VERSIÓN 3.0

COMPONENTES XML CLÁSICOS

Índice de contenido

1 Introducción

OpenXava permite crear aplicaciones de gestión Java usando POJOs y anotaciones Java 5. Pero en sus primeras encarnaciones (v1.0 y v2.0) OpenXava usaba componentes de negocio definidos con XML y generaba el código de la aplicación desde estos XMLs.

Estos componentes XML todavía se soportan. Es decir, si tenemos una aplicación OpenXava desarrollada con v1.x o v2.x podemos actualizar a la última versión de OpenXava. OpenXava soporta componentes XML, y generación de EJB2, POJOs + Hibernate e incluso Java 1.4.

Esta guía es una referencia completa de la sintaxis XML para los componentes de negocio de OpenXava.

2 Modelo

La capa del modelo en una aplicación orientada a objetos es la que contiene la lógica de negocio, esto es la estructura de los datos con los que se trabaja y todos los cálculos, validaciones y procesos asociados a esos datos.

OpenXava es un marco orientado al modelo, en donde el modelo es lo más importante, y todo lo demás (p. ej. la interfaz gráfica) depende de él.

La forma de definir el modelo en OpenXava es mediante XML y un poquito de Java. OpenXava a partir de nuestra definición genera el código Java que implementa ese modelo.

2.1 Implementación en Java

Actualmente OpenXava genera código para las siguiente 4 alternativas:

1. Clases de Java convencionales (los famosos POJOs) para el modelo usando Hibernate para la persistencia *(nuevo en v2.1)*.
2. Clases de Java convencionales para el modelo usando EJB3 JPA (**J**ava **P**ersistence **A**PI) para la persistencia.
3. Los clásicos EntityBeans de EJB2 para el modelo y la persistencia.
4. POJOs + Hibernate dentro de un contenedor EJB.

La opción 2 es la opción por defecto *(nuevo en v3.0)* y la más adecuada para la mayoría de los casos. La opción 1 es también buena, especialmente si necesitamos usar Java 1.4. La opción 3 es para soportar todas las aplicaciones OpenXava escritas con EJB2 (EJB2 era la única opción en OpenXava 1.x). La opción 4 puede ser útil en ciertas circunstancias. Podemos ver como configurar esto en *OpenXavaTest/properties/xava.properties*.

2.2 Componente de negocio

Como bien hemos visto la unidad básica para crear una aplicación OpenXava es el componente de negocio. Un componente de negocio se define en un archivo XML. La estructura de un componente de negocio en OpenXava es:

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

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

<componente nombre="NombreComponente">

    <!-- Modelo -->
    <entidad>...</entidad>
    <agregado nombre="...">...</agregado>
    <agregado nombre="...">...</agregado>
    ...

    <!-- Vista -->
```

```
    <vista>...</vista>
    <vista nombre="...">...</vista>
    <vista nombre="...">...</vista>
    ...

    <!-- Datos tabulares -->
    <tab>...</tab>
    <tab nombre="...">...</tab>
    <tab nombre="...">...</tab>
    ...

    <!-- Mapeo objeto relacional -->
    <mapeo-entidad tabla="...">...</mapeo-entidad>
    <mapeo-agregado agregado="..." tabla="...">...</mapeo-agregado>
    <mapeo-agregado agregado="..." tabla="...">...</mapeo-agregado>
    ...

</component>
```

La primera parte del componente, la parte de entidad y de los agregados, es la que se usa para definir el modelo. En este capítulo vamos a ver la sintaxis completa de esta parte.

2.3 Entidad y agregados

La definición de la entidad y de los agregados es prácticamente idéntica. La entidad es el objeto principal que representa al concepto de negocio, mientras que los agregados son objetos adicionales necesarios para definir el concepto de negocio pero que por sí solos no pueden tener vida propia. Por ejemplo, al definir un componente `Factura`, los datos de cabecera de la factura estarían en la entidad, mientras que para las líneas podríamos poner un agregado `LineaFactura`; podemos notar como el ciclo de vida de un línea de factura está ligado a la factura, esto es una línea de factura sin factura no tiene sentido, y compartir una línea entre varias facturas no es posible, por eso lo modelamos como un agregado.

Formalmente, la relación entre A y B es agregación, y B puede ser modelada como un agregado cuando:

- Podemos decir que A *tiene* un B.
- Si A es borrado su B es borrado también.
- B no es compartido.

A veces un mismo concepto puede ser modelado como agregado o entitad en otro componente. Por ejemplo, el concepto dirección. Si la dirección es compartida por varias personas se debería modelar como una referencia a otra entidad, mientras que sí cada persona tiene su propia dirección puede que lo más práctico sea un agregado.

2.4 Entidad

La sintaxis de la entidad es como sigue:

```
<entidad>
     <bean ... />                              (1)
     <ejb ... />                               (2)
     <implementa .../> ...                     (3)
     <propiedad .../> ...                      (4)
     <referencia .../> ...                     (5)
     <coleccion .../> ...                      (6)
     <metodo .../> ...                         (7)
     <buscador .../> ...                       (8)
     <calculador-poscrear .../> ...            (9)
     <calculador-poscargar .../> ...          (10)
     <calculador-posmodificar .../> ...       (11)
     <calculador-preborrar .../> ...          (12)
     <validador .../> ...                     (13)
     <validador-borrar .../> ...              (14)
</entidad>
```

(1)`bean` (uno, opcional): Permite usar un JavaBean existente (una simple clase Java, uno de esos famosos POJOs). Esto aplica si usamos JPA o Hibernate como gestor de persistencia. En este caso la generación de código para POJO y mapeo de Hibernate para este componente no se produce.

(2)`ejb` (uno, opcional): Permite usar un EJB existente. Esto solo aplica si usamos EJB CMP2 para gestionar la persistencia. En este caso la generación de código EJB para este componente no se produce. No aplica a EJB3.

(3)`implementa` (varios, opcional): Para que el código generado implemente una o varias interfaces Java cualquiera.

(4)`propiedad` (varias, opcional): Las propiedades representan propiedades Java (con su *setter* y *getter*) en el código generado.

(5)`referencia` (varias, opcional): Referencias a otros modelos, podemos referenciar a la entidad de otro componente o a un agregado del nuestro.

(6)`coleccion` (varias, opcional): Colecciones de referencias, en el código generado se convierten en una propiedad que devuelve un `java.util.Collection`.

(7)`metodo` (varios, opcional): Para crear un método en el código generado, en este caso la lógica del método estará contenida en un calculador (`Icalculator`).

(8)`buscador` (varios, opcional): Usado para crear métodos de búsqueda. Los buscadores son métodos estáticos localizados en las clase POJO. En el caso de generación EJB2 se generan *finders* de EJB2.

(9)`calculador-poscrear` (varios, opcional): Lógica a ejecutar después de hacer el objeto persistente. En Hibernate en el evento `PreInsertEvent`, en EJB2 en el método `ejbPostCreate`.

(10)`calculador-poscargar` (varios, opcional): Lógica a ejecutar justo después de cargar el estado del objeto desde el almacenamiento persistente. En Hibernate en el evento `PostLoadEvent`, en EJB2 en el método `ejbLoad`.

(11)`calculador-posmodificar` (varios, opcional): Lógica a ejecutar después de modificar el objeto persistente y antes de grabarlo en el almacenamiento persistente. En Hibernate en el evento `PreUpdateEvent`, en EJB2 en el método `ejbStore`.

(12)`calculador-preborrar` (varios, opcional): Lógica a ejecutar justo antes de borrar el objeto persistente. En Hibernate en el evento `PreDeleteEvent`, en EJB2 en el método `ejbRemove`.

(13)`validador` (varios, opcional): Ejecuta una validación a nivel de modelo. Este validador puede recibir el valor de varias propiedades del modelo. Para validar una sola propiedad es preferible poner el validador a nivel de propiedad.

(14)`validador-borrar` (varios, opcional): Se ejecuta antes de borrar, y tiene la posibilidad de vetar el borrado del objeto.

2.5 Bean (1)

Con `<bean/>` podemos especificar que deseamos usar nuestra propia clase Java.

Por ejemplo:

```
<entidad>
     <bean clase="org.openxava.test.modelo.Familia"/>
     ...
```

De esta forma tan simple podemos escribir nuestro propio código Java en vez de dejar que OpenXava lo genere.

Por ejemplo podemos escribir la clase `Familia` como sigue:

```
package org.openxava.test.modelo;

import java.io.*;

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

     private String oid;
     private int codigo;
     private String descripcion;

     public String getOid() {
            return oid;
     }
```

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

	public int getCodigo() {
		return codigo;
	}
	public void setCodigo(int codigo) {
		this.codigo = codigo;
	}

	public String getDescripcion() {
		return descripcion;
	}
	public void setDescripcion(String descripcion) {
		this.descripcion = descripcion;
	}

}
```

Si queremos referenciar desde el código generado por OpenXava a nuestro propio código necesitamos que nuestra clase Java implemente una interfaz (`IFamilia` en este caso) que extienda de `IModel` (ver `org.openxava.test.Family` en *OpenXavaTest/src*).

Adicionalmente tenemos que definir el mapeo usando Hibernate:

```
<?xml version="1.0"?>

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

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

  <class
	name="Familia"
	table="XAVATEST@separator@FAMILIA">

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

	<property name="codigo" column="CODIGO"/>
	<property name="descripcion" column="DESCRIPCION"/>
```

```
  </class>

</hibernate-mapping>
```

Podemos poner este archivo en la carpetea hibernate de nuestro proyecto. Además en esta carpeta tenemos el archivo *hibernate.cfg.xml* que hemos de editar de esta forma:

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

De esta forma tan sencilla podemos envolver nuestras clases y mapeos de hibernate existentes usando OpenXava. Por supuesto, si estamos creando un sistema nuevo es mucho mejor dejar la generación del código en manos de OpenXava.

2.6 EJB (2)

Con `<ejb/>` podemos especificar que queremos usar un EJB propio.

Por ejemplo:

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

De esta forma tan sencilla podemos escribir nuestro código EJB a mano, en vez de usar el código que OpenXava genera.

El código EJB lo podemos escribir manualmente de forma íntegra (pero eso solo si somos hombre de verdad), si somos programadores al uso(quiero decir vagos) preferiremos usar los asistentes de un IDE, o mejor aún XDoclet. Si optamos por usar XDoclet, podemos poner nuestras clases XDoclet en el paquete `modelo` (o cualquier otro paquete, depende del valor que demos a la variable `model.package` en nuestro *build.xml*) de la carpeta *src* de nuestro proyecto; y nuestro código XDoclet se generará junto con el resto de código OpenXava.

Para nuestro ejemplo podriamos escribir una clase `FamiliaBean` como sigue:

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

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

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

/**
 * @ejb:bean name="Familia" type="CMP" view-type="remote"
 *   jndi-name="OpenXavaTest/ejb/openxava.test/Familia"
 * @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@FAMILIA"
 *
 * @author Javier Paniza
 */
abstract public class FamiliaBean
     extends org.openxava.ejbx.EJBReplicableBase // (1)
     implements javax.ejb.EntityBean {

     private UUIDCalculator calculadorOid = 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 "CODIGO"
       */
     public abstract int getCodigo();
     /**
      * @ejb:interface-method
      */
```

```
	public abstract void setCodigo(int newCodigo);

	/**
	 * @ejb:interface-method
	 * @ejb:persistent-field
	 *
	 * @jboss:column-name "DESCRIPCION"
	 */
	public abstract String getDescripcion();
	/**
	  * @ejb:interface-method
	  */
	public abstract void setDescripcion(String newDescripcion);

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

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

Al escribir nuestro EJB tenemos que observar dos pequeñas restricciones:

(1)La clase ha de descender de `org.openxava.ejbx.EJBReplicableBase`

(2)Ha de haber al menos un `ejbCreate` (con su `ejbPostCreate`) que reciba como argumento un mapa y asigne sus valores al bean como en este ejemplo.

Sí, sí, es un poco intrusivo, pero ¿no son los EJB la culminación de la intrusión?

2.7 Implementa (3)

Con `<implementa/>` especificamos una interfaz Java que será implementada por el código generado. Como sigue:

```
<entidad>
     <implementa interfaz="org.openxava.test.modelo.IConNombre"/>
     ...
     <propiedad nombre="nombre" tipo="String" requerido="true"/>
     ...
```

Y podemos hacer nuestra interfaz Java:

```
package org.openxava.test.modelo;

import java.rmi.*;

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

     String getNombre() throws RemoteException;

}
```

Hemos de tener cuidado para que el código generado implemente nuestra interfaz. En este caso tenemos una propiedad `nombre` que generará un método `getNombre()` y por ende se implementará la interfaz.

En nuestro código generado nos encontramos una interfaz `ICliente` como sigue:

```
public interface ICliente extends org.openxava.test.modelo.IConNombre {
     ...
}
```

In the POJO generated code you can see:

```
public class Cliente implements Serializable, org.openxava.test.modelo.ICliente {
     ...
}
```

En el código EJB generado (si es que generamos EJB) podremos observar en la interfaz remota

```
public interface ClienteRemote extends
      org.openxava.ejbx.EJBReplicable,
      org.openxava.test.modelo.ICliente
```

y la clase del bean también se ve afectada

```
abstract public class ClienteBean extends EJBReplicableBase
      implements
            org.openxava.test.modelo.ICliente,
            EntityBean
```

Esta jugosa característica hace del polimorfismo un invitado privilegiado de OpenXava.

Se puede ver como OpenXava genera una interfaz por cada componente. Es bueno usar estas interfaces en lugar de los POJOs o las interfaces remotas de EJB2. Todo el código generado de esta forma puede usarse en una versión POJO y EJB2 al mismo tiempo. Esto también facilitará una posible migración del código de EJB2 a POJO. Aunque, si trabajamos usando POJOs exclusivamente podemos escoger trabajar directamente con las clases POJO ignorando las interfaces, al gusto.

2.8 Propiedad (4)

Una propiedad OpenXava corresponde exactamente a una propiedad Java. Representa parte del estado de un objeto que se puede consultar y en algunos casos cambiar. El objeto no tiene la obligación de guardar físicamente la información de la propiedad, solo de devolverla cuando se le pregunte.

La sintaxis para definir una propiedad es:

```
<propiedad
      nombre="nombrePropiedad"                (1)
      etiqueta="etiqueta"                     (2)
      tipo="tipo"                             (3)
      estereotipo="ESTEREOTIPO"               (4)
      longitud="longitud"                     (5)
      escala="longitud"                       (6)   nuevo en v2.0.4
      requerido="true|false"                  (7)
      clave="true|false"                      (8)
      oculta="true|false"                     (9)
      clave-busqueda="true|false"            (10)   nuevo en v2.2.4
      version="true|false"                   (11)   nuevo en v2.2.3
>
      <valores-posibles .../>                (12)
      <calculador .../>                      (12)
      <calculador-valor-defecto .../>        (13)
      <validador .../> ....                  (14)
</propiedad>
```

(1)`nombre` (obligado): Es el nombre que tendrá la propiedad en Java, por lo tanto ha de seguir la normativa para nombres de propiedad Java, entre ellas empezar por minúscula. Se desaconseja el uso de subrayados (_).

(2)`etiqueta` (opcional): Etiqueta que se mostrará al usuario final. Es **mucho mejor** usar los archivos *i18n*.

(3)`tipo` (opcional): Corresponde a un tipo Java. Todos los tipos válidos para una propiedad Java son válidos aquí, esto incluye clases definidas por nosotros mismos. Solo hemos de proveer un conversor para grabar en la base de datos y un editor para visualizar en HTML; así que cosas como una `java.sql.Connection` o así puede que sean complicadas de tratar, pero no imposible. Es opcional, pero solo si hemos especificado `<bean/>` o `<ejb/>` antes o asignamos un estereotipo con un tipo asociado.

(4)`estereotipo` (opcional): Permite especificar un comportamiento especial para cierta propiedades.

(5)`longitud` (opcional): Longitud en caracteres de la propiedad. Especialmente útil a la hora de generar interfaces gráficas. Si no especificamos longitud asume un valor por defecto asociado al tipo o estereotipo que se obtiene de *default-size.xml* o *longitud-defecto.xml*.

(6)`escala` (opcional): *(nuevo en v2.0.4)* Escala (tamaño de la parte decimal) de la propiedad. Solo aplica a propiedades numéricas. Si no especificamos escala asume un valor por defecto asociado al tipo o estereotipo que se obtiene de *default-size.xml* o *longitud-defecto.xml*.

(7)`requerido` (opcional): Indica si esa propiedad es requerida. Por defecto es `true` para las propiedades clave ocultas *(nuevo en v2.1.3)* o sin calculador valor por defecto al crear y `false` en todos los demás casos. Al grabar OpenXava comprobará si las propiedades requeridas están presentes, si no lo están no se producirá la grabación y se devolverá una lista de errores de validación. La lógica para determinar si una propiedad está presente o no se puede configurar creando un archivo *validators.xml* o *validadores.xml* en nuestro proyecto. Podemos ver la sintaxis en *OpenXava/xava/validators.xml*.

(8)`clave` (opcional): Para indicar si una propiedad forma parte de la clave. Al menos una propiedad (o referencia) ha de ser clave. La combinación de propiedades (y referencias) clave se debe mapear a un conjunto de campos en la base de datos que no tengan valores repetidos, típicamente con la clave primaria.

(9)`oculta` (opcional): Una propiedad oculta es aquella que tiene sentido para el desarrollador pero no para el usuario. Las propiedades ocultas se excluyen cuando se generan interfaces gráficas automáticas, sin embargo a nivel de código generado están presentes y son totalmente funcionales, incluso si se les hace alusión explicita podrían aparecer en una interfaz gráfica.

(10)`clave-busqueda` (opcional): *(nuevo en v2.2.4)* Las propiedades clave de búsqueda se usan por los usuarios para buscar los objetos. Son editables en la interfaz de usuario de las referencias permitiendo al usuario teclear su valor para buscar. OpenXava usa las propiedades clave (`clave="true"`) para buscar por defecto, y si la propiedades clave (`clave="true"`) están ocultas usa la primera propiedad en la vista. Con `clave-busqueda` podemos elegir las propiedades para buscar explicitamente.

(11)`version` (opcional): *(nuevo en v2.2.3)* Una propiedad versión se usa para el control de concurrencia optimista. Si queremos control de concurrencia solo necesitamos tener una propiedad marcada como `version="true"` en nuestro componente. Solo podemos especificar

una propiedad de versión por componente. Los siguientes tipos son soportados para propiedades versión: `int`, `Integer`, `short`, `Short`, `long`, `Long`, `Timestamp`. Las propiedades de versión también se consideran ocultas.

(12)`valores-posibles` (uno, opcional): Para indicar que la propiedad en cuestión solo puede tener un conjunto de valore fijos.

(13)`calculador` (uno, opcional): Para implementar la lógica de una propiedad calculada. Una propiedad calculada solo tiene *getter* y no se almacena en la base de datos.

(14)`calculador-valor-defecto` (uno, opcional): Para implementar la lógica para calcular el valor inicial de la propiedad. Una propiedad con `calculador-valor-defecto` sí tiene *setter* y es persistente.

(15)`validador` (varios, opcional): Indica la lógica de validación a ejecutar sobre el valor a asignar a esta propiedad antes de crear o modificar.

2.8.1 Estereotipo

Un estereotipo es la forma de determinar un comportamiento especifico dentro de un tipo. Por ejemplo, un nombre, un comentario, una descripción, etc. todos corresponden al tipo Java `java.lang.String` pero si queremos que los validadores, logitud por defecto, editores visuales, etc. sean diferente en cada caso y necesitamos afinar más; lo podemos hacer asignando un esterotipo a cada uno de estos casos. Es decir, podemos tener los estereotipos NOMBRE, TEXTO_GRANDE o DESCRIPCION y asignarlos a nuestras propiedades.

El OpenXava viene configurado con los siguientes estereotipos:

- 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 (instrucciones en 3.8.2) *nuevo en v2.0*
- RELLENADO_CON_CEROS, ZEROS_FILLED *nuevo en v2.0.2*
- TEXTO_HTML, HTML_TEXT (texto con formato editable) *nuevo en v2.0.3*
- ETIQUETA_IMAGEN, IMAGE_LABEL (imagen que depende del contenido de la propiedad) *nuevo en v2.1.5*
- EMAIL *nuevo en 2.2.3*
- TELEFONO, TELEPHONE *nuevo en 2.2.3*
- WEBURL *nuevo en 2.2.3*
- IP *nuevo en 2.2.4*
- ISBN *nuevo en 2.2.4*

- TARJETA_CREDITO, CREDIT_CARD *nuevo en 2.2.4*
- LISTA_EMAIL, EMAIL_LIST *nuevo en 2.2.4*

Vamos a ver como definiríamos un estereotipo propio. Crearemos uno llamado NOMBRE_PERSONA para representar nombres de persona.

Editamos (o creamos) el archivo *editors.xml* o *editores.xml* en nuestra carpeta *xava*. Y añadimos

```
<editor url="editorNombrePersona.jsp">
    <para-estereotipo estereotipo="NOMBRE_PERSONA"/>
</editor>
```

De esta forma indicamos que editor se ha de ejecutar para editar y visualizar propiedades con el estereotipo NOMBRE_PERSONA.

También podemos editar *stereotype-type-default.xml* o *tipo-estereotipo-defecto.xml* y añadir la línea:

```
<para estereotipo="NOMBRE_PERSONA" tipo="String"/>
```

Además es útil indicar la longitud por defecto, eso se hace editando *default-size.xml* o *longitud-defecto.xml*:

```
<para-estereotipo nombre="NOMBRE_PERSONA" longitud="40"/>
```

Y así si no ponemos longitud asumirá 40 por defecto.

Menos común es querer cambiar el validador para `requerido`, pero si queremos cambiarlo lo podemos hacer añadiendo a *validators.xml* o *validadores.xml* de nuestro proyecto lo siguiente:

```
<validador-requerido>
     <clase-validador clase="org.openxava.validators.NotBlankCharacterValidator"/>
     <para-estereotipo estereotipo="NOMBRE_PERSONA"/>
</validador-requerido>
```

Ahora podemos definir propiedades con estereotipo NOMBRE_PERSONA:

```
<propiedad nombre="nombre" estereotipo="NOMBRE_PERSONA" requerido="true"/>
```

En este caso asume 40 longitud y tipo `String`, así como ejecutar el validador `NotBlankCharacterValidator` para comprobar que es requerido.

2.8.2 Estereotipo GALERIA_IMAGENES *(nuevo en v2.0)*

Si queremos que una propiedad de nuestro componente almacene una galería de imágenes. Solo necesitamos declarar que nuestra propiedad sea del estereotipo GALERIA_IMAGENES. De esta manera:

```
<propiedad nombre="fotos" estereotipo="GALERIA_IMAGENES"/>
```

Además, en el mapeo tenemos que mapear la propiedad contra una columna adecuada para almacenar una cadena (`String`) con 32 caracteres de longitud (`VARCHAR(32)`).

Y ya está todo.

Pero, para que nuestra aplicación soporte este estereotipo necesitamos configurar nuestro sistema.

Lo primero es crear a tabla en la base de datos para almacenar las imágenes:

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

CREATE INDEX IMAGENES01
      ON IMAGENES (GALLERY);
```

El tipo de la columna `IMAGE` puede ser un tipo más adecuado para almacenar `byte []` en el caso de nuestra base de datos (por ejemplo `LONGVARBINARY`) .

El nombre de la tabla puede ser cualquiera que queramos. Especificamos el nombre de la tabla y nombre de esquema *(nuevo en v2.2.4)* en el archivo de configuración (un archivo *.properties* en la raiz de nuestro proyecto OpenXava). De esta forma:

```
images.schema=MIESQUEMA
images.table=IMAGENES
```

Y finalmente necesitamos definir el mapeo en nuestro archivo *hibernate/hibernate.cfg.xml*, así:

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

Después de todo esto ya podemos usar el estereotipo GALERIA_IMAGENES en los componentes de nuestra aplicación.

2.8.3 Concurrencia y propiedad versión *(nuevo en v2.2.3)*

Concurrencia es la habilidad de una aplicación para permitir que varios usuarios graben datos al mismo tiempo sin perder información. OpenXava usa un esquema de concurrencia optimista. Usando concurrencia optimista los registros no se bloquean permitiendo un alto nivel de concurrencia sin perder la integridad de la información.

Por ejemplo, si un usuario A lee un registro y entonces un usuario B lee el mismo registro, lo modifica y graba los cambios, cuando el usuario A intente grabar el registro recibirá un error y tendrá que refrescar los datos y reintentar su modificación.

Para activar el soporte de concurrencia para un componente OpenXava solo necesitamos declarar una propiedad usando `version="true"`, de esta manera:

```
<propiedad nombre="version" tipo="int" version="true"/>
```

Esta propiedad es para uso del mecanismo de persistencia (Hibernate o JPA), ni nuestra aplicación ni usuarios deberían acceder directamente a ella.

2.8.4 Valores posibles

El elemento `<valores-posibles/>` permite definir una propiedad que solo puede contener los valores indicados. Digamos que es algo así como el `enum` de C (o Java 5).

Es fácil de usar, veamos un ejemplo:

```
<propiedad nombre="distancia">
     <valores-posibles>
           <valor-posible valor="local"/>
           <valor-posible valor="nacional"/>
           <valor-posible valor="internacional"/>
     </valores-posibles>
</propiedad>
```

La propiedad `distancia` solo puede valer `local`, `nacional` o `internacional`, y como no hemos puesto `requerido="true"` también la podemos dejar en blanco. Como se ve no es necesario indicar el tipo, asume `int` por defecto.

A nivel de interfaz gráfico la implementación web actual usa un combo. La etiqueta para cada valor se obtienen de los archivos *i18n*.

A nivel de código Java generado crea una propiedad `distancia` de tipo `int` que puede tener valor 0 (sin valor), 1 (local), 2 (nacional) o 3 (internacional).

A nivel de base datos por defecto guarda el entero, pero esto se puede configurar fácilmente para poder usar sin problemas bases de datos legadas. Ver más de esto último en el capítulo 6.

2.8.5 Calculador

Un calculador indica que lógica hay que usar cuando se llame al método *getter* de la propiedad. Las propiedades que definen un calculador son de solo lectura (solo tienen *getter*) y no son persistentes (no tienen correspondencia con ninguna columna de la tabla de base de datos).

Así se define una propiedad calculada:

```
<propiedad nombre="precioUnitarioEnPesetas" tipo="java.math.BigDecimal" longitud="18">
     <calculador clase="org.openxava.test.calculadores.CalculadorEurosAPesetas">
           <poner propiedad="euros" desde="precioUnitario"/>
     </calculador>
</propiedad>
```

Ahora cuando nosotros (o el OpenXava para llenar una interfaz gráfica) llamamos a `getPrecioUnitarioEnPesetas()` el sistema ejecuta el calculador `CalculadorEurosAPesetas`, pero antes llena el valor de la propiedad `euros` de `CalculadorEurosAPesetas` con el valor de `precioUnitario` del objeto actual.

Puede ser instructivo ver el código del calculador:

```
package org.openxava.test.calculadores;

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

/**
 * @author Javier Paniza
 */
public class CalculadorEurosAPesetas 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;
	}

}
```

Notamos dos cosas, primero `(1)` que un calculador ha de implementar `org.openxava.calculators.ICalculator`, y que `(2)` el método `calculate()` es el que ejecuta la lógica que devolverá la propiedad.

Según lo visto ahora podemos usar el código generado de la siguiente forma:

```
Producto producto = ...
producto.setPrecioUnitario(2);
BigDecimal result = producto.getPrecioUnitarioEnPesetas();
```

Y `result` contendría 332.772.

Podemos definir un calculador sin poner `desde` al definir valores para sus propiedades, como sigue:

```
<propiedad nombre="cantidadLineas" tipo="int" longitud="3">
	<calculador clase="org.openxava.test.calculadores.CalculadorCantidadLineas">
		<poner propiedad="año"/>
```

```
            <poner propiedad="numero"/>
      </calculador>
</propiedad>
```

En este caso la propiedad `año` y `numero` de calculador `CalculadorCantidadLineas` se llenan desde las propiedades del mismo nombre en el objeto actual.

El atributo `desde` soporta propiedades calificadas, como sigue:

```
<agregado nombre="Direccion">
      <propiedad nombre="calle" tipo="String" longitud="30" requerido="true"/>
      <propiedad nombre="codigoPostal" tipo="int" longitud="5" requerido="true"/>
      <propiedad nombre="poblacion" tipo="String" longitud="20" requerido="true"/>
      <reference nombre="provincia" requerido="true"/>
      <propiedad nombre="comoCadena" tipo="String">
            <calculador clase="org.openxava.calculators.ConcatCalculator">
                  <poner propiedad="string1" desde="calle"/>
                  <poner propiedad="int2" desde="codigoPostal"/>
                  <poner propiedad="string3" desde="poblacion"/>
                  <poner propiedad="string4" desde="provincia.nombre"/> (1)
                  <poner propiedad="int5" desde="cliente.codigo"/>            (2)
            </calculador>
      </propiedad>
</agregado>
```

La propiedad `string4` (1) del calculador se llena usando el valor de `nombre` de la `provincia` (que es una referencia), esto es una propiedad calificada (`referencia.propiedad`). En el caso de `int5` (2) podemos ver como la referencia `cliente` no esta declarada en `Direccion`, porque es referenciada desde la entidad `Cliente`, por lo tanto `Direccion` tiene un referencia implicita a su modelo contenedor (su padre) que podemos usar en el atributo `desde` *(nuevo en v2.0.4)*. Esto es, la propiedad `int5` se llena con el `codigo` del `cliente` el cual tiene esta dirección.

También podemos asignar un valor fijo a una propiedad de un calculador:

```
<propiedad nombre="nombreCompleto" tipo="String">
      <calculador clase="org.openxava.calculators.ConcatCalculator">
            <poner propiedad="string1" desde="id"/>
            <poner propiedad="separator" valor=" - "/>
            <poner propiedad="string2" desde="name"/>
      </calculador>
</propiedad>
```

En este caso la propiedad `separator` de `ConcatCalculator` se le da un valor fijo.

Otra característica interesante de los calculadores es que podemos acceder directamente al objeto modelo (entidad o agregado) que contiene el calculador:

```
<propiedad nombre="sumaImportes" estereotype="DINERO">
     <calculador clase="org.openxava.test.calculadores.CalculadorSumaImportes"/>
</propiedad>
```

Y el calculador:

```
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 CalculadorSumaImportes implements IModelCalculator { // (1)

     private IFactura factura;

     public Object calculate() throws Exception {
            Iterator itLineas = factura.getLineas().iterator();
            BigDecimal result = new BigDecimal(0);
            while (itLineas.hasNext()) {
                  ILineaFactura linea = (ILineaFactura) itLineas.next();
                  result = result.add(linea.getImporte());
            }
            return result;
     }

     public void setModel(Object modelo) throws RemoteException { // (2)
            factura = (IFactura) modelo;
     }

}
```

Este calculador implementa `IModelCalculator (1)` *(nuevo en v2.0)* y por ello tiene un método `setModel (2)`, este método se llama antes de llamar al método `calculate()` y así desde el método

`calculate()` podemos acceder al modelo (en este caso a la factura) que contiene la propiedad o método. A pesar de su nombre también puede recibir un objeto que esté actuando como agregado.

Entre el código generado por OpenXava encontramos una interfaz por cada concepto de negocio que es implementada por la clase POJO, por la interfaz remota y por la clase del bean. Esto es para Factura tendriamos una interfaz `IFactura` implementada por `Factura` (clase POJO), `FacturaRemote` (interfaz remota EJB2) y `FacturaBean` (clase del bean EJB2), estos dos últimos solo si generamos código EJB2. En los calculadores `IModelCalculator` es aconsejable moldear a esta interfaz, de esta forma el mismo calculador funcionará con POJOs, interfaces remotos EJB2 y clases de bean EJB2. Si desarrollamos usando solo POJOs (puede que esto sea lo normal) podemos elegir moldear directamente a la clase POJO, que en esta caso sería `Factura`.

Este tipo de calculadores es menos reutilizable que los que reciben propiedades simples, pero a veces es práctico usarlos. ¿Por qué es menos reutilizable? Por ejemplo, si usamos `IFactura` para calcular un descuento, ese calculador solo podría se aplicado a facturas, pero si usamos un calculador que recibe `cantidad` y `porcentajeDescuento` como propiedades simples este último calculador podría ser aplicado a facturas, albaranes, pedidos, etc.

Desde un calculador también se puede acceder directamente a una conexión JDBC, aquí un ejemplo:

```
<propiedad nombre="cantidadLineas" tipo="int" longitud="3">
     <calculador clase="org.openxava.test.calculadores.CalculadorCantidadLineas">
           <poner propiedad="año"/>
           <poner propiedad="numero"/>
     </calculador>
</propiedad>
```

Y la clase del calculador:

```
package org.openxava.test.calculadores;

import java.sql.*;

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

/**
 * @author Javier Paniza
 */
public class CalculadorCantidadLineas implements IJDBCCalculator {      // (1)

     private IConnectionProvider provider;
     private int año;
     private int numero;

     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_LINEAFACTURA " +
                        "where FACTURA_AÑO = ? and FACTURA_NUMERO = ?");

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

      public int getAño() {
            return año;
      }

      public int getNumero() {
            return numero;
      }

      public void setAño(int año) {
            this.año = año;
      }

      public void setNumero(int numero) {
            this.numero = numero;
      }

}
```

Para usar JDBC tenemos que implementar `IJDBCCalculator (1)` y entonces recibiremos un `IConnectionProvider (2)` que podremos usar dentro de `calculate()`. Ya sé que el código JDBC

es feo y engorroso, pero en ocasiones puede ayudar a resolver algún problema de rendimiento.

Los calculadores nos permiten de forma elegante insertar nuestra propia lógica en un sistema en el que todo el código es generado; y como se puede ver promueven la creación de código reutilizable ya que la naturaleza de los calculadores (simples y configurables) permite que se usen una y otra vez para definir propiedades calculadas o métodos. Esta filosofía, la de clases simples y configurables que se pueden enchufar en varios lugares es lo que sustenta todo el marco de trabajo OpenXava.

OpenXava dispone de un conjunto de calculadores incluidos de uso genérico, que se pueden encontrar en `org.openxava.calculators`.

2.8.6 Calculador valor defecto

Con `<calculador-valor-defecto/>` podemos asociar una lógica a una propiedad, pero en este caso la propiedad es de lectura y escritura y persistente. Este calculador sirve para calcular su valor inicial. Por ejemplo:

```
<propiedad nombre="año" tipo="int" clave="true" longitud="4" requerido="true">
     <calculador-valor-defecto
           clase="org.openxava.calculators.CurrentYearCalculator"/>
</propiedad>
```

En este caso cuando el usuario intente crear una factura nueva (por ejemplo) se encontrará que el campo año ya tiene un valor, que el usuario puede cambiar si desea.

Podemos indicar que calcule el valor justo antes de crear (insertar en la base datos) el objeto por primera vez; eso se hace así:

```
<propiedad nombre="oid" tipo="String" clave="true" oculta="true">
     <calculador-valor-defecto
           clase="org.openxava.calculators.UUIDCalculator"
           al-crear="true"/>
</propiedad>
```

Al poner `al-crear="true"` conseguimos este efecto.

Un uso típico de `al-crear="true"` es para generar identificadores automáticamente. En el ejemplo anterior, un identificador único de tipo `String` y 32 caracteres es generado. Podemos usar otras técnicas de generación, por ejemplo, una *sequence* de base de datos se puede definir así:

```
<propiedad nombre="id" clave="true" tipo="int" oculta="true">
     <calculador-valor-defecto
           clase="org.openxava.calculators.SequenceCalculator" al-crear="true">
           <poner propiedad="sequence" valor="XAVATEST_SIZE_ID_SEQ"/>
     </calculador-valor-defecto>
</propiedad>
```

O quizás queramos usar una columna *identity* (auto incremento) como clave:

```
<propiedad nombre="id" clave="true" tipo="int" oculta="true">
      <calculador-valor-defecto
            clase="org.openxava.calculators.IdentityCalculator" al-crear="true"/>
</propiedad>
```

`SequenceCalculator` *(nuevo en v2.0.1)* e `IdentityCalculator` *(nuevo en v2.0.2)* no funcionan con EJB2. Funcionan con Hibernate y EJB3.

Si definimos una propiedad clave y oculta sin calculador valor defecto con `al-crear="true"` entonces se usan las técnicas *identity*, *sequence* o *hilo* automáticamente dependiendo de las capacidades de la base de datos subyacente. Así:

```
<propiedad nombre="oid" tipo="int" oculta="true" clave="true"/>
```

Esto solo funciona con Hibernate y EJB3, no con EJB2.

Por lo demás funciona exactamente igual que `<calculador/>` visto en la sección anterior.

2.8.7 Validador

El validador ejecuta la lógica de validación sobre el valor que se vaya a asignar a esa propiedad antes de grabar. Una propiedad puede tener varios validadores.

```
<propiedad nombre="description" tipo="String" longitud="40" requerido="true">
      <validador clase="org.openxava.test.validadores.ValidadorExcluirCadena">
            <poner propiedad="cadena" valor="MOTO"/>
      </validador>
      <validador clase="org.openxava.test.validadores.ValidadorExcluirCadena"
            solo-al-crear="true">
            <poner propiedad="cadena" valor="COCHE"/>
      </validador>
</propiedad>
```

La forma de configurar el validador (con los `<poner/>`) es exactamente igual como en los calculadores. Con el atributo `solo-al-crear="true"` se puede definir que esa validación solo se ejecute cuando se crea el objeto, y no cuando se modifica.

El código del validador es:

```
package org.openxava.test.validadores;

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

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

```
public class ValidadorExcluirCadena implements IPropertyValidator { // (1)

	private String cadena;

	public void validate(
		Messages errores,		// (2)
		Object valor,			// (3)
		String nombreObjecto,		// (4)
		String nombrePropiedad)		// (5)
		throws Exception {
		if (valor==null) return;
		if (valor.toString().indexOf(getCadena()) >= 0) {
			errores.add("excluir_cadena",
				nombrePropiedad, nombreObjeto, getCadena());
		}
	}

	public String getCadena() {
		return cadena==null?"":cadena;
	}

	public void setCadena(String cadena) {
		this.cadena = cadena;
	}

}
```

Un validador ha de implementar `IPropertyValidator (1)`, esto le obliga a tener un método `validate()` en donde se ejecuta la validación de la propiedad. Los argumentos del método `validate()` son:

(2)`Messages errores`: Un objeto de tipo `Messages` que representa un conjunto de mensajes (una especie de colección inteligente) y es donde podemos añadir los problemas de validación que encontremos.

(3)`Object valor`: El valor a validar.

(4)`String nombreObjeto`: Nombre del objeto al que pertenece la propiedad a validar. Útil para usarlo en los mensajes de error.

(5)`String nombrePropiedad`: Nombre de la propiedad a validar. Útil para usarlo en los mensajes de error.

Como se ve cuando encontramos un error de validación solo tenemos que añadirlo (con `errores.add()`) enviando un identificador de mensaje y los argumentos. Para que este validador produzca un mensaje significativo tenemos que tener en nuestro archivo de mensajes i18n la siguiente entrada:

```
excluir_cadena={0} no puede contener {2} en {1}
```

Si el identificador que se envía no está en el archivo de mensajes, sale tal cual al usuario; pero lo recomendado es siempre usar identificadores del archivo de mensajes.

La validación es satisfactoria si no se añaden mensajes y se supone fallida si se añaden. El sistema recolecta todos los mensajes de todos los validadores antes de grabar y si encuentra los visualiza al usuario y no graba.

El paquete `org.openxava.validators` contiene algunos validadores de uso común.

2.8.8 Validador por defecto *(nuevo en v2.0.3)*

Podemos definir validadores por defecto para las propiedades de cierto tipo o estereotipo. Para esto se usa el archivo *xava/validadores.xml* de nuestro proyecto para definir en él los validadores por defecto.

Por ejemplo, podemos definir en nuestro *xava/validadores.xml* lo siguiente:

```
<validadores>
     <validador-defecto>
           <clase-validador
                 clase="org.openxava.test.validadores.ValidadorNombrePersona"/>
           <para-estereotipo stereotipo="NOMBRE_PERSONA"/>
     </validador-defecto>
</validadores>
```

En este caso estamos asociando el validador `ValidadorNombrePersona` al estereotipo NOMBRE_PERSONA. Ahora si definimos una propiedad como la siguiente:

```
<propiedad nombre="nombre" estereotipo="NOMBRE_PERSONA" requerido="true"/>
```

Esta propiedad será validada usando `ValidadorNombrePersona` aunque la propiedad misma no defina ningun validador. `ValidadorNombrePersona` se aplica a todas las propiedades con el estereotipo NOMBRE_PERSONA.

Podemos también asignar validadores por defecto a un tipo.

En el archivo *validadores.xml* podemos definir también los validadores para determinar si un valor requerido está presente (ejecutado cuando usamos `requerido="true"`). Además podemos asignar nombre (alias) a las clases de los validadores.

Podemos aprender más sobre los validadores examinando *OpenXava/xava/validators.xml* y *OpenXavaTest/xava/validators.xml*.

2.9 Referencia (5)

Una referencia hace que desde una entidad o agregado se pueda acceder otra entidad o agregado. Una referencia se traduce a código Java como una propiedad (con su *getter* y su *setter*) cuyo tipo es el del modelo al que se referencia. Por ejemplo un `Cliente` puede tener una referencia a su `Comercial`, y así podemos escribir código Java como éste:

```
ICliente cliente = ...
cliente.getComercial().getNombre();
```

para acceder al nombre del comercial de ese cliente.

La sintaxis para definir referencias es:

```
<referencia
      nombre="nombre"                          (1)
      etiqueta="etiqueta"                      (2)
      modelo="modelo"                          (3)
      requerido="true|false"                   (4)
      clave="true|false"                       (5)
      cometido-destino="cometido destino"      (6)
>
      <calculador-valor-defecto .../>          (7)
</referencia>
```

(1)`nombre` (opcional, obligada si no se especifica `modelo`): Es el nombre que tendrá la referencia en Java, por lo tanto ha de seguir la normativa para nombres de miembros Java, entre ellas empezar por minúscula. Si no especificamos `nombre` asume el nombre del modelo pero en con la primera letra minúscula. Se desaconseja usar subrayado (_).

(2)`etiqueta` (opcional): Etiqueta que se mostrará al usuario final. Es **mucho mejor** usar los archivos i18n.

(3)`modelo` (opcional, obligada si no se especifica `nombre`): Es el nombre del modelo a referenciar. Puede ser el nombre de otro componente, en cuyo caso es una referencia a entidad, o el nombre de un agregado del componente en el que estamos. Si no especificamos `modelo` asume el `nombre` de la referencia pero con la primera letra en mayúscula.

(4)`requerido` (opcional): Indica si la referencia es requerida. Al grabar OpenXava comprobará si las referencias requeridas están presentes, si no lo están no se producirá la grabación y se devolverá una lista de errores de validación.

(5)`clave` (opcional): Para indicar si la referencia forma parte de la clave. La combinación de propiedades y referencias clave se debe mapear a un conjunto de campos en la base de datos que no tengan valores repetidos, típicamente con la clave primaria.

(6)`cometido-destino` (opcional): Usado solo en referencia dentro de colecciones. Ver más adelante.

(7)`calculador-valor-defecto` (uno, opcional): Para implementar la lógica para calcular el valor inicial de la referencia. Este calculador ha de devolver el valor de la clave, que puede ser un dato simple (solo si la clave del objeto referenciado es simple) o un objeto clave (un objeto especial que envuelve la clave primaria y que genera OpenXava).

Un pequeño ejemplo de uso de referencias:

```
<referencia modelo="Direccion" requerido="true"/>                    (1)
<referencia nombre="comercial"/>                                     (2)
```

```
<referencia nombre="comercialAlternativo" model="Comercial"/>     (3)
```

(1)Una referencia a un agregado llamado `Direccion`, el nombre de la referencia será `direccion`.

(2)Una referencia a la entidad del componente `Comercial`. Se deduce el modelo a partir del nombre.

(3)Una referencia llamada `comercialAlternativo` a la entidad del componente `Comercial`.

Suponiendo que esto está en un componente llamado `Cliente`, podriamos escribir:

```
ICliente cliente = ...
Direccion direccion = cliente.getDireccion();
IComercial comercial = cliente.getComercial();
IComercial comercialAlternativo = cliente.getComercialAlternativo();
```

2.9.1 Calculador valor por defecto para una referencia

En una referencia `<calculador-valor-defecto/>` funciona como en una propiedad, solo que hay que devolver el valor de la clave de la referencia, y no se admite `al-crear="true"`.

Por ejemplo, en el caso de una referencia con clave simple podemos poner:

```
<referencia nombre="familia" modelo="Familia2" requerido="true">
     <calculador-valor-defecto clase="org.openxava.calculators.IntegerCalculator">
           <poner propiedad="value" valor="2"/>
     </calculador-valor-defecto >
</referencia>
```

El método `calculate()` de este calculador es:

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

Como se puede ver se devuelve un entero, es decir, el valor para familia por defecto es la familia cuyo código es el 2.

En el caso de clave compuesta sería así:

```
<referencia nombre="almacen" modelo="Almacen">
     <calculador-valor-defecto
           clase="org.openxava.test.calculadores.CalculadorDefectoAlmacen"/>
</referencia>
```

Y el código del calculador:

```
package org.openxava.test.calculadores;

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

```
/**
 * @author Javier Paniza
 */
public class CalculadorDefectoAlmacen implements ICalculator {

     public Object calculate() throws Exception {
            Almacen clave = new Almacen();
            clave.setCodigo(4);
            clave.setCodigoZona(4);
            return clave; // Funciona con POJOs y EJB2
            // return new AlmacenKey(new Integer(4), 4); // Funciona solo con EJB2
     }

}
```

Devuelve un objeto de tipo `Almacen` (o `AlmacenKey`, si usamos solo EJB).

2.10 Colección (6)

Con `<coleccion/>` definimos una colección de referencias a entidades o agregados. Esto se traduce en una propiedad Java que devuelve `java.util.Collection`.

Aquí la sintaxis para definir una colección:

```
<coleccion
     nombre="nombre"                 (1)
     etiqueta="etiqueta"             (2)
     minimo="N"                      (3)
     maximo="N"                      (4)   nuevo en v2.0.3
>
     <referencia ... />              (5)
     <condicion ... />               (6)
     <orden ... />                   (7)
     <calculador ... />              (8)
     <calculador-posborrar ... />    (9)
</coleccion>
```

(1)`nombre` (obligado): Es el nombre que tendrá la colección en Java, por lo tanto ha de seguir la normativa para nombres de miembro Java, entre ellas empezar por minúscula. Se desaconseja usar subrayado (_).

(2)`etiqueta` (opcional): Etiqueta que se mostrará al usuario final. Es **mucho mejor** usar los archivos *i18n*.

(3)`minimo` (opcional): Indica el número mínimo de elementos esperados. Esto se valida antes de grabar.

(4)`maximo` (opcional): *(nuevo en v2.0.3)* Indica el número máximo de elementos esperados.

(5)`referencia` (obligada): Con la sintaxis vista en el subtema anterior.

(6)`condicion` (opcional): Para restringir los elementos que aparecen en la colección.

(7)`orden` (opcional): Para que los elementos de la colección aparezcan en un determinado orden.

(8)`calculador` (opcional): Permite especificar nuestra propia lógica Java para generar la colección. Si se especifica `calculador` no se puede poner ni `condicion` ni `orden`.

(9)`calculador-posborrar` (opcional): Permite ejecutar cierta lógica de negocio justo después de haber eliminado un elemento de la colección.

Vamos a ver algunos ejemplos. Empecemos por uno simple:

```
<coleccion nombre="albaranes">
     <referencia modelo="Albaran"/>
</coleccion>
```

Si ponemos esto dentro de una `Factura`, estamos definiendo una colección de los `albaranes` asociados a esa `Factura`. La forma de relacionarlo se hace en la parte del mapeo objeto-relacional, algo que se verá en el capítulo 6.

Ahora podemos escribir código como este:

```
IFactura factura = ...
for (Iterator it = factura.getAlbaranes().iterator(); it.hasNext();) {
     IAlbaran albaran = (IAlbaran) it.next();
     albaran.hacerAlgo();
}
```

Para hacer algo con todos los albaranes asociados a una factura.

Vamos a ver otro ejemplo más complejo, también dentro de `Factura`:

```
<coleccion nombre="lineas" minimo="1">        (1)
     <referencia modelo="LineaFactura"/>
     <orden>${tipoServicio} desc</orden>      (2)
     <calculador-posborrar                    (3)
           clase="org.openxava.test.calculadores.CalculadorPosborrarLineaFactura"/>
</coleccion>
```

En este caso tenemos una colección de agregados, las líneas de las facturas. La diferencia entre una colección de agregados y una de referencia, es que cuando se borrar la entidad principal los elementos de las colecciones de agregados también se borran. Esto es al borrar una factura sus líneas se borran automáticamente.

(1)La restricción de `minimo="1"` hace que sea obligado que haya al menos una línea para que la factura sea válida.

(2)Con `orden` obligamos a que las `lineas` se devuelvan ordenadas por `tipoServicio`.

(3)Con `calculador-posborrar` indicamos un calculador a ejecutar justo después de borrar un

elemento de la colección. Veamos el código de este calculador:

```
package org.openxava.test.calculadores;

import java.rmi.*;

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

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

	private IFactura factura;

	public Object calculate() throws Exception {
		factura.setComentario(factura.getComentario() + "DETALLE BORRADO");
		return null;
	}

	public void setModel(Object modelo) throws RemoteException {
		this.factura = (IFactura) modelo;
	}

}
```

Como se ve es un calculador normal y corriente, como el que hemos visto que se puede asignar a una propiedad calculada. Lo único que hay que tener en cuenta es que el calculador se aplica a la entidad contenedora (es este caso a `Factura`) y no al elemento de la colección. Es decir, si nuestro calculador implementa `IModelCalculator` recibirá una `Factura` y no una `LineaFactura`. Esto es lógico porque como se ejecuta después de borrar la línea esa línea ya no existe.

Tenemos libertad completa para definir como se obtienen los datos de una colección, con `condicion` podemos sobreescribir la condición por defecto que genera OpenXava:

```
<!-- Otros transportistas del mismo almacén -->
<coleccion nombre="compañeros">
	<referencia modelo="Transportista"/>
	<condicion>
		${almacen.codigoZona} = ${this.almacen.codigoZona} AND
		${almacen.numero} = ${this.almacen.numero} AND
		NOT (${numero} = ${this.numero})
	</condicion>
</coleccion>
```

Si ponemos esta colección dentro de `Transportista`, podemos obtener todos los transportista del mismo almacén menos él mismo, es decir, la lista de sus compañeros. Es de notar como podemos usar `this` en la condición para referenciar al valor de una propiedad del objeto actual.

Si con esto no tenemos suficiente, podemos escribir completamente la lógica que devuelve la colección. La colección anterior también se podría haber definido así:

```
<!-- Lo mismo que 'compañeros' pero implementado con un calculador  -->
<coleccion nombre="compañerosCalculados">
     <referencia modelo="Transportista"/>
     <calculador
           clase="org.openxava.test.calculadores.CalculadorCompañerosTransportista"/>
</coleccion>
```

Y aquí el código del calculador:

```
package org.openxava.test.calculadores;

import java.rmi.*;

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

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

     private ITransportista transportista;

     public Object calculate() throws Exception {
           // Usando Hibernate
           int codigoZonaAlmacen = transportista.getAlmacen().getCodigoZona();
           int codigoAlmacen = transportista.getAlmacen().getCodigo();
           Session sesion = XHibernate.getSession();
           Query query = sesion.createQuery("from Transportista as o where " +
                       "o.almacen.codigoZona = :zonaAlmacen AND " +
                       "o.almacen.codigo = :codigoAlmacen AND " +
                       "NOT (o.codigo = :codigo)");
           query.setInteger("zonaAlmacen", codigoZonaAlmacen);
           query.setInteger("codigoAlmacen", codigoAlmacen);
           query.setInteger("codigo", transportista.getCodigo());
           return query.list();
```

```
            /* Usando EJB3 JPA
            EntityManager manager = XPersistence.getManager();
            Query query = manager.createQuery("from Transportista o where " +
                  "o.almacen.codigoZona = :zonaAlmacen AND " +
                  "o.almacen.codigo = :codigoAlmacen  AND " +
                  "NOT (o.codigo = :codigo) ");
            query.setParameter("zonaAlmacen", codigoZonaAlmacen);
            query.setParameter("codigoAlmacen", codigoAlmacen);
            query.setParameter("codigo", transportista.getCodigo());
            return query.getResultList();
            */

            /* Usando EJB2
            return TransportistaUtil.getHome().findCompañerosOfTransportista(
                  transportista.getAlmacenKey().getCodigoZona(),
                  transportista.getAlmacenKey().get_Codigo(),
                  new Integer(transportista.getCodigo())
            );
            */
      }

      public void setModel(Object modelo) throws RemoteException {
            transportista = (ITransportista) modelo;
      }

}
```

Como se ve es un calculador convencional. Obviamente ha de devolver una `java.util.Collection` cuyos elementos sean de tipo `ITransportista`.

Las referencias de las colecciones se asumen bidireccionales, esto quiere decir que si en un `Comercial` tengo una colección `clientes`, en `Cliente` tengo que tener una referencia a `Comercial`. Pero si en `Cliente` tengo más de una referencia a `Comercial` (por ejemplo, `comercial` y `comercialAlternativo`) OpenXava no sabe cual escoger, para eso tenemos el atributo `cometido-destino` de `referencia`. En este caso pondríamos:

```
<coleccion nombre="clientes">
      <referencia modelo="Cliente" cometido-destino="comercial"/>
</coleccion>
```

Para indicar que es la referencia `comercial` y no `comercialAlternativo` la que vamos a usar para esta colección.

En el caso de colección de referencias a entidad tenemos que definir nosotros las referencia en la otra parte, pero en el caso de colección de referencias a agregados no es necesario, porque en los agregados se genera automáticamente una referencia a su contenedor.

2.11 Método (7)

Con `<metodo/>` podemos definir un método qué será incluido en el código generado.

La sintaxis para definir un método es:

```
<metodo
     nombre="nombre"                  (1)
     tipo="tipo"                      (2)
     argumentos="argumentos"          (3)
     excepciones="excepciones"        (4)
>
     <calculador ... />               (5)
</metodo>
```

(1)`nombre` (obligado): Es el nombre que tendrá el método en Java, por lo tanto ha de seguir la normativa para miembros Java, entre ellas empezar por minúscula.

(2)`tipo` (opcional, por defecto `void`): Corresponde a un tipo Java. Todos los tipos válidos como tipo de retorno de un método en Java son válidos aquí.

(3)`argumentos` (opcional): Lista de argumentos del método en formato Java.

(4)`excepciones` (opcional): Lista de excepciones que puede lanzar el método en formato Java.

(5)`calculador` (obligado): Implementa la lógica que ejecuta el método.

Definir un método es sencillo:

```
<metodo nombre="incrementarPrecio">
     <calculador clase="org.openxava.test.calculadores.CalculadorIncrementarPrecio"/>
</metodo>
```

E implementarlo depende de lo que se quiera hacer. En este caso:

```
package org.openxava.test.calculadores;

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

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

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

     private IProducto producto;
```

```
	public Object calculate() throws Exception {
		producto.setPrecioUnitario(	// (1)
			producto.getPrecioUnitario().
				multiply(new BigDecimal("1.02")).setScale(2));
		return null;			// (2)
	}

	public void setModel(Object modelo) throws RemoteException {
		this.producto = (IProducto) modelo;
	}

}
```

Todo lo que se dijo para los calculadores cuando se habló de las propiedades aplica a los métodos también, con los siguientes matices:

(1)Un calculador para un método tiene autoridad moral para cambiar el estado del objeto.

(2)Si el tipo que se devuelve es `void` hay que acabar con un `return null`.

Ahora podemos usar el método de la forma esperada:

```
IProducto producto = ...
producto.setPrecioUnitario(new BigDecimal("100"));
producto.incrementarPrecio();
BigDecimal nuevoPrecio = producto.getPrecioUnitario();
```

Y en `nuevoPrecio` tendremos 102.

Otro ejemplo de método, ahora un poco más complejo:

```
<metodo nombre="getPrecio" tipo="BigDecimal"
	argumentos="String pais, BigDecimal tarifa"
	excepciones="ProductoException, PrecioException">
	<calculador clase="org.openxava.test.calculadores.CalculadorPrecioExportacion">
		<poner propiedad="euros" desde="precioUnitario"/>
	</calculador>
</metodo>
```

En este caso es de notar que tanto en argumentos como en excepciones se usa el formato Java, ya que lo que se pone ahí es insertado directamente en el código generado.

El calculador quedaría de la siguiente forma:

```
package org.openxava.test.calculadores;

import java.math.*;
```

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

/**
 * @author Javier Paniza
 */

public class CalculadorPrecioExportacion implements ICalculator {

	private BigDecimal euros;
	private String pais;
	private BigDecimal tarifa;

	public Object calculate() throws Exception {
		if ("España".equals(pais) || "Guatemala".equals(pais)) {
			return euros.add(tarifa);
		}
		else {
			throw new PriceException("Pais no registrado");
		}
	}

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

	public BigDecimal getTarifa() {
		return tarifa;
	}
	public void setTarifa(BigDecimal decimal) {
		tarifa = decimal;
	}

	public String getPais() {
		return pais;
	}
	public void setPais(String string) {
		pais = string;
```

```
    }
}
```

Cada argumento se asigna a una propiedad del mismo nombre en el calculador, es decir el valor del primer argumento, `pais`, se asigna a la propiedad `pais`, y el valor del segundo, `tarifa`, a la propiedad `tarifa`. Y, por supuesto, se pueden configurar valores a otras propiedades de calculador con `<poner/>` como es usual para los calculadores.

Y para usar el método:

```
IProducto producto = ...
BigDecimal precio = producto.getPrecio("España", new BigDecimal("100")); // funciona
producto.getPrecio("El Puig", new BigDecimal("100")); // lanza PrecioException
```

Los métodos son la salsa de los objetos, sin ellos solo serían caparazones tontos alrededor de los datos. Cuando sea posible es mejor poner la lógica de negocio en los métodos (capa del modelo) que en las acciones (capa del controlador).

2.12 Buscador (8)

Un buscador es un método especial que nos permite encontrar un objeto o una colección de objetos que cumplen un solo criterio. En la versión POJO un buscador es un método estático generado en la clase POJO. En la versión EJB2 un buscador corresponde con un *finder* del *home*.

La sintaxis para definir buscadores es:

```
<buscador
    nombre="nombre"                 (1)
    argumentos="argumentos"         (2)
    coleccion="(true|false)"        (3)
>
    <condicion ... />               (4)
    <orden ... />                   (5)
</buscador>
```

(1)`nombre` (obligado): Es el nombre que tendrá el método *finder* en Java, por lo tanto ha de seguir la normativa para miembros Java, entre ellas empezar por minúscula.

(2)`argumentos` (obligado): Lista de argumentos del método en formato Java. Lo más aconsejable es usar tipos de datos simples.

(3)`coleccion` (opcional, por defecto `false`): Indica si el resultado va a ser un solo objeto o una colección.

(4)`condicion` (opcional): Una condición con sintaxis SQL/EJBQL en la que podemos usar los nombres de las propiedades entre `${}`.

(5)`orden` (opcional): Una ordenación con sintaxis SQL/EJBQL en la que podemos usar los nombres de las propiedades entre `${}`.

Algunos ejemplos:

```
<buscador nombre="byCodigo" argumentos="int codigo">
      <condicion>${codigo} = {0}</condicion>
</buscador>

<buscador nombre="byNombreLike" argumentos="String nombre" coleccion="true">
      <condicion>${nombre} like {0}</condicion>
      <orden>${nombre} desc</orden>
</buscador>

<buscador
      nombre="byNombreLikeYRelacionConComercial"
      argumentos="String nombre, String relacionConComercial"
      coleccion="true">
      <condicion>${nombre} like {0} and ${relacionConComercial} = {1}</condicion>
      <orden>${nombre} desc</orden>
</buscador>

<buscador nombre="normales" argumentos="" coleccion="true">
      <condicion>${tipo} = 1</condicion>
</buscador>

<buscador nombre="fijos" argumentos="" coleccion="true">
      <condicion>${tipo} = 2</condicion>
</buscador>

<buscador nombre="todos" argumentos="" coleccion="true"/>
```

Esto genera un conjunto de métodos *finders* disponibles desde la clase POJO y el *home* EJB2 correspondiente. Estos métodos se pueden usar así:

```
// POJO, tanto JPA como Hibernate
ICliente cliente = Cliente.findByCodigo(8);
Collection javieres = Cliente.findByNombreLike("%JAVI%");

// EJB2
ICliente cliente = ClienteUtil.getHome().findByCodigo(8);
Collection javieres = ClienteUtil.getHome().findByNombreLike("%JAVI%");
```

2.13 Calculador poscrear (9)

Con `<calculador-poscrear/>` podemos indicar que se ejecute cierta lógica justo después de crear el objeto como persistente.

Su sintaxis es:

```
<calculador-poscrear
     clase="clase">            (1)
     <poner ... /> ...         (2)
</calculador-poscrear>
```

(1)`clase` (obligado): Clase del calculador. Un calculador que implemente `ICalculator` u alguno de sus derivados.

(2)`poner` (varios, opcional): Para establecer valor a las propiedades del calculador antes de ejecutarse.

Un ejemplo sencillo sería:

```
<calculador-poscrear
     clase="org.openxava.test.calculadores.CalculadorPoscrearTipoAlbaran">
     <poner propiedad="sufijo" valor="CREADO"/>
</calculador-poscrear>
```

Y ahora la clase del calculador:

```
package org.openxava.test.calculadores;

import java.rmi.*;

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

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

     private ITipoAlbaran tipoAlbaran;
     private String sufijo;

     public Object calculate() throws Exception {
            tipoAlbaran.setDescripcion(tipoAlbaran.getDescripcion() + " " + sufijo);
            return null;
     }

     public void setModel(Object modelo) throws RemoteException {
            tipoAlbaran = (ITipoAlbaran) modelo;
     }

     public String getSufijo() {
```

```
            return sufijo;
      }
      public void setSufijo(String sufijo) {
            this.sufijo = sufijo;
      }

}
```

En este caso cada vez que se graba por primera vez un `TipoAlbaran`, justo después se añade un sufijo a su descripción.

Como se ve es exactamente igual que cualquier otro calculador (como para propiedades calculadas o métodos) solo que este se ejecuta después de crear.

2.14 Calculador posmodificar (11)

Con `<calculador-posmodificar/>` podemos indicar que se ejecute cierta lógica justo después de modificar un objeto y justo antes de actualizar su contenido en la base de dato, esto es justo antes de hacer el UPDATE.

Su sintaxis es:

```
<calculador-posmodificar
      clase="clase">              (1)
      <poner ... /> ...           (2)
</calculador-posmodificar>
```

(3)`clase` (obligado): Clase del calculador. Un calculador que implemente `ICalculator` u alguno de sus derivados.

(4)`poner` (varios, opcional): Para establecer valor a las propiedades del calculador antes de ejecutarse.

Un ejemplo sencillo sería:

```
<calculador-posmodificar
      clase="org.openxava.test.calculators.CalculadorPosmodificarTipoAlbaran"/>
```

Y ahora la clase del calculador:

```
package org.openxava.test.calculadores;

import java.rmi.*;

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

/**
 * @author Javier Paniza
```

```
 */

public class CalculadorPosmodificarTipoAlbaran implements IModelCalculator {

	private ITipoAlbaran tipoAlbaran;

	public Object calculate() throws Exception {
		tipoAlbaran.setDescripcion(tipoAlbaran.getDescripcion() + " MODIFICADO");
		return null;
	}

	public void setModel(Object modelo) throws RemoteException {
		tipoAlbaran = (ITipoAlbaran) modelo;
	}

}
```

En este caso cada vez que se modifica un `TipoAlbaran` se añade un sufijo a su descripción.

Como se ve es exactamente igual que cualquier otro calculador (como para propiedades calculadas o métodos) solo que este se ejecuta después de modificar.

2.15 Calculadores poscargar y preborrar (10, 12)

La sintaxis y comportamiento de los calculadores poscargar y preborrar son iguales que en el caso de los calculadores poscrear y posmodificar.

2.16 Validador (13)

Este validador permite poner una validación a nivel de modelo. Cuando necesitamos hacer una validación sobre varias propiedades del modelo, y esta validación no corresponde lógicamente a ninguna de ellas se puede usar este tipo de validación.

Su sintaxis es:

```
<validador
	clase="validador"                (1)
	nombre="nombre"                  (2)
	solo-al-crear="true|false" (3)
>
	<poner ... /> ...                (4)
</validador>
```

(1)`clase` (opcional, obligada si no se especifica nombre): Clase que implementa la validación. Ha de ser del tipo `IValidator`.

(2)`nombre` (opcional, obligada si no se especifica clase): Este nombre es el que tenga el validador en el archivos *xava/validators.xml* o *xava/validadores.xml*, del proyecto OpenXava o de nuestro

propio proyecto.

(3)`solo-al-crear` (opcional): Si `true` el validador es ejecutado solo cuando estamos creando un objeto nuevo, no cuando modificamos uno existente. El valor por defecto es `false`.

(4)`poner` (varios, opcional): Para establecer valor a las propiedades del validador antes de ejecutarse.

Un ejemplo:

```
<validador clase="org.openxava.test.validadores.ValidadorProductoBarato">
     <poner propiedad="limite" valor="100"/>
     <poner propiedad="descripcion"/>
     <poner propiedad="precioUnitario"/>
</validador>
```

Y el código del validador:

```
package org.openxava.test.validadores;

import java.math.*;

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

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

     private int limite;
     private BigDecimal precioUnitario;
     private String descripcion;

     public void validate(Messages errores) {                           // (2)
            if (getDescripcion().indexOf("CHEAP") >= 0
                   getDescripcion().indexOf("BARATO") >= 0
                   getDescripcion().indexOf("BARATA") >= 0) {
                   if (getLimiteBd().compareTo(getPrecioUnitario()) < 0) {
                          errores.add("producto_barato", getLimiteBd()); // (3)
                   }
            }
     }

     public BigDecimal getPrecioUnitario() {
            return precioUnitario;
```

```
	}

	public void setPrecioUnitario(BigDecimal decimal) {
		precioUnitario = decimal;
	}

	public String getDescripcion() {
		return descripcion==null?"":descripcion;
	}

	public void setDescripcion(String string) {
		descripcion = string;
	}

	public int getLimite() {
		return limite;
	}

	public void setLimite(int i) {
		limite = i;
	}

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

}
```

Este validador ha de implementar `IValidator` (1), lo que le obliga a tener un método `validate(Messages messages)` (2). En este método solo hay que añadir identificadores de mensajes de error (3) (cuyos textos estarán en los archivos *i18n*), si en el proceso de validación (es decir en la ejecución de todos los validadores) hubiese al menos un mensaje de error, OpenXava no graba la información y visualiza los mensajes al usuario.

En este caso vemos como se accede a `descripcion` y `precioUnitario`, por eso la validación se pone a nivel de módelo y no a nivel de propiedad individual, porque abarca más de una propiedad.

2.17 Validador borrar (14)

El `<validador-borrar/>` también es un validador a nivel de modelo, la diferencia es que se ejecuta antes de borrar el objeto, y tiene la posibilidad de vetar el borrado.

Su sintaxis es:

```
<validador-borrar
	clase="validador"          (1)
```

```
      nombre="nombre"            (2)
>
      <poner ... /> ...          (3)
</validador-borrar>
```

(1)`clase` (opcional, obligada si no se especifica `nombre`): Clase que implementa la validación. Ha de ser del tipo `IRemoveValidator`.

(2)`nombre` (opcional, obligada si no se especifica `clase`): Este nombre es el que tenga el validador en el archivos *xava/validators.xml* o *xava/validadores.xml*, del proyecto OpenXava o de nuestro propio proyecto.

(3)`poner` (varios, opcional): Para establecer valor a las propiedades del calculador antes de ejecutarse.

Un ejemplo puede ser:

```
<validador-borrar
      clase="org.openxava.test.validadores.ValidadorBorrarTipoAlbaran"/>
```

Y el validador:

```
package org.openxava.test.validadores;

import java.util.*;

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

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

      private ITipoAlbaran tipoAlbaran;

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

      public void validate(Messages errores) throws Exception {
            if (!tipoAlbaran .getAlbaranes().isEmpty()) {
                  errores.add("no_borrar_tipo_albaran_si_albaranes");   // (3)
            }
      }
```

```
}
```

Como se ve tiene que implementar `IRemoveValidator` (1) lo que le obliga a tener un método `setEntity()` (2) con el recibirá el objeto que va a ser borrado. Si hay algún error de validación se añade al objeto de tipo `Messages` enviado a `validate()` (3). Si después de ejecutar todas las validaciones el OpenXava detecta al menos 1 error de validación no realizará el borrado del objeto y enviará la lista de mensajes al usuario.

En este caso si se comprueba si hay albaranes que usen este tipo de albarán antes de poder borrarlo.

2.18 Agregado

La sintaxis de un agregado es como sigue:

```
<agregado nombre="agregado">           (1)
     <bean clase="claseBean"/>         (2)
     <ejb ... />                       (3)
     <implementa .../>
     <propiedad .../> ...
     <referencia .../> ...
     <coleccion .../> ...
     <metodo .../> ...
     <buscador .../> ...
     <calculador-poscrear .../> ...
     <calculador-posmodificar .../> ...
     <validador .../> ...
     <validador-borrar .../> ...
</agregado>
```

(1)`nombre` (obligado): Cada agregado tiene que tener un nombre único. La normativa para poner el nombre es la misma que para un nombre de clase Java, es decir, empieza por mayúscula y cada palabra nueva también.

(2)`bean` (uno, opcional): Permite especificar una clase escrita por nosotros para implementar el agregado. La clase ha de ser un JavaBean, es decir una clase de Java normal y corriente con *getters* y *setters* para las propiedades. Normalmente no se suele usar ya que es mucho mejor que OpenXava genere el código por nosotros.

(3)`ejb` (uno, opcional): Permite usar un EJB2 existente para implementar un agregado. Esto se usa en el caso de querer tener una colección de agregados. Normalmente no se suele usar ya que es mucho mejor que OpenXava genere el código por nosotros.

En un componente puede haber cuantos agregados deseemos. Y podemos referenciarlos desde la entidad o desde otro agregado.

2.18.1 Referencia a agregado

El primer ejemplo es un agregado `Direccion` que es referenciado desde la entidad principal.

En la entidad principal pondremos:

```
<referencia nombre="direccion" modelo="Direccion" requerido="true"/>
```

Y a nivel de componente definiremos el agregado.

```
<agregado nombre="Direccion">
     <implementa interfaz="org.openxava.test.ejb.IConMunicipio"/>              (1)
     <propiedad nombre="calle" tipo="String" longitud="30" requerido="true"/>
     <propiedad nombre="codigoPostal" tipo="int" longitud="5" requerido="true"/>
     <propiedad nombre="municipio" tipo="String" longitud="20" requerido="true"/>
     <referencia nombre="provincia" requerido="true"/>                         (2)
</agregado>
```

Como se ve un agregado puede implementar una interfaz (1) y contener referencias (2), entre otras cosas, en realidad todo lo que se soporta en `<entidad/>` se soporta aquí.

El código resultante se puede usar así, para leer:

```
ICliente cliente = ...
Direccion direccion = cliente.getDireccion();
direccion.getCalle(); // para obtener el valor
```

O así para establecer una nueva dirección

```
// para establecer una nueva dirección
Direccion direccion = new Direccion(); // es un JavaBean, nunca un EJB2
direccion.setCalle(“Mi calle”);
direccion.setCodigoPostal(46001);
direccion.setMunicipio(“Valencia”);
direccion.setProvincia(provincia);
cliente.setDireccion(direccion);
```

En este caso que tenemos una referencia simple, el código generado es un simple JavaBean, cuyo ciclo de vida esta asociado a su objeto contenedor, es decir, la `Direccion` se borrará y creará junto al `Cliente`, jamas tendrá vida propia ni podrá ser compartida por otro `Cliente`.

2.18.2 Colección de agregados

Ahora un ejemplo de una colección de agregados. En la entidad principal (por ejemplo de `Factura`) podemos poner:

```
<coleccion nombre="lineas" minimo="1">
     <referencia modelo="LineaFactura"/>
</coleccion>
```

Y definimos el agregado `LineaFactura`:

```
<agregado nombre="LineaFactura">
```

```
    <propiedad nombre="oid" tipo="String" clave="true" oculta="true">
        <calculador-valor-defecto
            clase="org.openxava.test.calculadores.CalculadorOidLineaFactura"
            al-crear="true"/>
    </propiedad>
    <propiedad nombre="tipoServicio">
        <valores-posibles>
            <valor-posible valor="especial"/>
            <valor-posible valor="urgente"/>
        </valores-posibles>
    </propiedad>
    <propiedad nombre="cantidad" tipo="int"
        longitud="4" requerido="true"/>
    <propiedad nombre="precioUnitario"
        estereotipo="DINERO" requerido="true"/>
    <propiedad nombre="importe"
        estereotype="DINERO">
        <calculador
            clase="org.openxava.test.calculadores.CalculadorImporteLinea">
            <poner propiedad="precioUnitario"/>
            <poner propiedad="cantidad"/>
        </calculador>
    </propiedad>
    <referencia modelo="Producto" requerido="true"/>
    <propiedad nombre="fechaEntrega" tipo="java.util.Date">
        <calculador-valor-defecto
            clase="org.openxava.calculators.CurrentDateCalculator"/>
    </propiedad>
    <referencia nombre="vendidoPor" modelo="Comercial"/>
    <propiedad nombre="observaciones" estereotipo="MEMO"/>

    <validador clase="org.openxava.test.validadores.ValidadorLineaFactura">
        <poner propiedad="factura"/>
        <poner propiedad="oid"/>
        <poner propiedad="producto"/>
        <poner propiedad="precioUnitario"/>
    </validador>

</agregado>
```

Como podemos ver un agregado es tan complejo como una entidad, con calculadores, validadores, referencias y todo lo que queramos. En el caso de un agregado usado en una colección, como este caso, automáticamente se añade una referencia al contenedor, es decir, aunque no lo hayamos

definido, `LineaFactura` tiene una referencia a `Factura`.

En el código generado nos encontraremos en `Factura` una colección `LineaFactura`. La diferencia entre una colección de referencias y una de agregados está en que al borrar la factura se borraran sus líneas asociadas, y también en el estilo de la interfaz gráfica (la interfaz gráfica se ve en el capítulo 4).

(Nuevo en la guía de referencia v2.1.1: Explicación de las referencias implícitas) Cada agregado tiene una referencia implícita a su modelo contenedor. Es decir, `LineaFactura` tiene una referencia llamada `factura`, incluso si la referencia no está declarada (aunque puede declararse, de forma opcional, por propósito de refinamiento, por ejemplo, para hacerla clave). Esta referencia puede usarse en el código Java, como sigue:

```
LineaFactura linea = ... ;
linea.getFactura(); // Para obtener el padre
```

O puede usarse en el código XML, de esta forma:

```
<propiedad nombre="oid" tipo="String" clave="true" oculta="true">
     <calculador-valor-defecto
           clase="org.openxava.test.calculadores.CalculadorLineaFacturaOid"
           al-crear="true">
           <poner propiedad="añoFactura" desde="factura.año"/>          (1)
           <poner propiedad="numeroFactura" desde="factura.numeror"/>   (1)
     </calculador-valor-defecto>
</propiedad>
```

En este caso usamos `factura` en el atributo `desde` (1) aunque `factura` no está declarada en `LineaFactura`. *Nuevo en v2.1.1: usar referencia a propiedades clave del modelo padre en el atributo desde (es decir, desde="factura.año")*

3 Vista

OpenXava genera a partir del modelo una interfaz gráfica de usuario por defecto. Para muchos casos sencillos esto es suficiente, pero muchas veces es necesario modelar con más precisión la forma de la interfaz de usuario o vista. En este capítulo vamos a ver cómo.

La sintaxis para definir una vista es:

```
<vista
      nombre="nombre"                  (1)
      etiqueta="etiqueta"              (2)
      modelo="modelo"                  (3)
      miembros="miembros"              (4)
>
      <propiedad ... /> ...            (5)
      <vista-propiedad ... /> ...      (6)
      <vista-referencia ... /> ...     (7)
      <vista-coleccion ... /> ...      (8)
      <miembros ... /> ...             (9)
</vista>
```

(1)`nombre` (opcional): El nombre identifica a la vista, y puede ser usado desde otro lugares de OpenXava (por ejemplo desde *aplicacion.xml*) o desde otro componente. Si no se pone nombre se asume que es la vista por defecto, es decir la forma normal de visualizar el objeto.

(2)`etiqueta` (opcional): Etiqueta que se muestra al usuario si es necesario, cuando se visualiza la vista. Es **mucho mejor** usar los archivos *i18n*.

(3)`modelo` (opcional): Si es una vista de un agregado de este componente se pone aquí el nombre del agregado. Si no se pone nada se supone que es una vista de la entidad.

(4)`miembros` (opcional): Lista de miembros a visualizar. Por defecto visualiza todos los miembros no ocultos en el orden en que están declarados en el modelo. Este atributo es excluyente con el elemento `miembros` visto más adelante.

(5)`propiedad` (varios, opcional): Permite definir propiedades de la vista, es decir, información que se puede visualizar de cara al usuario y tratar programáticamente, pero no forma parte del modelo.

(6)`vista-propiedad` (varias, opcional): Para definir la forma en que queremos que se visualice cierta propiedad.

(7)`vista-referencia` (varias, opcional): Para definir la forma en que queremos que se visualice cierta referencia.

(8)`vista-coleccion` (varias, opcional): Para definir la forma en que queremos que se visualice cierta colección.

(9)`miembros` (uno, opcional): Indica los miembros que tienen que salir y como tienen que estar dispuestos en la interfaz gráfica. Es excluyente con el atributo `miembros`. Dentro de `miembros` podemos usar los elementos `seccion` y `grupo` (ver sección 4.1) para indicar la disposición; o el elemento `accion` *(desde v2.0.3)* para mostrar un vínculo asociado a una acción propia dentro de

la vista (ver sección 4.6).

3.1 Disposición

Por defecto (es decir si no definimos ni siquiera el elemento `<view/>` en nuestro componente) se visualizan todos los miembros del objeto en el orden en que están en el modelo, y se disponen uno debajo del otro.

Por ejemplo, un modelo así:

```
<entidad>
     <propiedad nombre="codigoZona" clave="true"
            longitud="3" requerido="true" tipo="int"/>
     <propiedad nombre="codigoOficina" clave="true"
            longitud="3" requerido="true" tipo="int"/>
     <propiedad nombre="codigo" clave="true"
            longitud="3" requerido="true" tipo="int"/>
     <propiedad nombre="nombre" tipo="String"
            longitud="40" requerido="true"/>
</entidad>
```

Generaría una vista con este aspecto:

Podemos escoger que miembros queremos que aparezcan y en que orden, con el atributo `miembros`:

```
<vista miembros="codigoZona; codigoOficina; codigo"/>
```

En este caso ya no aparece el `nombre` en la vista.

Los miembros también se pueden especificar mediante el elemento `miembros` que es excluyente con el atributo `miembros`, así:

```
<vista>
     <miembros>
            codigoZona, codigoOficina, codigo;
            nombre
     </miembros>
</vista>
```

Podemos observar como separamos los nombres de miembros con comas y punto y comas, esto nos sirve para indicar la disposición, con la coma el miembro se pone a continuación, y con punto y coma en la línea siguiente, esto es la vista anterior quedaría así:

3.1.1 Grupos

Con los grupos podemos agrupar un conjunto de propiedades relacionadas, y esto tiene un efecto visual:

```
<vista>
	<miembros>
		<grupo nombre="id">
			codigoZona, codigoOficina, codigo
		</grupo>
		; nombre
	</miembros>
</vista>
```

En este caso el resultado sería:

Se puede observar como las tres propiedades puestas en el grupo aparecen dentro de un marquito, y como `nombre` aparece fuera. El punto y coma antes de `nombre` es para que aparezca abajo, si no aparecería a continuación.

Podemos poner varios grupos en una vista:

```
<grupo nombre="cliente">
	tipo;
	nombre;
</grupo>
<grupo nombre="comercial">
	comercial;
	relacionConComercial;
</grupo>
```

En este caso se visualizan uno al lado del otro:

Si queremos que aparezca uno debajo del otro debemos poner un punto y coma después del grupo, como sigue:

```
<grupo nombre="cliente">
      tipo;
      nombre;
</grupo>;
<grupo nombre="comercial">
      comercial;
      relacionConComercial;
</grupo>
```

En este caso se visualizaría así:

Anidar grupos está soportado. Esta interesante característica permite disponer los elementos de la interfaz gráfica de una forma simple y flexible. Por ejemplo, si definimos una vista como ésta:

```
<miembros>
      factura;
      <grupo nombre="datosAlbaran">
            tipo, codigo;
            fecha;
            descripcion;
            envio;
            <grupo nombre="datosTransporte">
```

```
                distancia; vehiculo; modoTransporte; tipoConductor;
            </grupo>
            <grupo nombre="datosEntregadoPor">
                entregadoPor;
                transportista;
                empleado;
            </grupo>
        </grupo>
</miembros>
```

Obtendremos lo siguiente:

Nuevo en v2.0.4: A veces es útil distribuir los miembros alineándolos por columnas, como en una tabla. Por ejemplo, la siguiente vista:

```
<vista nombre="Importes">
    <miembros>
        año, numero;
        <grupo nombre="importes">
            descuentoCliente, descuentoTipoCliente, descuentoAño;
            sumaImportes, porcentajeIVA, iva;
        </grupo>
    </miembros>
</vista>
```

...será visualizada como sigue:

Esto es feo. Sería mejor tener la información alineada por columnas. Podemos definir el grupo de

esta forma:

```
<vista nombre="Importes">
      <miembros>
            año, numero;
            <grupo nombre="importes" alineado-por-columnas="true">       (1)
                  descuentoCliente, descuentoTipoCliente, descuentoAño;
                  sumaImportes, porcentajeIVA, iva;
            </grupo>
      </miembros>
</vista>
```

Y así obtendriamos el siguiente resultado:

Ahora, gracias al atributo `alineado-por-columnas` (1), los miembros están alineado por columnas.

El atributo `alineado-por-columnas` esta disponible tambien para las secciones (ver abajo).

3.1.2 Secciones

Además de en grupo los miembros se pueden organizar en secciones, veamos un ejemplo en el componente `Factura`:

```
<vista>
      <miembros>
            año, numero, fecha, pagada;
            descuentoCliente, descuentoTipoCliente, descuentoAño;
            comentario;
            <seccion nombre="cliente">cliente</seccion>
            <seccion nombre="lineas">lineas</seccion>
            <seccion nombre="importes">sumaImportes; porcentajeIVA; iva</seccion>
            <seccion nombre="albaranes">albaranes</seccion>
      </miembros>
</vista>
```

El resultado visual sería:

Las secciones se convierten en pestañitas que el usuario puede pulsar para ver la información contenida en esa sección. Podemos observar también como en la vista indicamos todo tipo de

miembros (y no solo propiedades), así `cliente` es una referencia, lineas una colección de `agregados` y `albaranes` una colección de referencias a entidad.

Se permiten secciones anidadas (*nuevo en v2.0*). Por ejemplo, podemos definir una vista como ésta:

```
<vista nombre="SeccionesAnidadas">
     <miembros>
           año, numero
           <seccion nombre="cliente">cliente</seccion>
           <seccion nombre="datos>
                 <seccion nombre="lineas">lineas</seccion>
                 <seccion nombre"cantidades">
                       <seccion nombre="iva">porcentajeIVA; iva</seccion>
                       <seccion nombre="sumaImportes">sumaImportesm</seccion>
                 </seccion>
           </seccion>
           <seccion nombre="albaranes">albaranes</seccion>
     </miembros>
</vista>
```

En este caso podemos obtener una interfaz gráfica como esta:

Nuevo en v2.0.4: Al igual que en los grupos, las secciones permiten usar el atributo `alineado-`

`por-columna`, así:

```
<seccion nombre="importes" alineado-por-columnas="true"> ... </seccion>
```

Con el mismo efecto que en el caso de los grupos. Ver la sección 4.1.2.

3.1.3 Filosofía para la disposición

Es de notar tenemos grupos y no marcos y secciones y no pestañas. Porque en las vista de OpenXava intentamos mantener un nivel de abstracción alto, es decir, un grupo es un conjunto de propiedades relacionadas semánticamente, y las secciones nos permite dividir la información en partes cuando tenemos mucha y posiblemente no se pueda visualizar toda a la vez, el que los grupos se representen con marquitos y las secciones con pestañas es una cuestión de implementación, pero el generador del interfaz gráfico podría escoger usar un árbol u otro control gráfico para representar las secciones, por ejemplo.

3.2 Vista propiedad

Con `<vista-propiedad/>` podemos refinar la forma de visualización y comportamiento de una propiedad en la vista:

Tiene esta sintaxis:

```
<vista-propiedad
      propiedad="nombrePropiedad"                          (1)
      etiqueta="etiqueta"                                  (2)
      solo-lectura="true|false"                            (3)
      formato-etiqueta="NORMAL|PEQUENA|SIN_ETIQUETA" (4)
      editor="nombreEditor"                                (5) nuevo en v2.1.3
      longitud-visual="longitud"                           (6) nuevo en v2.2.1
>
      <al-cambiar ... />                                   (7)
      <accion ... /> ...                                   (8)
</vista-propiedad>
```

(1)`propiedad` (obligado): Normalmente el nombre de una propiedad del modelo, aunque también puede ser el nombre de una propiedad propia de la vista.

(2)`etiqueta` (opcional): Para modificar la etiqueta que se sacará en esta vista para esta propiedad. Para esto es **mucho mejor** usar los archivos i18n.

(3)`solo-lectura` (opcional): Si la ponemos a `true` esta propiedad no será nunca editable por el usuario en esta vista. Una alternativa a esto es hacer la propiedad editable/no editable programáticamente usando `org.openxava.view.View`.

(4)`formato-etiqueta` (opcional): Forma en que se visualiza la etiqueta para esta propiedad.

(5)`editor` (opcional): *Nuevo en v2.1.3*. Nombre del editor a usar para visualizar la propiedad en esta vista. El editor tiene que estar declarado en *OpenXava/xava/default-editors.xml* o *xava/editores.xml* de nuestro proyecto.

(6)`longitud-visual` (opcional): *Nuevo en v2.2.1*. La longitud en caracteres del editor en la interfaz

de usuario usado para visualizar esta propiedad. El editor mostrará solo los caracteres indicados con `longitud-visual` pero permite que el usuario introduzca hasta el total de la longitud de la propiedad. Si `longitud-visual` no se especifica se asume el valor de la longitud de la propiedad.

(7)`al-cambiar` (uno, opcional): Acción a realizar cuando cambia el valor de esta propiedad.

(8)`accion` (varias, opcional): Acciones (mostradas como vínculos, botones o imágenes al usuario) asociadas (visualmente) a esta propiedad y que el usuario final puede ejecutar.

3.2.1 Formato de etiqueta

Un ejemplo sencillo para cambiar el formato de la etiqueta:

```
<vista modelo="Direccion">
     <vista-propiedad propiedad="codigoPostal" formato-etiqueta="PEQUENA"/>
</vista>
```

En este caso el código postal lo visualiza así:

El formato `NORMAL` es el que hemos visto hasta ahora (con la etiqueta grande y la izquierda) y el formato `SIN_ETIQUETA` simplemente hace que no salga etiqueta.

3.2.2 Evento de cambio de valor

Si queremos reaccionar al evento de cambio de valor de una propiedad podemos poner:

```
<vista-propiedad propiedad="transportista.codigo">
     <al-cambiar clase="org.openxava.test.acciones.AlCambiarTransportistaEnAlbaran"/>
</vista-propiedad>
```

Podemos observar como la propiedad puede ser calificada, es decir en este caso reaccionamos al cambio del código del transportista (que es una referencia).

El código que se ejecutará será:

```
package org.openxava.test.acciones;

import org.openxava.actions.*;

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

     public void execute() throws Exception {
           if (getNewValue() == null) return;                              // (2)
```

```
            getView().setValue("observaciones",                                    // (3)
                  "El transportista es " + getNewValue());
            addMessage("transportista_cambiado");
      }

}
```

La acción ha implementar `IOnChangePropertyAction` aunque es más cómodo hacer que descienda de `OnChangePropertyBaseAction` (1). Dentro de la acción tenemos disponible `getNewValue()` (2) que proporciona el nuevo valor que ha introducido el usuario, y `getView()` (3)que nos permite acceder programáticamente a la vista (cambiar valores, ocultar miembros, hacerlos editables, o lo que queramos).

3.2.3 Acciones de la propiedad

También podemos especificar acciones que el usuario puede pulsar directamente:

```
<vista-propiedad propiedad="numero">
      <accion accion="Albaranes.generarNumero"/>
</vista-propiedad>
```

En este caso en vez de la clase de la acción se pone un identificador que consiste en el nombre de controlador y nombre de acción. Esta acción ha de estar registrada en *controladores.xml* de la siguiente forma:

```
<controlador nombre="Albaranes">
      ...
      <accion nombre="generarNumero" oculta="true"
            clase="org.openxava.test.acciones.GenerarNumeroAlbaran">
            <usa-objeto nombre="xava_view"/>
      </accion>
      ...
</controlador>
```

Las acciones se visualizan con un vínculo o imagen al lado del editor de la propiedad. Como sigue:

Por defecto el vínculo de la acción aparece solo cuando la propiedad es editable, ahora bien si la propiedad es de solo-lectura o calculada entonces está siempre disponible. Podemos usar el atributo `siempre-activa` *(nuevo v2.0.3)* a `true` para que el vínculo esté siempre presente, incluso si la propiedad no es editable. Como sigue:

```
<accion accion="Albaranes.generarNumero" siempre-activa="true"/>
```

El atributo `siempre-activa` es opcional y su valor por defecto es `false`.

El código de la acción anterior es:

```
package org.openxava.test.acciones;

import org.openxava.actions.*;

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

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

}
```

Una implementación simple pero ilustrativa. Se puede usar cualquier acción definida en *controladores.xml* y su funcionamiento es el normal para una acción OpenXava. En el capítulo 7 veremos más detalles sobre los controladores.

Opcionalmente podemos hacer nuestra acción una `IPropertyAction` *(nuevo v2.0.2)* (esto está disponible solo para acciones usadas en `<vista-propiedad/>`), the esta forma la vista contenedora y el nombre de la propiedad son inyectados en la acción por OpenXava. La clase de la acción anterior se podría reescribir así:

```
package org.openxava.test.acciones;

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

/**
 * @author Javier Paniza
 */
public class GenerarNumeroAlbaran
	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;
    }
}
```

Esta acción implementa `IPropertyAction` (1), esto requiere que la clase tenga los métodos `setProperty()` (3) y `setView()` (4), estos valores serán inyectados en la acción antes de llamar al método `execute()`, donde pueden ser usados (2). En este caso no necesitas inyectar el objeto `xava_view` al definir la acción en *controladores.xml*. La vista inyectada por `setView()` (4) es la vista más interna que contiene la propiedad, por ejemplo, si la propiedad está dentro de un agregado es la vista de ese agregado, no la vista principal del módulo. De esta manera podemos escribir acciones más reutilizables.

3.2.4 Escoger un editor *(nuevo en v2.1.3)*

Un editor visualiza la propiedad al usuario y le permite editar su valor. OpenXava usa por defecto el editor asociado al estereotipo o tipo de la propiedad, pero podemos especificar un editor concreto para visualizar una propiedad en una vista.

Por ejemplo, OpenXava usa un combo para editar las propiedades de tipo `valores-posibles`, pero si queremos visualizar una propiedad de este tipo en alguna vista concreta usando un *radio button* podemos definir esa vista de esta forma:

```
<vista nombre="TipoConRadioButton">
    <vista-propiedad propiedad="tipo" editor="ValidValuesRadioButton"/>
    <miembros>codigo; tipo; nombre; direccion</miembros>
</vista>
```

En este caso para visualizar/editar se usará el editor `ValidValuesRadioButton`, en lugar de del editor por defecto. `ValidValueRadioButton` está definido en *OpenXava/xava/default-editors.xml* como sigue:

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

Este editor está incluido con OpenXava, pero nosotros podemos crear nuestro propios editores con nuestro propios JSPs y declararlos en el archivo *xava/editores.xml* de nuestro proyecto.

Esta característica es para cambiar el editor solo en una vista. Si lo que se pretende es cambiar el editor para un estereotipo, tipo o una propiedad de un modelo a nivel de aplicación entonces lo mejor es configurarlo usando el archivo *xava/editors.xml*.

3.3 Vista referencia

Con `<vista-referencia/>` modificamos la forma en que se visualiza una referencia.

Su sintaxis es:

```
<vista-referencia
    referencia="referencia"           (1)
```

```
      vista="vista"                    (2)
      solo-lectura="true|false"        (3)
      marco="true|false"               (4)
      crear="true|false"               (5)
      modificar="true|false"           (6)   nuevo en v2.0.4
      buscar="true|false"              (7)
      como-agregado="true|false"       (8)   nuevo en v2.0.3
>
      <busqueda-al-cambiar ... />      (9)   nuevo en v2.2.5
      <accion-buscar ... />           (10)
      <lista-descriptiones ... />     (11)
      <accion ... /> ...              (12)   nuevo en v2.0.1
</vista-referencia>
```

(1)`referencia` (obligado): Nombre de la referencia del modelo de la que se quiere personalizar la visualización.

(2)`vista` (opcional): Si omitimos este atributo usa la vista por defecto del objeto referenciado para visualizarlo, con este atributo podemos indicar que use otra vista.

(3)`solo-lectura` (opcional): Si la ponemos a `true` esta referencia no será nunca editable por el usuario en esta vista. Una alternativa a esto es hacer la propiedad editable/no editable programáticamente usando `org.openxava.view.View`.

(4)`marco` (opcional): Si el dibujador de la interfaz gráfica usa un marco para envolver todos los datos de la referencia con este atributo se puede indicar que dibuje o no ese marco, por defecto sí que sacará el marco.

(5)`crear` (opcional): Indica si el usuario ha de tener opción para crear o no un nuevo objeto del tipo referenciado. Por defecto vale `true`.

(6)`modificar` (opcional): *(nuevo en v2.0.4)* Indica si el usuario ha de tener opción para modificar o no el objeto actualmente referenciado. Por defecto vale `true`.

(7)`buscar` (opcional): Indica si el usuario va a tener un vínculo para poder realizar búsquedas con una lista, filtros, etc. Por defecto vale `true`.

(8)`como-agregado` (opcional): *(nuevo en v2.0.3)* Por defecto `false`. Por defecto en el caso de una referencia a un agregado el usuario puede crear y editar sus datos, mientras que en el caso de una referencia a una entidad el usuario escoge una entidad existente. Si ponemos `como-agregado` a `true` entonce la interfaz de usuario para referencias a entidad se comporta como en el caso de los agregados, permitiendo al usuario crear un nuevo objeto y editar sus datos directamente. No tiene efecto en el caso de una referencia a agregado. ¡Ojo! Si borramos una entidad sus entidades referenciadas no se borran, incluso si estamos usando `como-agregado="true"`.

(9)`busqueda-al-cambiar` (una, opcional): *(nuevo en v2.2.5)* Nos permite especificar nuestra propia acción de búsqueda cuando el usuario teclea una clave nueva.

(10)`accion-buscar` (una, opcional): Nos permite especificar nuestra propia acción de búsqueda cuando el usuario pulsa en el vínculo para buscar.

(11)`lista-descripciones`: Permite visualizar los datos como una lista descripciones, típicamente

un combo. Práctico cuando hay pocos elementos del objeto referenciado.

(12)`accion` (varias, opcional): *(nuevo en v2.0.1)* Acciones (mostradas como vínculos, botones o imágenes al usuario) asociadas (visualmente) a esta referencia y que el usuario final puede ejecutar. Funciona como ene el caso de `<vista-propiedad/>`, mirar la sección 4.2.3.

Si no usamos `<vista-referencia/>` OpenXava dibuja la referencia usando su vista por defecto. Por ejemplo si tenemos una referencia así:

```
<entidad>
...
     <referencia nombre="familia" modelo="Familia" requerido="true"/>
...
</entidad>
```

La interfaz gráfica tendrá el siguiente aspecto *(vínculo para modificar nuevo en v2.0.4)*:

3.3.1 Escoger vista

La modificación más sencilla sería especificar que vista del objeto referenciado queremos usar:

```
<vista-referencia referencia="factura" vista="Simple"/>
```

Para esto en el componente `Factura` tenemos que tener una vista llamada simple:

```
<componente nombre="Factura">
...
     <vista nombre="Simple">
          <miembros>
               año, numero, fecha, descuentoAño;
          </miembros>
     </vista>
...
</componente>
```

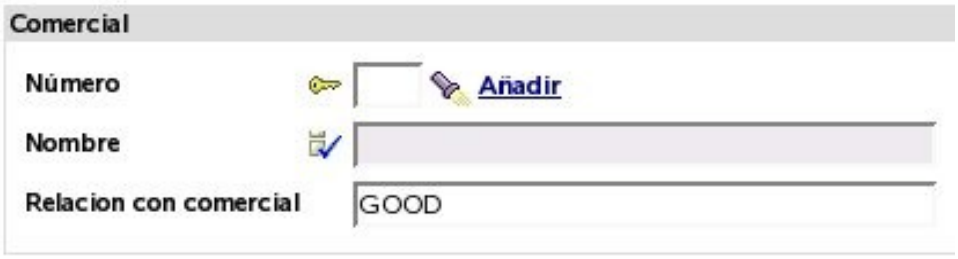
Y así en lugar de usar la vista de la `Factura` por defecto, que supuestamente sacará toda la información, visualizará ésta:

3.3.2 Personalizar el enmarcado

Si combinamos `marco="false"` con un `grupo` podemos agrupar visualmente una propiedad que no forma parte de la referencia, por ejemplo:

```
<vista-referencia referencia="comercial" marco="false"/>
<miembros>
...
     <grupo nombre="comercial">
           comercial;
           relacionConComercial;
     </grupo>
...
</miembros>
```

Así obtendríamos:



3.3.3 Acción de búsqueda propia

El usuario puede buscar un nuevo valor para la referencia simplemente tecleando el código y al salir del editor recupera el valor correspondiente; por ejemplo, si el usuario teclea "1" en el campo del código de comercial, el nombre (y demás datos) del comercial "1" serán automaticamente rellenados. También podemos pulsar la linternita, en ese caso vamos a una lista en donde podemos filtrar, ordenar, etc, y marcar el objeto deseado.

Para definir nuestra propia rutina de búsqueda podemos usar `<accion-buscar/>`, como sigue:

```
<vista-referencia referencia="comercial">
     <accion-buscar accion="MiReferencia.buscar"/>
</vista-referencia>
```

Ahora al pulsar la linternita ejecuta nuestra acción, la cual tenemos que tener definida en *controladores.xml*:

```
<controlador nombre="MiReferencia">
     <accion nombre="buscar" oculta="true"
           clase="org.openxava.test.acciones.MiAccionBuscar"
           imagen="images/search.gif">
           <usa-objeto nombre="xava_view"/>
           <usa-objeto nombre="xava_referenceSubview"/>
           <usa-objeto nombre="xava_tab"/>
```

```
            <usa-objeto nombre="xava_currentReferenceLabel"/>
      </accion>
      ...
</controlador>
```

Lo que hagamos en `MiAccionBuscar` ya es cosa nuestra. Podemos, por ejemplo, refinar la acción por defecto de busqueda para filtrar la lista usada para buscar, como sigue:

```
package org.openxava.test.acciones;

import org.openxava.actions.*;

/**
 * @author Javier Paniza
 */

public class MiAccionBuscar extends ReferenceSearchAction {

      public void execute() throws Exception {
            super.execute();    // El comportamiento por defecto para buscar
            getTab().setBaseCondition("${codigo} < 3"); // Añadir un filtro a la lista
      }

}
```

Veremos más acerca de las acciones en el capítulo 7.

3.3.4 Acción de creación propia

Si no hemos puesto `crear="false"` el usuario tendrá un vínculo para poder crear un nuevo objeto. Por defecto muestra la vista por defecto del componente referenciado y permite introducir valores y pulsar un botón para crearlo. Si queremos podemos definir nuestras propias acciones (entre ellas la de crear) en el formulario a donde se va para crear uno nuevo, para esto hemos de tener un controlador llamado como el componente con el sufijo `Creation`. Si OpenXava ve que existe un controlador así lo usa en vez del de por defecto para permitir crear un nuevo objeto desde una referencia. Por ejemplo, podemos poner en nuestro *controladores.xml*:

```
<!--
Puesto que su nombre es AlmacenCreation (nombre modelo + Creation) es usado
por defecto para crear desde referencias, en vez de NewCreation.
La accion 'new' es ejecutada automáticamente.
-->
<controlador nombre="AlmacenCreation">
      <hereda-de controlador="NewCreation"/>
      <accion nombre="new" oculta="true"
```

```
            clase="org.openxava.test.actions.CrearNuevoAlmacenDesdeReferencia">
            <usa-objeto nombre="xava_view"/>
      </accion>
</controlador>
```

En este caso cuando en una referencia a `Almacen` pulsemos el vínculo 'crear' irá a la vista por defecto de `Almacen` y mostrará las acciones de `AlmacenCreation`.

Sí tenemos una acción `new`, ésta se ejecuta automáticamente antes de nada, la podemos usar para iniciar la vista si lo necesitamos.

3.3.5 Acción de modificación propia *(nuevo en v2.0.4)*

Si no hemos puesto `modificar="false"` el usuario tendrá un vínculo para poder actualizar el objeto actualmente referenciado. Por defecto muestra la vista por defecto del componente referenciado y permite modificar valores y pulsar un botón para actualizarlo. Si queremos podemos definir nuestras propias acciones (entre ellas la de actualizar) en el formulario a donde se va para modificar, para esto hemos de tener un controlador llamado como el componente con el sufijo `Modification`. Si OpenXava ve que existe un controlador así lo usa en vez del de por defecto para permitir modificar el objeto referenciado desde una referencia. Por ejemplo, podemos poner en nuestro *controladores.xml*:

```
<!--
Dado que su nombre es AlmacenModification (nombre modelo + Modification) es usado
por defecto para modificar desde referencias, en lugar de Modification.
La acción 'search' se ejecuta automáticamente.
-->
<controlador nombre="AlmacenModification">
      <hereda-de controlador="Modification"/>
      <accion nombre="search" oculta="true"
            clase="org.openxava.test.actions.ModificarAlmacenDesdeReferencia">
            <usa-objeto nombre="xava_view"/>
      </accion>
</controlador>
```

En este caso cuando en una referencia a `Almacen` pulsemos el vínculo 'modificar' irá a la vista por defecto de `Almacen` y mostrará las acciones de `AlmacenModification`.

Sí tenemos una acción `search`, ésta se ejecuta automáticamente antes de nada, la podemos usar para iniciar la vista con los datos del objeto actualmente referenciado.

3.3.6 Lista descripciones (combos)

Con `<lista-descripciones/>` podemos instruir a OpenXava para que visualice la referencia como una lista de descripciones (actualmente como un combo). Esto puede ser práctico cuando hay pocos valores y haya un nombre o descripción significativo. La sintaxis es:

```
<lista-descripciones
```

```
    propiedad-descripcion="propiedad"                (1)
    propiedades-descripcion="propiedades"            (2)
    depende-de="depende"                             (3)
    condicion="condicion"                            (4)
    ordenado-por-clave="true|false"                  (5)
    orden="orden"                                    (6)
    formato-etiqueta="NORMAL|PEQUENA|SIN_ETIQUETA" (7)
/>
```

(1)`propiedad-descripcion` (opcional): Indica que propiedad es la que tiene que aparecer en la lista, si no se especifica asume la propiedad `description`, `descripcion`, `name` o `nombre`. Si el objeto referencia no tiene ninguna propiedad llamada así entonces es obligado especificar aquí un nombre de propiedad.

(2)`propiedad-descripciones` (opcional): Como propiedad-descripcion (y además exclusiva con ella) pero permite poner una lista de propiedades separadas por comas. Al usuario le aparecen concatenadas.

(3)`depende-de` (opcional): Se usa junto con `condicion` para hacer que el contenido de la lista dependa del valor de otro miembro visualizado en la vista principal (si simplemente ponemos el nombre del miembro) o en la misma vista (si ponemos `this.` delante del nombre de miembro).

(4)`condicion` (opcional): Permite poner una condición (al estilo SQL) para filtrar los valores que aparecen en la lista de descripciones.

(5)`ordenado-por-clave` (opcional): Por defecto los datos salen ordenados por descripción, pero si ponemos está propiedad a `true` saldrán ordenados por clave.

(6)`orden` (opcional): Permite poner un orden (al estilo SQL) para los valores que aparecen en la lista de descripciones.

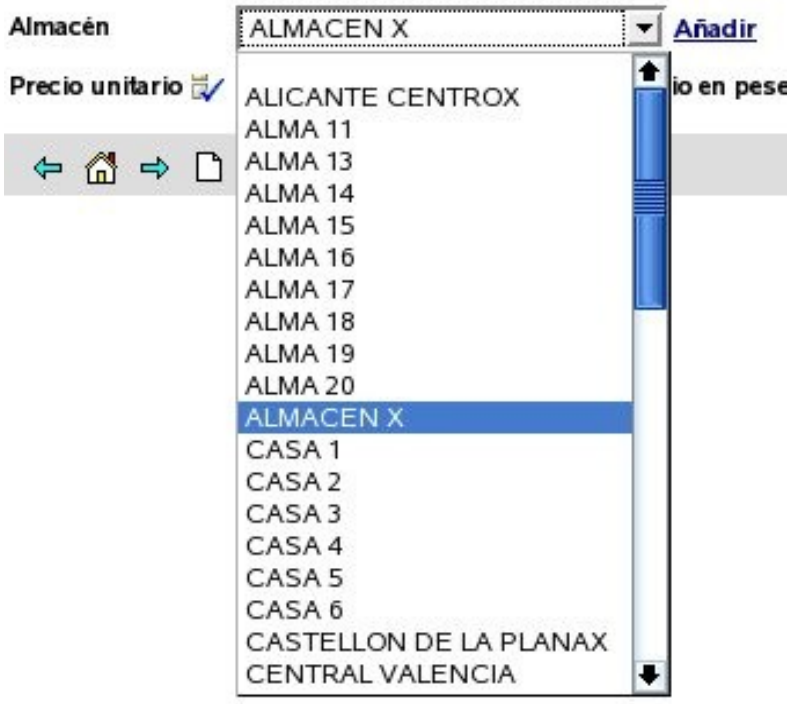
(7)`formato-etiqueta` (opcional): Forma en que se visualiza la etiqueta para esta referencia. Ver sección 4.2.1.

El uso más simple es:

```
<vista-referencia referencia="almacen">
    <lista-descripciones/>
</vista-referencia>
```

Que haría que una referencia a `Almacen` se representara así:

En un principio saca todos los almacenes, aunque en realidad usa la `condicion-base` y `filtro` especificados en el `tab` por defecto de `Almacen`. Veremos como funcionan los tabs en el capítulo 5.



Si queremos, por ejemplo, que se visualice un combo con las familias de productos y según la familia que se escoja se rellene el combo de las subfamilias, podemos hacer algo así:

```
<vista-referencia referencia="familia">
     <lista-descripciones ordenado-por-clave="true"/>      (1)
</vista-referencia>

<vista-referencia referencia="subfamilia" crear="false">   (2)
     <lista-descripciones
           propiedad-descripcion="descripcion"             (3)
           depende-de="familia"                            (4)
           condicion="${familia.codigo} = ?"               (5)
           orden="${descripcion} desc"/>                   (6)
</vista-referencia>
```

Se visualizarán 2 combos uno con todas las familias y otro vacío, y al seleccionar una familia el otro combo se rellenará con todas las subfamilias de esa familia.

En el caso de `Familia` (1) se visualiza la propiedad `descripcion` de `Familia`, ya que si no lo indicamos por defecto visualiza una propiedad llamada '`descripcion`' o '`nombre`'. En este caso los datos aparecen ordenados por clave y no por descripción. En el caso de `Subfamilia` indicamos que no muestre el vínculo para crear una nueva `subfamilia` (2) y que la propiedad a visualizar es `descripcion` (aunque esto lo podríamos haber omitido). Con `depende-de` (4) hacemos que este combo dependa de la referencia `familia`, cuando cambia `familia` en la interfaz gráfica, rellenará esta lista de descripciones aplicando la condición de `condicion` (5) y enviando como argumento (para rellenar el interrogante) el nuevo valor de `familia`. Y las entradas están ordenadas

descendentemente por descripción (6).

En `condicion` y `orden` ponemos los nombres de las propiedades entre `${}` y los argumentos como `?`, los operadores de comparación son los de SQL.

Podemos especificar una lista de propiedades para que aparezca como descripción:

```
<vista-referencia referencia="comercialAlternativo" solo-lectura="true">
     <lista-descripciones propiedades-descripcion="nivel.descripcion, nombre"/>
</vista-referencia>
```

En este caso en el combo se visualizará una concatenación de la descripción del nivel y el nombre. Además vemos como podemos usar propiedades calificadas (`nivel.descripcion`) también.

En el caso de poner una referencia `lista-descripciones` como `solo-lectura` se visualizará la descripción (en este caso `nivel.descripcion + nombre`) como si fuera una propiedad simple de texto y no como un combo.

3.3.7 Búsqueda de referencia al cambiar *(new in v2.2.5)*

El usuario puede buscar el valor de una referencia simplemente tecleando su clave. Por ejemplo, si hay una referencia a `Subfamilia`, el usuario puede teclear el código de subfamilia y automáticamente se cargará la información de la subfamilia en la vista. Esto se hace usando una acción "al cambiar" que hace la búsqueda. Podemos especificar nuestra propia acción para buscar cuando la clave cambia usando `busqueda-al-cambiar`, justo así:

```
<vista-referencia referencia="subfamilia">
     <busqueda-al-cambiar clase="org.openxava.test.acciones.BuscarAlCambiarSubfamilia"/>
</vista-referencia>
```

Esta acción se ejecuta para realizar la búsqueda, en vez de la acción por defecto, cuando el usuario cambia el código de subfamilia.
El código a ejecutar es:

```
package org.openxava.test.acciones;

import org.openxava.actions.*;

/**
 *
 * @author Javier Paniza
 */
public class BuscarAlCambiarSubfamilia
    extends OnChangeSearchAction {  // 1

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

```
        }
        super.execute();
    }

}
```

La acción implementa `IOnChangePropertyAction`, mediante `OnChangeSearchAction` (1), aunque es una referencia. Recibe el cambio de la propiedad clave de la referencia; en este caso `subfamilia.codigo`.

Este caso es un ejemplo de refinamiento del comportamiento de la búsqueda al cambiar, porque extiende de `OnChangeSearchAction`, que es la acción por defecto para buscar, y llama a `super.execute()`. También es posible hacer una acción al cambiar convencional (extendiendo de `OnChangePropertyBaseAction` por ejemplo) anulando completamente la lógica de búsqueda.

3.4 Vista colección

Sirve para refinar la presentación de una colección. Aquí su sintaxis:

```
<vista-coleccion
     coleccion="coleccion"                  (1)
     vista="vista"                          (2)
     solo-lectura="true|false"              (3)
     solo-edicion="true|false"              (4)
     crear-referencia="true|false"          (5)
     modificar-referencia="true|false"      (6) nuevo en v2.0.4
     como-agregado="true|false"             (7) nuevo en v2.0.2
>
     <propiedades-lista ... />              (8)
     <estilo-fila ... />                    (9) nuevo en v2.2.2
     <accion-editar ... />                 (10)
     <accion-ver ... />                    (11)
     <accion-nuevo ... />                  (12) nuevo en v2.0.2
     <accion-grabar ... />                 (13) nuevo en v2.0.2
     <accion-ocultar-detalle ... />        (14) nuevo en v2.0.2
     <accion-quitar ... />                 (15) nuevo en v2.0.2
     <accion-quitar-seleccionados ... />   (16) nuevo en v2.1
     <accion-lista ... /> ...              (17)
     <accion-detalle ... /> ...            (18)
</vista-coleccion>
```

(1)`coleccion` (obligado): Indica la colección de la que se quiere personalizar la presentación.

(2)`vista` (opcional): La vista del objeto referenciado que se ha de usar para representar el detalle. Por defecto usa la vista por defecto.

(3)`solo-lectura` (opcional): Por defecto `false`, si la ponemos a `true` solo podremos visualizar los elementos de la colección, no podremos ni añadir, ni borrar, ni modificar los elementos.

(4)`solo-edicion` (opcional): Por defecto `false`, si la ponemos a `true` podemos modificar los elementos existentes, pero no podemos añadir nuevos ni eliminar.

(5)`crear-referencia` (opcional): Por defecto `true`, si la ponemos a `false` el usuario final no tendrá el vínculo que le permite crear objetos del tipo del objeto referenciado. Esto solo aplica en el caso de colecciones de referencias a entidad.

(6)`modificar-referencia` (opcional): *(nuevo en v2.0.4)* Por defecto `true`, si la ponemos a `false` el usuario final no tendrá el vínculo que le permite modificar objetos del tipo del objeto referenciado. Esto solo aplica en el caso de colecciones de referencias a entidad.

(7)`como-agregado` (opcional): *(nuevo en v2.0.2)* Por defecto `false`. Por defecto las colecciones de agregados permiten al usuario crear y añadir elementos, mientras que las colecciones de entidades permiten solo escoger entidades existentes para añadir (o quitar) de la colección. Si ponemos `como-agregado` a `true` entonces la colección de entidades se comportan como una colección de agregados, permitiendo al usuario añadir objetos y editarlos directamente. No tiene efecto en el caso de una colección de agregados.

(8)`propiedades-lista` (una, opcional): Indica las propiedades que han de salir en la lista al visualizar la colección. Podemos calificar las propiedades. Por defecto saca todas las propiedades persistentes del objeto referenciado (sin incluir referencias ni calculadas).

(9)`estilo-lista` (varios, opcional): *Nuevo en v2.2.2.* Para dar un estilo especial a algunas filas. Se comporta igual que en el caso del `Tab`. Para más detalles ver la sección 5.1 sobre resaltar filas. No funciona para colecciones calculadas.

(10)`accion-editar` (una, opcional): Permite sobreescribir la acción que inicia la edición de un elemento de la colección. Esta es la acción mostrada en cada fila cuando la colección es editable.

(11)`accion-ver` (una, opcional): Permite sobreescribir la acción para visualizar un elemento de la colección. Esta es la acción mostrada en cada fila cuando la colección es de solo lectura.

(12)`accion-nuevo` (una, opcional): *(nuevo en v2.0.2)* Permite definir nuestra propia acción para empezar a añadir un nuevo elemento en la colección. Ésta es la acción que se ejecuta al pulsar en el vínculo 'Añadir'.

(13)`accion-grabar` (una, opcional): *(nuevo en v2.0.2)* Permite definir nuestra propia acción para grabar el elemento de la colección. Ésta es la acción que se ejecuta al pulsar el vínculo 'Grabar detalle'.

(14)`accion-ocultar-detalle` (una, opcional): *(nuevo en v2.0.2)* Permite definir nuestra propia acción para ocultar la vista de detalle. Ésta es la acción que se ejecuta al pulsar el vínculo 'Cerrar'.

(15)`accion-quitar` (una, opcional): *(nuevo en v2.0.2)* Permite definir nuestra propia acción para borrar un elemento de la colección. Ésta es la acción que se ejecuta al pulsar en el vínculo 'Quitar detalle'.

(16)`accion-quitar-seleccionados` (una, opcional): *(nuevo en v2.1)* Permite definir nuestra propia acción para quitar los elementos seleccionados de la colección. Ésta es la acción que se ejecuta al seleccionar algunas filas y pulsar en el vínculo 'Quitar seleccionados'.

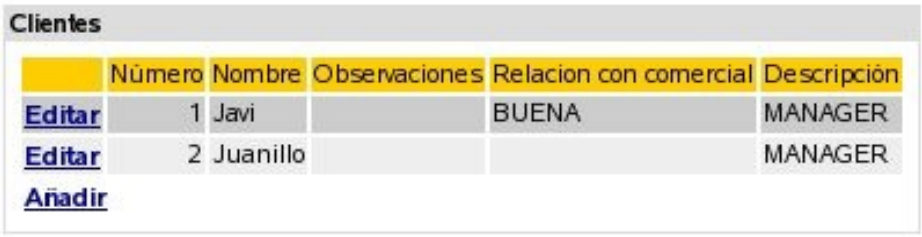
(17)`accion-lista` (varias, opcional): Para poder añadir acciones en el modo lista; normalmente acciones cuyo alcance es la colección entera.

(18)`accion-detalle` (varias, opcional): Para poder añadir acciones en detalle, normalmente acciones cuyo alcance es el detalle que se está editando.

Si no usamos `<vista-coleccion/>` una colección se visualiza usando las propiedades persistentes en el modo lista y la vista por defecto para representar el detalle; aunque lo más normal es indicar como mínimo que propiedades salen en la lista y que vista se ha de usar para representar el detalle:

```
<vista-coleccion coleccion="clientes" vista="Simple">
     <propiedades-lista>
           codigo, nombre, observaciones,
           relacionConComercial, comercial.nivel.descripcion
     </propiedades-lista>
</vista-coleccion>
```

De esta forma la colección se visualiza así:



Podemos ver como en la lista de propiedades podemos poner propiedades calificadas (como `comercial.nivel.descripcion`).

Al pulsar 'Editar' se visualizará el detalle usando la vista `Simple` de `Cliente`; para eso

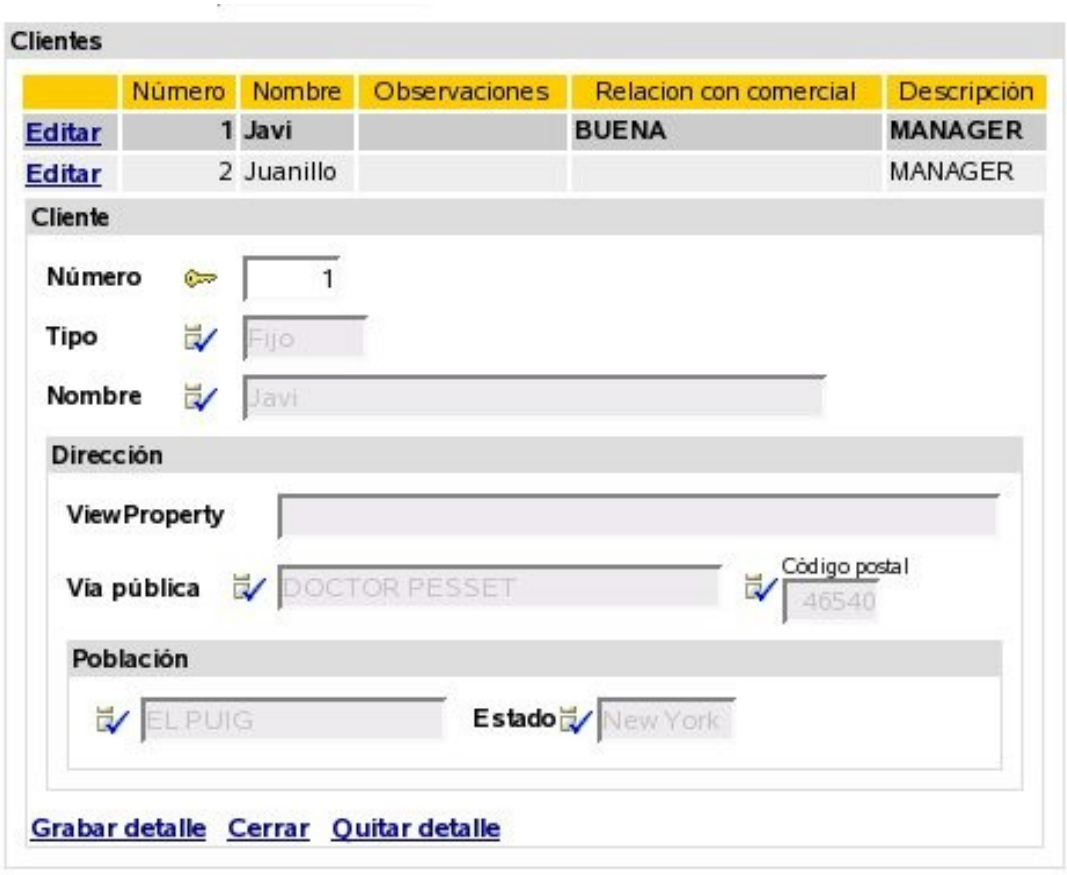
hemos de tener una vista llamada `Simple` en el componente `Cliente` (el modelo de los elementos de la colección).

Este vista se usa también cuando el usuario pulsa en 'Añadir' en una colección de agregados, pero en el caso de una colección de entidades OpenXava no muestra esta vista, en su lugar muestra una lista de entidades a añadir *(nuevo en v2.2)*.

Si la vista `Simple` de `Cliente` es así:

```
<vista nombre="Simple" miembros="codigo; tipo; nombre; direccion"/>
```

Al pulsar detalle aparecerá:



3.4.1 Acción de editar/ver detalle propia

Podemos refinar fácilmente el comportamiento cuando se pulse el vínculo 'Editar':

```
<vista-coleccion coleccion="lineas">
     <accion-editar accion="Facturas.editarLinea"/>
</vista-coleccion>
```

Hemos de definir `Facturas.editarLinea` en *controladores.xml*:

```
<controlador nombre="Facturas">
     ...
     <accion nombre="editarLinea" oculta="true"
           clase="org.openxava.test.acciones.EditarLineaFactura">
           <usa-objeto nombre="xava_view"/>
     </accion>
     ...
</controlador>
```

Y nuestra acción puede ser así:

```
package org.openxava.test.acciones;
```

```
import java.text.*;

import org.openxava.actions.*;

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

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

}
```

En este caso queremos solamente refinar y por eso nuestra acción desciende de (1) `EditElementInCollectionAction`. Nos limitamos a poner un valor por defecto en la propiedad `remarks`. Es de notar que para acceder a la vista que visualiza el detalle podemos usar el método `getCollectionElementView()` (2).

También es posible eliminar la acción para editar de la interfaz de usuario *(nuevo en v2.2.1)*, de esta manera:

```
<vista-coleccion coleccion="lineas">
     <accion-editar accion=""/>
</vista-coleccion>
```

Sólo necesitamos poner una cadena vacía como valor para la acción. Aunque en la mayoría de los casos es suficiente declarar la colección como de `solo-lectura`.

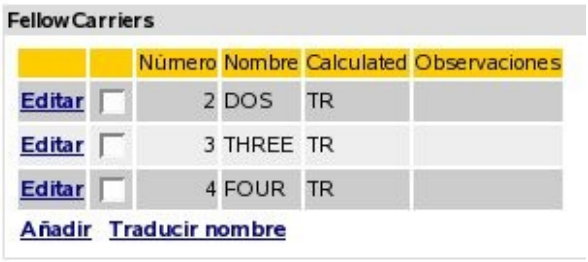
La técnica para refinar una acción 'ver' (la acción para cada fila cuando la colección es de solo lectura) es la misma pero usando `<accion-ver/>` en vez de `<accion-editar/>`.

3.4.2 Acciones de lista propias

Añadir nuestras propias acciones de lista (acciones que aplican a la colección entera) es fácil:

```
<vista-coleccion coleccion="compañeros" vista="Simple">
     <accion-lista accion="Transportistas.traducirNombre"/>
</vista-coleccion>
```

Ahora aparece un nuevo vínculo al usuario:



Vemos además como ahora hay una casilla de chequeo en cada línea *(desde v2.1.4 la caseilla de chequeo está siempre presente en todas las colecciones)*.

Falta definir la acción en *controladores.xml*:

```
<controlador nombre="Transportistas">
     <accion nombre="traducirNombre"
           clase="org.openxava.test.acciones.TraducirNombreTransportista">
     </accion>
</controlador>
```

Y el código de nuestra acción:

```
package org.openxava.test.acciones;

import java.util.*;

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

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

     public void execute() throws Exception {
           Iterator it = getSelectedObjects().iterator();              // (2)
           while (it.hasNext()) {
                 ITransportista transportista = (ITransportista) it.next();
                 transportista.traducir();
           }
     }

}
```

La acción desciende de `CollectionBaseAction` (1), de esta forma tenemos a nuestra disposición métodos como `getSelectedObjects()` (2) que ofrece una colección de los objetos seleccionados por el usuario. Hay disponible otros métodos como `getObjects()` (todos los objetos de la colección), `getMapValues()` (los valores de la colección en formato de mapa) y

`getMapsSelectedValues()` (los valores seleccionados de la colección en formato de mapa).

Como en el caso de la acciones de detalle (ver la siguiente sección) puedes usar `getCollectionElementView()`.

También es posible usar acciones para el modo lista (ver la sección 7.5) como acciones de lista para una colección *(nuevo en v2.1.4)*.

3.4.3 Acciones de lista por defecto *(nuevo en v2.1.4)*

Si queremos añadir alguna acciones de lista a todas las colecciones de nuestra aplicación hemos de crear un controlador llamado `DefaultListActionsForCollections` en nuestro propio *xava/controladores.xml* como sigue:

```
<controlador nombre="DefaultListActionsForCollections">
	<hereda-de controlador="Print"/>
	<accion nombre="exportarComoXML"
		clase="org.openxava.test.acciones.ExportarComoXML">
	</accion>
</controlador>
```

De esta forma todas las colecciones tendrán las acciones del controlador `Print` (para exportar a Excel y generar informes PDF) y nuestra propia acción `ExportarComoXML`. Esto tiene el mismo efecto que el elemento `<accion-lista/>` (en sección 4.4.2) pero aplica a todas las colecciones a la vez.

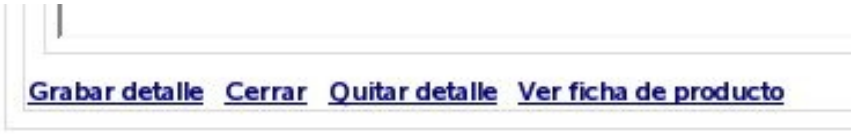
Esta característica no aplica a las colecciones calculadas *(nuevo en v2.2.1)*.

3.4.4 Acciones de detalle propias

También podemos añadir nuestras propias acciones a la vista de detalle usada para editar cada elemento. Estas sería acciones que aplican a un solo elemento de la colección. Por ejemplo:

```
<vista-coleccion coleccion="lineas">
	<accion-detalle accion="Facturas.verProducto"/>
</vista-coleccion>
```

Esto haría que el usuario tuviese a su disposición otro vínculo al editar el detalle:



Debemos definir la acción en *controladores.xml*:

```
<controlador nombre="Facturas">
	...
	<accion nombre="verProducto" oculta="true"
		clase="org.openxava.test.acciones.VerProductoDesdeLineaFactura">
		<usa-objeto nombre="xava_view"/>
		<use-objeto nombre="xavatest_valoresFactura"/>
```

```
    </accion>
    ...
</controlador>
```

Y el código de nuestra acción:

```
package org.openxava.test.acciones;

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

import org.openxava.actions.*;

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

    private Map valoresFactura;

    public void execute() throws Exception {
        try {
            setValoresFactura(getView().getValues());
            Object codigo =
                getCollectionElementView().getValue("producto.codigo");// (2)
            Map clave = new HashMap();
            clave.put("codigo", codigo);
            getView().setModelName("Producto");                              // (3)
            getView().setValues(clave);
            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 [] { "ProductoDesdeFactura" };
	}

	public String getCustomView() {
		return SAME_VIEW;
	}

	public Map getValoresFactura() {
		return valoresFactura;
	}

	public void setValoresFactura(Map map) {
		valoresFactura = map;
	}

}
```

Vemos como desciende de `CollectionElementViewBaseAction` (1) y así tiene disponible la vista que visualiza el elemento de la colección mediante `getCollectionElementView()` (2). También podemos acceder a la vista principal mediante `getView()` (3). En el capítulo 7 se ven más detalles acerca de como escribir acciones.

Además, usando la vista devuelta por `getCollectionElementView()` podemos añadir y borrar programaticamente acciones de detalle y de lista (*nuevo en v2.0.2*) con `addDetailAction()`, `removeDetailAction()`, `addListAction()` y `removeListAction()`, ver API doc para `org.openxava.view.View`.

3.4.5 Refinar comportamiento por defecto para la vista de colección *(nuevo en v2.0.2)*

Usando `<accion-nuevo/>`, `<accion-grabar/>`, `<accion-ocultar-detalle/>`, `<accion-quitar-seleccionados/>` y `<accion-quitar/>` podemos refinar el comportamiento por defecto para una vista de colección. Por ejemplo, si queremos refinar el comportamiento de la acción de grabar un detalle podemos definir nuestra vista de esta forma:

```
<vista-coleccion coleccion="detalles">
	<accion-grabar accion="DetallesAlbaran.grabar"/>
</vista-coleccion>
```

Debemos tener la acción `DetallesAlbaran.grabar` en *controladores.xml*:

```
<controlador nombre="DetallesAlbaran">
	<accion nombre="grabar"
```

```
            clase="org.openxava.test.acciones.GrabarDetalleAlbaran">
            <usa-objeto nombre="xava_view"/>
      </accion>
</controlador>
```

Y definir la clase acción para grabar:

```
package org.openxava.test.acciones;

import org.openxava.actions.*;

/**
 *
 * @author Javier Paniza
 */

public class GrabarDetalleAlbaran extends SaveElementInCollectionAction { // (1)

      public void execute() throws Exception {
            super.execute();
            // Aquí nuestro código                                    // (2)
      }

}
```

El caso más común es extender el comportamiento por defecto, para eso hemos de extender la clase original para grabar un detalle de una colección (1), esto es la acción `SaveElementInCollection`, entonces llamamos a `super` desde el método `execute()` (2), y después escribimos nuestro propio propio código.

Nuevo en v2.2.1: También es posible eliminar cualquiera de estas acciones de la interfaz gráfica, por ejemplo, podemos defina una `<vista-coleccion/>` de esta manera:

```
<vista-coleccion coleccion="detalles">
      <accion-quitar-seleccionados accion=""/>
</vista-coleccion>
```

En este caso la acción para quitar los elementos seleccionadas no aparecerá en la interfaz de usuario. Como se ve, sólo es necesario declarar una cadena vacía como nombre de la acción.

3.5 Propiedad de vista

Poniendo `<propiedad/>` dentro de una vista podemos usar una propiedad que no existe en el modelo, pero que sí nos interesa que se visualice al usuario. Podemos usarlas para proporcionar controles al usuario para manejar la interfaz gráfica.

Un ejemplo:

```
<vista>
	<propiedad nombre="entregadoPor">
		<valores-posibles>
			<valor-posible valor="empleado"/>
			<valor-posible valor="transportista"/>
		</valores-posibles>
		<calculador-valor-defecto
			 clase="org.openxava.calculators.IntegerCalculator">
			<poner propiedad="value" valor="0"/>
		</calculador-valor-defecto>
	</propiedad>

	<vista-propiedad propiedad="entregadoPor">
		<al-cambiar clase="org.openxava.test.actiones.AlCambiarEntregadoPor"/>
	</vista-propiedad>
	...
</vista>
```

Podemos observar como la sintaxis es exactamente igual que en el caso de definir una propiedad en la parte del modelo, podemos incluso hacer que sea un `<valores-posibles/>` y que tenga un `<calculador-valor-defecto/>`. Después de haber definido la propiedad podemos usarla en la vista como una propiedad más, asignandole una propiedad `al-cambiar` por ejemplo y por supuesto poniendola en `<miembros/>`.

3.6 Acciones de la vista (nuevo en v2.0.3)

Además de poder asociar acciones a una propiedad, referencia o colección, podemos tambien definir acciones arbitrarias en cualquier parte de nuestra vista. Para poder hacer esto se usa el elemento `accion`, de esta manera:

```
<miembros>
	codigo;
	tipo;
	nombre, <accion accion="Clientes.cambiarEtiquetaDeNombre"/>;
	...
</miembros>
```

El efecto visual sería:

Podemos ver el vínculo 'Cambiar nombre de etiqueta' que ejecutará la acción

`Clientes.cambiarEtiquetaDeNombre` al pulsarlo.

Si la vista contenedora de la acción no es editable, la acción no estará presente. Si queremos que la acción esté siempre activa, incluso si la vista no está editable, hemos de usar el atributo `siempre-activa`, como sigue:

```
<accion accion="Clientes.cambiarEtiquetaDeNombre" siempre-activa="true"/>
```

La forma normal de exponer las acciones al usuario es mediante los controladores (acciones en la barra), lo controladores son reutilizables entre vistas, pero puede que a veces necesitemos una acción específica a una vista, y queramos visualizarla dentro de la misma (no en la barra de botones), para estos casos el elemento `accion` puede ser útil.

Podemos ver más acerca de las acciones en el capítulo 7.

3.7 Componente transitorio: Solo para crear vistas (nuevo en v2.1.3)

En OpenXava no se puede tener vistas que no estén asociadas a un modelo. Así que si queremos dibujar una interfaz gráfica arbitraria, lo que hemos de hacer es crear un componente, declararlo transitorio *(nuevo en v2.1.3)* y a partir de él definir una vista.

Un componente transitorio no está asociada a ninguna tabla de la base de datos, normalmente se usa solo para visualizar interfaces de usuario no relacionadas con ninguna tabla de la base de datos.

Un ejemplo puede ser:

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

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

<!--
     Ejemplo de componente OpenXava transitorio (no persistente).

     Puede ser usado, por ejemplo, para visualizar un diálogo,
     o cualquier otra interfaz gráfica.
-->

<componente nombre="FiltrarPorSubfamilia">

     <entidad>
           <referencia nombre="subfamilia" modelo="Subfamilia2" requerido="true"/>
     </entidad>

     <vista nombre="Familia1">
           <vista-referencia referencia="subfamilia" crear="false">
                 <lista-descripciones condicion="${familia.codigo} = 1"/>
           </vista-referencia>
     </vista>
```

```
    <vista nombre="Familia2">
        <vista-referencia referencia="subfamilia" crear="false">
            <lista-descripciones condicion="${familia.codigo} = 2"/>
        </vista-referencia>
    </vista>

    <vista nombre="ConFormularioDeSubfamilia">
        <vista-referencia referencia="subfamily" buscar="false"/>
    </vista>

    <transitorio/>                                                    (1)

</componente>
```

Para definir un componente como transitorio solo necesitamos poner `<transitorio/>` al final de la definición del componente (1), justo en la parte para los mapeos. No hemos de poner el mapeo ni declarar propiedades como clave.

De esta forma podemos hacer un diálogo que puede servir, por ejemplo, para lanzar un listado de familias o productos filtrado por subfamilias.

Podemos así tener un generador de cualquier tipo de interfaz gráficas sencillo y bastante flexible, aunque no queramos que la información visualizada sea persistente.

4 Datos tabulares

Datos tabulares son aquellos que se visualizan en formato de tabla. Cuando creamos un módulo de OpenXava convencional el usuario puede gestionar la información sobre ese componente con una lista como ésta:

Esta lista permite al usuario:

- Filtrar por cualquier columna o combinación de ellas.
- Ordenar por cualquier columna con un simple click.
- Visualizar los datos paginados, y así podemos leer eficientemente tablas de millones de registros.
- Personalizar la lista: añadir, quitar y cambiar de orden las columnas (con el lapicito que hay en la parte superior izquierdas). Las personalizaciones se recuerdan por cada usuario.
- Acciones genéricas para procesar la lista: Como la de generar un informe en PDF, exportar a Excel o borrar los registros seleccionados.

La lista por defecto suele ir bien, y además el usuario puede personalizarsela. Sin embargo, a veces conviene modificar el comportamiento de la lista. Esto se hace mediante `<tab/>` dentro de la definición de un componente.

La sintaxis de `tab` es:

```
<tab
     nombre="nombre"            (1)
>
     <filtro ... />             (2)
     <estilo-fila ... /> ...    (3)
     <propiedades ... />        (4)
     <condicion-base ... />     (5)
     <orden-defecto ... />      (6)
</tab>
```

(1)`nombre` (opcional): Podemos definir varios tabs para un componente, y ponerle un nombre a cada uno. Este nombre se usará después para indicar que tab queremos usar (normalmente en

aplicación.xml al definir un módulo).

(2)`filtro` (uno, opcional): Permite definir programáticamente un filtro a realizar sobre los valores que introduce el usuario cuando quiere filtrar.

(3)`estilo-fila` (varios, opcional): Una forma sencilla de especificar una estilo de visualización diferente para ciertas filas. Normalmente para resaltar filas que cumplen cierta condición.

(4)`propiedades` (uno, opcional): La lista de propiedades a visualizar inicialmente. Pueden ser calificadas.

(5)`condicion-base` (una, opcional): Es una condición que aplicará siempre a los datos visualizados añadiendose a las que pueda poner el usuario.

(6)`orden-defecto` (uno, opcional): Para especificar el orden en que aparece los datos en la lista inicialmente.

4.1 Propiedades iniciales y resaltar filas

La personalización más simple es indicar las propiedades a visualizar inicialmente:

```
<tab>
      <estilo-fila estilo="highlight" propiedad="tipo" valor="fijo"/>
      <propiedades>
            nombre, tipo, comercial.nombre,
            direccion.municipio, comercial.nivel.descripcion
      </propiedades>
</tab>
```

Vemos como podemos poner propiedades calificadas (que pertenecen a referencias) hasta cualquier nivel. Estas serán las propiedades que salen la primera vez que se ejecuta el módulo, después cada usuario puede escoger cambiar las propiedades que quiere ver.

En este caso vemos también como se indica un `<estilo-fila/>`; estamos diciendo que aquellos objetos cuya propiedad `tipo` tenga el valor `fijo` han de usar el estilo `highlight`. El estilo ha de definirse en la hoja de estilos CSS. El estilo `highlight` ya viene predefinido con OpenXava, pero se pueden añadir más. El resultado visual del anterior tab es:

		Nombre	Tipo	Comercial	Población	Nivel comercial
Filtrar		empieza por		empieza por	empieza por	empieza por
Detalle	☐	Javi	Fijo	MANUEL CHAVARRI	EL PUIG	MANAGER
Detalle	☐	Juanillo	Normal	MANUEL CHAVARRI	VALENCIA	MANAGER
Detalle	☐	Carmelo	Normal		EL PUIG	
Detalle	☐	Cuatrero	Normal	JUANVI LLAVADOR	VALENCIA	MANAGER

1 Hay 4 registros en la lista

4.2 Filtros y condición base

Una técnica habitual es combinar un filtro con una condición base:

```
<tab nombre="Actuales">
      <filtro clase="org.openxava.test.filtros.FiltroAñoActual"/>
      <propiedades>
```

```
            año, numero, sumaImportes, iva, cantidadLineas, pagada, cliente.nombre
      </propiedades>
      <condicion-base>${año} = ?</condicion-base>
</tab>
```

La condición tiene la sintaxis SQL, ponemos `?` para los argumentos y los nombres de propiedades entre `${}`. En este caso usamos el filtro para dar valor al argumento. El código del filtro es:

```
package org.openxava.test.filtros;

import java.util.*;

import org.openxava.filters.*;

/**
 * @author Javier Paniza
 */

public class FiltroAñoActual implements IFilter {                // (1)

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

```
            return r;
      }

}
```

Un filtro recoge los argumentos que el usuario teclea para filtrar la lista y los procesa devolviendo lo que al final se envía a OpenXava para que haga la consulta. Como se ve ha de implementar `IFilter` (1) lo que lo obliga a tener un método llamado `filter` (2) que recibe un objeto que el valor de los argumentos y devuelve los argumentos que al final serán usados. Estos argumentos pueden ser nulo (3), si el usuario no ha metidos valores, un objeto simple (5), si el usuario a introducido solo un valor o un array de objetos (4), si el usuario a introducidos varios valores. El filtro ha de contemplar bien todos los casos. En el ejemplo lo que hacemos es añadir delante el año actual, y así se usa como argumento a la condición que hemos puesto en nuestro tab.

Resumiendo el tab que vemos arriba solo sacará las facturas correspondientes al año actual.

Podemos ver otro caso:

```
<tab nombre="AñoDefecto">
      <filtro clase="org.openxava.test.filtros.FiltroAñoDefecto"/>
      <propiedades>
            año, numero, cliente.numero, cliente.nombre,
            sumaImportes, iva, cantidadLineas, pagada, importancia
      </propiedades>
      <condicion-base>${año} = ?</condicion-base>
</tab>
```

En este caso el filtro es:

```
package org.openxava.test.filtros;

import java.util.*;

import org.openxava.filters.*;

/**
 * @author Javier Paniza
 */

public class FiltroAñoDefecto extends BaseContextFilter {                    // (1)

      public Object filter(Object o) throws FilterException {
            if (o == null) {
                  return new Object [] { getAñoDefecto() };                  // (2)
            }
            if (o instanceof Object []) {
```

```
                List c = new ArrayList(Arrays.asList((Object []) o));
                c.add(0, getAñoDefecto());                          // (2)
                return c.toArray();
        }
        else {
                return new Object [] { getAñoDefecto(), o      };          // (2)
        }
    }

    private Integer getAñoDefecto() throws FilterException {
        try {
                return getInteger("xavatest_añoDefecto");                 // (3)
        }
        catch (Exception ex) {
                ex.printStackTrace();
                throw new FilterException(
                "Imposible obtener año defecto asociado a esta sesión");
        }
    }

}
```

Este filtro desciende de `BaseContextFilter`, esto le permite acceder al valor de los objetos de sesión de OpenXava. Vemos como usa un método `getAñoDefecto()` (2) que a su vez llama a `getInteger()` (3) el cual (al igual que `getString()`, `getLong()` o el más genérico `get()`) nos permite acceder al valor del objeto `xavatest_añoDefecto`. Esto objeto lo definimos en nuestro archivo *controladores.xml* de esta forma:

```
<objeto nombre="xavatest_añoDefecto" clase="java.lang.Integer" valor="1999"/>
```

Las acciones lo pueden modificar y tiene como vida la sesión del usuario y es privado para cada módulo. De esto se habla más profundamente en el capítulo 7.

Esto es una buena técnica para que en modo lista aparezcan unos datos u otros según el usuario o la configuración que éste haya escogido.

También es posible acceder a variables de entorno dentro de un filtro *(nuevo en v2.0)* de tipo `BaseContextFilter`, usando el método `getEnvironment()`, de esta forma:

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

Para aprender más sobre variable de entorno ver el *Capítulo 7 Controladores*.

4.3 Select íntegro

Tenemos la opción de poner el select completo para obtener los datos del tab:

```
<tab nombre="SelectIntegro">
     <propiedades>codigo, descripcion, familia</propiedades>
     <condicion-base>
           select ${codigo}, ${descripcion}, XAVATEST@separator@FAMILIA.DESCRIPCION
                 from   XAVATEST@separator@SUBFAMILIA, XAVATEST@separator@FAMILIA
                 where  XAVATEST@separator@SUBFAMILIA.FAMILIA =
                       XAVATEST@separator@FAMILIA.CODIGO
     </condicion-base>
</tab>
```

Esto es mejor usarlo solo en casos de extrema necesidad. No suele ser necesario, y al usarlo el usuario no podrá personalizarse la vista.

4.4 Orden por defecto

Por último, establecer un orden por defecto es harto sencillo:

```
<tab nombre="Simple">
     <propiedades>año, numero, fecha</properties>
     <orden-defecto>${año} desc, ${numero} desc</orden-defecto>
</tab>
```

Este orden es solo el inicial, el usuario puede escoger otro con solo pulsar la cabecera de una columna.

5 Mapeo objeto/relacional

Con el mapeo objeto relacional declaramos en que tablas y columnas de nuestra base de datos relacional se guarda la información de nuestro componente.

Para los que estén familiarizado con herramientas O/R decir que esta información se usa para generar el código y archivos xml necesarios para el mapeo. Actualmente se genera código para:

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

Para los que no estén familiarizados con herramientas O/R decir que una herramienta de este tipo nos permite trabajar con objetos, en vez de con tablas y columnas y genera automáticamente el código SQL necesario para leer y actualizar la base de datos.

OpenXava genera un conjunto de clases java que representa la capa del modelo de nuestra aplicación (los conceptos de negocio con sus datos y funcionalidad). Nosotros podemos usar estos objetos directamente sin necesidad de acceder directamente a la base de datos con SQL, pero para eso tenemos que definir con precisión como se mapean nuestras clases a nuestras tablas, y eso es lo que se hace en la parte del mapeo.

5.1 Mapeo de entidad

La sintaxis para mapear la entidad principal es:

```
<mapeo-entidad tabla="tabla">                  (1)
     <mapeo-propiedad ... /> ...               (2)
     <mapeo-referencia ... /> ...              (3)
     <mapeo-propiedad-multiple ... /> ...      (4)
</mapeo-entidad>
```

(1)`tabla` (obligado): Para relacionar la entidad principal del componente con esa tabla.

(2)`mapeo-propiedad` (varios, opcional): Mapea una propiedad con una columna de la tabla de base de datos.

(3)`mapeo-referencia` (varios, opcional): Mapea una referencia con una o más columna de la tabla de base de datos.

(4)`mapeo-propiedad-multiple` (varios, opcional): Mapea una propiedad con varias columnas de la tabla de base de datos. Para cuando propiedad corresponde a varias columnas.

Un ejemplo sencillo de mapeo puede ser:

```
<mapeo-entidad tabla="XAVATEST@separator@TIPOALBARAN">
     <mapeo-propiedad propiedad-modelo="codigo" columna-tabla="CODIGO"/>
     <mapeo-propiedad propiedad-modelo="descripcion" columna-tabla="DESCRIPCION" />
</mapeo-entidad>
```

Nada más fácil.

Vemos como en el nombre de tabla la ponemos calificada (con el nombre de colección/esquema delante). También vemos que como separador en lugar de un punto ponermos `@separator@`, esto es útil porque podemos dar valor a `separator` en nuestro *build.xml* y así una misma aplicación puede ir contra bases de datos que soportan y no soportan las colecciones o esquemas.

5.2 Mapeo propiedad

La sintaxis para mapear una propiedad es:

```
<mapeo-propiedad
      propiedad-modelo="propiedad"      (1)
      columna-tabla="columna"           (2)
      tipo-cmp="tipo">                  (3)
      <conversor ... />                 (4)
</mapeo-propiedad>
```

(1)`propiedad-modelo` (obligada): El nombre de una propiedad definida en la parte del modelo.

(2)`columna-tabla` (obligada): Nombre de la columna de la tabla.

(3)`tipo-cmp` (opcional): Indica el tipo Java que usará internamente nuestro objeto para guardar la propiedad. Esto nos permite usar tipos Java más cercanos a lo que tenemos en la base de datos sin ensuciar nuestro modelo. Se suele usar en combinación con un conversor.

(4)`conversor` (uno, opcional): Permite indicar nuestra propia lógica para convertir del tipo usado en Java al tipo de la db.

Hemos visto ya ejemplos de un mapeo sencillo de propiedad con columna. Un caso más avanzado sería el uso de un conversor. Un conversor se usa cuando el tipo de Java y el tipo de la base de datos no coincide, en ese caso usar un conversor es una buena idea. Por ejemplo supongamos que en la base de datos el código postal es de tipo VARCHAR mientras que en Java nos interesa que sea un `int`. Un `int` de Java no se puede asignar directamente a una columna VARCHAR de base de datos, pero podemos poner un conversor para convertir ese `int` en un `String`. Veamos:

```
<mapeo-propiedad
      propiedad-modelo="codigoPostal"
      columna-tabla="CP"
      tipo-cmp="String">
      <conversor clase="org.openxava.converters.IntegerStringConverter"/>
</mapeo-propiedad>
```

Con `tipo-cmp` indicamos el tipo al que convertirá nuestro conversor y es el tipo que tendrá el atributo interno de la clase generada, ha de ser un tipo cercano (asignable directamente con JDBC) al de la columna de la tabla.

El código del conversor será:

```
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");
            }
     }

}
```

Un conversor ha de implementar `IConverter` (1), esto le obliga a tener un método `toDB()` (2), que recibe el objeto del tipo que usamos en Java (en este caso un Integer) y devuelve su representación con otro tipo más cercano a la base de datos (en este caso String y por ende asignable a una columna VARCHAR). El método `toJava()` tiene el cómetido contrario, coge el objeto en el formato de la base de datos y tiene que devolver un objeto del tipo que se usa en Java.

Ante cualquier problemas podemos lanzar una `ConversionException`.

Vemos como este conversor está en `org.openxava.converters`, es decir es un conversor genérico que viene incluido en la distribución de OpenXava. Otro conversor genérico bastante útil es `ValidValuesLetterConverter`, que permite mapear las propiedades de tipo `valores-posibles`. Por ejemplo podemos tener una propiedad como ésta:

```
<entidad>
...
     <propiedad nombre="distancia">
            <valores-posibles>
```

```
                <valor-posible valor="local"/>
                <valor-posible valor="nacional"/>
                <valor-posible valor="internacional"/>
            </valores-posibles>
        </propiedad>
...
</entidad>
```

Los `valores-posibles` generan una propiedad Java de tipo `int` donde el 0 se usa para indicar un valor vacío, el 1 será 'local', el 2 'nacional' y el 3 'internacional'. ¿Pero que ocurre si en la base de datos se almacena una L para local, una N para nacional y una I para internacional? Podemos hacer lo siguiente en el mapeo:

```
<mapeo-propiedad
      propiedad-modelo="distancia" columna-tabla="DISTANCIA" tipo-cmp="String">
      <conversor clase="org.openxava.converters.ValidValuesLetterConverter">
            <poner propiedad="letters" valor="LNI"/>
      </conversor>
</mapeo-propiedad>
```

Al poner 'LNI' como valor para `letters`, hace corresponder la L con 1, la N con 2 y la I con 3. Vemos como el que se puedan configurar propiedades del conversor (como se hacía con los calculadores, validadores, etc) nos permite hacer conversores reutilizables.

5.3 Mapeo referencia

La sintaxis para mapear una referencia es:

```
<mapeo-referencia
      referencia-modelo="referencia"          (1)
>
      <detalle-mapeo-referencia ... /> ...    (2)
</mapeo-referencia>
```

(1)`referencia` (obligada): La referencia que se quiere mapear.

(2)`detalle-mapeo-referencia` (varias, obligada): Para mapear una columna de la tabla con un propiedad de la clave para obtener la referencia. Si la clave del objeto referenciado es múltiple habrá varios `detalle-mapeo-referencia`.

Hacer un mapeo de una referencia es sencillo. Por ejemplo si tenemos una referencia como ésta:

```
<entidad>
      ...
      <referencia nombre="factura" modelo="Factura"/>
      ...
</entidad>
```

Podemos mapearla de esta forma:

```
<mapeo-entidad tabla="XAVATEST@separator@ALBARAN">
      <mapeo-referencia referencia-modelo="factura">
            <detalle-mapeo-referencia
                  columna-tabla="FACTURA_AÃ‘O"
                  propiedad-modelo-referenciado="aÃ±o"/>
            <detalle-mapeo-referencia
                  columna-tabla="FACTURA_NUMERO"
                  propiedad-modelo-referenciado="numero"/>
      </mapeo-referencia>
      ...
</mapeo-entidad>
```

FACTURA_NUMERO y FACTURA_AÑO son columnas de la tabla ALBARAN que nos permiten acceder a su factura, digamos que es la clave ajena, aunque el que esté declarada como tal en la base de datos no es preceptivo. Estas columnas las tenemos que relacionar con las propiedades clave en `Factura` , como sigue:

```
<entidad>
      <propiedad nombre="aÃ±o" tipo="int" clave="true" longitud="4" requerido="true">
            <calculador-valor-defecto
                  clase="org.openxava.calculators.CurrentYearCalculator"/>
      </propiedad>
      <propiedad nombre="numero" tipo="int"
            clave="true"  longitud="6" requerido="true"/>
      ...
```

Si tenemos una referencia a un modelo cuyo clave incluye referencia para definirlo en el mapeo lo hacemos como sigue:

```
<mapeo-referencia referencia-modelo="albaran">
      <detalle-mapeo-referencia
            columna-tabla="ALBARAN_FACTURA_AÃ‘O"
            propiedad-modelo-referenciado="factura.aÃ±o"/>
      <detalle-mapeo-referencia
            columna-tabla="ALBARAN_FACTURA_NUMERO"
            propiedad-modelo-referenciado="factura.numero"/>
      <detalle-mapeo-referencia
            columna-taba="ALBARAN_TIPO"
            propiedad-modelo-referenciado="tipo.codigo"/>
      <detalle-mapeo-referencia
            columna-tabla="ALBARAN_NUMERO"
            propiedad-modelo-referenciado="numero"/>
```

```
</mapeo-referencia>
```

Como se ve al indicar la propiedad del modelo referenciado podemos calificarla.

También es posible usar conversores cuando mapeamos referencias:

```
<mapeo-referencia referencia="permisoConducir">
     <detalle-mapeo-referencia
            columna-tabla="PERMISOCONDUCIR_TIPO"
            propiedad-modelo-referenciado="tipo"
            tipo-cmp="String">  (1) <!-- En esta caso esta linea puede ser omitida -->
            <conversor clase="org.openxava.converters.NotNullStringConverter"/> (2)
     </detalle-mapeo-referencia>
     <detalle-mapeo-referencia
            columna-tabla="PERMISOCONDUCIR_NIVEL"
            propiedad-modelo-referenciado="nivel"/>
</mapeo-referencia>
```

Podemos usar el conversor igual que en el caso de una propiedad simple (1). La diferencia en el caso de las referencias es que si no asignamos conversor no se usa ningún conversor por defecto, esto es porque aplicar de forma indiscriminada conversores sobre información que se usa para buscar puede ser problemático. Podemos usar `tipo-cmp` (1) *(nuevo en v2.0)* para indicar que tipo para el atributo es usado internamente en nuestro objeto para almacenar el valor. Esto nos permite usar un tipo Java más cercano al de la base de datos, ; `tipo-cmp` no es necesario si el tipo de la base de datos es compatible con el tipo Java.

5.4 Mapeo propiedad multiple

Con `<mapeo-propiedad-multiple/>` podemos hacer que varias columnas de la tabla de base de datos correspondan a una propiedad en Java. Esto es útil, por ejemplo cuando tenemos propiedades cuyo tipo Java son clases definidas por nosotros que tienen a su vez varias propiedades susceptibles de ser almacenadas, y también se usa mucho cuando nos enfrentamos a esquemas de bases de datos legados.

La sintaxis de este tipo de mapeo es:

```
<mapeo-propiedad-multiple
     propiedad-modelo="propiedad"      (1)
>
     <conversor ... />                 (2)
     <campo-cmp ... /> ...             (3)
</mapeo-propiedad-multiple>
```

(1)`propiedad-modelo` (obligada): Nombre de la propiedad que se quiere mapear.

(2)`conversor` (uno, obligado): Clase con la lógica para hacer la conversión de Java a base de datos y viceversa. Ha de implementar `IMultipleConverter`.

(3)`campo-cmp` (varios, obligado): Hace corresponder cada columna de la base de datos con una

propiedad del conversor.

Un ejemplo típico sería usar el conversor genérico `Date3Converter`, que permite almacenar en la base de datos 3 columnas y en Java una propiedad `java.util.Date`.

```
<mapeo-propiedad-multiple propiedad-modelo="fechaEntrega">
      <conversor clase="org.openxava.converters.Date3Converter"/>
      <campo-cmp
            propiedad-conversor="day" columna-tabla="DIAENTREGA" tipo-cmp="int"/>
      <campo-cmp
            propiedad-conversor="month" columna-tabla="MESENTREGA" tipo-cmp="int"/>
      <campo-cmp
            propiedad-conversor="year" columna-tabla="AÑOENTREGA" tipo-cmp="int"/>
</mapeo-propiedad-multiple>
```

DIAENTREGA, MESENTREGA y AÑOENTREGA son las tres columnas que en la base de datos guardan la fecha de entrega, y `day`, `month` y `year` son propiedades que podemos encontrar en `Date3Converter`. Y aquí `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 objetoJava) throws ConversionException {  // (3)
            if (objetoJava == null) {
                  setDay(0);
```

```
                setMonth(0);
                setYear(0);
                return;
            }
            if (!(objetoJava instanceof java.util.Date)) {
                throw new ConversionException("conversion_db_utildate_expected");
            }
            java.util.Date fecha = (java.util.Date) objetoJava;
            Calendar cal = Calendar.getInstance();
            cal.setTime(fecha);
            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;
    }

}
```

El conversor tiene que implementar `IMultipleConverter` (1) lo que le obliga a tener un método `toJava()` (2) que a partir de lo contenido en sus propiedades (en este caso `year`, `month` y `day`) ha

de devolver un objeto Java con el valor de la propiedad mapeada (en nuestro ejemplo el valor de la `fechaEntrega`); y el método `toDB()` (3) que recibe el valor de la propiedad Java (el de `fechaEntrega`) y tiene que desglosarlo para que se queda almacenado en las propiedades del conversor (`year`, `month` y `day`).

5.5 Mapeo de referencias a agregados

Una referencia a un agregado contiene información que en el modelo relacional se guarda en la misma tabla que la entidad principal. Por ejemplo si tenemos un agregado `Direccion` asociado a un `Cliente`, los datos de la dirección se guardan en la misma tabla que los del cliente. ¿Cómo se expresa eso en OpenXava? Es muy sencillo. En el modelo nos encontramos:

```
<entidad>
      ...
      <referencia nombre="direccion" modelo="Direccion" requerido="true"/>
      ...
</entidad>

<agregado nombre="Direccion">
      <implementa interfaz="org.openxava.test.ejb.IConMunicipio"/>
      <propiedad nombre="calle" tipo="String" longitud="30" requerido="true"/>
      <propiedad nombre="codigoPostal" tipo="int" longitud="5" requerido="true"/>
      <propiedad nombre="municipio" tipo="String" longitud="20" requerido="true"/>
      <referencia nombre="provincia" requerido="true"/>
</agregado>
```

Sencillamente una referencia a un agregado, y para mapearlo lo hacemo así:

```
<mapeo-entidad tabla="XAVATEST@separator@CLIENTE">
      ...
      <mapeo-propiedad propiedad-modelo="direccion_calle" columna-tabla="CALLE"/>
      <mapeo-propiedad
            propiedad-modelo="direccion_codigoPostal"
            columna-tabla="CP" tipo-cmp="String">
            <conversor clase="org.openxava.converters.IntegerStringConverter"/>
      </mapeo-propiedad>
      <mapeo-propiedad
            propiedad-modelo="direccion_municipio" columna-tabla="MUNICIPIO"/>
      <mapeo-referencia referencia-modelo="direccion_municipio">
            <detalle-mapeo-referencia
                  columna-tabla="PROVINCIA" propiedad-modelo-referenciado="codigo"/>
      </mapeo-referencia>
</mapeo-entidad>
```

Vemos como los miembros del agregado se mapean en el mapeo de la entidad que lo contienen, solo que ponemos como prefijo el nombre de la referencia a agregado con un subrayado (en esta caso

`direccion_`). Podemos observar como podemos mapear referencias y propiedades y usar conversores.

5.6 Mapeo de agregados usados en colecciones

En el caso de que tengamos una colección de agregados, supongamos las líneas de una factura, obviamente la información de las líneas se guarda en una tabla diferente que la información de cabecera de la factura. En este caso los agregados han de tener su propio mapeo. Veamos el ejemplo de las líneas de una factura.

En la parte de modelo del componente `Factura` tenemos:

```
<entidad>
      ...
      <coleccion nombre="lineas" minimo="1">
            <referencia modelo="LineaFactura"/>
      </coleccion>
      ...
</entidad>

<agregado nombre="LineaFactura">
      <propiedad nombre="oid" tipo="String" clave="true" oculta="true">
            <calculador-valor-defecto
                  clase="org.openxava.test.calculadores.CalculadorOidLineaFactura"
                  al-crear="true"/>
      </propiedad>
      <propiedad nombre="tipoServicio">
            <valores-posibles>
                  <valor-posible valor="especial"/>
                  <valor-posible valor="urgente"/>
            </valores-posibles>
      </propiedad>
      <propiedad nombre="cantidad" tipo="int" longitud="4" requerido="true"/>
      <propiedad nombre"precioUnitario" estereotipo="DINERO" requerido="true"/>
      <propiedad nombre="importe" estereotipo="DINERO">
            <calculador clase="org.openxava.test.calculadores.CalculadorImporteLinea">
                  <poner propiedad="precioUnitario"/>
                  <poner propiedad="cantidad"/>
            </calculador>
      </propiedad>
      <referencia modelo="Producto" requerido="true"/>
      <propiedad nombre="fechaEntrega" tipo="java.util.Date">
            <calculador-valor-defecto
                  clase="org.openxava.calculators.CurrentDateCalculator"/>
      </propiedad>
```

```
    <referencia nombre="vendidoPor" modelo="Comercial"/>
    <propiedad nombre="observaciones" estereotipo="MEMO"/>
</agregado>
```

Vemos una colección de `LineaFactura` que es un agregado, `LineaFactura` se ha de mapear así:

```
<mapeo-agregado agregado="LineaFactura" tabla="XAVATEST@separator@LINEAFACTURA">
    <mapeo-referencia referencia="factura"> (1)
        <detalle-mapeo-referencia
            columna-tabla="FACTURA_AÃ‘O"
            propiedad-modelo-referenciado="aÃ±o"/>
        <detalle-mapeo-referencia
            columna-tabla="FACTURA_NUMERO"
            propiedad-modelo-referenciado="numero"/>
    </mapeo-referencia>
    <mapeo-propiedad propiedad-modelo="oid" columna-tabla="OID"/>
    <mapeo-propiedad propiedad-modelo="tipoServicio" columna-tabla="TIPOSERVICIO"/>
    <mapeo-propiedad
        propiedad-modelo="precioUnitario" columna-tabla="PRECIOUNITARIO"/>
    <mapeo-propiedad propiedad-modelo="cantidad" columna-tabla="CANTIDAD"/>
    <mapeo-referencia referencia="producto">
        <detalle-mapeo-referencia
            columna-tabla="PRODUCTO_CODIGO"
            propiedad-modelo-referenciado="codigo"/>
    </mapeo-referencia>
    <mapeo-propiedad-multiple propiedad-modelo="fechaEntrega">
        <conversor clase="org.openxava.converters.Date3Converter"/>
        <campo-cmp
            propiedad-conversor="day"
            columna-tabla="DIAENTREGA" tipo-cmp="int"/>
        <campo-cmp
            propiedad-conversor="month"
            columna-tabla="MESENTREGA" tipo-cmp="int"/>
        <campo-cmp
            propiedad-conversor="year"
            columna-tabla="AÃ‘OENTREGA" tipo-cmp="int"/>
    </mapeo-propiedad-multiple>
    <mapeo-referencia referencia="vendidoPor">
        <detalle-mapeo-referencia
            columna-tabla="VENDIDOPOR_CODIGO"
            propiedad-modelo-referenciado="codigo"/>
    </mapeo-referencia>
    <mapeo-propiedad
```

```
            propiedad-modelo="observaciones" columna-tabla="OBSERVACIONES"
</aggregate-mapping>
```

Los mapeos de agregado se ponen a continuación del mapeo de entidad, y debe haber tantos como agregados usados en colecciones. El mapeo de un agregado tiene exactamente las misma posibilidades que el mapeo de entidad ya visto, con la salvedad de que necesitamos definir el mapeo de la referencia al objeto contenedor aunque ella no esté en el modelo. Esto es aunque nosotros no definamos en `LineaFactura` una referencia a `Factura`, el OpenXava la añade automáticamente y por ende nosotros en el mapeo tenemos que reflejarlo (1).

5.7 Conversores por defecto

Vemos como podemos declarar un conversor a cada mapeo de propiedad. Pero ¿qué pasa cuando no declaramos conversor? En realidad en OpenXava todas las propiedades (a excepción de las que son clave) tienen un conversor aunque no se indique explícitamente. Los conversores por defecto están definidos en el archivo *OpenXava/xava/default-converters.xml*, que tiene un contenido como este:

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

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

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

<converters>

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

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

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

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

     <for-type type="java.lang.Boolean"
```

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

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

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

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

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

</converters>
```

Si usamos una propiedad de un tipo que no tenemos definido aquí por defecto se le asigna el conversor `NoConversionConverter`, que es un conversor tonto que no hace nada.

En el caso de las propiedades clave no se asigna conversor en absoluto, aplicar conversor a las propiedades claves puede ser problemático en ciertas circunstancias, pero si aun así lo queremos podemos declarar explicitamente en nuestro mapeo un conversor para una propiedad clave y se le aplicará.

Si queremos modificar el comportamiento de los conversores por defecto para nuestra aplicación no debemos modificar este archivo sino crear uno llamado *converters.xml* o *conversores.xml* en el directorio *xava* de nuestro proyecto. Podemos asignar también conversor por defecto a un estereotipo (usando `<para-estereotipo/>` o `<for-stereotype/>`).

5.8 Mapeo por defecto (nuevo en v2.1.3)

Desde la versión 2.1.3 OpenXava permite definir componentes sin mapeo, y se asume un mapeo por defecto para él. Por ejemplo, podemos escribir:

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

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

<componente nombre="Alumno">

    <entidad>
```

```
        <propiedad nombre="numero" tipo="int" clave="true"
            longitud="2" requerido="true"/>
        <propiedad nombre="nombre" tipo="String"
            longitud="40" requerido="true"/>
        <referencia nombre="profesor"/>
    </entidad>

</componente>
```

Este componente está mapeado a la tabla `Alumno`, y las propiedades `numero` y `nombre` están mapeadas a las columnas `numero` y `nombre`. La referencia profesor está mapeada a una columna llamada `profesor_numero` (si la propiedad clave de `Profesor` se llama `numero`).

5.9 Filosofia del mapeo objeto-relacional

OpenXava ha nacido y se ha desarrollado en un entorno en el que nos hemos visto obligados a trabajar con bases de datos existentes sin poder modificar su estructura, esto hace que:

- Ofrezca gran flexibilidad para mapear contra bases de datos legadas.
- No ofrezca algunas características propias de la OOT y que requieren poder manipular el esquema, como por ejemplo el soporte de herencia o consultas polimórficas.

Otras característica importante del mapeo de OpenXava es que las aplicaciones son 100% portables entre JBoss CMP2 y Websphere CMP2 sin tener que reescribir nada por parte del desarrollador. Además, la portabilidad entre Hibernate, JPA y EJB2 de una aplicación es muy alta, los mapeos y todos los controladores automáticos son portables al 100%, obviamente el código propio escrito con EJB2, JPA o Hibernate no lo es tanto.

6 Aspectos

6.1 Introducción a AOP

AOP (Programación Orientada a Aspectos) introduce una nueva forma de reutilizar código. Realmente complementa algunas deficiencias de la programación orientada a objetos tradicional.

¿Qué problema resuelve AOP? A veces tenemos funcionalidad que es común a un grupo de clases pero usar herencia no es práctico (en Java solo contamos con herencia simple) ni ético (porque se rompe la relación *es-un*). Además el sistema puede esta ya escrito, o quizá necesitamos poder incluir o no la funcionalidad bajo demanda. AOP es una manera fácil de resolver estos problemas.

¿Qué es un aspecto? Un aspecto es un puñado de código que puede ser esparcido a nuestro gusto por la aplicación.

El lenguaje Java ofrece un soporte completo de AOP mediante el proyecto AspectJ.

OpenXava añade algún soporte para el concepto de *aspectos* desde la versión 1.2.1. De momento el soporte es bastante limitado y OpenXava está todavía muy lejos de un marco de trabajo AOP al uso, pero el soporte de aspectos dentro de OpenXava se ha mostrado útil.

6.2 Definición de aspectos

El archivo *aspectos.xml* dentro de la carpeta *xava* de nuestro proyecto se usa para definir los aspectos.

Su sintaxis es:

```
<aspectos>
     <aspecto ... /> ...              (1)
     <aplicar ... /> ...              (2)
</aspectos>
```

(1)`aspecto` (varios, opcional): Para definir aspectos.

(2)`aplicar` (varios, opcional): Para aplicar los aspectos definidos a los modelos seleccionados.

Con `aspecto` (1) podemos definir un aspecto (es decir un grupo de características) con un nombre, y usando `aplicar` (2) conseguimos que ese conjunto de modelos (entidades o agregados) obtengan esas características automáticamente.

Veamos la sintaxis para un aspecto:

```
<aspecto
     nombre="nombre"                        (1)
>
     <calculador-poscrear .../> ...         (2)
     <calculador-poscargar .../> ...        (3)
     <calculador-posmodificar .../> ...     (4)
     <calculador-preborrar .../> ...        (5)
```

```
</aspecto>
```

(1)`nombre` (obligado): Nombre para este aspecto. Tiene que ser único.

(2)`calculador-poscrear` (varios, opcional): Todos los modelos con este aspecto tendrán este `calculador-poscrear` implícitamente.

(3)`calculador-poscargar` (varios, opcional): Todos los modelos con este aspecto tendrán este `calculador-poscargar` implícitamente.

(4)`calculador-posmodify` (varios, opcional): Todos los modelos con este aspecto tendrán este `calculador-posmodify` implícitamente.

(5)`calculador-preborrar` (varios, opcional): Todos los modelos con este aspecto tendrán este `calculador-preborrar` implícitamente.

Además, tenemos que asignar el aspecto que hemos definido a nuestros modelos. La sintaxis para eso es:

```
<aplicar
     aspecto="aspecto"                (1)
     para-modelos="modelos"           (2)
     excepto-para-modelos="modelos"   (3)
/>
```

(1)`aspecto` (obligado): El nombre del aspecto que queremos aplicar.

(2)`para-modelos` (opcional): Una lista de modelos, separados por coma, para aplicarles este aspecto. Es exclusivo con el atributo `excepto-para-modelos`.

(3)`excepto-para-modelos` (opcional): Una lista de modelos, separados por comas, para excluir cuando se aplique este aspecto. De esta forma el aspecto es aplicado a todos los modelos menos los indicados. Es exclusivo con el atributo `para-modelos`.

Si no usamos ni `para-modelos` ni `excepto-para-modelos` entonces el aspecto se aplicará a todos lo modelos de la aplicación. Los modelos son los nombres de componente (para sus entidades) o agregados.

Un ejemplo simple puede ser:

```
<aspecto nombre="MiAspecto">
     <calculador-poscrear
           clase="com.miempresa.miaplicacion.calculadores.MiCalculador"/>
</aspecto>
<aplicar aspecto="MiAspecto"/>
```

De esta forma tan sencilla podemos hacer que cuando un nuevo objeto se cree (se grabe en la base de datos por primera vez) la lógica en `MiCalculador` sea ejecutada. Y esto para todos los modelos.

Como se ve, solo unos pocos calculadores son soportados de momento. Esperamos extender las posibilidades de los *aspectos* en OpenXava en el futuro. De todas formas, estos calculadores ahora ofrecen posibilidades interesantes. Veamos un ejemplo en la siguientes sección.

6.3 AccessTracking: Una aplicación práctica de los aspectos

La distribución actual de OpenXava incluye el proyecto *AccessTracking*. Este proyecto define un aspecto que nos permite llevar la pista de todos los acceso a la información en nuestra aplicación. Actualmente, este proyecto permite que nuestras aplicaciones cumplan con la Ley de Protección de Datos española, incluyendo los datos con un nivel de seguridad alto. Aunque es lo suficientemente genérico para ser útil en una amplia variedad de circunstancia.

6.3.1 La definición del aspecto

Podemos encontrar la definición del aspecto en *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>
```

Cuando aplicamos este aspecto a nuestro componentes el código de `AccessTrackingCalculator` es ejecutado cada vez que un objeto es creado, cargado, modificado o borrado. `AccessTrackingCalculator` escribe un registro en una tabla de la base de datos con información acerca del acceso.

Para poder aplicar este aspecto necesitamos escribir en nuestro *aspectos.xml* algo como esto:

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

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

<aspectos>

     <aplicar aspecto="AccessTracking" para-modelos="Almacen, Factura"/>

</aspectos>
```

De esta forma este aspecto es aplicado a `Almacen` y `Factura`. Todos los accesos a estas entidades serán registrados en una tabla de base de datos.

6.3.2 Configurar AccessTracking

Si queremos usar el aspecto `AccessTracking` en nuestro proyecto hemos de seguir los siguientes pasos de configuración:

- Añadir *AccessTracking* como proyecto referenciado.
- Crear la tabla en nuestra base de datos para almacenar el registro de los accesos. Podemos encontrar los CREATE TABLEs en el archivo *AccessTracking/data/access-tracking-db.script*.
- Hemos de incluir la propiedad `hibernate.dialect` en nuestros archivos de configuración. Puedes ver ejemplos de esto en *OpenXavaTest/jboss-hypersonic.properties* y otros archivos *OpenXavaTest/xxx.properties*.
- Dentro del proyecto *AccessTracking* necesitamos seleccionar una configuración (editando build.xml) y regenerar código hibernate (usando la tarea ant `generateHibernate`) para *AccessTracking*.
- Editamos el archivo de nuestro proyecto *build/ejb/META-INF/MANIFEST.MF* para añadir los siguientes jars al classpath: `./lib/tracking.jar ./lib/ehcache.jar ./lib/antlr.jar ./lib/asm.jar ./lib/cglib.jar ./lib/hibernate3.jar ./lib/dom4j.jar.` (Este paso no es necesario si solo usamos POJOs, y no usamos EJB CMP2, *nuevo en v2.0*)

También necesitamos modificar la tarea ant `crearJarEJB` (solo si usamos EJB2 CMP) y `desplegarWar` de nuestro *build.xml* de esta forma:

```
<target name="crearJarEJB">     <!-- 'crearJarEJB' solo si usamos EJB2 CMP -->
     ...
     <ant antfile="../AccessTracking/build.xml" target="createEJBTracker"/>
</target>
<target name="desplegarWar">
     <ant antfile="../AccessTracking/build.xml" target="createTracker"/>
     ...
</target>
```

Después de todo esto, hemos de aplicar el aspecto en nuestra aplicación. Creamos un archivo *xava/aspectos.xml* en nuestro proyecto:

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

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

<aspectos>

     <aplicar aspecto="AccessTracking"/>

</aspectos>
```

Ahora solo nos queda desplegar el war para nuestro proyecto. *(nuevo en v2.0)*

En el caso de que usemos EJB2 CMP tenemos que regenerar código, desplegar EJB y desplegar el war para nuestro proyecto.

Los accesos son registrados en una tabla con el nombre TRACKING.ACCESS. Si lo deseamos podemos desplegar el módulo web o el portlet del proyecto *AccessTracking* para tener una aplicación web para examinar los accesos.

Para más detalle podemos echar un vistazo al proyecto *OpenXavaTest*.

7 Miscelánea

7.1 Relaciones de muchos-a-muchos

En OpenXava no existe el concepto directo de relación muchos-a-muchos, en lugar de eso en OpenXava solo existen colecciones. Aun así modelar una relación muchos-a-muchos con OpenXava is fácil. Solo necesitamos definir colecciones en ambas partes de la relación.

Por ejemplo, si tenemos clientes y provincias, y un cliente puede trabajar en varias provincias, y, obviamente, en una provincia pueden trabajar varios clientes, tenemos un caso de relación muchos-a-muchos (usando la nomenclatura relacional). Suponiendo que tenemos una tabla CLIENTE (sin referencia a provincia), una tabla PROVINCIA (sin referencia a clientes) y una tabla CLIENTE_PROVINCIA (para vincular ambas tablas), este caso podemos modelarlo así:

```
<componente nombre="Cliente">
      <entidad>
      ...
            <coleccion nombre="provincias">                    (1)
                  <referencia modelo="ClienteProvincia"/>
            </coleccion>
      ...
      </entidad>

      <agregado nombre="ClienteProvincia">                     (2)
            <referencia nombre="cliente" clave="true"/>        (3)
            <referencia nombre="provincia" clave="true"/>      (4)
      </agregado>
      ...
</componente>
```

Estamos definiendo en `Cliente` una colección de agregados (1), cada agregado (`ClienteProvincia`) (2) contiene una referencia a `Provincia` (4) y, por supuesto, una referencia a su entidad contenedora (`Cliente`) (3).

Ahora podemos mapear la colección de la forma habitual:

```
<componente nombre="Cliente">
      ...
      <mapeo-agregado agregado="ClienteProvincia" tabla="CLIENTE_PROVINCIA">
            <mapeo-referencia referencia="cliente">
                  <detalle-mapeo-referencia
                        columna-tabla="CLIENTE"                (1)
                        propiedad-modelo-referenciado="codigo"/>
            </mapeo-referencia>
            <mapeo-referencia referencia="provincia">
                  <detalle-mapeo-referencia
```

```
                        columna-tabla="PROVINCIA"                (2)
                        propiedad-modelo-referenciado="id"/>
            </mapeo-referencia>
      </mapeo-agregado>
</componente>
```

`ClienteProvincia` lo mapeamos a CLIENTE_PROVINCIA, una tabla que contiene dos columnas, una para apuntar a CLIENTE (1) y otra para apuntar a PROVINCIA (2).

A nivel de modelo y mapeo ya lo tenemos todo listo, pero la intefaz gráfica que genera OpenXava por defecto es un poco fea en este caso. Aunque con los siguientes retoques en la parte de la vista nuestra colección muchos-a-muchos puede quedar bastante bien:

```
<componente nombre="Cliente">
      ...
      <vista>
            ...
            <vista-coleccion coleccion="provincias">
                  <propiedades-lista>
                        provincia.id, provincia.nombre                  (1)
                  </list-properties>
            </vista-coleccion>

            <miembros>
                  ...
                  provincias
                  ...
            </miembros>

      </vista>

      <vista modelo="ClienteProvincia">
            <vista-referencia referencia="provincia" marco="false"/>    (2)
      </vista>
      ...
</componente>
```

En esta vista podemos ver como definimos explicitamente (1) las propiedades a mostrar en la lista de la colección `provincias`. Esto es necesario porque tenemos que mostrar las propiedades de `Provincia`, no las de `ClienteProvincia`. Adicionalmente, definimos que la referencia a `Provincia` en `ClienteProvincia` se muestre sin marcos (2), de esta forma evitamos dos feos marcos anidados.

De esta manera podemos definir una colección que mapea a una relación de muchos a muchos en la base de datos. Si queremos que la relación sea bidireccional solo tenemos que crear una colección

`clientes` en la entidad `Provincia`, esta colección puede ser de type aggregado `ProvinciaCliente` y tiene que mapearse a la tabla CLIENTE_PROVINCIA. Todo exactamente igual que en el ejemplo aquí mostrado.