscount;" +
    "amountsSum, vatPercentage, vat;" +
  "}"
)
```

With the same effect as in the group case.

Layout philosophy

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

Rules for view annotations

You can annotate a member (property, reference or collection) with several annotations that refine its display style and behaviour. Moreover you can define that effect of these annotations only applies to some views.

For example if you have an entity as this one:

```
@Entity
@Views({
    @View( members= "number; type; name; address" ),
    @View( name="A", members= "number; type; name; address; seller" ),
    @View( name="B", members= "number; type; name; seller; alternateSeller" ),
    @View( name="C", members="number; type; name; address; deliveryPlaces" )
})
public class Customer {
```

And you want the *name* property will be read only. You can annotate it in this way:

```
@ReadOnly
private String name;
```

In this way *name* is read only in all views. However, you may want that *name* will be read only only on views *B* and *C*, then you can define the member as following:

```
@ReadOnly(forViews="B, C")
private String name;
```

Another way for defining this same case is:

```
@ReadOnly(notForViews="DEFAULT, A")
private String name;
```

Using *notForViews* you indicate the views where *name* property is not read only. *DEFAULT* is used for referencing to the default view, the view with no name.

Some annotations have one or more values, for example for indicating which view of the referenced type will be used for displaying a reference you use the *@ReferenceView* annotation:

```
@ReferenceView("Simple")
private Seller seller;
```

In this case when the seller is displayed the view *Simple*, defined in *Seller* class, is used.

What if you want to use *Simple* view of *Seller* only in *B* view of *Customer*? It's easy:

```
@ReferenceView(forViews="B", value="Simple")
private Seller seller;
```

What if you want to use *Simple* view of *Seller* only in *B* view of *Customer* and the *VerySimple* view of *Seller* for *A* view of *Customer*? In this case you have to use several `@ReferenceView` grouping them with `@ReferenceViews`, just in this way:

```
@ReferenceViews({
    @ReferenceView(forViews="B", value="Simple"),
    @ReferenceView(forViews="A", value="VerySimple")
})
```

These rules apply to all the annotations in this chapter, except `@View` and `@Views`.

Property customization

You can refine the visual aspect and behavior of a property in a view using the next annotations:

```
@ReadOnly // 1
@LabelFormat // 2
@DisplaySize // 3
@OnChange // 4
@Action // 5
@Editor // 6
private type propertyName;
```

All these annotations follow the rules for view annotations and all they are optionals. OpenXava always assumes a correct default values if they are omitted.

1. **@ReadOnly** (OX): If you mark a property with this annotation it never will be editable by the final user in this view. An alternative to this is to make the property editable or not editable programmatically using `org.openxava.view.View`.
2. **@LabelFormat** (OX): Format to display the label of this property. Its value can be `LabelFormatType.NORMAL`, `LabelFormatType.SMALL` or `LabelFormatType.NO_LABEL`.

3. **@DisplaySize** (OX): The size in characters of the editor in the User Interface used to display this property. The editor display only the characters indicated by *@DisplaySize* but it allows to the user to entry until the total size of the property. If *@DisplaySize* is not specified, the value of the size of the property is assumed.
4. **@OnChange** (OX): Action to execute when the value of this property changes. Only one *@OnChange* action per view is allowed.
5. **@Action** (OX): Actions (showed as links, buttons or images to the user) associated (visually) to this property and that the final user can execute. It's possible to define several *@Action* for each view.
6. **@Editor** (OX): Name of the editor to use for displaying the property in this view. The editor must be declared in *OpenXava/xava/default-editors.xml* or *xava/editors.xml* of your project.

Label format

A simple example of using *@LabelFormat*:

```
@LabelFormat(LabelFormatType.SMALL)
private int zipCode;
```

In this case the zip code is displayed as:

Zip code

46540

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

Property value change event

If you wish to react to the event of a value change of a property you can use *@OnChange* as following:

```
@OnChange(OnChangeCustomerNameAction.class)
private String name;
```

The code to execute is:

```

package org.openxava.test.actions;

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

/**
 * @author Javier Paniza
 */
public class OnChangeCustomerNameAction extends OnChangePropertyBaseAction { // 1

    public void execute() throws Exception {
        String value = (String) getNewValue(); // 2
        if (value == null) return;
        if (value.startsWith("Javi")) {
            getView().setValue("type", Customer.Type.STEADY); // 3
        }
    }
}

```

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

Actions of property

You can also specify actions (*@Action*) that the user can click directly:

```

@Action("Delivery.generateNumber")
private int number;

```

In this case instead of an action class you have to write the action identifier that is the controller name and the action name. This action must be registered in *controllers.xml* in this way:

```

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

```

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



By default the action link is present only when the property is editable, but if the property is read only (`@ReadOnly`) or calculated then it is always present. You can use the attribute `alwaysEnabled` to `true` so that the link is always present, even if the property is not editable. As following:

```
@Action(value="Delivery.generateNumber", alwaysEnabled=true)
```

The attribute `alwaysEnabled` is optional and its default value is `false`. The code of previous action is:

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

import org.openxava.actions.*;

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

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

}
```

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

Optionally you can make your action an `IPropertyAction` (this is only available for actions associated to properties with `@Action` annotation), thus the container view and the property name are injected in the action by OpenXava. The above action class could be rewritten in this way:

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

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

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

```

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

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

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

```

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

Choosing an editor

An editor display the property to the user and allows him to edit its value. OpenXava uses by default the editor associated to the stereotype or type of the property, but you can specify a concrete editor for display a property using *@Editor*.

For example, OpenXava uses a combo for editing the properties of type *enum*, but if you want to display a property of this type in some particular view using a radio button you can define that view in this way:

```

@Editor(forViews="TypeWithRadioButton", value="ValidValuesRadioButton")
private Type type;
public enum Type { NORMAL, STEADY, SPECIAL };

```

In this case for displaying/editing the editor *ValidValuesRadioButton* will be used, instead of default one. *ValidValueRadioButton* is defined in *OpenXava/xava/default-editors.xml* as following:

```

<editor name="ValidValuesRadioButton" url="radioButtonEditor.jsp">

```

This editor is included with OpenXava, but you can create your own editors with your custom JSP code and declare them in the file *xava/editors.xml* of

your project.

This feature is for changing the editor only in one view. If you want to change the editor for a type, stereotype or a property of a model at application level then it's better to configure it using *xava/editors.xml* file.

Reference customization

You can refine the visual aspect and behavior of a reference in a view using the next annotations:

```
@ReferenceView    // 1
@ReadOnly        // 2
@NoFrame         // 3
@NoCreate        // 4
@NoModify        // 5
@NoSearch        // 6
@AsEmbedded      // 7
@SearchAction    // 8
@DescriptionsList // 9
@LabelFormat     // 10
@Action          // 11
@OnChange        // 12
@OnChangeSearch  // 13
@ManyToOne
private type referenceName;
```

All these annotations follow the rules for view annotations and all they are optionals. OpenXava always assumes a correct default values if they are omitted.

1. **@ReferenceView** (OX): If you omit this annotation, then the default view of the referenced object is used. With this annotation you can indicate that it uses another view.
2. **@ReadOnly** (OX): If you use this annotation the reference never will be editable by final user in this view. An alternative is to make the property editable/uneditable programmatically using *org.openxava.view.View*.
3. **@NoFrame** (OX): If the reference is displayed with no frame. By default the references are displayed with frame.
4. **@NoCreate** (OX): By default the final user can create new objects of the referenced type from here. If you use this annotation this will not be possible.
5. **@NoModify** (OX): By default the final user can modify the current referenced object from here. If you use this annotation this will not be possible.

6. **@NoSearch** (OX): By default the user will have a link to make searches with a list, filters, etc. If you use this annotation this will not be possible.
7. **@AsEmbedded** (OX): By default in the case of a reference to an embeddable the user can create and edit its data, while in the case of a reference to an entity the user can only to choose an existing entity. If you put *@AsEmbedded* then the user interface for references to entities behaves as a in the embedded case, allowing to the user to create a new object and editing its data directly. It has no effect in case of a reference to embeddables. Warning! If you remove an entity its referenced entities are not removed, even if they are displayed using *@AsEmbedded*.
8. **@SearchAction** (OX): Allows you to specify your own action for searching when the user click in the search link. Only one by view is allowed.
9. **@DescriptionsList** (OX): Display the data as a list of descriptions, typically as a combo. Useful when there are few elements of the referenced object.
10. **@LabelFormat** (OX): Format to display the label of the reference. It only applies if this reference is annotated with *@DescriptionsList*. Works as in property case.
11. **@Action** (OX): Actions (showed as links, buttons or images to the user) associated (visually) to this reference and that the final user can execute. Works as in property case. You can define several actions for each reference in the same view.
12. **@OnChange** (OX): Action to execute when the value of this reference changes. Only one *@OnChange* action by view is allowed.
13. **@OnChangeSearch** (OX): Allows you to specify your own action for searching when the user type a new key. Only one by view is allowed.

If you do not use any of these annotations OpenXava draws a reference using the default view. For example, if you have a reference like this:

```
@ManyToOne
private Family family;
```


The user interface will look like this:

Choose view

The most simple customization is to specify the view of the referenced object that you want to use. This is done by means of `@ReferenceView`:

```
@ManyToOne(fetch=FetchType.LAZY)
@ReferenceView("Simple")
private Invoice invoice;
```

In the *Invoice* entity you must have a view named *Simple*:

```
@Entity
@Views({
    ...
    @View(name="Simple", members="year, number, date, yearDiscount:"),
    ...
})
public class Invoice {
```

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

Customizing frame

If you combine `@NoFrame` with groups you can group visually a property that is not a part of a reference with that reference, for example:

```
@View( members=
    ...
```

```

"seller [" +
"  seller;" +
"  relationWithSeller;" +
"]" +
...
)
public class Customer {
...
@ManyToOne(fetch=FetchType.LAZY)
@NoFrame
private Seller seller;
...
}

```

And the result:

Seller

Number	    
Name	 MANUEL CHAVARRI
Relation with seller	 BUENA

Custom search action

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

To define your custom search logic you have to use `@SearchAction` in this way:

```

@ManyToOne(fetch=FetchType.LAZY) @SearchAction("MyReference.search")
private Seller seller;

```

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

```

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

```

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

```

package org.openxava.test.actions;

import org.openxava.actions.*;

/**
 * @author Javier Paniza
 */

public class MySearchAction extends ReferenceSearchAction {

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

}

```

You will learn more about actions in chapter 7.

Custom creation action

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

```

<!--
Because its name is WarehouseCreation (model name + Creation) it is used

```

```

by default for create from reference, instead of NewCreation.
Action 'new' is executed automatically.
-->
<controller name="WarehouseCreation">
  <extends controller="NewCreation"/>
  <action name="new" hidden="true"
    class="org.openxava.test.actions.CreateNewWarehouseFromReferenceAction">
    <use-object name="xava_view"/>
  </action>
</controller>

```

In this case when the user clicks on the 'create' link, the user is directed to the default view of *Warehouse* and the actions in *WarehouseCreation* 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.

Custom modification action

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

```

<!--
Because its name is WarehouseModification (model name + Modification) it is used
by default for modifying from reference, instead of Modification.
The action 'search' is executed automatically.
-->
<controller name="WarehouseModification">
  <extends controller="Modification"/>
  <action name="search" hidden="true"
    class="org.openxava.test.actions.ModifyWarehouseFromReferenceAction">
    <use-object name="xava_view"/>
  </action>
</controller>

```

In this case when the user clicks on the 'modify' link, the user is directed to the default view of *Warehouse* and the actions in *WarehouseModification* will be allowed.

If you have an action called `'search'`, it will be executed automatically before all. It is used to initialize the view with the object to modify.

Descriptions list (combos)

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

The syntax is:

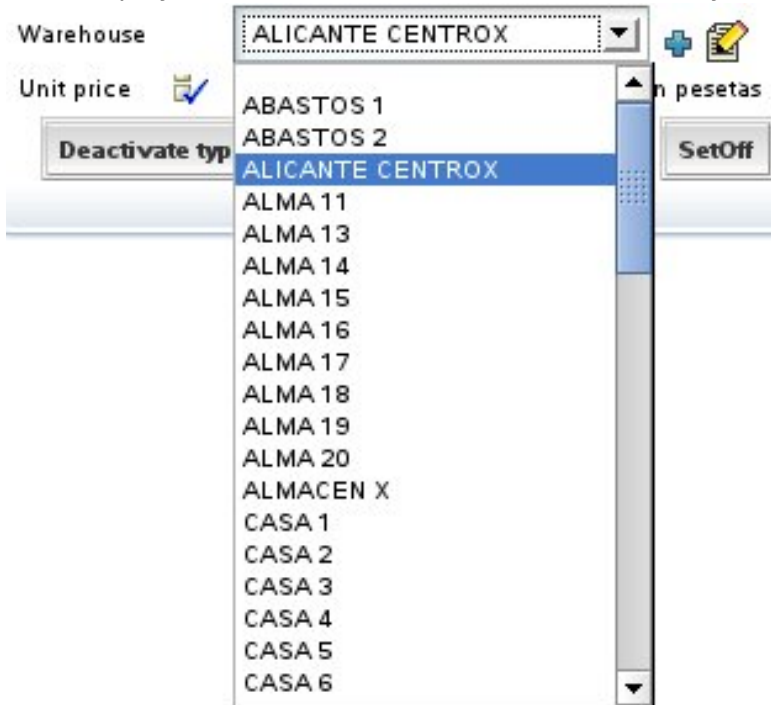
```
@DescriptionsList(
    descriptionProperties="properties", // 1
    depends="depends",                // 2
    condition="condition",           // 3
    orderByKey="true|false",         // 4
    order="order"                    // 5
)
```

1. **descriptionProperties** (optional): The property or properties to show in the list, if not specified, the property named *description*, *descripcion*, *name* or *nombre* is assumed. If the referenced object does not have a property called this way then it is required to specify a property name here. It allows to set more than one property separated by commas. To the final user the values are concatenated.
2. **depends** (optional): It's used in together with *condition*. It can be achieve that the list content depends on another value displayed in the main view (if you simply type the name of the member) or in the same view (if you type *this.* before the name of the member).
3. **condition** (optional): Allows to specify a condition (with SQL style) to filter the values that are shown in the description list.
4. **orderByKey** (optional): By default the data is ordered by description, but if you set this property to *true* it will be ordered by key.
5. **order** (optional): Allows to specify an order (with SQL style) for the values that are shown in the description list.

The simplest usage is:

```
@ManyToOne(fetch=FetchType.LAZY)
@DescriptionsList
private Warehouse warehouse;
```

That displays a reference to warehouse in this way:



In this case it shows all warehouses, although in reality it uses the *baseCondition* and the *filter* specified in the default *@Tab* of *Warehouse*. You will see more about tabs in chapter 5.

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

```
@ManyToOne(fetch=FetchType.LAZY)
@DescriptionsList(orderByKey=true) // 1
private Family family;

@ManyToOne(fetch=FetchType.LAZY) @NoCreate // 2
@DescriptionsList(
    descriptionProperties="description", // 3
    depends="family", // 4
    condition="{family.number} = ?" // 5
    order="{description} desc" // 6
)
private Subfamily subfamily;
```

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

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

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

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

You can specify several properties to be shown as description:

```
@ManyToOne(fetch=FetchType.LAZY)
@ReadOnly
@DescriptionsList(descriptionProperties="level.description, name")
private Seller alternateSeller;
```

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

If you use *@ReadOnly* in a reference annotated as *@DescriptionsList*, then the description (in this case *level.description* + *name*) is displayed as a simple text property instead of using a combo.

Reference value change event

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

```
@ManyToOne(fetch=FetchType.LAZY)
@OnChange(OnChangeCarrierInDeliveryAction.class)
private Carrier carrier;
```

In this case your action listens to the change of carrier number. 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;
        getView().setValue("remarks", "The carrier is " + getNewValue());
        addMessage("carrier_changed");
    }
}

```

The action implements *OnChangePropertyAction*, by means of *OnChangePropertyBaseAction* (1), although it's a reference. We receive the change of the key property of the reference; in this case *carrier.number*. The rest is as in the property case.

Reference search on change event

The user can search the value of a reference simply typing its key. For example, if there is a reference to *Subfamily*, the user can type the subfamily number and automatically the subfamily data is loaded in the view. This is done using a default on change action that does the search. You can specify your own action for search when key change using *@OnChangeSearch* annotation, just in this way:

```

@ManyToOne(fetch=FetchType.LAZY)
@OnChangeSearch(OnChangeSubfamilySearchAction.class)
private Subfamily subfamily;

```

This action is executed for doing the search, instead of the standard action, when the user changes the subfamily number.

The code to execute is:

```

package org.openxava.test.actions;

import org.openxava.actions.*;

/**
 *
 * @author Javier Paniza
 */

```



```

public class OnChangeSubfamilySearchAction
    extends OnChangeSearchAction { // 1

    public void execute() throws Exception {
        if (getView().getValueInt("number") == 0) {
            getView().setValue("number", new Integer("1"));
        }
        super.execute();
    }
}

```

The action implements *IONChangePropertyAction*, by means of *OnChangeSearchAction* (1), although it's a reference. It receives the change of the key property of the reference; in this case *subfamily.number*.

This case is an example of refining the behaviour of on change search, because it extends from *OnChangeSearchAction*, that is the default action for searching, and calls to *super.execute()*. Also it's possible to do a regular on change action (extending from *OnChangePropertyBaseAction* for example) overriding completely the search logic.

Collection customization

You can refine the visual aspect and behavior of a collection in a view using the next annotations:

```

@CollectionView // 1
@ReadOnly // 2
@EditOnly // 3
@NoCreate // 4
@NoModify // 5
@AsEmbedded // 6
@ListProperties // 7
@RowStyle // 8
@EditAction // 9
@ViewAction // 10
@NewAction // 11
@SaveAction // 12
@HideDetailAction // 13
@RemoveAction // 14
@RemoveSelectedAction // 15
@ListAction // 16
@DetailAction // 17
@OneToMany/@ManyToMany
private Collection collectionName;

```

All these annotations follow the rules for view annotations and all they are optional. OpenXava always assumes a correct default values if they are omitted.

1. **@CollectionView** (OX): The view of the referenced object (each collection element) which is used to display the detail. By default the default view is used.
2. **@ReadOnly** (OX): If you set it then the final user only can view collection elements, he cannot add, delete or modify elements.
3. **@EditOnly** (OX): If you set it then the final user can modify existing elements, but not add or remove collection elements.
4. **@NoCreate** (OX): If you set it then the final user doesn't get the link to create new objects of the referenced object type. It does not apply to embedded collections.
5. **@NoModify** (OX): If you set it then the final user doesn't get the link to modify the objects of the referenced object type. It does not apply to embedded collections.
6. **@AsEmbedded** (OX): By default the embedded collections (with cascade type REMOVE or ALL) allow the users to create and to edit elements, while the regular collections allow only to choose existing entities to add to (or remove from) the collection. If you put *@AsEmbedded* then the collection behaves as a embedded collection even though it hasn't cascade type REMOVE or ALL, allowing to the user to add objects and editing them directly. It has no effect in case of embedded collections.
7. **@ListProperties** (OX): Properties to show in the list for visualization of the collection. You can qualify the properties. By default it shows all persistent properties of the referenced object (excluding references and calculated properties).
8. **@RowStyle** (OX): To give a special style to some rows. Behaves equals that in the Tab case. It does not works for calculated collections. It's possible to define several *@RowStyle* for each view.
9. **@EditAction** (OX): Allows you to define your custom action to begin the editing of a collection element. This is the action showed in each row of the collection, if the collection is editable. Only one *@EditAction* per view is allowed.
10. **@ViewAction** (OX): Allows you to define your custom action to view a collection element. This is the action showed in each row, if the collection is read only. Only one *@ViewAction* per view is allowed.

11. **@NewAction** (OX): Allows you to define your custom action to start adding a new element to the collection. This is the action executed on click in 'Add' link. Only one *@ViewAction* per view is allowed.
12. **@SaveAction** (OX): Allows you to define your custom action to save the collection element. This is the action executed on click in 'Save detail' link. Only one *@SaveAction* per view is allowed.
13. **@HideDetailAction** (OX): Allows you to define your custom action to hide the detail view. This is the action executed on click in 'Close' link. Only one *@HideDetailAction* per view is allowed.
14. **@RemoveAction** (OX): Allows you to define your custom action to remove the element from the collection. This is the action executed on click in 'Remove detail' link. Only one *@RemoveAction* per view is allowed.
15. **@RemoveSelectedAction** (OX): Allows you to define your custom action to remove the selected elements from the collection. This is the action executed when a user select some rows and then click in 'Remove selected' link. Only one *@RemoveSelectedAction* per view is allowed.
16. **@ListAction** (OX): To add actions in list mode; usually actions which scope is the entire collection. It's possible to define several *@ListAction* for each view.
17. **@DetailAction** (OX): To add actions in detail mode, usually actions which scope is the detail that is being edited. It's possible to define several *@DetailAction* for each view.

If you do not use any of these annotations then the collection is displayed using the persistent properties in list mode and the default view to represent the detail; although in typical scenarios the properties of the list and the view for detail are specified:


```
@CollectionView("Simple"),
@ListProperties("number, name, remarks, relationWithSeller, seller.level.description, type")
@OneToMany(mappedBy="seller")
private Collection<Customer> customers;
```


And the collection is displayed:

	Number	Name	Remarks	Relation with seller	Seller level	Type
	1	Javi		BUENA	MANAGER	Steady
	2	Juanillo			MANAGER	Normal

1 **There are 2 records in list ([Hide them](#))**

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

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

This view is also used if the user click on  ('Add') in an embedded collection, otherwise OpenXava does not show this view, instead it shown a list of entities to add.




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


```
@View(name="Simple", members="number; type; name; address")
```


On clicking in a detail the following will be shown:

Customers



Customer



Little code   

Type 




Name 

Address

Street  Zip code 


City  State 

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

	Number	Name	Remarks	Relation with seller	Seller level	Type
	=	starts	starts	starts	starts	
	1	Javi		BUENA	MANAGER	Steady
	2	Juanillo			MANAGER	Normal

1 There are 2 records in list ([Hide them](#))

Custom edit/view action

You can refine easily the behavior when the  ('Edit') link is clicked using `@EditAction`:

```
@EditAction("Invoice.editDetail")
@OneToMany (mappedBy="invoice", cascade=CascadeType.REMOVE)
private Collection<InvoiceDetail> details;
```

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

```
<controller name="Invoice">
  ...
  <action name="editDetail" hidden="true"
    image="images/edit.gif"
    class="org.openxava.test.actions.EditInvoiceDetailAction">
    <use-object name="xava_view"/>
  </action>
```

```
...
</controller>
```

And finally write your action:

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

import java.text.*;

import org.openxava.actions.*;

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

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

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

Also it's possible to remove the edit action from the User Interface, in this way:

```
@EditAction("")
@OneToMany(mappedBy="invoice", cascade=CascadeType.REMOVE)
private Collection<InvoiceDetail> details;
```

You only need to put an empty string as value for the action. Although in most case it's enough to define the collection as *@ReadOnly*.

The technique to refine the view action (the action for each row, if the collection is read only) is the same but using *@ViewAction* instead of *@EditAction*.

Custom list actions

Adding our custom list actions (actions that apply to entire collections) is easy using *@ListAction*:

```
@ListAction("Carrier.translateName")
private Collection<Carrier> fellowCarriers;
```

Now a new link is shown to the user:

The screenshot shows a web application interface. At the top, there is a link labeled "Translate name" with a red arrow icon and a green checkmark icon. Below this is a table with the following columns: "Number", "Name", "Remarks", and "Calculated". The table contains three rows of data:

Number	Name	Remarks	Calculated
2	DOS		TR
3	TRES		TR
4	CUATRO		TR

At the bottom of the table, there is a status bar that reads: "1 There are 3 records in list (Hide them)".

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

```
<controller name="Carrier">
  ...
  <action name="translateName" hidden="true"
    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()) {
            Carrier carrier = (Carrier) it.next();
        }
    }
}
```

```

        carrier.translate();
    }
}

```

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

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

Also it's possible to use actions for list mode as list actions for a collection.

Default list actions

If you want to add some custom list actions to all the collection of your application you can do it creating a controller called *DefaultListActionsForCollections* in your own *xava/controllers.xml* file as following:

```

<controller name="DefaultListActionsForCollections">
  <extends controller="Print"/>
  <action name="exportAsXML"
    class="org.openxava.test.actions.ExportAsXMLAction">
  </action>
</controller>

```

In this way all the collections will have the actions of *Print* controller (for export to Excel and generate PDF report) and your own *ExportAsXMLAction*. This has the same effect of *@ListAction* (look at *custom list actions* section) but it applies to all collections at once.


This feature does not apply to calculated collections.

Custom detail actions

Also you can add your custom actions to the detail view used for editing each element. This is accomplish by means of *@DetailAction* annotation. These actions are applicable only to one element of collection. For example:


```
@DetailAction("Invoice.viewProduct")
@OneToMany(mappedBy="invoice", cascade=CascadeType.REMOVE)
private Collection<InvoiceDetail> details;
```

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



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

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

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

And the code of your action:

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

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

import org.openxava.actions.*;

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

    private Map invoiceValues;

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

```

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

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

public String getCustomView() {
    return SAME_VIEW;
}

public Map getInvoiceValues() {
    return invoiceValues;
}

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

```

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

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

Refining collection view default behavior

Using *@NewAction*, *@SaveAction*, *@HideDetailAction*, *@RemoveAction* and *@RemoveSelectedAction* you can refine the default behavior of collection view. For example if you want to refine the behavior of save a detail action you can define your view in this way:

```
@SaveAction("DeliveryDetail.save")
@OneToMany (mappedBy="delivery", cascade=CascadeType.REMOVE)
private Collection<DeliveryDetail> details;
```

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

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

And define your action class for saving:

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

import org.openxava.actions.*;

/**
 * @author Javier Paniza
 */

public class SaveDeliveryDetailAction extends SaveElementInCollectionAction { // 1

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

}
```

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

Also it's possible to remove any of these actions from User Interface, for example, you can define a collection in this way:

```
@RemoveSelectedAction("")
@OneToMany (mappedBy="delivery", cascade=CascadeType.REMOVE)
private Collection<DeliveryDetail> details;
```

In this case the action for removing the selected elements in the collection will be missing in the User Interface. As you see, only it's needed to declare an empty string as the name of the action.

Transient properties for UI controls

With *@Transient (JPA)* you define a property that is not stored in database but you want to show to the user. You can use it to provide UI controls to allow the user to manage his user interface.

An example:

```
@Transient
@DefaultValueCalculator(value=EnumCalculator.class,
    properties={
        @PropertyValue(name="enumType", value="org.openxava.test.model.Delivery$DeliveredBy")
        @PropertyValue(name="value", value="CARRIER")
    }
)
@OnChange(OnChangeDeliveryByAction.class)
private DeliveredBy deliveredBy;
public enum DeliveredBy { EMPLOYEE, CARRIER }
```

You can see that the syntax is exactly the same as in the case of a regular property of an entity; you can even use *enum* and *@DefaultValueCalculator*. After defining the property you can use it in the view as usual, for example with *@OnChange* or putting it as member of a view.

View actions

In addition of associating actions to a property, reference or collection, you also can define arbitrary actions inside your view, in any place. In order to do this we use the qualified name of action using brackets () as suffix, in this way:

```
@View( members=
    "number;" +
    "type;" +
    "name, Customer.changeNameLabel();" +
    ...
```

The visual effect will be:

Code		1	
Type		Steady	
Name		Javi	Change name label

You can see the link 'Change name label' that will execute the action

Customer.changeNameLabel on click on it.

If the container view of the action is not editable, the action is not present. If you want that the action is always enabled, even if the view is not editable, you have to use put the word ALWAYS between the brackets, as following:

```
@View( name="Simple", members=
    "number;" +
    "type;" +
    "name, Customer.changeNameLabel(ALWAYS);" +
    ...
```

The standard way to expose actions to the user is using the controllers (actions in a bar), the controllers are reusable between views, but sometimes you will need an action specific to a view, and you want display it inside the view (not in the button bar), for these cases the view actions may be useful. See more about actions in chapter 7.

Transient class: Only for creating views

In OpenXava it is not possible to have a view without model. Thus if you want to draw an arbitrary user interface, you need to create a class, not to declare it as entity and define your view in it.

An transient class is not associated to any table of the database, typically it's used only for display User Interfaces not related to any data in database.

An example can be:

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

import javax.persistence.*;

import org.openxava.annotations.*;

/**
 * Example of an transient OpenXava model class (not persistent). <p>
 *
 * This can be used, for example, to display a dialog,
 * or any other graphical interface.<p>
 *
 * Note that is not marked as @Entity <br>
 *
 * @author Javier Paniza
 */

@Views({
    @View(name="Family1", members="subfamily"),
    @View(name="Family2", members="subfamily"),
```

```

    @View(name="WithSubfamilyForm", members="subfamily"),
    @View(name="Range", members="subfamily; subfamilyTo")
  })
  public class FilterBySubfamily {

    @ManyToOne(fetch=FetchType.LAZY) @Required
    @NoCreate(forViews="Family1, Family2")
    @NoModify(forViews="Family2, WithSubfamilyForm")
    @NoSearch(forViews="WithSubfamilyForm")
    @DescriptionsLists({
      @DescriptionsList(forViews="Family1",
        condition="{family.number} = 1", order="{number} desc"
      ),
      @DescriptionsList(forViews="Family2",
        condition="{family.number} = 2"
      )
    })
    private Subfamily subfamily;

    @ManyToOne(fetch=FetchType.LAZY)
    private Subfamily subfamilyTo;

    public Subfamily getSubfamily() {
      return subfamily;
    }

    public void setSubfamily(Subfamily subfamily) {
      this.subfamily = subfamily;
    }

    public Subfamily getSubfamilyTo() {
      return subfamilyTo;
    }

    public void setSubfamilyTo(Subfamily subfamilyTo) {
      this.subfamilyTo = subfamilyTo;
    }
  }
}

```

For defining a model class as transient you only need to define a regular Java class without *@Entity* annotation. You mustn't put the mapping annotations nor declare properties as key.

This way you can design a dialog that can be useful, for example, to print a report of families or products filtered by subfamily.

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.

.....

.....

Chapter 5: Tabular data

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

The screenshot shows the 'OpenXavaTest - Warehouses management' interface. It features a table with the following data:

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

Below the table, there are pagination controls showing '1 2 3 4 5 6' and a message: 'There are 63 records in list (Hide them)'. At the bottom, there are three buttons: 'Delete selected', 'Selected to lowercase', and 'Change page row count'.

This list allows user to:

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

The default list is enough for many cases, moreover the user can customize it. Nevertheless, sometimes it is convenient to modify the list behavior. For this you have the `@Tab` annotation within the entity definition.

The syntax of `@Tab` is:

```
@Tab(
    name="name",           // 1
    filter=filter class,  // 2
    rowStyles=array of @RowStyle, // 3
    properties="properties", // 4
    baseCondition="base condition", // 5
    defaultOrder="default order" // 6
)
public class MyEntity {
```

1. **name** (optional): You can define several tabs in a entity (using `@Tabs` annotation), 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** (optional): Allows to define programmatically some logic to apply to the values entered by user when he filters the list data.
3. **rowStyles** (optional): A simple way to specify a different visual style for some rows. Normally to emphasize rows that fulfill certain condition. You specify an array of `@RowStyle`, in this way you can use several styles for a tab.
4. **properties** (optional): The list of properties to show initially. Can be qualified (that is you can specify `referenceName.propertyName` at any depth level).
5. **baseCondition** (optional): Condition to be fulfilled by the displayed data. It's added to the user condition if needed.
6. **defaultOrder** (optional): To specify the initial order for data.

Initial properties and emphasize rows

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









```
@Tab(
    rowStyles=@RowStyle(style="highlight", property="type", value="steady"),
    properties="name, type, seller.name, address.city, seller.level.description, address.state.name"
)
```

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 `@RowStyle`; 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 of Address	Seller level	State of Address
		starts ▾	▾	starts ▾	starts ▾	starts ▾	starts ▾
	<input type="checkbox"/>	Javi	Steady	MANUEL CHAVARRI	EL PUIG	MANAGER	NEW YORK
	<input type="checkbox"/>	Juanillo	Normal	MANUEL CHAVARRI	VALENCIA	MANAGER	COLORADO
	<input type="checkbox"/>	Carmelo	Normal		EL PUIG		NEW YORK
	<input type="checkbox"/>	Cuatrero	Normal		VALENCIA		NEW YORK

1

There are 4 records in list ([Hide them](#))

Filters and base condition

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

```
@Tab(name="Current",
  filter=CurrentYearFilter.class,
  properties="year, number, amountsSum, vat, detailsCount, paid, customer.name",
  baseCondition="${year} = ?"
)
```

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

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

import java.util.*;

import org.openxava.filters.*;

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

```

public class CurrentYearFilter implements IFilter { // (1)

    public Object filter(Object o) throws FilterException { // (2)
        Calendar cal = Calendar.getInstance();
        cal.setTime(new java.util.Date());
        Integer year = new Integer(cal.get(Calendar.YEAR));
        Object [] r = null;
        if (o == null) { // (3)
            r = new Object[1];
            r[0] = year;
        }
        else if (o instanceof Object []) { // (4)
            Object [] a = (Object []) o;
            r = new Object[a.length + 1];
            r[0] = year;
            for (int i = 0; i < a.length; i++) {
                r[i+1]=a[i];
            }
        }
        else { // (5)
            r = new Object[2];
            r[0] = year;
            r[1] = o;
        }

        return r;
    }
}

```

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

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

Another case:

```

@Tab(name="DefaultYear",
    filter=DefaultYearFilter.class,
    properties="year, number, customer.number, customer.name, amountsSum, " +
        "vat, detailsCount, paid, importance",
    baseCondition="{$year} = ?"

```

```
)
```

In this case the filter is:

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

import java.util.*;

import org.openxava.filters.*;

/**
 * @author Javier Paniza
 */
public class DefaultYearFilter extends BaseContextFilter {           // (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 environment variables inside a filter of type *BaseContextFilter*, using *getEnvironment()* method, just in this way:

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

For learning more about environment variables see the chapter 7 about controllers.

Pure SQL select

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

```
@Tab(name="CompleteSelect",
  properties="number, description, family",
  baseCondition=
    "select" +
    "  ${number}, ${description}, XAVATEST@separator@FAMILY.DESCRPTION " +
    "from " +
    "  XAVATEST@separator@SUBFAMILY, XAVATEST@separator@FAMILY " +
    "where " +
    "  XAVATEST@separator@SUBFAMILY.FAMILY = " +
    "  XAVATEST@separator@FAMILY.NUMBER"
)
```

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

Default order

Finally, setting a default order is very easy:

```
@Tab(name="Simple", properties="year, number, date",
  defaultOrder="${year} desc, ${number} desc"
)
```

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

.....

.....

Chapter 6: Object/relational mapping

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

Object/relational tools allow you to work with objects instead of tables and columns, and to generate automatically the SQL code to read and update the database. In this way you do not need direct access to the SQL database. Of course you have to define precisely how to map your classes to your tables, and this work is done using JPA mapping annotations.

The OpenXava entities are JPA entities, therefore the object/relational mapping in OpenXava is done by means of the Java Persistence API (JPA). This chapter shows the more basic mapping techniques and some special cases. If you want to learn more about JPA you can look at the documentation of Hibernate Annotations (the JPA implementation used by OpenXava by default), or whatever JPA manual you want.

Entity mapping

The `@Table` annotation specifies the primary table for the annotated entity. Additional tables may be specified using `@SecondaryTable` or `@SecondaryTables` annotation.

If no `@Table` annotation is specified for an entity class, the default values apply.

Example:

```
@Entity
@Table(name="CUST", schema="XAVATEST")
public class Customer {
```

Property mapping

The `@Column` annotation is used to specify a mapped column for a persistent property or field. If no `@Column` annotation is specified, the default values are applied.

A simple example:

```
@Column(name="DESC", length=512)
private String description;
```

An example annotating the getter:

```
@Column(name="DESC", nullable=false, length=512)
public String getDescription() { return description; }
```

Other examples:

```
@Column(name="DESC",
        columnDefinition="CLOB NOT NULL",
        table="EMP_DETAIL")
@Lob
private String description;

@Column(name="ORDER_COST", updatable=false, precision=12, scale=2)
private BigDecimal cost;
```

Reference mapping

The *@JoinColumn* annotation is used to specify a mapped column for a reference.

Example:

```
@ManyToOne
@JoinColumn(name="CUST_ID")
private Customer customer;
```

If you need to define a mapping for the composite foreign keys use *@JoinColumns*. This annotation groups *@JoinColumn* annotations for the same reference.

When the *@JoinColumns* annotation is used, both the *name* and the *referencedColumnName* elements must be specified in each such *@JoinColumn* annotation.

Example:

```
@ManyToOne
@JoinColumns({
    @JoinColumn(name="INV_YEAR", referencedColumnName="YEAR"),
    @JoinColumn(name="INV_NUMBER", referencedColumnName="NUMBER")
})
private Invoice invoice;
```


Collection mapping

When you use `@OneToMany` for a collection the mapping depends of the reference used in the other part of the association, that is, usually it's not needed to do anything. But if you are using `@ManyToMany`, maybe it's useful to declare the `@JoinTable`, as following:

```
@ManyToMany
@JoinTable(name="CUSTOMER_STATE",
    joinColumns=@JoinColumn(name="CUSTOMER"),
    inverseJoinColumns=@JoinColumn(name="STATE")
)
private Collection<State> states;
```

If `@JoinTable` is missing the default values apply.

Embedded reference mapping

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

Just using `@AttributeOverrides` annotations, in this way:

```
@Embedded
@AttributeOverrides({
    @AttributeOverride(name="street", column=@Column("ADDR_STREET")),
    @AttributeOverride(name="zip", column=@Column("ADDR_ZIP")),
    @AttributeOverride(name="city", column=@Column("ADDR_CITY")),
    @AttributeOverride(name="country", column=@Column("ADDR_COUNTRY"))
})
private Address address;
```

If you do not use `@AttributeOverrides` default values are assumed.

Type conversion

The type conversion between Java and the relational database is a work for the JPA implementation (OpenXava uses Hibernate by default). Usually, the default type conversion is good for most cases, but if you work with legacy database perhaps you need to use the tips here.

Given that OpenXava uses the type conversion facility provided by Hibernate you can learn more on Hibernate documentation.

Property conversion

When the type of a Java property and the type of its corresponding column in DB do not match you need to write a Hibernate Type in order to do your custom type conversion.

For example, if you have a property of type *String []*, and you want to store its value concatenating it in a single table column of VARCHAR type. Then you must declare the conversion for your property in this way:

```
@Type(type="org.openxava.test.types.RegionsType")
private String [] regions;
```

The conversion logic in *RegionsType* is:

```
package org.openxava.test.types;

import java.io.*;
import java.sql.*;

import org.apache.commons.logging.*;
import org.hibernate.*;
import org.hibernate.usertype.*;
import org.openxava.util.*;

/**
 *
 * @author Javier Paniza
 */

public class RegionsType implements UserType { // 1

    public int[] sqlTypes() {
        return new int[] { Types.VARCHAR };
    }

    public Class returnedClass() {
        return String[].class;
    }

    public boolean equals(Object obj1, Object obj2) throws HibernateException {
        return Is.equal(obj1, obj2);
    }

    public int hashCode(Object obj) throws HibernateException {
        return obj.hashCode();
    }

    public Object nullSafeGet(ResultSet resultSet, String[] names, Object owner) // 2
```

```

    throws HibernateException, SQLException
    {
        Object o = resultSet.getObject(names[0]);
        if (o == null) return new String[0];
        String dbValue = (String) o;
        String [] javaValue = new String [dbValue.length()];
        for (int i = 0; i < javaValue.length; i++) {
            javaValue[i] = String.valueOf(dbValue.charAt(i));
        }
        return javaValue;
    }

    public void nullSafeSet(PreparedStatement ps, Object value, int index) // 3
        throws HibernateException, SQLException
    {
        if (value == null) {
            ps.setString(index, "");
            return;
        }
        String [] javaValue = (String []) value;
        StringBuffer dbValue = new StringBuffer();
        for (int i = 0; i < javaValue.length; i++) {
            dbValue.append(javaValue[i]);
        }
        ps.setString(index, dbValue.toString());
    }

    public Object deepCopy(Object obj) throws HibernateException {
        return obj == null?null:((String []) obj).clone();
    }

    public boolean isMutable() {
        return true;
    }

    public Serializable disassemble(Object obj) throws HibernateException {
        return (Serializable) obj;
    }

    public Object assemble(Serializable cached, Object owner) throws HibernateException {
        return cached;
    }

    public Object replace(Object original, Object target, Object owner) throws HibernateException {
        return original;
    }
}

```

The type converter has to implement *org.hibernate.usertype.UserType* (1). The main methods are *nullSafeGet* (2) for read from database and to convert to Java, and *nullSafeSet* (3) for writing the Java value into database. OpenXava has generic Hibernate type converters in the *org.openxava.types*

package ready to use. One of them is *EnumLetterType*, that allows to map properties of *enum* type. For example, if you have a property like this:

```
private Distance distance;
public enum Distance { LOCAL, NATIONAL, INTERNATIONAL };
```

In this Java property 'LOCAL' is 1, 'NATIONAL' is 2 and 'INTERNATIONAL' is 3 when the property is stored in database. But what happens, if in the database a single letter ('L', 'N' or 'I') is stored? In this case you can use *EnumLetterType* in this way:

```
@Type(type="org.openxava.types.EnumLetterType",
    parameters={
        @Parameter(name="letters", value="LNI"),
        @Parameter(name="enumType", value="org.openxava.test.model.Delivery$Distance")
    }
)
private Distance distance;
public enum Distance { LOCAL, NATIONAL, INTERNATIONAL };
```

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

Multiple column conversion

With *CompositeUserType* you can map several table columns to a single Java property. This is useful if you have properties of custom class that have itself several attributes to store. Also it is used when you have to deal with legate database schemes.

A typical example is the generic converter *Date3Type*, that allows to store in the database 3 columns and in Java a single property of type *java.util.Date*.

```
@Type(type="org.openxava.types.Date3Type")
@Column(columns = {
    @Column(name="YEARDELIVERY"),
    @Column(name="MONTHDELIVERY"),
    @Column(name="DAYDELIVERY")
})
private java.util.Date deliveryDate;
```

DAYDELIVERY, MONTHDELIVERY and YEAREDELIVERY are 3 columns in database that store the delivery date. And here *Date3Type*:

```

package org.openxava.types;

import java.io.*;
import java.sql.*;

import org.hibernate.*;
import org.hibernate.engine.*;
import org.hibernate.type.*;
import org.hibernate.usertype.*;
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 Date3Type implements CompositeUserType { // 1

    public String[] getPropertyNames() {
        return new String[] { "year", "month", "day" };
    }

    public Type[] getPropertyTypes() {
        return new Type[] { Hibernate.INTEGER, Hibernate.INTEGER, Hibernate.INTEGER };
    }

    public Object getPropertyValue(Object component, int property) throws HibernateException { // 2
        java.util.Date date = (java.util.Date) component;
        switch (property) {
            case 0:
                return Dates.getYear(date);
            case 1:
                return Dates.getMonth(date);
            case 2:
                return Dates.getYear(date);
        }
        throw new HibernateException(XavaResources.getString("date3_type_only_3_properties"));
    }

    public void setPropertyValue(Object component, int property, Object value)
        throws HibernateException // 3
    {
        java.util.Date date = (java.util.Date) component;
        int intValue = value == null?0:((Number) value).intValue();
        switch (property) {
            case 0:
                Dates.setYear(date, intValue);
            case 1:
                Dates.setMonth(date, intValue);
            case 2:
                Dates.setYear(date, intValue);
        }
    }
}

```

```

        throw new HibernateException(XavaResources.getString("date3_type_only_3_properties"));
    }

    public Class returnedClass() {
        return java.util.Date.class;
    }

    public boolean equals(Object x, Object y) throws HibernateException {
        if (x==y) return true;
        if (x==null || y==null) return false;
        return !Dates.isDifferentDay((java.util.Date) x, (java.util.Date) y);
    }

    public int hashCode(Object x) throws HibernateException {
        return x.hashCode();
    }

    public Object nullSafeGet(ResultSet rs, String[] names, SessionImplementor session, Object owner)
        throws HibernateException, SQLException // 4
    {
        Number year = (Number) Hibernate.INTEGER.nullSafeGet( rs, names[0] );
        Number month = (Number) Hibernate.INTEGER.nullSafeGet( rs, names[1] );
        Number day = (Number) Hibernate.INTEGER.nullSafeGet( rs, names[2] );

        int iyear = year == null?0:year.intValue();
        int imonth = month == null?0:month.intValue();
        int iday = day == null?0:day.intValue();

        return Dates.create(iday, imonth, iyear);
    }

    public void nullSafeSet(PreparedStatement st, Object value, int index, SessionImplementor session)
        throws HibernateException, SQLException // 5
    {
        java.util.Date d = (java.util.Date) value;
        Hibernate.INTEGER.nullSafeSet(st, Dates.getYear(d), index);
        Hibernate.INTEGER.nullSafeSet(st, Dates.getMonth(d), index + 1);
        Hibernate.INTEGER.nullSafeSet(st, Dates.getDay(d), index + 2);
    }

    public Object deepCopy(Object value) throws HibernateException {
        java.util.Date d = (java.util.Date) value;
        if (value == null) return null;
        return (java.util.Date) d.clone();
    }

    public boolean isMutable() {
        return true;
    }

    public Serializable disassemble(Object value, SessionImplementor session)
        throws HibernateException
    {
        return (Serializable) deepCopy(value);
    }

```

```

public Object assemble(Serializable cached, SessionImplementor session, Object owner)
    throws HibernateException
{
    return deepCopy(cached);
}

public Object replace(Object original, Object target, SessionImplementor session, Object owner)
    throws HibernateException
{
    return deepCopy(original);
}
}

```

As you see the type converter implements *CompositeUserType* (1). The key methods are *getPropertyValue* (2) and *setPropertyValue* (3) to get and to set values in the properties of the object of the composite type, and *nullSafeGet* (4) and *nullSafeSet* (5) for reading and storing this object from and to database.

Reference conversion

Reference conversion is not supported directly by Hibernate. But in some very rare circumstances maybe you need to do conversion in the reference. In this section we explain how to do it.

For example, you may have a reference to driver licence using two columns, `DRIVINGLICENCE_LEVEL` and `DRIVINGLICENCE_TYPE`, and the `DRIVINGLICENCE_TYPE` column does not admit null, but it's possible that the object can have no reference to driving licence in which case the column `DRIVINGLICENCE_TYPE` hold an empty string. This is not a normal case if you design the database using foreign keys, but if the database was designed by a RPG programmer, for example, this was done in this way, because RPG programmer are not used to cope with nulls.

That is, you need a conversion for `DRIVINGLICENCE_TYPE`, for transform null to empty string. This can be achieve with a code like this:

```

// We apply conversion (null into an empty String) to DRIVINGLICENCE_TYPE column
// In order to do it, we create drivingLicence_level and drivingLicence_type
// We make JoinColumns not insertable nor updatable, we modify the get/setDrivingLicence methods
// and we create a drivingLicenceConversion() method.
@ManyToOne(fetch=FetchType.LAZY)
@JoinColumns({ // 1
    @JoinColumn(name="DRIVINGLICENCE_LEVEL", referencedColumnName="LEVEL",

```

```

        insertable=false, updatable=false),
        @JoinColumn(name="DRIVINGLICENCE_TYPE", referencedColumnName="TYPE",
            insertable=false, updatable=false)
    ))
    private DrivingLicence drivingLicence;
    private Integer drivingLicence_level; // 2
    private String drivingLicence_type; // 2

    public DrivingLicence getDrivingLicence() { // 3
        // In this way because the column for type of driving lincence does not admit null
        try {
            if (drivingLicence != null) drivingLicence.toString(); // to force load
            return drivingLicence;
        }
        catch (EntityNotFoundException ex) {
            return null;
        }
    }

    public void setDrivingLicence(DrivingLicence licence) { // 4
        // In this way because the column for type of driving lincence does not admit null
        this.drivingLicence = licence;
        this.drivingLicence_level = licence==null?null:licence.getLevel();
        this.drivingLicence_type = licence==null?null:licence.getType();
    }

    @PrePersist @PreUpdate
    private void drivingLicenceConversion() { // 5
        if (this.drivingLicence_type == null) this.drivingLicence_type = "";
    }

```

First, you have to use `@JoinColumn` with `insertable=false` and `updatable=false` on all `@JoinColumn` (1), in this way the reference is readed from database, but it is not write. Also define plain properties for storing the foreign key of the reference (2).

Now you must write a getter, `getDrivingLicence()` (3), for returning null when the reference is not found; and a setter, `setDrivingLicence()` (4), for assigning the key of the reference to the correspondng plain properties.

Finally, you have to write a callback method, `drivingLincenceConversion()` (5), to do the conversion work. This method will be automatically executed on create and update.

This example shows how it's possible to wrap legate databases simply using a little of programming and some basic resources from JPA.

.....

.....

Chapter 7: Controllers

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 *default-controllers.xml* that contains a group of generic controllers that can be used in your applications.

The *controllers.xml* file contains an element of type `<controllers/>` with the syntax:

```
<controllers>
  <env-var ... /> ...      <!-- 1 -->
  <object ... /> ...      <!-- 2 -->
  <controller ... /> ...  <!-- 3 -->
</controllers>
```

1. **env-var** (several, optional): Variable that contains configuration information. This variable can be accessed from the actions and filters, and its value can be overwritten in each module.
2. **object** (several, optional): Defines Java object with session scope; that is objects that are created for an user and exist during his session.
3. **controller** (several, required): A controller is a group of actions.

Environment variables

The environment variables contain configuration information. These variables can be accessed from the actions and filters, and its value can be overwritten in each module. Its syntax is:

```
<env-var
  name="name"          <!-- 1 -->
  value="value"       <!-- 2 -->
/>
```

1. **name** (required): Name of the environment variable in uppercase and using underscore to separate words.
2. **value** (required): Value for the environment variable.

These are some example:

```
<env-var name="MYAPPLICATION_DEFAULT_YEAR" value="2007"/>
<env-var name="MYAPPLICATION_COLOR" value="RED"/>
```

Session objects

The Java objects declared in *controllers.xml* have session scope; that is, they are objects that are created for a user and exist during his session. It's syntax is:

```
<object
  name="objectName"           <!-- 1 -->
  class="objectType"         <!-- 2 -->
  value="initialValue"       <!-- 3 -->
  scope="module|global"     <!-- 4 New in v2.1 -->
/>
```

1. **name** (required): Name of the object, usually you use the application name as prefix to avoid name collision in large projects.
2. **class** (required): Full qualified Java class for this object.
3. **value** (optional): Initial value for the object.
4. **scope** (optional): (*New in v2.1*) The default value is module. If you use module scope each module will have its own copy of this object. If you use global scope the same object will be shared by all modules of all OpenXava applications (running in the same .war).

Defining session objects is very easy, you can see the defined ones in *OpenXava/xava/default-controllers.xml*:

```
<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"/>
```

These objects are used by OpenXava in order to work, although it is quite normal that you use some of these from your actions. If you want to create your own objects you can do it in your *controllers.xml* in the *xava* directory of your project.

The controller and its actions

The syntax of controller is:

```
<controller
  name="name"           <!-- 1 -->
>
  <extends ... /> ...   <!-- 2 -->
  <action ... /> ...    <!-- 3 -->
</controller>
```

1. **name** (required): Name of the controller.
2. **extends** (several, optional): Allows to use multiple inheritance, to do this the controller inherits all actions from other controller(s).
3. **action** (several, required): Implements the logic to execute when the final user clicks a button or link.

The controllers consist of actions, and actions are the main things. Here is its syntax:

```
<action
  name="name"           <!-- 1 -->
  label="label"         <!-- 2 -->
  description="description" <!-- 3 -->
  mode="detail|list|ALL" <!-- 4 -->
  image="image"         <!-- 5 -->
  class="class"         <!-- 6 -->
  hidden="true|false"   <!-- 7 -->
  on-init="true|false"  <!-- 8 -->
  on-each-request="true|false" <!-- 9 New in v2.1.2 -->
  before-each-request="true|false" <!-- 10 New in v2.2.5 -->
  by-default="never|if-possible|almost-always|always" <!-- 11 -->
  takes-long="true|false" <!-- 12 -->
  confirm="true|false"  <!-- 13 -->
  keystroke="keystroke" <!-- 14 New in v2.0.1 -->
>
  <set ... /> ...       <!-- 15 -->
  <use-object ... /> ... <!-- 16 -->
</action>
```

1. **name** (required): Action name that must be unique within its controller, but it can be repeated in other controllers. When you reference an action always use the format *ControllerName.actionName*.
2. **label** (optional): Button label or link text. It's much better to use i18n files.
3. **description** (optional): Description text of the action. It's much better to use i18n files.
4. **mode** (optional): Indicates in which mode the action has to be visible. The default value is ALL, that means that this action is always visible.

5. **image** (optional): URL of the image associated with this action. In the current implementation if you specify an image, it is shown to user in link format.
6. **class** (optional): Implements the logic to execute. Must implement *IAction* interface.
7. **hidden** (optional): A hidden action is not shown in the button bar, although it can be used in all other places, for example to associate it to an event, as action of a property, in collections, etc. The default is *false*.
8. **on-init** (optional): If you set this property to *true*, then the action will be executed automatically on initiating the module. The default is *false*.
9. **on-each-request** (optional): (*New in v2.1.2*) If you set this property to *true*, then the action will be executed automatically on each request of the user, that is, on first module execution and before each user action execution. In the moment of execution all OpenXava session objects are setup and ready to use. That is, from this action you can use *xava_view* and *xava_tab*. The default is *false*.
10. **before-each-request** (optional): (*New in v2.2.5*) If you set this property to *true*, then the action will be executed automatically before each request of the user, that is, on first module execution and before each user action execution, but before the OpenXava session objects are setup and ready to use. That is, from this action you cannot use *xava_view* or *xava_tab*. The default is *false*.
11. **by-default** (optional): Indicates the weight of this action on choosing the action to execute as the default one. The default action is executed when the user presses ENTER. The default is *never*.
12. **takes-long** (optional): If you set it to *true*, then you are indicating that this action takes long time in executing (minutes or hours). In the current implementation OpenXava shows a progress bar. The default is *false*.
13. **confirm** (optional): If you set it to *true*, then before executing the action a dialog is shown to the user to ask if he is sure to execute it. The default is *false*.
14. **keystroke** (optional): (*New in v2.0.1*) Defines a keystroke that the user can press for executing this action. The possible values are the same as for *javax.swing.KeyStroke*. Examples: "control A", "alt x", "F7".

15. **set** (several, optional): Sets a value of action properties. Thus the same action class can be configured in different ways and it can be used in several controllers.
16. **use-object** (several, optional): Assigns a session object to an action property just before executing the action. After the execution the property value is put back into the context again (update the session object, thus you can update even immutable objects).

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

A plain controller might look like this:

```
<controller name="Remarks">
  <action name="hideRemarks"
    class="org.openxava.test.actions.HideShowPropertyAction">
    <set property="property" value="remarks" />
    <set property="hide" value="true" />
    <use-object name="xava_view"/>
  </action>
  <action name="showRemarks" mode="detail"
    class="org.openxava.test.actions.HideShowPropertyAction">
    <set property="property" value="remarks" />
    <set property="hide" value="false" />
    <use-object name="xava_view"/>
  </action>
  <action name="setRemarks" mode="detail"
    class="org.openxava.test.actions.SetPropertyValueAction">
    <set property="property" value="remarks" />
    <set property="value" value="Hell in your eyes" />
    <use-object name="xava_view"/>
  </action>
</controller>
```

Now you can include this controller into the module that you want; this is made by editing in `xava/application.xml` the module in which you can use these actions:

```
<module name="Deliveries">
  <model name="Delivery"/>
  <controller name="Typical"/>
  <controller name="Remarks"/>
</module>
```

Thus you have in your module the actions of *Typical* (CRUD and printing) plus these defined by you in the controller named *Remarks*. The top button bar of the module will have this aspect:



Ant the bottom button bar:

You can note as actions with image are located on top and actions without image are located at bottom.

You can write code for *hideRemarks* like this:

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

import org.openxava.actions.*;

/**
 * @author Javier Paniza
 */
public class HideShowPropertyAction extends ViewBaseAction { // 1

    private boolean hide;
    private String property;

    public void execute() throws Exception { // 2
        getView().setHidden(property, hide); // 3
    }

    public boolean isHide() {
        return hide;
    }

    public void setHide(boolean b) {
        hide = b;
    }

    public String getProperty() {
        return property;
    }

    public void setProperty(String string) {
        property = string;
    }
}
```

```
}

```

An action must implement `IAction`, but usually it extends from a base class that implements this interface. The base action more basic is `BaseAction` that implements most of the method of `IAction` except `execute()`. In this case you use `ViewBaseAction` as base class. `ViewBaseAction` has the property `view` of type `View`. This joined to the next declaration in action...

```
<use-object name="xava_view"/>

```

...allows to manage the view (the user interface) from an action using view. The `<use-object />` gets the session object `xava_view` and assigns it to the property `view` (removing the prefix `xava_`, in general removes the prefix `myapplication_` before assigning object to property) of your action just before calling `execute()`.

Now inside the `execute()` method you can use `getView()` as you want (3), in this case for hiding a property. You can see all View possibilities in the JavaDoc of `org.openxava.view.View`.

With...

```
<set property="property" value="remarks" />
<set property="hide" value="true" />

```

you can set constant values to the properties of your action.

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/default-controllers.xml`:

```
<controller name="Typical">
  <extends controller="Print"/>
  <extends controller="CRUD"/>
</controller>

```

When you assign the controller `Typical` to a module this module will have available all actions of `Print` controller (to generate PDF reports and export to Excel) and `CRUD` controller (to Create, Read, Update and Delete)

You can use inheritance to refine the way a standard controller works, e. g. like this:


```

<controller name="Family">
  <extends controller="Typical"/>
  <action name="new" image="images/new.gif"
    class="org.openxava.test.actions.CreateNewFamilyAction">
  <use-object name="xava_view"/>
  </action>
</controller>

```

As you see the name of your action `new` matches with an action in *Typical* controller (in reality in *CRUD* controller from which extends *Typical*). In this case the original action is ignored and your action is used. Thus you can put your own logic to execute when a final user clicks the 'new' link.

List mode actions

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

An example can be:

```

<action name="deleteSelected" mode="list"           <!-- 1 -->
  confirm="true"                                   <!-- 2 -->
  class="org.openxava.actions.DeleteSelectedAction">
</action>

```

You set `mode="list"` in order to show it only in list mode (1). Since this action deletes records you require that the user must confirm explicitly before the action is executed (2). It's not needed to include a `<use-object/>` for `xava_tab` (new in v2.1.4).

The action source code:

```

package org.openxava.actions;

import java.util.*;

import org.openxava.model.*;
import org.openxava.validators.*;

/**
 * @author Javier Paniza
 */

public class DeleteSelectedAction extends TabBaseAction implements IModelAction { // 1
  private String model;

  public void execute() throws Exception {
    int [] selectedOnes = getTab().getSelected(); // 2
  }
}

```

```

    if (selected0nes != null) {
        for (int i = 0; i < selected0nes.length; i++) {
            Map key = (Map)
                getTab().getTableModel().getObjectAt(selected0nes[i]);
            try {
                MapFacade.remove(model, key);           // (3)
            }
            catch (ValidationException ex) {
                addError("no_delete_row", new Integer(i), key);// (4)
                addErrors(ex.getErrors());
            }
            catch (Exception ex) {
                addError("no_delete_row", new Integer(i), key);
            }
        }
        getTab().deselectAll();                          // 5
        resetDescriptionsCache();                         // 6
    }

    public void setModel(String modelName) {              // 7
        this.model = modelName;
    }
}

```

This action is a standard action of OpenXava, but it allows you to see the things that you can do within an action in list mode. You can observe (1) how the action extends from *TabBaseAction* and implements *IModelAction*. Since it extends from *TabBaseAction* (*new in v2.1.4*) it has a group of utilities and you don't need to implement all methods of *IAction*; and as it implements *IModelAction* this action has a method called *setModel()* (7) that receives the model name (the name of OpenXava component) before executing it.

You can access to the *Tab* using the *getTab()* method (2); this method is implemented in *TabBaseAction* and it allows you to access to the *xava_tab* session object. By means of *getTab()* you are allowed to manage the list of displayed objects. For example, with *getTab().getSelected()* (2) 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* (3). *MapFacade* allows you to access the data model using Java maps (*java.util.Map*). This is useful, if you get data from *Tab* or *View* in *Map* format and you want to update the model (and therefore the database) with it, or vice versa. All generic classes of OpenXava use *MapFacade* to manage

the model and you also can use *MapFacade*. As general design tip: working with maps is useful in the case of generic logic, but if you need to program specific things it is better to use directly the object of model layer. For more details have a look at the JavaDoc of *org.openxava.model.MapFacade*.

You see here how to display messages to the user with *addError()*. The *addError()* method receives the id of an entry in your i18n files and the argument to send to the message. The added messages are displayed to the user as errors. If you want to add warning or informative messages you can use *addMessage()* whose behavior is like *addError()*. The i18n files that hold errors and messages must be called *MyProject-messages.properties* and the language suffix (*_en*, *_ca*, *_es*, *_it*, etc). You can see the examples in *OpenXavaTest/xava/i18n*. All not caught exceptions produces a generic error messages, except if the not caught exception is of the type *ValidationException*. In this case the message exception is displayed.

The *resetDescriptionsCache()* (6) method deletes all cache entries used by OpenXava to display descriptions list (combos). It's a good idea to call it whenever data is updated.

You can see more possibilities in *org.openxava.actions.BaseAction* and *org.openxava.actions.TabBaseAction* JavaDoc.

Since v2.1.4 this type of actions can also be used as *@ListAction* (*<list-action/>* of a *<collection-view/>*).

Overwriting default search

When a module is shown in list mode and the user clicks to display a detail, then OpenXava searches the corresponding object and displays it in detail. Now, if in detail mode the user fills the key fields and clicks on search (the binoculars), it also does the same. And when the user navigates by the records clicking the next or previous buttons then it does the same search. How can you customize this search? Let's see that:

You only need to define the module in *xava/application.xml* this way:

```
<module name="Deliveries">
  <env-var name="XAVA_SEARCH_ACTION" value="Deliveries.search"i>
  <model name="Delivery"/>
  <controller name="Typical"/>
  <controller name="Remarks"/>
  <controller name="Deliveries"/>
</module>
```

You see how it is necessary to define an environment variable named `XAVA_SEARCH_ACTION` that contains the action that you want to use for searching. This action is defined in `xava/controllers.xml`:

```
<controller name="Deliveries">
  <action name="search" mode="detail"
    by-default="if-possible" hidden="true"
    class="org.openxava.test.actions.SearchDeliveryAction"
    keystroke="F8">
    <use-object name="xava_view"/>
  </action>
  ...
</controller>
```

And its code:

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

import java.util.*;

import org.openxava.actions.*;
import org.openxava.util.*;

/**
 * @author Javier Paniza
 */

public class SearchDeliveryAction extends 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 the database (or through EJB2, EJB3 JPA or Hibernate) and fill the view. Most times it is better that it extends

SearchByViewKeyAction (1) and within *execute()* write a *super.execute()* (2).

OpenXava comes with 3 predefined search actions:

- **CRUD.searchByViewKey**: This is the default one. It does a search using the key values in the view, it executes no event.
- **CRUD.searchExecutingOnChange**: Works as *searchByViewKey* but it executes the *@OnChange/on-change* actions after search data.
- **CRUD.searchReadOnly**: Works as *searchByViewKey* but it set the detail view to not editable state on searching. Useful for creating read only modules.

If you want that the *@OnChange/on-change* actions will be executed on search then you must define your module this way:

```
<module name="Products3ChangeActionsOnSearch">
  <env-var name="XAVA_SEARCH_ACTION" value="CRUD.searchExecutingOnChange"/>
  <model name="Product3"/>
  <view name="WithDescriptionsList"/>
  <controller name="Typical"/>
  <controller name="Products3"/>
  <mode-controller name="Void"/>
</module>
```

As you see, simply by setting the value of the *XAVA_SEARCH_ACTION* environment variable.

Initialize a module with an action

By setting *on-create="true"* when you define an action, you configure that this action will be executed automatically when the module is executed for the first time. This is a chance to initialize the module. Let's see an example. In your *controllers.xml* you write:

```
<controller name="Invoices2002">
  <action name="init" on-init="true" hidden="true"
    class="org.openxava.test.actions.InitDefaultYearTo2002Action">
    <use-object name="xavatest_defaultYear"/>
    <use-object name="xava_tab"/>
  </action>
  ...
</controller>
```

And in your action:

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

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

```

/**
 * @author Javier Paniza
 */

public class InitDefaultYearTo2002Action extends BaseAction {

    private int defaultYear;
    private Tab tab;

    public void execute() throws Exception {
        setDefaultYear(2002);           // 1
        tab.setTitleVisible(true);      // 2
        tab.setTitleArgument(new Integer(2002)); // 3
    }

    public int getDefaultYear() {
        return defaultYear;
    }

    public void setDefaultYear(int i) {
        defaultYear = i;
    }

    public Tab getTab() {
        return tab;
    }

    public void setTab(Tab tab) {
        this.tab = tab;
    }

}

```

In this action you set the default year to 2002 (1), you make the title list visible (2) and you assign a value as an argument to that title (3). The list title is defined in the `i18n` files, usually it's used for reports, but you can show it in list mode too.

Calling another module

Sometimes it's convenient to call programmatically one module from another one. For example, imagine that you want to show a list of customers and when the user clicks on one customer, then a list of its invoices is displayed and the user can choose an invoice to edit. One way to obtain this effect is to have a module with only list mode and when the user clicks on a detail, the user is directed to an invoices module that shows only the invoices of

the chosen customer. Let's see it. First you need to define the module in *application.xml* this way:

```
<module name="InvoicesFromCustomers">
  <env-var name="XAVA_LIST_ACTION" value="Invoices.listOfCustomer"/> <!-- 1 -->
  <model name="Customer"/>
  <controller name="Print"/>
  <controller name="ListOnly"/>                                <!-- 2 -->
  <mode-controller name="Void"/>                               <!-- 3 -->
</module>
```

In this module only the list is shown (without detail part), for this you set the mode controller to *Void* (3) thus 'detail' and 'list' links are not displayed; and also you add a controller called *ListOnly* (2) in order to show the list mode, and only the list mode (if you only set the mode controller to *Void* the detail, and only the detail is displayed). Moreover you declare the variable *XAVA_LIST_ACTION* to define your custom action. When the user clicks the link in each row, then your own action will be executed. You must declare this action in *controllers.xml*:

```
<controller name="Invoices">
  <action name="listOfCustomer" hidden="true"
    class="org.openxava.test.actions.ListCustomerInvoicesAction">
    <use-object name="xava_tab"/>
  </action>
  ...
</controller>
```

And the action code:

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

import java.util.*;

import org.openxava.actions.*;
import org.openxava.controller.*;
import org.openxava.tab.*;

/**
 * @author Javier Paniza
 */
public class ListCustomerInvoicesAction extends BaseAction
    implements IChangeModuleAction,                                // 1
               IModuleContextAction {                             // 2

    private int row;                                             // 3
    private Tab tab;
    private ModuleContext context;

    public void execute() throws Exception {
        Map customerKey = (Map) tab.getTableModel().getObjectAt(row); // 4
    }
}
```

```

        int customerNumber = ((Integer) customerKey.get("number")).intValue();
        Tab invoiceTab = (Tab)
            context.get("OpenXavaTest", getNextModule(), "xava_tab"); // 5
        invoiceTab.setBaseCondition("${customer.number} = "+customerNumber); // 6
    }

    public int getRow() { // 3
        return row;
    }
    public void setRow(int row) { // 3
        this.row = row;
    }
}

    public Tab getTab() {
        return tab;
    }
    public void setTab(Tab tab) {
        this.tab = tab;
    }
}

    public String getNextModule() { // 7
        return "CustomerInvoices";
    }
}

    public void setContext(ModuleContext context) { // 8
        this.context = context;
    }
}

    public boolean hasReinitNextModule() { // 9
        return true;
    }
}
}

```

In order to change to another module the action implements *IChangeModuleAction* (1) thus forces the action to have a method called *getNextModule()* (7). This will indicate to which module OpenXava will switch after executing this action. The method *hasReinitNextModule()* (9) indicates, whether you want that the target module has re-initiated on changing to it.

On the other hand this action implements *IModuleContextAction* (2) too and therefore it receives an object of type *ModuleContext* with the method *setContext()* (8). *ModuleContext* allows you to access the session objects of others modules. This is useful to configure the target module before changing to it.

Another detail is that the action specified in `XAVA_LIST_ACTION` must have a property named `row` (3); before executing the action this property is filled with the row number that user has clicked.

If you keep in mind the above details it is easy to understand the action:

- Gets the key of the object associated to the clicked row (4), to do this it uses the tab of the current module.
- Accesses to the tab of the target module using context (5).
- Sets the base condition of the tab of target module using the key obtained from current tab.

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.

Go to a JSP page

The automatic view generator of OpenXava is good for most cases, but it can be required to display a JSP page hand-written by you. You can do this with an action like this:

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

import org.openxava.actions.*;

/**
 * @author Javier Paniza
 */

public class MySearchAction extends BaseAction implements INavigationAction { // 1

    public void execute() throws Exception {

    }

    public String[] getNextControllers() { // 2
        return new String [] { "MyReference" };
    }

    public String getCustomView() { // 3
        return "doYouWishSearch.jsp";
    }

    public void setKeyProperty(String s) {

    }

}
```

In order to go to a custom view (in this case a JSP page) your action has to implement *INavigationAction* (*ICustomViewAction* is enough). This way you can indicate with *getNextControllers()* (2) the next controllers to use and with *getCustomView()* (3) the JSP page to display (3).

Generating a custom report with JasperReports

OpenXava allows the final user to generate their own reports from the list model. The user can perform filtering, ordering, adding/removing fields, changing the positions of the fields and then generate a PDF report of the list.

But in all non-trivial business application you need to create programatically your own reports. You can do that easily by using JasperReports and then by integrating the reports into your OpenXava application with the action *JasperReportBaseAction*.

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

Then you can write your action to print the report in this way:

```

package org.openxava.test.actions;

import java.util.*;

import net.sf.jasperreports.engine.*;
import net.sf.jasperreports.engine.data.*;

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

/**
 * Report of products of the selected subfamily. <p>
 *
 * Uses JasperReports. <br>
 *
 * @author Javier Paniza
 */
public class FamilyProductsReportAction extends JasperReportBaseAction { // 1

    private ISubfamily2 subfamily;

    public Map getParameters() throws Exception { // 2
        Messages errors =
            MapFacade.validate("FilterBySubfamily", getView().getValues());
        if (errors.contains()) throw new ValidationException(errors); // 3
        Map parameters = new HashMap();
        parameters.put("family", getSubfamily().getFamily().getDescription());
        parameters.put("subfamily", getSubfamily().getDescription());
        return parameters;
    }

    protected JRDataSource getDataSource() throws Exception { // 4
        return new JRBeanCollectionDataSource(getSubfamily().getProductsValues());
    }

    protected String getJRXML() { // 5
        return "Products.jrxml"; // To read from classpath
        // return "/home/javi/Products.jrxml"; // To read from file system
    }

    private ISubfamily2 getSubfamily() throws Exception {
        if (subfamily == null) {
            int subfamilyNumber = getView().getValueInt("subfamily.number");
            // Using JPA, the usual with OX3
            subfamily = XPersistence.getManager().find(
                Subfamily2.class, new Integer(subfamilyNumber));
            // Using Hibernate, the usual with OX2, but still supported
        }
    }
}

```

```

    // subfamily = (ISubfamily2)
    //     XHibernate.getSession().get(
    //         Subfamily2.class, new Integer(subfamilyNumber));
    // Using EJB2, the usual with OX1, but still supported
    //subfamily =     Subfamily2Util.getHome().
    //     findByPrimaryKey(new Subfamily2Key(subfamilyNumber));
    }
    return subfamily;
}
}
}

```

Your action has to extend *JasperReportBaseAction* (1) and it has to overwrite the next three method:

- **getParameters()** (2): A *Map* with the parameters to send to the report, in this case we validate the input data (using *MapFacade.validate()*) before (3).
- **getDataSource()** (4): A *JRDataSource* with data to print. In this case it is a collection of JavaBeans obtained calling a method of the model object. If you use EJB CMP2 Entities be careful and do not loop over an EJB2 Entity 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 can be in the classpath. You may have a source code folder called reports in your project to hold these files. Other option is put this file in the file system (*new in v2.0.3*), this is achieved by specifying the full path of file, for example: */home/javai/Products.jrxml*.

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

You can find more examples of JasperReport actions in the *OpenXavaTest* project, as *InvoiceReportAction* for printing an Invoice.

Uploading and processing a file from client (multipart form)

This feature allows you to process in your OpenXava application a binary file (or several) provided by the client. This is implemented in a HTTP/HTML context using HTML multipart forms, although the OpenXava code is technologically neutral, hence your action will be portable to another environment with no recoding.

In order to upload a file the first step is creating an action to direct to a form where the user can choose his file. This action must implements *ILoadFileAction* in this way:

```
public class ChangeImageAction extends BaseAction implements ILoadFileAction { // 1
    ...
    public void execute() throws Exception { // 2
    }

    public String[] getNextControllers() { // 3
        return new String [] { "LoadImage" };
    }

    public String getCustomView() { // 4
        return "xava/editors/changeImage";
    }

    public boolean isLoadFile() { // 5
        return true;
    }

    ...
}
```

An *ILoadFileAction* (1) action is also an *INavigationAction* action that allows you to navigate to another controller (3) and to a custom view (4). The new controller (3) usually will have an action of type *IProcessLoadedFileAction*. The method *isLoadFile()* (5) returns *true* in case that you want to navigate to the form to upload the file, you can use the logic in *execute()* (2) to determine this value. The custom view is (4) a JSP with your own form to upload the file.

An example of a JSP for a custom view is:

```
<%@ include file="./imports.jsp"%>

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

<table>
<th align='left' class=<%=style.getLabel()%>>
<fmt:message key="enter_new_image"/>
</th>
<td>
<input name = "newImage" class=<%=style.getEditor()%> type="file" size='60' />
</td>
</table>
```

As you see, the HTML form is not specified, because the OpenXava module already has the form.

The last piece is the action for processing the uploaded files:

```

public class LoadImageAction extends BaseAction
    implements INavigationAction, IProcessLoadedFileAction {           // 1

    private List fileItems;
    private View view;
    private String newImageProperty;

    public void execute() throws Exception {
        Iterator i = getFileItems().iterator();                       // 2
        while (i.hasNext()) {
            FileItem fi = (FileItem)i.next();                         // 3
            String fileName = fi.getName();
            if (!Is.emptyString(fileName)) {
                getView().setValue(getNewImageProperty(), fi.get()); // 4
            }
        }
    }

    public String[] getNextControllers() {
        return DEFAULT_CONTROLLERS;
    }

    public String getCustomView() {
        return DEFAULT_VIEW;
    }

    public List getFileItems() {
        return fileItems;
    }

    public void setFileItems(List fileItems) {                       // 5
        this.fileItems = fileItems;
    }
    ...
}

```

The action implements *IProcessLoadedFileAction* (1), thus the action must have a method *setFileItem()* (5) to receive the list of uploaded files. This list can be processed in *execute()* (2). The elements of the collection are of type *org.apache.commons.fileupload.FileItem* (4) (from fileupload project of apache commons). Only calling to *get()* (4) in the file item you will access to the content of the uploaded file.

Override the default controllers (new in v2.0.3)

The controllers in *OpenXava/xava/default-controllers.xml* (before v2.0.3 it was *OpenXava/xava/controllers.xml*) are used by OpenXava to give to application a default behavior. Many times the easier way to override the default behavior of OpenXava is creating our own controllers and use them in our applications,

that is you can create a controller called *MyTypical*, and using it in your modules instead of *Typical* that comes with OpenXava.

Another option is override a default controller of OpenXava. In order to override a default controller you only need to create in your application a controller with the same name of default one. For example, if you want refine the collections behavior for your application then you have to create a *Collection* controller in your *xava/controllers.xml*, as following:

```
<controller name="Collection">
  <action name="new"
    class="org.openxava.actions.CreateNewElementInCollectionAction">
  </action>
  <action name="hideDetail"
    class="org.openxava.test.actions.MyHideDetailElementInCollection"> <!-- 1 -->
  </action>
  <action name="save"
    class="org.openxava.actions.SaveElementInCollectionAction">
    <use-object name="xava_view"/>
  </action>
  <action name="remove"
    class="org.openxava.actions.RemoveElementFromCollectionAction">
    <use-object name="xava_view"/>
  </action>
  <action name="edit"
    class="org.openxava.actions.EditElementInCollectionAction">
    <use-object name="xava_view"/>
  </action>
  <action name="view"
    class="org.openxava.actions.EditElementInCollectionAction">
    <use-object name="xava_view"/>
  </action>
</controller>
```

In this case we only override the behavior of *hideDetail* (1) action. But we must declare all actions of the original controller, because OpenXava rely on all these actions for working; we cannot remove or rename actions.

All action types

You have seen until now that the behavior of your actions depends on which interfaces they implement. Next the available interfaces for actions are enumerated:

- **IAction**: Basic interface to be implemented by all actions.
- **IChainAction**: Allows you to chain actions, that is when the execution of the action finishes, then the next action is executed immediately.

- **IChainActionWithArgv**: (*New in v2.2*) It's a refinement of *IChainAction*. It allows to send as arguments values for filling properties of the chained action before execute it.
- **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 the User Interface (*new in v2.0*).
- **IJDBCAction**: Allows to use JDBC in an action directly. It receives an *IConnectionProvider*. Works like an *IJBCCalculator* (see chapter 3).
- **ILoadFileAction**: Navigates to a view that allows the final user to load a file.
- **IModelAction**: An action that receives the model name.
- **IModuleContextAction**: Gets a *ModuleContext* in order to access the session objects of other modules.
- **INavigationAction**: Extends from *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 EJB2. Well used it can be a good substitute for a *SessionBean*.
- **IRequestAction**: Receives a servlet request. This type of actions links your application to the Servlet/JSP technology, hence it is better avoiding it. But sometimes a little bit of flexibility is needed.
- **IShowActionAction, IShowActionsAction**: Allows to show an action or a group of actions previously hidden in an *IHideAction(s)Action* (*new in v2.0*).

If you wish to learn more about actions the best thing you can do is to have a look at the JavaDoc API of the package *org.openxava.actions* and to try out the examples of *OpenXavaTest* project.

.....

.....

Chapter 8: Application

An application is the software that the final user can use. For now you have seen how to define the pieces that make up an application (mainly the components and the actions), now you will learn how to assemble these pieces in order to create applications.

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

The file syntax is:

```
<application
  name="name"           <!-- 1 -->
  label="label"        <!-- 2 -->
>
  <default-module ... /> ... <!-- 3 New in v2.2.2 -->
  <module ... /> ...      <!-- 4 -->
</application>
```

1. **name** (required): Name of the application.
2. **label** (optional): It's much better to use i18n files.
3. **default-module** (one, optional): *New in v2.2.2*. For defining the controllers for the default (automatically generated for each component) modules.
4. **module** (several, optional): Each module executable by final user.

In short, an application is a set of modules. Let's see how define a module:

```
<module
  name="name"           <!-- 1 -->
  folder="folder"      <!-- 2 -->
  label="label"        <!-- 3 -->
  description="description" <!-- 4 -->
>
  <env-var ... /> ...   <!-- 5 -->
  <model ... />        <!-- 6 -->
  <view ... />         <!-- 7 -->
  <web-view ... />    <!-- 8 -->
  <tab ... />         <!-- 9 -->
  <controller ... /> ... <!-- 10 -->
  <mode-controller ... /> <!-- 11 -->
  <doc ... />         <!-- 12 -->
</module>
```

1. **name** (required): Unique identifier of the module within this application.

2. **folder** (optional): Folder in which the module will reside. It's a tip to classify the modules. For the moment it's used to generate a folder structure for JetSpeed2 but its use can be amplified in future. You can use / or . to indicate subfolders (for example, "invoicing/reports" or "invoicing.reports").
3. **label** (optional): Short name to be shown to the user. It's much better to use i18n files.
4. **description** (optional): Long description to be shown to the user. It's much better to use i18n files.
5. **env-var** (several, optional): Allows you to define variables with a value that can be accessed by actions. Thus you can have actions configurable by module.
6. **model** (one, optional): Name of the component used in this module. If you leave it blank, then it is required to set the value to *web-view*.
7. **view** (one, optional): The view used to display the detail. If you leave it blank, then the default view will be used.
8. **web-view** (one, optional): Allows you to define a JSP page to be used as a view.
9. **tab** (one, optional): The tab used in list mode. If you do not specify it, then the default tab will be used.
10. **controller** (several, optional): Controllers with the available actions used initially.
11. **mode-controller** (one, optional): Allows to define the behavior to switch from detail to list mode and vice versa, as well as to define a module without detail and view (with no modes).
12. **doc** (one, optional): It's mutually exclusive with all other elements. It allows you to define a module that contains documentation only and no logic. Useful for generating informational portlets for your application.

Typical module example

Defining a simple module can be like this:

```
<application name="Management">
  <module name="Warehouse" folder="warehousing">
    <model name="Warehouse"/>
    <controller name="Typical"/>
    <controller name="Warehouse"/>
  </module>
```

```
...
</application>
```

In this case you have a module that allows the user to create, to delete, to update, to search, to generate PDF reports and to export to Excel the warehouses data (thanks to *Typical* controller) and also to execute actions only for warehouses (thank to *Warehouse* controller). In the case the system generates a module structure (as in JetSpeed2 case) this module will be in folder "warehousing".

In order to execute this module you need to open your browser and go to:

```
http://localhost:8080/Management/xava/
module.jsp?application=Management&module=Warehouse
```

Also a portlet is generated to allow you to deploy the module as a JSR-168 portlet in a Java portal.

Default modules (new in v2.2.2)

OpenXava assumes a default module for each component in the application, although the module is not explicitly defined in application.xml.

That is, if you define a component Invoice.xml, you can open your browser and go to:

```
http://localhost:8080/Management/xava/
module.jsp?application=Management&module=Invoice
```

Also a portlet is generated to allow you to deploy the module as a JSR-168 portlet in a Java portal.

And all this without defining it in application.xml.

The controller for these default modules will be *Typical*, but you can change this default value using the default-module element in application.xml, in this way:

```
<application name="Management">
  <default-module>
    <controller name="ManagementCRUD"/>
  </default-module>
</application>
```

In this case all the default modules of the *Management* application will have the controller *ManagementCRUD* assigned to them.

If you want that certain module does not use these default controllers, you have two options:

1. To define a controller in your *controllers.xml* with the same name of the component.
2. To define explicitly the module in *application.xml*, as it's explained above.

To sum up, if you define a component, named *Customer*, for example, then you have a module named *Customer*, and also a portlet. This module will be defined in one of this ways:

1. If you define a module named *Customer* in *application.xml* then this module will be the valid one, else..
2. If you define a controller named *Customer* in *controllers.xml* a module will generated using the controller *Customer* as controller and the component *Customer* as model, else..
3. 3.If you define a *default-module* element in your *application.xml* then a module will generated using the controllers in *default-module* and the component *Customer* as model, else ...
4. as fallback a module with *Typical* as controller and *Customer* as model will be assumed.

Only detail module

A module with only detail mode, without list, is defined this way:

```
<module name="InvoiceNoList">
  <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.

Only list module

A module with only list mode, without detail, is defined this way:

```
<module name="FamilyListOnly">
  <env-var name="XAVA_LIST_ACTION" value=""/>      <!-- 1 New in v2.0.4 -->
  <model name="Family"/>
  <controller name="Typical"/>
  <controller name="ListOnly"/>                  <!-- 2 -->
  <mode-controller name="Void"/>                  <!-- 3 -->
</module>
```

Void (3) mode controller is for removing the "detail – list" links. Furthermore on defining *ListOnly* (2) as controller the module changes to list mode on init, so this is an only list module. Finally, setting `XAVA_LIST_ACTION` to empty string (1) the detail link in each row is missing (*new in v2.0.4*).

Documentation module

A documentation module can only display a HTML document. It's easy to define:

```
<module name="Description">
  <doc url="doc/description" languages="en,es"/>
</module>
```

This module shows the document *web/doc/description_en.html* or *web/doc/description_es.html* depending on the browser language. If the browser language is not English nor Spanish then it assumes English (the first specified language). If you do not specify the language, then the document *web/doc/description.html* is shown.

This is useful for informational portlets. This type of module has no effect outside a portal environment.

Read only module

A read only module, that is only for consulting data, not for modifying, can be defined as following:

```
<module name="CustomerReadOnly">
  <env-var name="XAVA_SEARCH_ACTION" value="CRUD.searchReadOnly"/> <!-- 1 -->
  <model name="Customer">
    <controller name="Print"/> <!-- 2 -->
</module>
```

Using *CRUD.searchReadOnly* (1) the user cannot edit the data, and using only *Print* controller (2) (without *CRUD* or *Typical*) the actions for saving, deleting, etc are not available. This is a simply consulting module.

The syntax for *application.xml* is not difficult. You can see more examples in *OpenXavaTest/xava/application.xml*.

.....

.....

Chapter 9: Customizing

User Interface generated by OpenXava is good for most cases, but sometimes you may need customizing some part of the user interface (creating your own editors) or create your own handmade user interface (using custom JSP views) completely.

Editors

Editors configuration

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

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

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

```
<?xml version = "1.0" encoding = "ISO-8859-1"?>
<!DOCTYPE editors SYSTEM "dtds/editors.dtd">
<editors>
  <editor .../> ...
</editors>
```

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

```
<editor
  url="url"                <!-- 1 -->
  format="true|false"     <!-- 2 -->
  depends-stereotypes="stereotypes" <!-- 3 -->
  depends-properties="properties" <!-- 4 -->
  frame="true|false"     <!-- 5 -->
>
  <property ... /> ...    <!-- 6 -->
  <formatter ... />     <!-- 7 -->
```



```

<for-stereotype ... /> ...      <!-- 8 -->
<for-type ... /> ...           <!-- 8 -->
<for-model-property ... /> ... <!-- 8 -->
</editor>

```

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

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

```

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

```

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

```

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

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

```

A JSP editor receives a set of parameters and has access to attributes that allows to configure it in order to work suitably with OpenXava. First you can see how it gets *propertyKey* (1) that is used as HTML id. From this id you can access to *MetaProperty* (2) (that contains meta information of the property to edit). The *fvalue* (3) attribute contains the value already formatted and ready to be displayed. *Align* (4) and *editable* (5) are obtained too. Also you need to obtain a JavaScript (6) fragment to put in the HTML editor. Although creating an editor directly with JSP is easy it is not usual to do it. It's more common to configure existing JSPs. For example, in your *xava/editors.xml* you can write:

```

<editor url="textEditor.jsp">
  <formatter class="org.openxava.formatters.UpperCaseFormatter">

```

```
<for-type type="java.lang.String"/>
</editor>
```

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

```
package org.openxava.formatters;

import javax.servlet.http.*;

/**
 * @author Javier Paniza
 */

public class UpperCaseFormatter implements IFormatter { // 1

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

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

}
```

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

Multiple values editors

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

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

```
@Stereotype("REGIONS")
private String [] regions;
```

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

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

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

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

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

The last is to write your JSP editor:

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

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

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

```

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

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

```

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

Custom editors and stereotypes for displaying combos

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

Let's see this.

You define the properties like this in your entity:

```

@Stereotype("FAMILY")
private int familyNumber;

@Stereotype("SUBFAMILY")
private int subfamilyNumber;

```

And in your *editors.xml* put:

```

<editor url="descriptionsEditor.jsp"                                <!-- 10 -->
  <property name="model" value="Family"/>                          <!-- 1 -->
  <property name="keyProperty" value="number"/>                    <!-- 2 -->
  <property name="descriptionProperty" value="description"/>      <!-- 3 -->
  <property name="orderByKey" value="true"/>                       <!-- 4 -->
  <property name="readOnlyAsLabel" value="true"/>                 <!-- 5 -->
  <for-stereotype stereotype="FAMILY"/>                            <!-- 11 -->
</editor>

<!-- It is possible to specify dependencies from stereotypes or properties -->
<editor url="descriptionsEditor.jsp"                                <!-- 10 -->

```

```

    depends-stereotypes="FAMILY">                                <!-- 12 -->
<!--
<editor url="descriptionsEditor.jsp" depends-properties="familyNumber">    <!-- 13 -->
-->
    <property name="model" value="Subfamily"/>                    <!-- 1 -->
    <property name="keyProperty" value="number"/>                 <!-- 2 -->
    <property name="descriptionProperties" value="number, description"/> <!-- 3 -->
    <property name="condition" value="\${familyNumber} = ?"/>    <!-- 6 -->
    <property name="parameterValuesStereotypes" value="FAMILY"/> <!-- 7 -->
    <!--
    <property name="parameterValuesProperties" value="familyNumber"/> <!-- 8 -->
-->
    <property name="descriptionsFormatter"                        <!-- 9 -->
        value="org.openxava.test.formatters.FamilyDescriptionsFormatter"/>
    <for-stereotype stereotype="SUBFAMILY"/>                       <!-- 11 -->
</editor>

```

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

In order to do that you need to assign to stereotypes (FAMILY and SUBFAMILY in this case(11)) the *descriptionsEditor.jsp* (10) editor and you configure it by assigning values to its properties. Some properties that you can set in this editor are:

1. **model**: Model to obtain data from. It can be the name of an entity (e.g. *Invoice*) or the name of the model used in an embedded collection (*Invoice.InvoiceDetail*).
2. **keyProperty** or **keyProperties**: Key property or list of key properties; this is used to obtain the value to assign to the current property. It is not required that they are the key properties of the model, although this is the typical case.
3. **descriptionProperty** or **descriptionProperties**: Property or list of properties to show in combo.
4. **orderByKey**: If it has to be ordered by the key, by default it is ordered by description. You can also use order with an order clause in SQL style if you need it.
5. **readOnlyAsLabel**: When it is read only, then it is rendered as label. The default is *false*.
6. **condition**: Condition to limit the data to be displayed. Has SQL format, but you can use the property names with *\\${}*, even qualified properties are supported. You can put arguments with *?*. This last

case is when this property depends on other ones and only obtain data when these other properties change.

7. **parameterValuesStereotypes**: List of stereotypes from which properties depend. It's used to fill the condition arguments and has to match with *depends-stereotypes* attribute (12).
8. **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).
9. **descriptionsFormatter**: Formatter for the descriptions displayed in a combo. It must implement *IFormatter*.

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

Custom JSP view and OpenXava taglibs

Obviously the better way to create user interfaces is using the view annotations explained in chapter 4. But, in extreme cases perhaps you have to define your view using JSP. OpenXava allows you to do it. And in order to help you to do it, you can use some JSP taglibs provided by OpenXava. Let's see an example.

Example

First you have to define in your module that you want to use your own JSP, in *application.xml*:

```
<module name="SellersJSP" folder="invoicing.variations">
  <model name="Seller"/>
  <view name="ForCustomJSP"/>           <!-- 1 -->
  <web-view url="custom-jsp/seller.jsp"/> <!-- 2 -->
  <controller name="Typical"/>
</module>
```

If you use *web-view* (2) on defining your module, OpenXava uses your JSP to render the detail, instead of generating the view automatically. Optionally you can define an OpenXava view using *view* (1), this view is used to know the events to throw and the properties to populate, if not it is specified the default view of the entity is used; although it's advisable to create an explicit OpenXava view for your JSP custom view, in this way you can control the

events, the properties to populate, the focus order, etc explicitly. You can put your JSP inside *web/custom-jsp* (or other of your choice) folder of your project, and it can be as this one:

```
<%@ include file="../xava/imports.jsp"%>

<table>
<tr>
  <td>Number: </td>
  <td>
    <xava:editor property="number"/>
  </td>
</tr>
<tr>
  <td>Name: </td>
  <td>
    <xava:editor property="name"/>
  </td>
</tr>

<tr>
  <td>Level: </td>
  <td>
    <xava:editor property="level.id"/>
    <xava:editor property="level.description"/>
  </td>
</tr>
</table>
```

You are free to create your JSP file as you like, but it can be useful to use OpenXava taglibs, in this case, for example the `<xava:editor/>` taglib is used, this renders an editor suitable for the indicated property, furthermore add the needed javascript to throw the events. If you use `<xava:editor/>`, you can manage the displayed data using *xava_view* (of *org.openxava.view.View* type) object, therefore all standard OpenXava controllers (including *CRUD*) work.

You can notice that qualified properties are allowed (as *level.id* or *level.description*) (*new in v2.0.1*), furthermore when you fill *level.id*, *level.description* is populated with the corresponding value. Yes, all the behaviour of an OpenXava view is available inside your JSP if you use the OpenXava taglibs.

Let's see the OpenXava taglibs.

xava:editor

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


```
<xava:editor
  propertyName="propertyName"      <!-- 1 -->
  editable="true|false"           <!-- 2 New in v2.0.1 -->
  throwPropertyChanged="true|false" <!-- 3 New in v2.0.1 -->
/>
```

1. **property** (required): It's the property of the model associated with the current module
2. **editable** (optional): *New in v2.0.1*. Forces to this editor to be editable, otherwise the appropriate default value is assumed.
3. **throwPropertyChanged** (optional): *New in v2.0.1*. Forces to this editor to throws property changed event, otherwise the appropriate default value is assumed.

This tag generates the needed JavaScript in order to allow your view to work in the same way as an automatic one. The qualified properties (properties of references) are supported (*new in v2.0.1*).

xava:action, xava:link, xava:image, xava:button

The `<xava:action/>` tag allows you to render an action (a button or a image that the user can click).

```
<xava:action action="controller.action" argv="argv"/>
```

The action attribute indicates the action to execute, and the *argv* attribute (optional) allows you to put values to some properties of the action before execute it. One example:

```
<xava:action action="CRUD.save" argv="resetAfter=true"/>
```

When the user clicks on it, then it executes the action *CRUD.save*, before it puts true to the *resetAfter* property of the action.

The action is rendered as an image, if it has an image associated. Otherwise it is rendered as a button. If you want to determine the render style, then you can use directly the next taglibs: `<xava:button/>`, `<xava:image/>` or `<xava:link/>` similars to `<xava:action/>`.

You can specify an empty string as action (*new in v2.2.1*), as following:

```
<xava:action action=""/>
```

In this case the tag has no effect and no error is produced. This feature may be useful if you fill the name of the action dynamically (that is `action="<%=mycode()%>"`), and the value can be empty in some cases.

xava:message (new in v2.0.3)

The `<xava:message/>` tag allows to show in HTML a message from the `i18n` resource files of OpenXava.

```
<xava:message key="message_key" param="messageParam" intParam="messageParam"/>
```

The message is searched first in the message resource files of your project (`YourProject/i18n/YourProject-messages.properties`) and if it is not found there then it's searched in the default OpenXava messages (`OpenXava/i18n/Messages.properties`).

The attributes `param` and `intParam` are optional. The attribute `intParam` is used when the value to send as parameter is of `int` type. If you use Java 5 you can use always `param` because `int` is automatically converted by autoboxing.

This tag only generates the message text, without any formatting HTML tags. An example:

```
<xava:message key="list_count" intParam="<%=totalSize%>"/>
```

xava:descriptionsList (new in v2.0.3)

The `<xava:descriptionsList/>` tag allows you to render a description list (a HTML combo) for your reference, in the same way that OpenXava does it when it generates the user interface automatically.

```
<xava:descriptionsList
  reference="referenceName" <!-- 1 -->
/>
```

1. **reference** (required): It's a reference of the model associated with the current module

This tag generates the needed JavaScript in order to allow your view to work in the same way as an automatic one.

An example:

```
<tr>
  <td>Level: </td>
  <td>
    <xava:descriptionsList reference="level"/>
  </td>
</tr>
```

In this case `level` is a reference of the current model (for example `Seller`). A combo is shown with all available levels.