



VERSION 3.0

CLASSIC XML COMPONENTS

Contents

1 Introduction.....	4
2 Model.....	5
2.1 Java Implementation.....	5
2.2 Business Component.....	5
2.3 Entity and aggregates.....	6
2.4 Entity.....	6
2.5 Bean.....	8
2.6 EJB (2).....	10
2.7 Implements (3).....	12
2.8 Property (4).....	14
2.8.1 Stereotype.....	15
2.8.2 IMAGES_GALLERY stereotype (new in v2.0).....	17
2.8.3 Concurrency and version property (new in v2.2.3).....	18
2.8.4 Valid values.....	18
2.8.5 Calculator.....	19
2.8.6 Default value calculator.....	24
2.8.7 Validator.....	25
2.8.8 Default validator (new in v2.0.3).....	27
2.9 Reference (5).....	28
2.9.1 Default value calculator in references.....	29
2.10 Collection (6).....	30
2.11 Method (7).....	35
2.12 Finder (8).....	38
2.13 Postcreate calculator (9).....	39
2.14 Postmodify calculator (11).....	41
2.15 Postload and preremove calculator (10, 12).....	42
2.16 Validator (13).....	42
2.17 Remove validator (14).....	44
2.18 Aggregate.....	46
2.18.1 Reference to aggregate.....	46
2.18.2 Collection of aggregates.....	47
3 View.....	50
3.1 Layout.....	51
3.1.1 Groups.....	52
3.1.2 Sections.....	55
3.1.3 Layout philosophy.....	57
3.2 Property view.....	57
3.2.1 Label format.....	58
3.2.2 Value change event.....	58
3.2.3 Actions of property.....	59
3.2.4 Choosing an editor (new in v2.1.3).....	61
3.3 Reference view.....	61
3.3.1 Choose view.....	64
3.3.2 Customizing frame.....	65
3.3.3 Custom search action.....	66

3.3.4 Custom creation action.....	67
3.3.5 Custom modification action (new in v2.0.4).....	67
3.3.6 Descriptions list (combos).....	68
3.3.7 Reference search on change event (new in v2.2.5).....	70
3.4 Collection view.....	71
3.4.1 Custom edit/view action.....	74
3.4.2 Custom list actions.....	75
3.4.3 Default list actions (new in v2.1.4).....	77
3.4.4 Custom detail actions.....	77
3.4.5 Refining collection view default behavior (new in v2.0.2).....	79
3.5 View property.....	80
3.6 View actions (new in v2.0.3).....	81
3.7 Transient component: Only for creating views (new in v2.1.3).....	82
4 Tabular data.....	84
4.1 Initial properties and emphasize rows.....	85
4.2 Filters and base condition.....	85
4.3 Pure SQL select.....	88
4.4 Default order.....	89
5 Object/relational mapping.....	90
5.1 Entity mapping.....	90
5.2 Property mapping.....	91
5.3 Reference mapping.....	93
5.4 Multiple property mapping.....	95
5.5 Reference to aggregate mapping.....	97
5.6 Aggregate used in collection mapping.....	98
5.7 Converters by default.....	100
5.8 Default mapping (new in v2.1.3).....	102
5.9 Object/relational philosophy.....	102
6 Aspects.....	103
6.1 Introduction to AOP.....	103
6.2 Aspects definition.....	103
6.3 AccessTracking: A practical application of aspects.....	104
6.3.1 The aspect definition.....	105
6.3.2 Setup AccessTracking.....	106
7 Miscellaneous.....	108
7.1 Many-to-many relationships.....	108

1 Introduction

OpenXava allows you to create Java Enterprise applications using POJOs and Java 5 annotations. But in its first incarnations (v1.0 and v2.0) OpenXava used business components defined using XML and it generated the code for the application from these XMLs.

These XML components are still supported. That is, if you have an OpenXava application developed with v1.x or v2.x you can update to the last OpenXava version. OpenXava supports XML components, and generation for EJB2, POJOs + Hibernate and even Java 1.4.

This guide is a complete reference to the XML syntax for OpenXava business component.

2 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 XML and a few of Java. OpenXava generates a complete Java implementation of your model from your definition.

2.1 Java Implementation

Currently OpenXava generates code for the next 4 alternatives:

1. Plain Java Classes (the so-called POJOs) for the model using Hibernate for persistence.
2. Plain Java Classes for the model using EJB3 JPA (**J**ava **P**ersistence **A**PI) for persistence (*new in v2.1*).
3. Classic EJB2 EntityBeans for the model and persistence.
4. POJOs + Hibernate inside an EJB2 container.

The option 2 is the default one (*new in v3.0*) and the best for the most cases. The option 1 is also good, specially if you need to use Java 1.4. The option 3 is for supporting all the OpenXava applications written using EJB2 (EJB2 was the only option in OpenXava 1.x). The option 4 can be useful in some circumstances. You can see how to configure this in *OpenXavaTest/properties/xava.properties*.

2.2 Business Component

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

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

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

<component name="ComponentName">

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

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

```

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

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

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

</component>

```

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

2.3 Entity and aggregates

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

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

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

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

2.4 Entity

The syntax of entity is:

```

<entity>
  <bean ... />
  (1)

```

```

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

```

- (1) `bean` (one, optional): Allows you to use an already existing JavaBean (a simple Java class, the so-called POJO). This applies if you use JPA or Hibernate as persistence engine. In this case the code generation for POJO and Hibernate mapping of this component will not be produced.
- (1) `ejb` (one, optional): Allows you to use an already existing EJB. This only applies if you use EJB CMP2 as persistence engine. In this case code generation for EJB code of this component will not be produced. It not apply to EJB3.
- (2) `implements` (several, optional): The generated code will implement this interface.
- (3) `property` (several, optional): The properties represent Java properties (with its *setters* and *getters*) in the generated code.
- (4) `reference` (several, optional): References to other models, you can reference to the entity of another component or an aggregate of itself.
- (5) `collection` (several, optional): Collection of references. In the generated code it is a property that returns a `java.util.Collection`.
- (6) `method` (several, optional): Creates a method in the generated code, in this case the method logic is in a calculator (`ICalculator`).
- (7) `finder` (several, optional): Used to generate finder methods. Finder methods are static method located in the POJO class. In the case of EJB2 generation EJB2 *finders* are generated.
- (8) `postcreate-calculator` (several, optional): Logic to execute after making an object persistent. In Hibernate in a `PreInsertEvent`, in EJB2 in the `ejbPostCreate` method.
- (9) `postload-calculator` (several, optional): Logic to execute just after load the state of an object from persistent storage. In Hibernate in a `PostLoadEvent`, in EJB2 in the `ejbLoad` method.
- (10) `postmodify-calculator` (several, optional): Logic to execute after modifying a persistent object and before storing its state in persistent storage. In Hibernate in a `PreUpdateEvent`, in EJB2 in the `ejbStore` method.
- (11) `preremove-calculator` (several, optional): Logic to execute just before removing a persistent from persistent storage. In Hibernate in `PreDeleteEvent`, in EJB2 in the `ejbRemove` method.

- (12)`validator` (several, optional): Executes a validation at model level. This validator can receive the value of various model properties. To validate a single property it is better to use a property level validator.
- (13)`remove-validator` (several, optional): It's executed before removal, and can deny the object removing.

2.5 Bean

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

For example:

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

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

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

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

import java.io.*;

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

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

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

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



```

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

```

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

Additionally you have to define the mapping using Hibernate:

```

<?xml version="1.0"?>

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

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

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

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

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

    </class>

</hibernate-mapping>

```

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

```

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

```

```
...
</session-factory>
...
```

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

2.6 EJB (2)

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

For example:

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

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

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

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

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

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

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

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

```

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

    private UUIDCalculator oidCalculator = new UUIDCalculator();

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

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

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

```

```

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

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

```

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

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

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

2.7 Implements (3)

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

```

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

```

And you can write your Java interface in this way:

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

import java.rmi.*;

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

    String getName() throws RemoteException;

}
```

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

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

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

    ...

}
```

In the POJO generated code you can see:

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

    ...

}
```

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

```
public interface CustomerRemote extends

    org.openxava.ejbx.EJBReplicable,

    org.openxava.test.model.ICustomer
```

and the EJB bean class is affected too

```
abstract public class CustomerBean extends EJBReplicableBase

    implements

        org.openxava.test.model.ICustomer,

        EntityBean
```

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

As you can see OpenXava generates an interface for each component. It's good that in your code you use these interfaces instead of POJO classes or EJB2 remote interfaces. All code made in this way can be used with POJO and EJB2 version on same time, or allows you to migrate from a EJB2 to a POJO version with little effort. Although, if you are using POJOs exclusively you may use the

POJOs classes directly and ignore the interfaces, as you wish.

2.8 Property (4)

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

The syntax to define a property is:

```
<property
  name="propertyName"           (1)
  label="label"                 (2)
  type="type"                   (3)
  stereotype="STEREOTYPE"      (4)
  size="size"                   (5)
  scale="scale"                 (6)   new in v2.0.4
  required="true|false"        (7)
  key="true|false"             (8)
  hidden="true|false"          (9)
  search-key="true|false"      (10)  new in v2.2.4
  version="true|false"         (11)  new in v2.2.3
>
<valid-values .../>           (12)
<calculator .../>           (13)
<default-value-calculator .../> (14)
<validator .../> .....      (15)
</property>
```

- (1) **name** (required): The property name in Java, therefore it must follow the Java convention for property names, like starting with lower-case. Using underline (_) is not advisable.
- (2) **label** (optional): Label showed to the final user. Is **much better** use the *il8n* files.
- (3) **type** (optional): It matches with a Java type. All types valid for a Java property are valid here, this include classes defined by you. You only need to provide a converter to allow saving in database and a editor to render as HTML; thus that things like `java.sql.Connection` or so can be a little complicated to manage as a property, but not impossible. It's optional, but only if you have specified `<bean/>` or `<ejb/>` or this property has a stereotype with a associated type.
- (4) **stereotype**(optional): Allows to specify an special behavior for some properties.
- (5) **size** (optional): Length in characters of property. Useful to generate user interfaces. If you do not specify the size, then a default value is assumed. This default value is associated to the stereotype or type and is obtained from *default-size.xml*.
- (6) **scale** (optional): (*new in v2.0.4*) 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*.

- (7)`required` (optional): Indicates if this property is required. By default this is `true` for key properties hidden (*new in v2.1.3*) or without default value calculator on create and `false` in all other cases. On saving OpenXava verifies if the required properties are present. If this is not the case, then saving is not done and a validation error list is returned. The logic to determine if a property is present or not can be configured by creating a file called `validators.xml` in your project. You can see the syntax in `OpenXava/xava/validators.xml`.
- (8)`key` (optional): Indicates that this property is part of the key. At least one property (or reference) must be key. The combination of key properties (and key references) must be mapped to a group of database columns that do not have duplicate values, typically the primary key.
- (9)`hidden` (optional): A hidden property has a meaning for the developer but not for the user. The hidden properties are excluded when the automatic user interface is generated. However at Java code level they are present and fully functional. Even if you put it explicitly into a view the property will be shown in the user interface.
- (10)`search-key` (optional): (*new in v2.2.4*) 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 key (`key="true"`) properties for searching by default, and if the key (`key="true"`) properties are hidden then it uses the first property in the view. With `search-key` you can choose explicitly the properties for searching.
- (11)`version` (optional): (*new in v2.2.3*) A version property is used for optimistic concurrency control. If you want control concurrency you only need to have a property marked as `version="true"` in your component. Only a single version property should be used per component. The following types are supported for version properties: `int`, `Integer`, `short`, `Short`, `long`, `Long`, `Timestamp`. The version properties are considered hidden.
- (12)`valid-values` (one, optional): To indicate that this property only can have a limited set of valid values.
- (13)`calculator` (one, optional): Implements the logic for a calculated property. A calculated property only has `getter` and is not stored in database.
- (14)`default-value-calculator` (one, optional): Implements the logic to calculate the default (initial) value for this property. A property with `default-value-calculator` has `setter` and it is persistent.
- (15)`validator` (several, optional): Implements the validation logic to execute on this property before modifying or creating the object that contains it.

2.8.1 Stereotype

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

OpenXava comes with these generic stereotypes:

- `DINERO`, `MONEY`
- `FOTO`, `PHOTO`, `IMAGEN`, `IMAGE`

- TEXTO_GRANDE, MEMO, TEXT_AREA
- ETIQUETA, LABEL
- ETIQUETA_NEGRITA, BOLD_LABEL
- HORA, TIME
- FECHAHORA, DATETIME
- GALERIA_IMAGENES, IMAGES_GALLERY (setup instructions in 3.8.2) *new in v2.0*
- RELLENADO_CON_CEROS, ZEROS_FILLED *new in v2.0.2*
- TEXTO_HTML, HTML_TEXT (text with editable format) *new in v2.0.3*
- IMAGE_LABEL, ETIQUETA_IMAGEN (image depending on property content) *new in v2.1.5*
- EMAIL *new in 2.2.3*
- TELEFONO, TELEPHONE *new in 2.2.3*
- WEBURL *new in 2.2.3*
- IP *new in 2.2.4*
- ISBN *new in 2.2.4*
- TARJETA_CREDITO, CREDIT_CARD *new in 2.2.4*
- LISTA_EMAIL, EMAIL_LIST *new in 2.2.4*

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

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

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

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

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

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

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

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

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

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

```
<required-validator>
```



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

Now everything is ready to define properties of stereotype PERSON_NAME:

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

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

2.8.2 IMAGES_GALLERY stereotype (new in v2.0)

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

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

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

And everything is done.

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

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

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

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

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

The name of the table is arbitrary. You need to specify the table name and schema (new in v2.2.4) name in your configuration file (a .properties file in the root of your OpenXava project). In this way:

```
images.schema=MYSHEMA
images.table=IMAGES
```

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

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

```
...
</session-factory>
</hibernate-configuration>
```

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

2.8.3 Concurrency and version property (*new in v2.2.3*)

Concurrency is the ability of the application to allow several users to save data at same time without losing data. OpenXava uses an optimistic concurrency schema. 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 OpenXava component you only need to declare a property using `version="true"`, in this way:

```
<property name="version" type="int" version="true"/>
```

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

2.8.4 Valid values

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

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

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

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

At user interface level the current implementation uses a combo. The label for each value is obtained from the `i18n` files.

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

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

2.8.5 Calculator

A calculator implements the logic to execute when the *getter* method of a calculated property is called. The calculated properties are read only (only have *getter*) and not persistent (they do not match with any column of database table).

A calculated property is defined in this way:

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

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

Seeing the calculator code may be instructive:

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

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

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

    private BigDecimal euros;

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

    public BigDecimal getEuros() {
        return euros;
    }

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

```
}
```

You can notice two things, first (1) a calculator must implement `org.openxava.calculators.ICalculator`, and (2) the method `calculate()` executes the logic to generate the value returned by the property.

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

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

And `result` will hold 332.772.

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

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

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

The `from` attribute supports qualified properties, as following:

```
<aggregate name="Address">
  <property name="street" type="String" size="30" required="true"/>
  <property name="zipCode" type="int" size="5" required="true"/>
  <property name="city" type="String" size="20" required="true"/>
  <reference name="state" required="true"/>
  <property name="asString" type="String">
    <calculator class="org.openxava.calculators.ConcatCalculator">
      <set property="string1" from="street"/>
      <set property="int2" from="zipCode"/>
      <set property="string3" from="city"/>
      <set property="string4" from="state.name"/>          (1)
      <set property="int5" from="customer.number"/>      (2)
    </calculator>
  </property>
</aggregate>
```

The property `string4` (1) of the calculator is filled using the value of `name` of the `state` (that is a reference), this a qualified property (`reference.property`). In the case of `int5` (2) you can see that `customer` reference is not declared in `Address`, because it is referenced from the `Customer` entity, therefore `Address` has an implicit reference to its container model (its parent) that you can

use in `from` attribute (*new in v2.0.4*). That is, the `int5` property is filled with the number of the customer which has this address.

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

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

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

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

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

And the calculator:

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

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

import javax.rmi.*;

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

/**
 * @author Javier Paniza
 */

public class AmountsSumCalculator implements IModelCalculator { // (1)

    private IInvoice invoice;

    public Object calculate() throws Exception {
        Iterator itDetails = invoice.getDetails().iterator();
        BigDecimal result = new BigDecimal(0);
    }
}
```

```

        while (itDetails.hasNext()) {
            IInvoiceDetail detail = (IInvoiceDetail) itDetails.next();
            result = result.add(detail.getAmount());
        }
        return result;
    }

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

```

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

Within the code generated by OpenXava you can find an interface for each business concept that is implemented by the POJO class, the EJB2 remote interface and the EJB2 Bean class. That is for Invoice you have a `IInvoice` interface implemented by `Invoice` (POJO class), `InvoiceRemote` (EJB2 remote interface) and `InvoiceBean` (EJB2 bean class), this last two only if you generate EJB2 code. In the calculator of type `IModelCalculator` it is advisable to cast to this interface, because in this cases the same calculator works with POJOs, EJB2 remote interface and EJB2 bean class. If you are developing with a POJO only version (maybe the normal case) you can cast directly to the POJO class, in this case `Invoice`.

This calculator type is less reusable than that which receives simple properties, but sometimes are useful. Why is it less reusable? For example, if you use `IInvoice` to calculate a discount, this calculator only could be applied to invoices, but if you uses a calculator that receives `amount` and `discountPercentage` as simple properties this last calculator could be applied to invoices, deliveries, orders, etc.

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

```

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

```

And the calculator class:

```

package org.openxava.test.calculators;

import java.sql.*;

```

```

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

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

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

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

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

            ps.setInt(1, getYear());
            ps.setInt(2, getNumber());
            ResultSet rs = ps.executeQuery();
            rs.next();
            Integer result = new Integer(rs.getInt(1));
            ps.close();
            return result;
        }
        finally {
            con.close();
        }
    }

    public int getYear() {
        return year;
    }

    public int getNumber() {
        return number;
    }
}

```

```

    }

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

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

```

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

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

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

2.8.6 Default value calculator

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

```

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

```

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

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

```

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

```


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

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

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

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

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

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

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

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

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

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

2.8.7 Validator

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

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

The technique to configure the validator (with `<set/>`) is exactly the same than in calculators. With the attribute `only-on-create="true"` you can define that the validation will be executed only

when the object is created, and not when it is modified.

The validator code is:

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

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

/**
 * @author Javier Paniza
 */

public class ExcludeStringValidator implements IPropertyValidator { // (1)

    private String string;

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

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

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

}
```

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

(2)`Messages errors`: A object of type `Messages` that represents a set of messages (like a smart collection) and where you can add the validation errors that you find.

(3)Object value: The value to validate.

(4)String objectName: Object name of the container of the property to validate. Useful to use in error messages.

(5)String propertyName: Name of the property to validate. Useful to use in error messages.

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

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

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

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

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

2.8.8 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:

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

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="true"`). 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*.

2.9 Reference (5)

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

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

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

The syntax of reference is:

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

- (1)`name` (optional, required if `model` is not specified): The name of reference in Java, hence must follow the rules to name members in Java, including start by lower-case. If you do not specify `name` the model name with the first letter in lower-case is assumed. Using underline (`_`) is not advisable.
- (2)`label` (optional): Label shown to the final user. It's **much better** use *i18n*.
- (3)`model` (optional, required if `name` is not specified): The model name to reference. It can be the name of another component, in which case it is a reference to entity, or the name of a aggregate of the current component. If you do not specify `model` the reference name with the first letter in upper-case is assumed.
- (4)`required` (optional): Indicates if the reference is required. When saving OpenXava verifies if the required references are present, if not the saving is aborted and a list of validation errors is returned.
- (5)`key` (optional): Indicates if the reference is part of the key. The combination of key properties and reference properties should map to a group of database columns with unique values, typically the primary key.
- (6)`role` (optional): Used only in references within collections. See below.
- (7)`default-value-calculator` (one, optional): Implements the logic for calculating the initial value of the reference. This calculator must return the key value, that can be a simple value (only if the key of referenced object is simple) or key object (a special object that wraps the key and is generated by OpenXava).

A little example of references use:

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

(1) A reference to an aggregate called `Address`, the reference name will be `address`.

(2) A reference to the entity of `Seller` component. The model is deduced from name.

(3) A reference called `alternateSeller` to the entity of component `Seller`.

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

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

2.9.1 Default value calculator in references

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

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

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

The `calculate()` method is:

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

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

In the case of composed key:

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

And the calculator code:

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

```

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

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

    public Object calculate() throws Exception {
        Warehouse key = new Warehouse();
        key.setNumber(4);
        key.setZoneNumber(4);
        return key; // This works with POJO and EJB2
        // return new WarehouseKey(new Integer(4), 4); // This only work with EJB2
    }
}

```

Returns an object of type `Warehouse`, (or `WarehouseKey` if you use only EJB2).

2.10 Collection (6)

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

Here syntax for collection:

```

<collection
  name="name" (1)
  label="label" (2)
  minimum="N" (3)
  maximum="N" (4) new v2.0.3
>
  <reference ... /> (5)
  <condition ... /> (6)
  <order ... /> (7)
  <calculator ... /> (8)
  <postremove-calculator ... /> (9)
</collection>

```

(1)name (required): The collection name in Java, therefore it must follow the rules for name members in Java, including starting with lower-case. Using underline (`_`) is not advisable.

(2)label (optional): Label shown to final user. Is **much better** to use *i18n* files.

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

- (4)`maximum` (optional): (*new v2.0.3*) Maximum number of expected elements.
- (5)`reference` (required): With the syntax you can see in the previous point.
- (6)`condition` (optional): Restricts the elements that appear in the collection.
- (7)`order` (optional): The elements in collections will be in the indicated order.
- (8)`calculator` (optional): Allows you to define your own logic to generate the collection. If you use this, then you cannot use neither `condition` nor `order`.
- (9)`postremove-calculator` (optional): Execute your custom logic just after an element is removed from collection.

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

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

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

Now you can write a code like this:

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

To do something with all deliveries associated to an invoice.

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

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

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

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

```

package org.openxava.test.calculators;

import java.rmi.*;

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

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

    private IInvoice invoice;

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

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

}

```

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

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

```

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

```

If you have this collection within `Carrier`, you can obtain with this collection all carriers of the

same warehouse but not himself, that is the list of his fellow workers. As you see you can use `this` in the condition in order to reference the value of a property of current object.

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

```
<!--
The same that 'fellowCarriers' but implemented with a calculator
-->
<collection name="fellowCarriersCalculated">
    <reference model="Carrier"/>
    <calculator class="org.openxava.test.calculators.FellowCarriersCalculator"/>
</collection>
```

And here the calculator code:

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

import java.rmi.*;

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

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

    private ICarrier carrier;

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

```

    /* Using EJB3 JPA
    EntityManager manager = XPersistence.getManager();
    Query query = manager.createQuery("from Carrier c where " +
        "c.warehouse.zone = :zone AND " +
        "c.warehouse.number = :warehouseNumber AND " +
        "NOT (c.number = :number) ");
    query.setParameter("zone", getWarehouse().getZone());
    query.setParameter("warehouseNumber", getWarehouse().getNumber());
    query.setParameter("number", getNumber());
    return query.getResultList();
    */

    /* Using EJB2
    return CarrierUtil.getHome().findFellowCarriersOfCarrier(
        carrier.getWarehouseKey().getZoneNumber(),
        carrier.getWarehouseKey().get_Number(),
        new Integer(carrier.getNumber())
    );
    */
}

public void setModel(Object model) throws RemoteException {
    carrier = (ICarrier) model;
}
}
}

```

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

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

```

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

```

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

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

2.11 Method (7)

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

The syntax for method is:

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

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

Defining a method is easy:

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

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

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

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

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

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

```

private IProduct product;

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

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

```

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

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

Now you can use the method in the expected way:

```

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

```

And in `newPrice` you have 102.

Another example, now a little bit more complex:

```

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

```

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

The calculator:

```

package org.openxava.test.calculators;

```

```

import java.math.*;

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

/**
 * @author Javier Paniza
 */

public class ExportPriceCalculator implements ICalculator {

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

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

    public BigDecimal getEuros() {
        return euros;
    }

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

    public BigDecimal getTariff() {
        return tariff;
    }

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

    public String getCountry() {
        return country;
    }

    public void setCountry(String string) {

```

```
        country = string;
    }
}
```

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

And to use the method:

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

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

2.12 Finder (8)

A finder is a special method that allows you to find an object or a collection of objects that follow some criteria. In POJO version a finder method is a generated static method in the POJO class. In the EJB2 version a finder matches with a *finder* in *home*.

The syntax for finder is:

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

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

Some examples:

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

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

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

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

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

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

This generates a set of *finder* methods available from POJO class and EJB2 home. This methods can be used this way:

```
// POJO, both JPA and Hibernate
ICustomer customer = Customer.findByNumber(8);
Collection javieres = Customer.findByNameLike("%JAVI%");

// EJB2
ICustomer customer = CustomerUtil.getHome().findByNumber(8);
Collection javieres = CustomerUtil.getHome().findByNameLike("%JAVI%");
```

2.13 Postcreate calculator (9)

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

Its syntax is:

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

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

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

A simple example is:

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

And now the calculator class:

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

import java.rmi.*;

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

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

    private IDeliveryType deliveryType;
    private String suffix;

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

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

    public String getSuffix() {
```



```

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

```

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

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

2.14 Postmodify calculator (11)

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

Its syntax is:

```

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

```

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

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

A simple example is:

```

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

```

And now the calculator class:

```

package org.openxava.test.calculators;

import java.rmi.*;

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

/**
 * @author Javier Paniza
 */

```

```

public class DeliveryTypePostmodifyCalculator implements IModelCalculator {

    private IDeliveryType deliveryType;

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

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

}

```

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

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

2.15 Postload and preremove calculator (10, 12)

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

2.16 Validator (13)

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

Its syntax is:

```

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

```

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

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

An example:

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

And the validator code:

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

import java.math.*;

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

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

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

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

    public BigDecimal getUnitPrice() {
        return unitPrice;
    }

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

```

    }

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

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

    public int getLimit() {
        return limit;
    }

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

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

```

This validator must implement `IValidator` (1), this forces you to write a `validate(Messages messages)` (2). In this method you add the error message ids (3) (whose texts are in the *i18n* 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.

2.17 Remove validator (14)

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

Its syntax is:

```

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

```

- (1) `class` (optional, required if `name` is not specified): Class that implements the validation logic. **Must implement** `IRemoveValidator`.
- (2) `name` (optional, required if `class` is not specified): This name is a validator name from the `xava/validators.xml` file of your project or from the OpenXava project.
- (3) `set` (several, optional): To set the value of the validator properties before executing it.

An example can be:

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

And the validator:

```
package org.openxava.test.validators;  
  
import java.util.*;  
  
import org.openxava.test.ejb.*;  
import org.openxava.util.*;  
import org.openxava.validators.*;  
  
/**  
 * @author Javier Paniza  
 */  
public class DeliveryTypeRemoveValidator implements IRemoveValidator { // (1)  
  
    private IDeliveryType deliveryType;  
  
    public void setEntity(Object entity) throws Exception { // (2)  
        this.deliveryType = (IDeliveryType) entity;  
    }  
  
    public void validate(Messages errors) throws Exception {  
        if (!deliveryType.getDeliveries().isEmpty()) {  
            errors.add("not_remove_delivery_type_if_in_deliveries"); // (3)  
        }  
    }  
}
```

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

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

2.18 Aggregate

The aggregate syntax is:

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

- (1)**name** (required): Each aggregate must have a unique name. The rules for this name are the same that for class names in Java, that is, to begin with upper-case and each new word starting with upper-case too.
- (2)**bean** (one, optional): Allows to specify a class written by you to implement the aggregate. The class has to be a JavaBean, that is a plain Java class with *getters* and *setters* for properties. Usually this is not used because it is much better that OpenXava generates the code for you.
- (3)**ejb** (one, optional): Allows to use existing EJB2 to implement an aggregate. This can be used only in the case of a collection of aggregates. Usually this is not used because it is much better that OpenXava generates the code for you.

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

2.18.1 Reference to aggregate

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

In the main entity you can write:

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

And in the component level you define:

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

```

    <property name="city" type="String" size="20" required="true"/>
    <reference name="state" required="true"/>
</aggregate>

```

(2)

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

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

```

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

```

Or in this other way to set a new address:

```

// to set a new address
Address address = new Address(); // it's a JavaBean, never an EJB2
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`.

2.18.2 Collection of aggregates

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

```

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

```

And define the `InvoiceDetail` aggregate:

```

<aggregate name="InvoiceDetail">
    <property name="oid" type="String" key="true" hidden="true">
        <default-value-calculator
            class="org.openxava.test.calculators.InvoiceDetailOidCalculator"
            on-create="true"/>
    </property>
    <property name="serviceType">
        <valid-values>

```

```

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

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

</aggregate>

```

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

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

(New in v2.1.1 Reference Guide: Explanation of implicit reference) Every aggregate has a implicit reference to its container model. That is, `InvoiceDetail` has a reference named `invoice`, even if the reference is not declared (although it can be optionally declared for refining purpose, for

example, for making it key). This reference can be used in Java code, as following:

```
InvoiceDetail detail = ... ;
detail.getInvoice(); // For obtaining the parent
```

Or it can be used in XML code, in this way:

```
<property name="oid" type="String" key="true" hidden="true">
  <default-value-calculator
    class="org.openxava.test.calculators.InvoiceDetail2OidCalculator"
    on-create="true">
    <set property="invoiceYear" from="invoice.year"/> (1)
    <set property="invoiceNumber" from="invoice.number"/> (1)
  </default-value-calculator>
</property>
```

In this case we use `invoice` in the `from` attribute (1) although `invoice` is not declared in `InvoiceDetail`. *New in v2.1.1: using reference to parent model key properties in a from attribute (that is, `from="invoice.year"`)*

3 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.

The syntax for view is:

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

- (1)`name` (optional): This name identifies the view, and can be used in other OpenXava places (for example in *application.xml*) or from another component. If the view has no name then the view is assumed as the default one, that is the natural form to display an object of this type.
- (2)`label` (optional): The label that is showed to the user, if needed, when the view is displayed. It's **much better** use the *i18n* files.
- (3)`model` (optional): If the view is for an aggregate of this component you need to specify here the name of that aggregate. If `model` is not specified then this view is for the main entity.
- (4)`members` (optional): List of members to show. By default it displays all members (excluding hidden ones) in the order in which are declared in the model. This attribute is mutually exclusive with the `members` element (that you will see below).
- (5)`property` (several, optional): Defines a property of the view, that is, information that can be displayed to the user and the programmer can work programmatically with it, but it is not a part of the model.
- (6)`property-view` (several, optional): Defines the format to display a property.
- (7)`reference-view` (several, optional): Defines the format to display a reference.
- (8)`collection-view` (several, optional): Defines the format to display a collection.
- (9)`members` (one, optional): Indicates the members to display and its layout in the user interface. Is mutually exclusive with the `members` attribute. Inside `members` you can use `section` and `group` elements (see section 4.1) for layout purposes; or `action` (*new in v2.0.3*) element for showing a link associated to a custom action inside your view (see section 4.6).

3.1 Layout

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

For example, a model like this:

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

Generates a view that looks like this:



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


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

In this case `name` is not shown.

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

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

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

Zone  Office  Number 

Name 


3.1.1 Groups


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

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

In this case the result is:

Id

Zone  Office  Number 






Name 

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

You can put several groups in a view:

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

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

<p>Customer</p> <p>Type  <input type="text" value="Normal"/></p> <p>Name  <input type="text"/></p>	<p>Seller</p> <p>Seller</p> <p>Number  <input type="text"/>  Add</p> <p>Name  <input type="text"/></p> <p>Relation with seller <input type="text" value="GOOD"/></p>
---	---

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

```

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

```

In this case the view is shown this way:

The image shows a user interface with two main sections: 'Customer' and 'Seller'.
 The 'Customer' section contains:
 - A 'Type' field with a dropdown menu currently showing 'Normal'.
 - A 'Name' field with a text input box.
 The 'Seller' section contains:
 - A 'Number' field with a key icon, a text input box, and an 'Add' button.
 - A 'Name' field with a text input box.
 - A 'Relation with seller' field with a dropdown menu currently showing 'GOOD'.

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

```






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

```




```
</members>
```


And the result will be:

Invoice


Year  Number    Date  Year discount €

Delivery data

Type   Number  [Generate](#)

Date  23/08/2005

Description

Shipment 

Transport data

Distance

Vehicle

Transport mode

Driver type



'Delivery by' data

Delivered by


New in v2.0.4: 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;
    <group name="amounts">
      customerDiscount, customerTypeDiscount, yearDiscount;
      amountsSum, vatPercentage, vat;
    </group>
  </members>
</view>
```

...will be displayed as following:

Year  Number 

Amounts

Customer discount	<input type="text" value="11.5"/>	€	Customer type discount	<input type="text" value="20"/>	€	Year discount	<input type="text" value="200"/>	€
Amounts sum	<input type="text" value="2,500"/>	€	VAT % 	<input type="text" value="16"/>	V.A.T.	<input type="text" value="400"/>	€	

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;
    <group name="amounts" aligned-by-columns="true">      (1)
      customerDiscount, customerTypeDiscount, yearDiscount;
      amountsSum, vatPercentage, vat;
    </group>
  </members>
</view>

```

And you will obtain this result:

Year Number

Amounts								
Customer discount	<input type="text" value="11.5"/>	€	Customer type discount	<input type="text" value="20"/>	€	Year discount	<input type="text" value="200"/>	€
Amounts sum	<input type="text" value="2,500"/>	€	VAT %	<input checked="" type="checkbox"/> <input type="text" value="16"/>	V.A.T.	<input type="text" value="400"/>	€	

Now, thanks to the `aligned-by-columns (1)` attribute, the members are aligned by columns.

The attribute `aligned-by-columns` is also available for the sections (see below).

3.1.2 Sections

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

```

<view>
  <members>
    year, number, date, paid;
    customerDiscount, customerTypeDiscount, yearDiscount;
    comment;
    <section name="customer">customer</section>
    <section name="details">details</section>
    <section name="amounts">amountsSum; vatPercentage; vat</section>
    <section name="deliveries">deliveries</section>
  </members>
</view>

```

The visual result is:

Year Number Date Paid

Customer discount € Customer type discount € Year discount €

Comment

Seller | Details | Amounts | Deliveries

Little code

Type

Name

Address

ViewProperty

Street Zip code State

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

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

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

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

Year Number

Customer | **Data** | **Deliveries**

Details | Amounts

V.A.T. | **Amounts sum**

VAT %

V.A.T. €

New in v2.0.4: As in the groups case, the sections allow using the attribute `aligned-by-columns`, like this:

```
<section name="amounts" aligned-by-columns="true"> ... </section>
```

With the same effect as in the group case. Look at section 4.1.2.

3.1.3 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.

3.2 Property view

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

It has this syntax:

```
<property-view
  property="propertyName"           (1)
  label="label"                     (2)
  read-only="true|false"           (3)
  label-format="NORMAL|SMALL|NO_LABEL" (4)
  editor="editorName"               (5) new in v2.1.3
  display-size="size"               (6) new in v2.2.1
>
  <on-change ... />                 (7)
  <action ... /> ...                 (8)
</property-view>
```

- (1)`property` (required): Usually the name of a model property, although it also can be the name of a property of the view itself.
- (2)`label` (optional): Modifies the label for this property in this view. To achieve this it is **much better** use the *i18n* files.
- (3)`read-only` (optional): If you set this property to `true` it never will be editable by the final user in this view. An alternative to this is to make the property editable or not editable programmatically using `org.openxava.view.View`.
- (4)`label-format` (optional): Format to display the label of this property.
- (5)`editor` (optional): *New in v2.1.3.* 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.
- (6)`display-size` (optional): *New in v2.2.1.* The size in characters of the editor in the User Interface

used to display this property. The editor display only the characters indicated by `display-size` but it allows to the user to entry until the total size of the property. If `display-size` is not specified, the value of the size of the property is assumed.

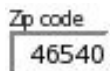
- (7)`on-change` (one, optional): Action to execute when the value of this property changes.
- (8)`action` (several, optional): Actions (showed as links, buttons or images to the user) associated (visually) to this property and that the final user can execute.

3.2.1 Label format

A simple example of using label format:

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

In this case the zip code is displayed as:



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

3.2.2 Value change event

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

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

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

The code to execute is:

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

import org.openxava.actions.*;

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

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

```

        addMessage("carrier_changed");
    }
}

```

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

3.2.3 Actions of property

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

```

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

```

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

```

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

```

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

Number  [Generate](#)

By default the action link is present only when the property is editable, but if the property is read-only or calculated then it is always present. You can use the attribute `always-enabled` (*new in v2.0.3*) to `true` so that the link is always present, even if the property is not editable. As following:

```

<action action="Deliveries.generateNumber" always-enabled="true"/>

```

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

```

package org.openxava.test.actions;

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

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

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

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

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

}

```

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

3.2.4 Choosing an editor (*new in v2.1.3*)

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 in a view.

For example, OpenXava uses a combo for editing the properties of type `valid-values`, 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:

```
<view name="TypeWithRadioButton">
  <property-view property="type" editor="ValidValuesRadioButton"/>
  <members>number; type; name; address</members>
</view>
```

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, stereoetype or a property of a model at application level then it's better to configure it using `xava/editors.xml` file.

3.3 Reference view

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

Its syntax is:

```
<reference-view
  reference="reference"           (1)
  view="view"                   (2)
  read-only="true|false"       (3)
  frame="true|false"           (4)
  create="true|false"          (5)
  modify="true|false"          (6)  new in v2.0.4
  search="true|false"          (7)
  as-aggregate="true|false"    (8)  new in v2.0.3
```

```

>
  <on-change-search ... />           (9)  new in v2.2.5
  <search-action ... />             (10)
  <descriptions-list ... />        (11)
  <action ... /> ...                (12)  new in v2.0.1
</reference-view>

```

- (1) `reference` (required): Name of the reference to refine its presentation.
- (2) `view` (optional): If you omit this attribute, then the default view of the referenced object is used. With this attribute you can indicate that it uses another view.
- (3) `read-only` (optional): If you set the value to `true` the reference never will be editable by final user in this view. An alternative is to make the property editable/uneditable programmatically using `org.openxava.view.View`.
- (4) `frame` (optional): If the reference is displayed inside a frame. The default value is `true`.
- (5) `create` (optional): If the final user can create new objects of the referenced type from here. The default value is `true`.
- (6) `modify` (optional): (*new in v2.0.4*) If the final user can modify the current referenced object from here. The default value is `true`.
- (7) `search` (optional): If the user will have a link to make searches with a list, filters, etc. The default value is `true`.
- (8) `as-aggregate` (optional): (*new in v2.0.3*) By default `false`. By default in the case of a reference to an aggregate 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 `as-aggregate` to `true` then the user interface for references to entities behaves as a in the aggregate 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 aggregates. Warning! If you remove an entity its referenced entities are not removed, even if they are displayed using `as-aggregate="true"`.
- (9) `on-change-search` (one, optional): (*new in v2.2.5*) Allows you to specify your own action for searching when the user type a new key.
- (10) `search-action` (one, optional): Allows you to specify your own action for searching when the user click on search link.
- (11) `descriptions-list`: Display the data as a list of descriptions, typically as a combo. Useful when there are few elements of the referenced object.
- (12) `action` (several, optional): (*new in v2.0.1*) Actions (showed as links, buttons or images to the user) associated (visually) to this reference and that the final user can execute. Works as in `<property-view/>` case, look at section 4.2.3.

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

```

<entity>
...

```

```

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

```

The user interface will look like this (*modify link new in v2.0.4*):

3.3.1 Choose view

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

```

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

```

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

```

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

```

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

3.3.2 Customizing frame

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

```

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

```

```
</members>
```

And the result:

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

3.3.3 Custom search action

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

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

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

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

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

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

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

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



```

/**
 * @author Javier Paniza
 */

public class MySearchAction extends ReferenceSearchAction {

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

}

```

You will learn more about actions in chapter 7.

3.3.4 Custom creation action

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

```

<!--
Because its name is 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.

3.3.5 Custom modification action (*new in v2.0.4*)

If you do not write `modify="false"` 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.

3.3.6 Descriptions list (combos)

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

The syntax is:

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

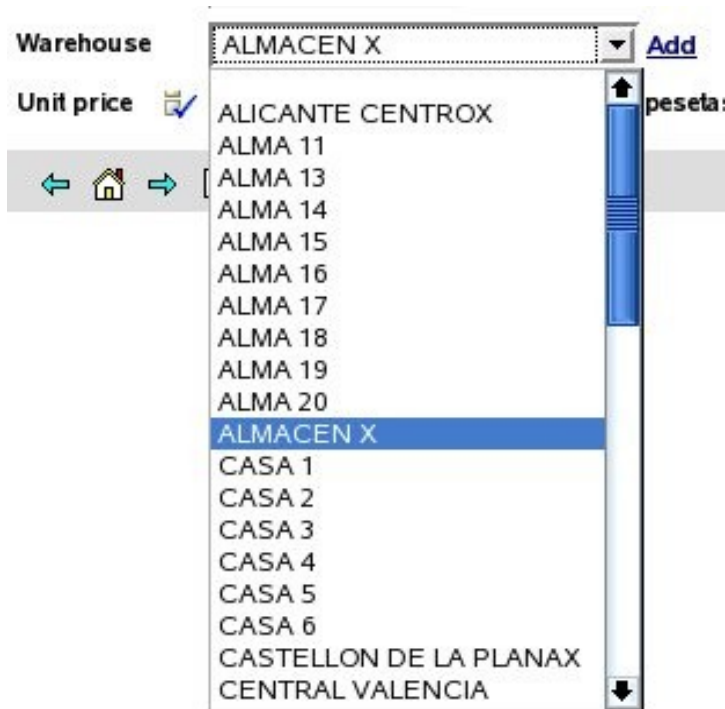
(1) `description-property` (optional): The property to show in the list, if not specified, the property named `description`, `descripcion`, `name` or `nombre` is assumed. If the referenced object does not have a property called this way then it is required to specify a property name here.

- (2) `description-properties` (optional): As `description-property` (and excluding with it) but allows to set more than one property separated by commas. To the final user the values are concatenated.
- (3) `depends` (optional): It's used in together with `condition`. It can be achieve that the list content depends on another value displayed in the main view (if you simply type the name of the member) or in the same view (if you type `this.` before the name of the member).
- (4) `condition` (optional): Allows to specify a condition (with SQL style) to filter the values that are shown in the description list.
- (5) `order-by-key` (optional): By default the data is ordered by description, but if you set this property to `true` it will be ordered by key.
- (6) `order` (optional): Allows to specify an order (with SQL style) for the values that are shown in the description list.
- (7) `label-format` (optional): Format to display the label of the reference. See section 4.2.1.

The most simple usage is:

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

That displays a reference to warehouse in this way:



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

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

implementation can look like this:

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

<reference-view reference="subfamily" create="false"> (2)
    <descriptions-list
        description-property="description"           (3)
        depends="family"                            (4)
        condition="\${family.number} = ?"/>         (5)
        order="\${description} desc"/>             (6)
</reference-view>
```

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

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

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

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

You can specify several properties to be shown as description:

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

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

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

3.3.7 Reference search on change event (*new in v2.2.5*)

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 `on-change-search`, just in this way:

```
<reference-view reference="subfamily">
```

```
<on-change-search class="org.openxava.test.actions.OnChangeSubfamilySearchAction" />
</reference-view>
```

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 {

    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 behavior 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.

3.4 Collection view

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

```
<collection-view
    collection="collection"           (1)
    view="view"                       (2)
    read-only="true|false"           (3)
    edit-only="true|false"           (4)
    create-reference="true|false"     (5)
    modify-reference="true|false"     (6) new in v2.0.4
```

```

    as-aggregate="true|false"          (7) new in v2.0.2
  >
  <list-properties ... />              (8)
  <row-style ... />                    (9) new in v2.2.2
  <edit-action ... />                  (10)
  <view-action ... />                  (11)
  <new-action ... />                   (12) new in v2.0.2
  <save-action ... />                  (13) new in v2.0.2
  <hide-detail-action ... />           (14) new in v2.0.2
  <remove-action ... />                (15) new in v2.0.2
  <remove-selected-action ... />       (16) new in v2.1
  <list-action ... /> ...              (17)
  <detail-action ... /> ...            (18)
</collection-view>

```

- (1) `collection` (required): The look of the collection with this name will be customized.
- (2) `view` (optional): The view of the referenced object (each collection element) which is used to display the detail. By default the default view is used.
- (3) `read-only` (optional): By default `false`; if you set it to `true`, then the final user only can view collection elements, he cannot add, delete or modify elements.
- (4) `edit-only` (optional): By default `false`; if you set it to `true`, then the final user can modify existing elements, but not add or remove collection elements.
- (5) `create-reference` (optional): By default `true`, if you set it to `false` then the final user doesn't get the link to create new objects of the referenced object type. This only applies in the case of an entity references collection.
- (6) `modify-reference` (optional): (*new in v2.0.4*) By default `true`, if you set it to `false` then the final user doesn't get the link to modify the objects of the referenced object type. This only applies in the case of an entity references collection.
- (7) `as-aggregate` (optional): (*new in v2.0.2*) By default `false`. By default the collections of aggregates allow the users to create and to edit elements, while the collections of entities allow only to choose existing entities to add to (or remove from) the collection. If you put `as-aggregate` to `true` then the collection of entities behaves as a collection of aggregates, allowing to the user to add objects and editing them directly. It has no effect in case of collections of aggregates.
- (8) `list-properties` (one, optional): Properties to show in the list for visualization of the collection. You can qualify the properties. By default it shows all persistent properties of the referenced object (excluding references and calculated properties).
- (9) `row-style` (several, optional): *New in v2.2.2*. To give a special style to some rows. Behaves equals that in the `Tab` case. See section 5.1 about emphasize rows for more details. It does not works for calculated collections.
- (10) `edit-action` (one, optional): Allows you to define your custom action to begin the editing of a collection element. This is the action showed in each row of the collection, if the collection is

editable.

- (11) `view-action` (one, optional): Allows you to define your custom action to view a collection element. This is the action showed in each row, if the collection is read only.
- (12) `new-action` (one, optional): (*new in v2.0.2*) Allows you to define your custom action to start adding a new element to the collection. This is the action executed on click in 'Add' link.
- (13) `save-action` (one, optional): (*new in v2.0.2*) Allows you to define your custom action to save the collection element. This is the action executed on click in 'Save detail' link.
- (14) `hide-detail-action` (one, optional): (*new in v2.0.2*) Allows you to define your custom action to hide the detail view. This is the action executed on click in 'Close' link.
- (15) `remove-action` (one, optional): (*new in v2.0.2*) Allows you to define your custom action to remove the element from the collection. This is the action executed on click in 'Remove detail' link.
- (16) `remove-selected-action` (one, optional): (*new in v2.1*) 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.
- (17) `list-action` (several, optional): To add actions in list mode; usually actions which scope is the entire collection.
- (18) `detail-action` (several, optional): To add actions in detail mode, usually actions which scope is the detail that is being edited.

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

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

And the collection is displayed:

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

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

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

This view is also used if the user click on 'Add' in a collection of aggregates, but in the case of a

collection of entities OpenXava does not show this view, instead it shown a list of entities to add (*new in v2.2*).

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:

The screenshot shows a web application interface. At the top, there is a table titled "Customers" with the following columns: Number, Name, Remarks, Relation with seller, and Description. The table contains two rows: one for "1 Javi" with "BUENA" as the relation and "MANAGER" as the description, and another for "2 Juanillo" with "MANAGER" as the description. Below the table, there is a "Customer" detail form. The form has several fields: "Number" (value: 1), "Type" (value: Steady), "Name" (value: Javi), "Address" (value: DOCTOR PESSET), "City" (value: EL PUIG), and "State" (value: New York). There is also a "Zip code" field with the value 46540. At the bottom of the form, there are three buttons: "Save detail", "Close", and "Remove detail".

3.4.1 Custom edit/view action

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

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

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

```
<controller name="Invoices">
  ...
  <action name="editDetail" hidden="true"
    class="org.openxava.test.actions.EditInvoiceDetailAction">
```



```

        <use-object name="xava_view"/>
    </action>
    ...
</controller>

```

And finally write your action:

```

package org.openxava.test.actions;

import java.text.*;

import org.openxava.actions.*;

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

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

```

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

Also it's possible to remove the edit action from the User Interface (*new in v2.2.1*), in this way:

```

<collection-view collection="details">
    <edit-action action=""/>
</collection-view>

```

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 `read-only`.

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

3.4.2 Custom list actions

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

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

Now a new link is shown to user:

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

And also you see that there is a check box in each row (*since v2.1.4 check box is always present in all collections*).

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

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

And the action code:

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

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

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

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

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

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

Also it's possible to use actions for list mode (see at section section 7.5) as list actions for a collection (*new in v2.1.4*).

3.4.3 Default list actions (*new in v2.1.4*)

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 `<list-action/>` element (in section 4.4.2) but it applies to all collections at once.

This feature does not apply to calculated collections (*new in v2.2.1*).

3.4.4 Custom detail actions

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

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

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



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

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

```
<controller name="Invoices">
  ...
  <action name="viewProduct" hidden="true"
    class="org.openxava.test.actions.ViewProductFromInvoiceDetailAction">
```

```

        <use-object name="xava_view"/>
        <use-object name="xavatest_invoiceValues"/>
    </action>
    ...
</controller>

```

And the code of your action:

```

package org.openxava.test.actions;

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

import org.openxava.actions.*;

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

    private Map invoiceValues;

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

```

```

        addError("system_error");
    }
}

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

public String getCustomView() {
    return SAME_VIEW;
}

public Map getInvoiceValues() {
    return invoiceValues;
}

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

```

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

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

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

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

```

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

```

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

```

<controller name="DeliveryDetails">
    <action name="save"
        class="org.openxava.test.actions.SaveDeliveryDetailAction">

```

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

And define your action class for saving:

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

import org.openxava.actions.*;

/**
 *
 * @author Javier Paniza
 */

public class SaveDeliveryDetailAction extends SaveElementInCollectionAction { // (1)

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

}
```

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

New in v2.2.1: Also it's possible to remove any of these actions from User Interface, for example, you can define a `<collection-view/>` in this way:

```
<collection-view collection="details">
    <remove-selected-action action=""/>
</collection-view>
```

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.

3.5 View property

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

An example:

```
<view>
```

```

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

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

```

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

3.6 View actions (new in v2.0.3)

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 `action` element, in this way:

```

<members>
  number;
  type;
  name, <action action="Customers.changeNameLabel"/>;
  ...
</members>

```

The visual effect will be:

Code	<input type="checkbox"/>	<input type="text" value=""/>	
Type	<input checked="" type="checkbox"/>	<input type="text" value="Steady"/>	
Name	<input checked="" type="checkbox"/>	<input type="text" value="Javi"/>	Change name label

You can see the link 'Change name label' that will execute the action `Customers.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 the attribute `always-enabled`, as following:

```
<action action="Customers.changeNameLabel" always-enabled="true"/>
```

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 `action` element may be useful.

See more about actions in chapter 7.

3.7 Transient component: Only for creating views (new in v2.1.3)

In OpenXava it is not possible to have a view without model. Thus if you want to draw an arbitrary user interface, you need to create a component, declare it as transient (*new in v2.1.3*) and define your view from it.

An transient component 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:

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

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

<!--
  Example of an OpenXava transient component (not persistent).

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

<component name="FilterBySubfamily">

  <entity>
    <reference name="subfamily" model="Subfamily2" required="true"/>
  </entity>

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

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



```
        </reference-view>
    </view>

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

    <transient/> (1)

</component>
```

For defining a component as transient you only need to put `<transient/>` at the end of the component definition (1), just in the part for the mappings. You mustn't put the mapping 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.

4 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 a web interface for managing warehouse data. At the top, there are buttons for 'Delete selected' and 'Selected to lowercase', and a link for 'Detail - List'. Below this is a table with columns for 'Zone', 'Warehouse number', and 'Name'. Each row has a 'Detail' link and a checkbox. The table contains 9 rows of data. Below the table, there are pagination controls '1 2 3 4 5' and a status message 'There are 49 objets in list'. At the bottom, there are more buttons for 'Delete selected' and 'Selected to lowercase', and a link for 'Detail - List'.

	Zone	Warehouse number	Name
Filter	=	=	starts
Detail		1	CENTRAL VALENCIA
Detail		1	VALENCIA SURETE
Detail		1	VALENCIA NORTE
Detail		2	CASTELLON DE LA PLANAX
Detail		3	ALICANTE CENTROX
Detail		4	ALMA 2
Detail		4	ALMA 3
Detail		4	ALMA 4
Detail		4	ALMA 5
Detail		4	ALMA 6

This list allows user to:

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

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

The syntax of `tab` is:

```
<tab
  name="name"           (1)
>
  <filter ... />       (2)
  <row-style ... /> ... (3)
  <properties ... />  (4)
  <base-condition ... /> (5)
  <default-order ... /> (6)
</tab>
```

(1)`name` (optional): You can define several tabs in a component, and set a name for each one. This

name is used to indicate the tab that you want to use (usually in *application.xml*).

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

4.1 Initial properties and emphasize rows

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

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

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

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

The visual effect of above is:

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

1

There are 4 objets in list

4.2 Filters and base condition

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

```
<tab name="Current">
  <filter class="org.openxava.test.filters.CurrentYearFilter"/>
```

```

<properties>
    year, number, amountsSum, vat, detailsCount, paid, customer.name
</properties>
<base-condition>${year} = ?</base-condition>
</tab>

```

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

```

package org.openxava.test.filters;

import java.util.*;

import org.openxava.filters.*;

/**
 * @author Javier Paniza
 */

public class CurrentYearFilter implements IFilter {           // (1)

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

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

Another case:

```

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

```

In this case the filter is:

```

package org.openxava.test.filters;

import java.util.*;

import org.openxava.filters.*;

/**
 * @author Javier Paniza
 */

public class DefaultYearFilter extends BaseContextFilter { // (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 (*new in v2.0*) of type `BaseContextFilter`, using `getEnvironment()` method, just in this way:

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

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

4.3 Pure SQL select

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

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

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

4.4 Default order

Finally, setting a default order is very easy:

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

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

5 Object/relational mapping

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

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

- Hibernate 3.x.
- EntityBeans CMP 2 of JBoss 3.2.x y 4.0.x.
- EntityBeans CMP 2 of Websphere 5, 5.1 y 6.

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

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

5.1 Entity mapping

The syntax to map the main entity is:

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

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

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

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

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

A plain example can be:

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

More easier impossible.

You see how the table name is qualified (with collection/schema name included). Also you see that the separator is @separator@ instead of a dot (.), this is useful because you can define the

separator value in your *build.xml* and thus the same application can run against databases with or without support for collections or schemes.

5.2 Property mapping

The syntax to map a property is:

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

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

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

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

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

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

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

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

The converter code is:

```
package org.openxava.converters;

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

```

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

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

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

```

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

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

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

```

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

```

```
...
</entity>
```

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

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

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

5.3 Reference mapping

The syntax to map a reference is:

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

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

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

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

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

You can map it this way:

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

```

        <reference-mapping-detail
            column="INVOICE_NUMBER"
            referenced-model-property="number" />
    </reference-mapping>
    ...
</entity-mapping>

```

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

```

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

```

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

```

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

```

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

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

```

<reference-mapping reference="drivingLicence">
    <reference-mapping-detail
        column="DRIVINGLICENCE_TYPE"
        referenced-model-property="type"
        cmp-type="String"> (1) <!-- In this case this line can be omitted -->

```

```

        <converter class="org.openxava.converters.NotNullStringConverter"/> (2)
    </reference-mapping-detail>
    <reference-mapping-detail
        column="DRIVINGLICENCE_LEVEL"
        referenced-model-property="level"/>
</reference-mapping>

```

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

5.4 Multiple property mapping

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

The syntax for this type of mapping is:

```

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

```

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

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

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

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

```

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

```

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

```

package org.openxava.converters;

import java.util.*;

import org.openxava.util.*;

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

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

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

    public void toDB(Object javaObject) throws ConversionException { // (3)
        if (javaObject == null) {
            setDay(0);
            setMonth(0);
            setYear(0);
            return;
        }
        if (!(javaObject instanceof java.util.Date)) {
            throw new ConversionException("conversion_db_utildate_expected");
        }
        java.util.Date date = (java.util.Date) javaObject;
        Calendar cal = Calendar.getInstance();
        cal.setTime(date);
        setDay(cal.get(Calendar.DAY_OF_MONTH));
        setMonth(cal.get(Calendar.MONTH) + 1);
        setYear(cal.get(Calendar.YEAR));
    }

    public int getYear() {
        return year;
    }

```

```

    }

    public int getDay() {
        return day;
    }

    public int getMonth() {
        return month;
    }

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

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

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

```

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

5.5 Reference to aggregate mapping

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

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

```

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

```

```

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

```

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

```

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

```

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

5.6 Aggregate used in collection mapping

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

Here the model part of Invoice:

```

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

<aggregate name="InvoiceDetail">
  <property name="oid" type="String" key="true" hidden="true">
    <default-value-calculator

```



```

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

```

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

```

<aggregate-mapping aggregate="InvoiceDetail" table="XAVATEST@separator@INVOICEDetail">
    <reference-mapping reference="invoice"> (1)
        <reference-mapping-detail
            column="INVOICE_YEAR"
            referenced-model-property="year" />
        <reference-mapping-detail
            column="INVOICE_NUMBER"
            referenced-model-property="number" />
    </reference-mapping>
    <property-mapping property="oid" column="OID" />
    <property-mapping property="serviceType" column="SERVICETYPE" />
    <property-mapping property="unitPrice" column="UNITPRICE" />
    <property-mapping property="quantity" column="QUANTITY" />
    <reference-mapping reference="product">

```

```

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

```

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

5.7 Converters by default

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

```

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

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

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

<converters>

```

```

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

<for-type type="int"
    converter-class="org.openxava.converters.IntegerNumberConverter"
    cmp-type="java.lang.Integer"/>

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

<for-type type="boolean"
    converter-class="org.openxava.converters.Boolean01Converter"
    cmp-type="java.lang.Integer"/>

<for-type type="java.lang.Boolean"
    converter-class="org.openxava.converters.Boolean01Converter"
    cmp-type="java.lang.Integer"/>

<for-type type="long"
    converter-class="org.openxava.converters.LongNumberConverter"
    cmp-type="java.lang.Long"/>

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

<for-type type="java.math.BigDecimal"
    converter-class="org.openxava.converters.BigDecimalNumberConverter"
    cmp-type="java.math.BigDecimal"/>

<for-type type="java.util.Date"
    converter-class="org.openxava.converters.DateUtilSQLConverter"
    cmp-type="java.sql.Date"/>

</converters>

```

If you use a property of a type that is not defined here, by default OpenXava will assign the converter `NoConversionConverter`, a silly converter that don't perform anything.

In the case of key properties and references no converter are assigned; applying a converter to key

properties can be problematic in certain circumstances, but even if you want to perform a conversion you can declare a converter explicitly in your mapping.

If you wish to modify the behavior of default converters in your application, you do not modify the OpenXava file, but you create your own *converters.xml* file in the folder *xava* of your project. You can assign a converter by default to a stereotype (using `<for-stereotype/>`).

5.8 Default mapping (new in v2.1.3)

Since version 2.1.3 OpenXava allows to define components without mapping, and it assumed a default mapping for it. For example, you can write:

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

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

<component name="Pupil">

  <entity>
    <property name="number" type="int" key="true"
      size="2" required="true"/>
    <property name="name" type="String"
      size="40" required="true"/>
    <reference name="teacher"/>
  </entity>

</component>
```

This component is mapped to the table `Pupil`, and the properties `number` and `name` are mapped to the columns `number` and `name`. The reference `teacher` is mapped to a column named `teacher_number` (if the key property of `Teacher` is named `number`).

5.9 Object/relational philosophy

OpenXava has been born and has been developed in an environment when it was necessary to work with legacy database without changing its structure. The result is that OpenXava:

- Provides great flexibility when mapping with legacy database.
- Does not provide some features natural for OOT and that requires to change database scheme, as inheritance support or polymorphic queries.

Another cool feature of OpenXava mapping is that applications are 100% portables from JBoss CMP2 to Websphere CMP2 without writing a single line of code. Furthermore, the portability between Hibernate, JPA and EJB2 version of an application is very high, the mapping and all automatic controllers are 100% portable, obviously the custom EJB2, JPA or Hibernate code is not so portable.

6 Aspects

6.1 Introduction to AOP

AOP (Aspect Oriented Programming) introduces a new way for reusing code. In fact *aspects* complement some shortcomings in traditional Object Oriented Programming.

Which problems does AOP resolve? Sometime you have a functionality that is common to a group of classes but using inheritance is not practical (in Java we only have single inheritance) or not ethical (because there isn't a *is-a* relationship). Moreover the system may be already written, or maybe you need to include or not this functionality on demand. AOP is an easy way to resolve these problems.

What is an aspect? An aspect is a bunch of code that can be scattered as you wish in your application.

The Java language has a complete AOP support by means of the AspectJ project.

OpenXava adds some support for the *aspects* concept since version 1.2.1. At the moment the support is small and OpenXava is still far away from an AOP framework, but the support of aspects in OpenXava is useful.

6.2 Aspects definition

The *aspects.xml* file inside the *xava* folder of your project is used to define aspects.

The file syntax is:

```
<aspects>
  <aspect ... /> ...           (1)
  <apply ... /> ...           (2)
</aspects>
```

(1)`aspect` (several, optional): To define aspects.

(2)`apply` (several, optional): To apply the defined aspects to the selected models.

With `aspect` (1) you can define an aspect (that is a group of features) with a name, and using `apply` (2) you achieve that a set of models (entities or aggregates) will have these features automatically.

Let's see the aspect syntax:

```
<aspect
  name="name"                 (1)
>
  <postcreate-calculator .../> ... (2)
  <postload-calculator .../> ...   (3)
  <postmodify-calculator .../> ... (4)
  <preremove-calculator .../> ...  (5)
</aspect>
```

- (1) `name` (required): Name for this aspect. It must be unique.
- (2) `postcreate-calculator` (several, optional): All model with this aspect will have this `postcreate-calculator` implicitly.
- (3) `postload-calculator` (several, optional): All model with this aspect will have this `postload-calculator` implicitly.
- (4) `postmodify-calculator` (several, optional): All model with this aspect will have this `postmodify-calculator` implicitly.
- (5) `preremove-calculator` (several, optional): All model with this aspect will have this `preremove-calculator` implicitly.

Furthermore, you need to assign the defined aspects to your models. The syntax to do that is:

```
<apply
  aspect="aspect"           (1)
  for-models="models"      (2)
  except-for-models="models" (3)
/>
```

- (1) `aspect` (required): The name of the aspect that you want to apply.
- (2) `for-models` (optional): A comma separated list of models to which the aspect is applied to. It's mutually exclusive with `except-for-models` attribute.
- (3) `except-for-models` (optional): A comma separated list of models to be excluded when apply this aspect. In this case the aspect applies to all models excepts the indicated ones. It's mutually exclusive with `for-models` attribute.

If you use neither `for-models` nor `except-for-models`, then the aspect will apply to all models in the application. Models are the names of components (for its entities) or aggregates.

A simple example may be:

```
<aspect name="MyAspect">
  <postcreate-calculator
    class="com.mycompany.myapplication.calculators.MyCalculator"/>
</aspect>
<apply aspect="MyAspect"/>
```

Whenever a new object is created (saved in database for the first time), then the logic of `MyCalculator` is executed. And this for all models.

At the moment only these few calculators are supported. We expect to extend the power of *aspects* for OpenXava in the future. Anyway the existing calculators offer interesting possibilities. Let's see an example in the next section.

6.3 AccessTracking: A practical application of aspects

The current OpenXava distribution includes the *AccessTracking* project. This project defines an aspect that allows you to track all access to the data in your application. Actually, this project allows

your application to comply the Spanish Data Protection Law (Ley de Protección de Datos) including high level security data. Although it's generic enough to be useful in a broad variety of scenarios.

6.3.1 The aspect definition

You can find the aspect definition in *AccessTracking/xava/aspects.xml*:

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

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

<!-- AccessTracking -->

<aspects>

    <aspect name="AccessTracking">
        <postcreate-calculator
            class="org.openxava.tracking.AccessTrackingCalculator">
            <set property="accessType" value="Create"/>
        </postcreate-calculator>
        <postload-calculator
            class="org.openxava.tracking.AccessTrackingCalculator">
            <set property="accessType" value="Read"/>
        </postload-calculator>
        <postmodify-calculator
            class="org.openxava.tracking.AccessTrackingCalculator">
            <set property="accessType" value="Update"/>
        </postmodify-calculator>
        <preremove-calculator
            class="org.openxava.tracking.AccessTrackingCalculator">
            <set property="accessType" value="Delete"/>
        </preremove-calculator>
    </aspect>
</aspects>
```

When you apply this aspect to your components, then the code of `AccessTrackingCalculator` is executed each time a object is created, loaded, modified or removed. `AccessTrackingCalculator` writes a record into a database table with information about the access.

In order to apply this aspect you need to write your *aspects.xml* like this:

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

<!DOCTYPE aspects SYSTEM "dtds/aspects.dtd">
```

```

<aspects>

    <apply aspect="AccessTracking" for-models="Warehouse, Invoice"/>

</aspects>

```

In this way this aspect is applied to `Warehouse` and `Invoice`. All access to these entities will be record in a database table.

6.3.2 Setup AccessTracking

If you want to use the `AccessTracking` aspect in your project you have to follow the next setup steps:

- Add `AccessTracking` as referenced project.
- Create the table in your database to store the tracking of accesses. You can find the CREATE TABLES in `AccessTracking/data/access-tracking-db.script` file.
- You have to include the `hibernate.dialect` property in your configuration files. You can see examples of this in `OpenXavaTest/jboss-hypersonic.properties` and other `OpenXavaTest/xxx.properties` files.
- Inside the `AccessTracking` project you need to select a configuration (editing `build.xml`) and regenerate hibernate code (using the ant target `generateHibernate`) for `AccessTracking` project.
- Edit the file of your project `build/ejb/META-INF/MANIFEST.MF` to add the next jars into the classpath: `./lib/tracking.jar ./lib/ehcache.jar ./lib/antlr.jar ./lib/asm.jar ./lib/cglib.jar ./lib/hibernate3.jar ./lib/dom4j.jar`. (This step isn't needed if you use only POJOs, not EJB CMP2, *new in v2.0*)

Also you need to modify the target `createEJBJars` (only if you are using EJB2 CMP) and `deployWar` of your `build.xml` in this way:

```

<target name="createEJBJars">    <!-- 'createEJBJars' only if you use EJB2 CMP -->
    ...
    <ant antfile="../AccessTracking/build.xml" target="createEJBTracker"/>
</target>
<target name="deployWar">
    <ant antfile="../AccessTracking/build.xml" target="createTracker"/>
    ...
</target>

```

After these steps, you have to apply the aspect in your application. Create a file in your project `xava/aspects.xml`:

```

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

```



```
<!DOCTYPE aspects SYSTEM "dtds/aspects.dtd">

<aspects>

    <apply aspect="AccessTracking"/>

</aspects>
```

Now you only have to deploy the war for your project. (*new in v2.0*)

In the case that you are using EJB2 CMP you have to regenerate the code, deploying EJB and deploying war for your project.

All access are recorded in a table with the name TRACKING.ACCESS. If you want you can deploy the module web or the portlet of *AccessTracking* project in order to have a web application to browse the accesses.

For more details you can have a look at the *OpenXavaTest* project.

7 Miscellaneous

7.1 Many-to-many relationships

In OpenXava there is no direct concept of a many-to-many relationship, only collections are available. Nevertheless modeling a many-to-many relationship in OpenXava is easy. You only need to define collections in both sides of the relationship.

For example, if you have customers and states, and a customer can work in several states, and, obviously, in a state several customers can work, then you have a many-to-many (using relational nomenclature) relationship. Suppose that you have a table CUSTOMER (without reference to state), a table STATE (without reference to customer) and a table CUSTOMER_STATE (to link both tables). Then you can model this case in this way:

```
<component name="Customer">
  <entity>
    ...
    <collection name="states"> (1)
      <reference model="CustomerState"/>
    </collection>
    ...
  </entity>

  <aggregate name="CustomerState"> (2)
    <reference name="customer" key="true"/> (3)
    <reference name="state" key="true"/> (4)
  </aggregate>
  ...
</component>
```

You define in `Customer` a collection of aggregates (1), each aggregate (`CustomerState`) (2) contains a reference to a `State` (4), and, of course, a reference of its container entity (`Customer`) (3).

Then you map this collection in the usual way:

```
<component name="Customer">
  ...
  <aggregate-mapping aggregate="CustomerState" table="CUSTOMER_STATE">
    <reference-mapping reference="customer">
      <reference-mapping-detail
        column="CUSTOMER" (1)
        referenced-model-property="number"/>
    </reference-mapping>
    <reference-mapping reference="state">
      <reference-mapping-detail
```

```

        column="STATE" (2)
        referenced-model-property="id"/>
    </reference-mapping>
</aggregate-mapping>
</component>

```

CustomerState is mapped to CUSTOMER_STATE, a table that only contains two columns, one to link to CUSTOMER (1) and other to link to STATE (2).

At model and mapping level all is right, but the User Interface generated by default by OpenXava is somewhat cumbersome in this case. Although with the next refinements to the view part your many-to-many collection will be just fine:

```

<component name="Customer">
    ...
    <view>
        ...
        <collection-view collection="states">
            <list-properties>state.id, state.name</list-properties> (1)
        </collection-view>

        <members>
            ...
            states
            ...
        </members>

    </view>

    <view model="CustomerState">
        <reference-view reference="state" frame="false"/> (2)
    </view>
    ...
</component>

```

In this view you can see how we define explicitly (1) the properties to be shown in list of the collection states. This is needed because we have to show the properties of the State, not the CustomerState ones. Additionally, we define that reference to State in CustomerState view to be showed without frames (2), this is to avoid two ugly nested frames.

By this easy way you can define a collection to map a many-to-many relationship in the database. If you want a bidirectional relationship only need to create a customers collection in State entity, this collection may be of the aggregate type StateCustomer and must be mapped to the table CUSTOMER_STATE. All in analogy to the example here.