GUÍA DE REFERENCIA

VERSIÓN **3**.**0**

Contenido

Capítulo 1: **Visión general**

Presentación

OpenXava es un marco de trabajo para desarrollar aplicaciones JavaEE/J2EE rápida y fácilmente.
La filosofía subyacente es definir con anotaciones de Java o con XML y programar con Java, pero cuanto más definimos y menos programamos mejor. El objetivo principal es hacer que las cosas más típicas en una aplicación de gestión sean fáciles de hacer, mientras que ofrecemos la flexibilidad suficiente para desarrollar las funciones más avanzadas y especificas.

A continuación se echa un vistazo a algunos conceptos básico de OpenXava.

Componente de negocio

Las piezas fundamentales para crear una aplicación OpenXava son los componentes, en el contexto de OpenXava un componente de negocio es una clase Java (aunque existe también una versión XML) que contiene toda la información necesaria sobre un concepto de negocio para poder crear aplicaciones sobre eso. Es decir toda la información que el sistema ha de saber sobre el concepto de factura se define en un archivo *Factura.java*. En un componente de negocio se define:

- La estructura de datos.
- Las validaciones, cálculos y en general toda la lógica de negocio asociada a ese concepto.
- Las posibles vista, esto es, la configuración de todos las posibles interfaces gráficas para este componente.
- Se define las posibilidades para la presentación tabular de los datos. Esto se usa para el modo lista (consultar y navegar por los datos), los listados, exportación a excel, etc.
- Mapeo objeto-relacional, lo que incluye información sobre las tablas de la base de datos y la forma de convertir a objetos la información que en ellas hay.

Esta forma de dividir es ideal para el trabajo en grupo, y permite desarrollar un conjunto de componentes interproyecto.

Controladores

Los componentes de negocio no definen lo que un usuario puede hacer con la aplicación; esto se define con los controladores. Los controladores están en el archivo *xava/controladores.xml* de cada proyecto; además OpenXava tiene un conjunto de controladores predefinidos en *OpenXava/xava/default-controllers.xml*.
Un controlador es un conjunto de acciones. Una acción es un botón o vínculo que el usuario puede pulsar.
Los controladores están separados de los componentes de negocio porque un mismo controlador puede ser asignado a diferentes componentes de negocio. Por ejemplo, un controlador para hacer un mantenimiento, imprimir en PDF, o exportar a archivos planos, etc. puede ser usado y reusado para facturas, clientes, proveedores, etc.

Aplicación

Una aplicación OpenXava es un conjunto de módulos. Un módulo une un componente de negocio con uno o más controladores. Cada módulo de la aplicación es lo que al final utiliza el usuario, y generalmente se configura como un portlet dentro de un portal.

Estructura de un proyecto

Un proyecto OpenXava típico suele contener las siguientes carpetas:

- [**raiz**]: En la raíz del proyecto nos encontraremos el *build.xml* (con las tareas Ant).
- **src**[**carpeta fuente**]: contiene el código fuente Java escrito por nosotros.
- **xava**: Los archivos XML para configurar nuestras aplicaciones OpenXava. Los principales son *aplicacion.xml* y *controladores.xml*.
- **i18n**: Archivos de recursos con las etiquetas y mensajes en varios idiomas.

- **properties**[**carpeta fuente**]: Archivos de propiedades para configurar nuestro aplicación.
- **data**: Útil para guardar los scripts para crear las tablas de nuestra aplicación, si aplicara.
- **web**: Contenido de la parte web. Normalmente archivos JSP, lib y classes. La mayoría del contenido es puesto automáticamente, pero es posible poner aquí nuestros propios archivos JSP.

Crear un proyecto nuevo

Una vez abierto el Eclipse y apuntando al *workspace* que viene en la distribución de OpenXava.
Usando el asistente de Eclipse apropiado hemos de crear un nuevo proyecto Java llamado *Gestion*. Ahora tenemos un proyecto Java vacío en el *workspace*, el siguiente paso es darle la estructura correcta para un proyecto OpenXava.
Vamos al proyecto *OpenXavaPlantilla* y editamos el archivo *CrearNuevoProyecto.xml* de esta forma:

```
<property name="proyecto" value="Gestion" />
```

Ahora hemos de ejecutar *CrearNuevoProyecto.xml* usando Ant. Podemos hacerlo con *Botón Derecho en CrearNuevoProyecto.xml > Run as > Ant Build*
Seleccionamos el proyecto *Gestion* y pulsamos F5 para refrescar.
Con esto ya tenemos nuestro proyecto listo para empezar a trabajar, pero antes de nada tenemos que tener una base de datos configurada.

Configurar base de datos

OpenXava genera una aplicación Java EE/J2EE pensada para ser desplegada en un servidor de aplicaciones Java (desde la v2.0 las aplicaciones OpenXava también funcionan en un simple servidor de servlets, como Tomcat). Dentro de OpenXava solo se indica el nombre JNDI de la fuente de datos, y en nuestro servidor de aplicaciones tenemos que configurar nosotros esa base datos. El configurar una fuente de datos en un servidor de aplicaciones es algo que va más allá de esta guía, sin embargo a continuación se da las instrucciones concretas para poder realizar este primer proyecto usando el Tomcat incluido en la distribución de OpenXava e Hypersonic como base de datos. Este Tomcat esta en la carpeta *openxava-3.x/tomcat*.
Con el Tomcat parado editar el archivo *context.xml* en el directorio de Tomcat *conf*, en ese archivo tenemos que añadir la siguiente entrada:

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

Lo importante aquí es el nombre JNDI, que es a lo único que se hace referencia desde OpenXava, en este caso *MiGestionDS*. Los atributos *driverClassName* y *url* dependen de nuestra base de datos, en este caso estamos usando Hypersonic.

Nuestro primer componente de negocio

Crear un componente de negocio OpenXava es fácil: La definición para cada componente es una clase Java con anotaciones. Para empezar vamos a crear una clase llamada *Almacen*:

- Nos ponemos en la carpeta *src* y usamos *Botón Derecho > New > Package*
- Creamos un paquete llamado *org.openxava.gestion.modelo*
- Nos ponemos en el paquete *org.openxava.gestion.modelo* y usamos *Botón Derecho > New > Class*
- Creamos una clase llamada *Almacen*

Ahora hemos de editar nuestra nueva clase y escribir el siguiente código:

```
package org.openxava.gestion.modelo;

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

@Entity
public class Almacen {

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

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

    public int getCodigo() {
        return codigo;
    }

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

```
    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

}
```

Esta clase contiene (y contendrá) toda la información necesaria por la aplicación sobre el concepto de negocio de Almacen. En este momento sólo tenemos la estructura de datos, pero en esta clase podemos poner el mapeo contra la base de datos, la lógica de negocio, la presentación visual, el manejo de datos tabulares, etc.

En realidad esta clase es una *Entity* que sigue el estándar EJB3. Para definir una clase como una entidad lo único que necesitamos es usar la anotación *@Entity* en la declaración de la clase.

Dentro de entidad vemos definidas un conjunto de propiedades, vamos a examinarlo:

```
@Id                                 // 1
@Column(length=3)                     // 2
@Required                            // 3
private int codigo;                    // 4
private int getCodigo() {              // 4
    return codigo;
}
private void setCodigo(int codigo) { // 4
    this.codigo = codigo;
}
```

Este es su significado:

1. **@Id**: Indica si esta propiedad forma parte de la clave. La clave identifica a cada objeto de forma única y normalmente coincide con la clave en la tabla de base de datos.
2. **@Column**(**length**=): Longitud de los datos visualizados. Es opcional, pero suele ser útil para hacer mejores interfaces gráficos y generar las tablas de la base de datos.
3. **@Required**: Indica si hay que validar la existencia de información en esta propiedad antes de crear o modificar.
4. La propiedad definida de la forma usual para una clase Java. Todo tipo válido para una propiedad Java se puede poner, lo que incluye tipos integrados, clases del JDK, clases propias, etc.

Las posibilidades de una propiedad van mucho más allá de lo que aquí se muestra, se puede ver una explicación más completa en el capitulo sobre el modelo.

La tabla

Antes de poder probar la aplicación hemos de crear la tabla en la base de datos:

- Arrancamos la base de datos: Desde la línea de órdenes vamos a la carpeta *openxava-3.x/tomcat/bin* y ejecutamos:
  - En Linux/Unix: *./start-hsqldb.sh gestion-db 1666*
  - En Windows: *start-hsqldb gestion-db 1666*
- Creamos la tabla:
  - Editamos *Gestion/build.xml*. Buscamos la tarea ant *actualizarEsquema*.
  - Ponemos el valor correcto para *schema.path*, en este caso *"../OpenXavaTest/lib/hsqldb.jar"*.
  - Ejecutamos la tarea ant *actualizarEsquema*.
- Arrancamos el Tomcat y ya está todo listo.

Ejecutar nuestra aplicación

Después de nuestro duro trabajo tenemos derecho a ver el fruto de nuestro sudor, así que allá vamos:

- Ejecutamos la tarea ant *desplegarWar.*
- Abrimos un navegador de internet y vamos a la dirección http://localhost:8080/Gestion/xava/module.jsp?application=Gestion&module=Almacen

Y ahora podemos jugar con nuestro módulo y ver como funciona.
También puedes desplegar el módulo como portlet JSR-168, de esta forma:

- Ejecutamos la tarea ant *generarPortlets*.
- Coge el archivo *Gestion.war* en la carpeta *openxava-3.x/workspace.dist/Gestion.dist* y despliegalo en tu portal.

Automatizando las pruebas

Aunque parece que lo más natural es probar la aplicación con un navegador e ir viendo lo mismo que verá el usuario; lo cierto es que es más productivo automatizar las pruebas, de esta forma a medida que nuestro sistema crece, lo tenemos atado y evitamos que al avanzar rompamos lo que ya teníamos.
OpenXava usa un sistema de pruebas basado en JUnit y HttpUnit. Las pruebas JUnit de OpenXava emulan el funcionamiento de un usuario real con un navegador, de esta forma podemos replicar de forma exacta las mismas pruebas que haríamos nosotros mismos con un navegador. La ventaja de este enfoque es que probamos de forma sencilla desde el interfaz gráfico al acceso a la base de datos.
Si probáramos el modulito manualmente normalmente crearíamos un registro nuevo, lo buscaríamos, lo modificaríamos y lo borraríamos. Vamos a hacer eso automáticamente.
En primer lugar crearemos un paquete en donde poner las pruebas, *org.openxava.gestion.pruebas*, y en este paquete pondremos una clase llamada *PruebaAlmacen*, y pegaremos en ella el siguiente código:

```
package org.openxava.gestion.pruebas;

import org.openxava.tests.*;

/**
 * @author Javier Paniza
 */

public class PruebaAlmacen extends ModuleTestBase {

    public PruebaAlmacen(String testName) {
        super(testName, "Gestion", "Almacen");    // 1
    }

    public void testCrearLeerModificarBorrar() throws Exception {
        // Creamos
        execute("CRUD.new");                      // 2
        setValue("codigo", "7");                  // 3
        setValue("nombre", "Almacen JUNIT");
        execute("CRUD.save");
        assertNoErrors();                          // 4
        assertValue("codigo", "");                // 5
        assertValue("nombre", "");

        // Leeemos
        setValue("codigo", "7");
        execute("CRUD.search");
        assertValue("codigo", "7");
        assertValue("nombre", "Almacen JUNIT");
```

```
        // Modificamos
        setValue("nombre", "Almacen JUNIT MODIFICADO");
        execute("CRUD.save");
        assertNoErrors();
        assertValue("codigo", "");
        assertValue("nombre", "");

        // Comprobamos modificado
        setValue("codigo", "7");
        execute("CRUD.search");
        assertValue("codigo", "7");
        assertValue("nombre", "Almacen JUNIT MODIFICADO");

        // Borramos
        execute("CRUD.delete");
        assertMessage("Almacen borrado satisfactoriamente"); // 6
    }

}
```

Podemos aprender de este ejemplo:

1. **Constructor**: En el constructor indicamos el nombre de la aplicación y el nombre del módulo.
2. **execute**: Permite simular la pulsación de un botón o vínculo. Como argumento se envía el nombre de la acción; los nombres de las acciones los podemos ver en *OpenXava/xava/default-controllers.xml* (los controladores predefinidos) y *Gestion/xava/controladores.xml* (los propios). También si paseamos el ratón sobre el vínculo el navegador nos mostrará la acción JavaScript a ejecutar, que contiene el nombre de acción OpenXava. Es decir *execute("CRUD.new")* es como pulsar el botón de nuevo en la interfaz gráfica.
3. **setValue**: Para asignar valor a un control del formulario. Es decir, *setValue("nombre", "Pepe")* tiene el mismo efecto que teclear en el campo de texto 'Pepe'. Los valores siempre son alfanuméricos, ya que se asignan a un formulario HTML.
4. **assertNoErrors**: Comprueba que no se hayan producido errores. En la interfaz gráfica los errores son mensajes en color rojo, que son añadidos por la lógica de la aplicación.
5. **assertValue**: Comprueba que el valor contenido en un elemento del formulario es el indicado.
6. **assertMessage**: Verifica que la aplicación ha producido el mensaje informativo indicado.

Se puede ver como de forma sencilla podemos probar que el mantenimiento funciona, escribir un código como este puede costar 5 minutos, pero a la

larga ahorra horas de trabajo, porque a partir de ahora podemos probarlo todo en 1 segundo, y porque nos va a avisar a tiempo cuando rompamos la gestión de Almacenes tocando otra cosa.
Para más detalle podemos ver el API JavaDoc de *org.openxava.tests.ModuleTestBase* y examinar los ejemplos que hay en *org.openxava.test.tests* de OpenXavaTest.
Por defecto la prueba se ejecuta contra el módulo en modo solitario (fuera del portal) (es decir desplegado con *desplegarWar*). Pero si lo deseamos es posible testear contra la versión de portlet (es decir desplegado con *generarPortlets*). Solo necesitamos editar el archivo *properties/xava-junit.properties* y escribir:

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

Esto para probar contra el portal Liferay. También es posible probar contra el portal JetSpeed2, mira en *OpenXavaTest/properties/xava-junit.properties* para ver cómo.

Las etiquetas

Ya nos funciona, pero hay un pequeño detalle que se ha quedado suelto. Posiblemente queramos definir las etiquetas que se mostrarán al usuario. La forma de hacerlo es escribiendo un archivo con todas las etiquetas, y así podemos traducir nuestro producto a otro idioma con facilidad.
Para definir las etiqueta solo tenemos que editar el archivo *EtiquetasGestion_es.properties* en la carpeta *i18n*. Editar ese archivo y añadir:

```
Almacen=Almacén
```

No es necesario poner todas las propiedades, porque los casos más comunes (codigo, nombre, descripcion y un largo etc) ya los tiene OpenXava incluidos en Español, Inglés, Polaco, Alemán, Francés, Indonesio y Catalán.
Si queremos una versión en otro idioma (inglés, por ejemplo) solo tenemos que copiar y pegar con el sufijo apropiado. Por ejemplo, podemos tener un *EtiquetasGestion_en.properties* con el siguiente contenido:

```
Almacen=Warehouse
```

Las etiquetas y mensaje por defecto de OpenXava están en *OpenXava/i18na/Labels.properties* y *OpenXava/i18n/Messages.properties*. Si queremos sobreescribir algunos de estos recursos no necesitamos editar estos archivos,

sino que podemos usar los mismos nombres de clave en los archivos de recursos de nuestro proyecto, entonces nuestras etiqueta y mensajes seran usadas en vez de las estándar de OpenXava *(nuevo en v2.0.3)*. Por ejemplo, si queremos cambiar el mensaje estandar en modo lista ''*Hay 23663 registros en la lista"* por otro, hemos de añadir a *MensajesGestion_es.properties* esta entrada:

```
# list_count está en Messages_en.properties de OpenXava, este es un ejemplo
# de sobreescritura de un mensaje  estándar de OpenXava
list_count=Hay {0} objetos en la lista
```

Ahora, nuestra aplicación mostrará *"Hay 23663 objetos en la lista"* en vez del mensaje por defecto de OpenXava "''*Hay 23663 registros en la lista".*
Para saber más sobre como definir las etiquetas de nuestros elementos OpenXava podemos echar un vistazo a los archivos de *OpenXavaTest/i18n*.

Capítulo 3: 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 simples clases Java (aunque también existe una versión XML) y un poquito de Java. OpenXava provee una aplicación completamente funcional a partir de la definición del modelo.

Componente de negocio

La unidad básica para crear aplicaciones OpenXava es el componente de negocio. Un componente de negocio se define usando una clase Java llamada *Entity.* Esta clase es una entidad EJB3 convencional, o con otras palabras, un POJO con anotaciones que sigue el estándar Java Persistence API (JPA).
JPA es el estándar de Java para la persistencia, es decir, para objetos que guardan su estado en una base de datos. Si sabes desarrollar usando POJOs con JPA, ya sabes como desarrollar aplicaciones OpenXava.
Usando una simple clase Java podemos definir un Componente de Negocio con:
Using this classes you can define a Business Component with:

- **Modelo**: Estrutura de datos, validaciones, calculos, etc.
- **Vista**: Cómo se puede mostrar el modelo al usuario.
- **Datos tabulares**: Cómo se muestra los datos de este componentes en modo lista (en formato tabular).
- **Mapeo objeto/relacional**: Cómo grabar y leer el estado de los objetos desde la base de datos.

Este capítulo explica cómo definir la parte del modelo, es decir, todo sobre la estructura, las validaciones, los cálculos, etc.

Entidad

Para define la parte del modelo hemos de definir a clase Java con anotaciones. Además de sus propias anotaciones, OpenXava sporta anotaciones de JPA e Hibernate Validator. Esta clase Java es una entidad, es decir, una clase persistente que representa concepto de negocio.
En este capítulo JPA se usa para indicar que es una anotación estándar de Java Persistent API, HV para indicar que es una anotación de Hibernate Validator, y OX para indicar que es una anotación de OpenXava.
Ésta es la sintáxis para una entidad:

```
@Entity                    //  1
@EntityValidator            //  2
@RemoveValidator            //  3
public class NombreEntidad {  //  4
  // Propiedades            //  5
  // Referencias            //  6
  // Colecciones            //  7
  // Métodos                //  8
  // Buscadores             //  9
  // Métodos de retrollamada  // 10
}
```

1. **@Entity** (JPA, uno, obligado): Indica que esta clase es una entidad JPA, con otras palabras, sus instancias serán objetos persistentes.
2. **@EntityValidator** (OX, 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.
3. **@RemoveValidator** (OX, varios, opcional): Se ejecuta antes de borrar, y tiene la posibilidad de vetar el borrado del objeto.
4. **Declaración de la clase**: Como en un clase de Java convencional. Podemos usar *extends* e *implements*.
5. **Propiedades**: Propiedades de Java convencionales. Representan el estado principal del objeto.
6. **Referencias**: Referencias a otras entidades.
7. **Colecciones**: Colecciones de referencias a otras entidades.
8. **Métodos**: Métodos Java con lógica de negocio.
9. **Buscadores**: Los buscadores son métodos estáticos que hacen búsquedas usando las prestaciones de consulta de JPA.
10. **Métodos de retrollamada**: Los métodos JPA de retrollamada (*callbacks*) para insertar lógica al crear, modificar, cargar, borrar, etc

Propiedades

Una propiedad 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:

```
@Stereotype                                          //  1
@Column(length=) @Max @Length(max=) @Digits(integerDigits=)  //  2
@Digits(fractionalDigits=)                             //  3
@Required @Min @Range(min=) @Length(min=)                //  4
@Id                                                 //  5
@Hidden                                              //  6
@SearchKey                                           //  7
@Version                                             //  8
@DefaultValueCalculator                                 //  9
@PropertyValidator                                     // 10
private tipo nombrePropiedad;                           // 11
public tipo getNombrePropiedad() { … }                 // 11
public void setNombrePropiedad(tipo nuevoValor) { … }      // 11
```

1. **@Stereotype** (OX, opcional): Permite especificar un comportamiento especial para cierta propiedades.
2. **@Column**(**length**=) (JPA), **@Max** (HV), **@Length**(**max**=) (HV), **@Digits**(**integerDigits**=) (HV, opcional, normalmente solo se usa una): Longitud en caracteres de la propiedad, excepto para *@Max* que es el valor máximo. 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*.
3. **@Digits**(**fractionalDigits**=) (HV, opcional): 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*.
4. **@Required** (OX), **@Min** (HV), **@Range**(**min**=) (HV), **@Length**(**min**=) (HV) (opcional, normalmente solo se usa una): Indica si esa propiedad es requerida. En el caso de *@Min*, *@Range* y *@Length* tenemos que poner un valor mayor que cero para *min* para que se asuma la propiedad como requerida. Por defecto es true para las propiedades clave ocultas *(nuevo en v2.1.3)* 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*.

5. **@Id** (JPA, 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.
6. **@Hidden** (OX, 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.
7. **@SearchKey** (OX, optional): 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 (*@Id*) para buscar por defecto, y si la propiedades clave (*@Id*) están ocultas usa la primera propiedad en la vista. Con *@SearchKey* podemos elegir las propiedades para buscar explicitamente.
8. **@Version** (JPA, opcional): Una propiedad versión se usa para el control de concurrencia optimista. Si queremos control de concurrencia solo necesitamos tener una propiedad marcada como *@Version* en nuestra entidad. Solo podemos especificar una propiedad de versión por entidad. Los siguientes tipos son soportados para propiedades versión: *int, Integer, short, Short, long, Long, Timestamp*. Las propiedades de versión también se consideran ocultas.
9. **@DefaultValueCalculator** (OX, uno, optional): Para implementar la lógica para calcular el valor inicial de la propiedad. Una propiedad con *@DefaultValueCalculator* sí tiene *setter* y es persistente.
10. **@PropertyValidator** (OX, varios, opcional): Indica la lógica de validación a ejecutar sobre el valor a asignar a esta propiedad antes de crear o modificar.
11. **Declaración de la propiedad**: Una declaración de propiedad Java normal y corriente con *getters* y *setters*. Podemos crear una propiedad calculada usando solo un *getter* sin campo ni *setter*. Cualquier tipo legal para JPA está permitido, solo hemos de proveer

un *Hibernate Type* para grabar en la base de datos y un editor OpenXava para dibujar como HTML.

Estereotipo

Un estereotipo (*@Stereotype*) 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)
- RELLENADO_CON_CEROS, ZEROS_FILLED
- TEXTO_HTML, HTML_TEXT (texto con formato editable)
- ETIQUETA_IMAGEN, IMAGE_LABEL (imagen que depende del contenido de la propiedad)
- EMAIL
- TELEFONO, TELEPHONE
- WEBURL
- IP
- ISBN
- TARJETA_CREDITO, CREDIT_CARD
- LISTA_EMAIL, EMAIL_LIST

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.
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:

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

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

Estereotipo GALERIA_IMAGENES

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:

```
@Stereotype("GALERIA_IMAGENES")
private String fotos;
```

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) .
Y finalmente necesitamos definir el mapeo en nuestro archivo *persistence/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.

Concurrencia y propiedad versión

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*, de esta manera:

```
@Version
private int version;
```

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

Enums

OpenXava suporta *enums* de Java 5. Un *enum* permite definir una propiedad que solo puede contener los valores indicados.
Es fácil de usar, veamos un ejemplo:

```
private Distancia distancia;
public enum Distancia { LOCAL, NACIONAL, INTERNACIONAL };
```

La propiedad *distancia* solo puede valer LOCAL, NACIONAL o INTERNACIONAL, y como no hemos puesto *@Required* también permite valor vacío (null).
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 base datos por defecto guarda el entero (0 para LOCAL, 1 para NACIONAL, 2 para INTERNACIONAL y null para cuando no hay valor), pero esto se puede configurar fácilmente para poder usar sin problemas bases de datos legadas. Ver más de esto último en el capítulo sobre mapeo.

Propiedades calculadas

Las propiedades calculadas son de solo lectura (solo tienen *getter*) y no persistentes (no se almacenan en ninguna columna de la tabla de base de datos).
Una propiedad calculada se define de esta manera:

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

De acuerdo con esta definición ahora podemos usar el código de esta manera:

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

Y *resultado* contendrá 332,772.
Cuando la propiedad *precioUnitarioEnPesetas* se visualiza al usuario no es editable, y su editor tiene una longitud de 10, indicado usando *@Max(9999999999L)* (2). También, dado que usamos *@Depends("precioUnitario")* (1) cuando el usuario cambie la propiedad *precioUnitario* en la interfaz de usuario la propiedad *precioUnitarioEnPesetas* será recalculada y su valor será refrescado de cara al usuario.
Desde una propiedad calculada tenemos acceso a conexiones JDBC. Un ejemplo:

```
@Max(999)
public int getCantidadLineas() {
    // Un ejemplo de uso de JDBC
    Connection con = null;
    try {
        con = DataSourceConnectionProvider.getByComponent("Factura").getConnection();  // 1
        String tabla = MetaModel.get("LineaFactura").getMapping().getTable();
        PreparedStatement ps = con.prepareStatement("select count(*) from " + tabla +
            " 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;
    }
    catch (Exception ex) {
        log.error("Problemas al calcular cantidad de líneas de una Factura", ex);
        // Podemos lanzar cualquier RuntimeException aquí
        throw new SystemException(ex);
    }
    finally {
        try {
            con.close();
        }
        catch (Exception ex) {
        }
    }
}
```

Es verdad, el código JDBC es feo y complicado, pero a veces puede ayudar a resolver problemas de rendimiento. La clase *DataSourceConnectionProvider* nos permite obtener la conexión asociada a la misma fuente de datos que la entidad indicada (en este caso *Factura*). Esta clase es para nuestra conveniencia, también podemos acceder a una conexión JDBC usando JNDI o cualquier otro medio que queramos. De hecho, en una propiedad calculada podemos escribir cualquier código que Java nos permita.

Calculador valor por defecto

Con *@DefaultValueCalculator* podemos asociar lógica a una propiedad, en este caso la propiedad es lectura y escritura. Este calculador se usa para calcular el valor inicial. Por ejemplo:

```
@DefaultValueCalculator(CurrentYearCalculator.class)
private int año;
```

En este caso cuando el usuario intenta crear una nueva factura (por ejemplo) se encontrará con que el campo de año ya tiene valor, que él puede cambiar si quiere. La lógica para generar este valor está en la clase *CurrentYearCalculator* class, así:

```
package org.openxava.calculators;

import java.util.*;

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

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

}
```

Es posible personalizar el comportamiento de un calculador poniendo el valor de sus propiedades, como sigue:

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

En este caso para calcular el valor por defecto OpenXava instancia *StringCalculator* y entonces inyecta el valor "BUENA" en la propiedad *string* de *StringCalculator*, y finalmente llama al método *calculate()* para obtener el valor por defecto para *relacionConComercial*. Como se ve, el uso de la anotación *@PropertyValue* permite crear calculadores reutilizable.
@PropertyValue permite inyectar valores desde otras propiedades visualizadas, de esta forma:

```
@DefaultValueCalculator(
    value=org.openxava.test.calculatodores.CalculadorObservacionesTransportista.class,
    properties={
        @PropertyValue(name="tipoPermisoConducir", from="permisoConducir.tipo")
    }
)
private String observaciones;
```

En este caso antes de ejecutar el calculador OpenXava llena la propiedad *permisoConducir* de *CalculadorObservacionesTransportista* con el valor de la propiedad visualizada *tipo* de la referencia *permisoConducir*. Como se ve el atributo *from* soporta propiedades calificadas (referencia.propiedad).
Además podemos usar *@PropertyValue* sin *from* ni *value*:

```
@DefaultValueCalculator(value=CalculadorPrecioDefectoProducto.class, properties=
    @PropertyValue(name="codigoFamilia")
)
```

En este caso OpenXava coge el valor de la propiedad visualizada *codigoFamilia* y lo inyecta en la propiedad *codigoFamilia* del calculador, es decir *@PropertyValue(name="codigoFamilia")* equivale a *@PropertyValue(name="codigoFamilia", from="codigoFamilia")*.
Desde un calculador tenemos acceso a conexiones JDBC, he aquí un ejemplo:

```
@DefaultValueCalculator(value=CalculadorCantidadLineas.class,
    properties= {
        @PropertyValue(name="año"),
        @PropertyValue(name="numero"),
    }
)
private int cantidadLineas;
```

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 nuestro calculador tiene que implementar *IJDBCCalculator* (1) y entonces recibirá un *IConnectionProvider* (2) que podemos usar dentro de *calculate()*.

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

Valores por defecto al crear

Podemos indicar que el valor sea calculado justo antes de crear (insertar en la base de datos) un objeto por primera vez.
Usualmente para las claves usamos el estándar JPA. Por ejemplo, si queremos usar una columna *identity* (auto incremento) como clave:

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

Podemos usar otras técnicas de generación, por ejemplo, una *sequence* de base de datos puede ser definida usando el estándar JPA de esta manera:

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

Si queremos generar un identificador único de tipo String y 32 caracteres, podemos usar una extensión de Hibernate de JPA:

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

Ver la sección 9.1.9 de la especificación JPA 1.0 (parte de JSR-220) para aprender más sobre *@GeneratedValues*.
Si queremos usar nuestra propia lógica para generar el valor al crear, o bien queremos generar un nuevo valor para propiedades que no son clave entonces no podemos usar el *@GeneratedValue* de JPA, aunque es fácil resolver estos casos con JPA. Solo necesitamos añadir este código a nuestra clase:

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

La anotación JPA *@PrePersist* hace que este método se ejecute antes de insertar datos por primera vez en la base de datos, en este método podemos calcular el valor para nuestra clave o incluso para propiedades no clave con nuestra propia lógica.

Validador de propiedad

Un validador de propiedad (*@PropertyValidator*) 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:

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

La forma de configurar el validador (con los *@PropertyValue*) es exactamente igual como en los calculadores. Con el atributo *onlyOnCreate="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:

1. **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.
2. **Object valor**: El valor a validar.
3. **String nombreObjeto**: Nombre del objeto al que pertenece la propiedad a validar. Útil para usarlo en los mensajes de error.
4. **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.

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:

```
@Required @Stereotype("NOMBRE_PERSONA")
private String nombre;
```

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 *@Required*). 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*.

Referencias

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:

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

para acceder al nombre del comercial de ese cliente.
La sintaxis para definir referencias es:

```
@Required                                          // 1
@Id                                               // 2
@DefaultValueCalculator                              // 3
```

```
@ManyToOne(                                          // 4
    optional=false                                    // 1
)
private tipo nombreReferencia;                          // 4
public tipo getNombreReferencia() { ... }               // 4
public void setNombreReferencia(tipo nuevoValor) { ... }  // 4
```

1. **@ManyToOne**(**optional**=**false**) (JPA), **@Required** (OX) (opcional, el JPA es el preferido): 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.
2. **@Id** (JPA, 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.
3. **@DefaultValueCalculator** (OX, one, 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).
4. **Declaración de la referencia**: Una declaración de referencia convencional de Java con sus *getters* y *setters*. La referencia se marca con *@ManyToOne (JPA)* y el tipo ha de ser otra entidad.

Un pequeño ejemplo de referencias:

```
@ManyToOne
private Comercial comercial;                  // 1
public Comercial getComercial() {
    return comercial;
}
public void setComercial(Comercial comercial) {
    this.comercial = comercial;
}

@ManyToOne(fetch=FetchType.LAZY)
private Comercial comercialAlternativo;        // 2
public Comercial getComercialAlternativo() {
    return comercialAlternativo;
}
public void setComercialAlternativo(Comercial comercialAlternativa) {
    this.comercialAlternativo = comercialAlternativo;
}
```

1. Una referencia llamada *comercial* a la entidad *Comercial*.

2. Una referencia llamada *comercialAlternativo* a la entidad *Comercial*. En este caso usamos *fetch=FetchType.LAZY*, de esta manera los datos son leidos de la base de datos bajo demanda. Este es el enfoque más eficiente, pero no es el valor por defecto en JPA, por tanto es aconsejable **usar siempre *fetch=FetchType.LAZY*** al declarar las referencias.

Si asumimos que esto está en una entidad llamada *Cliente*, podemos escribir:

```
Cliente cliente = …
Comercial comercial = cliente.getComercial();
Comercial comercialAlternativo = cliente.getComercialAlternativo();
```

Calculador valor por defecto en referencias

En una referencia *@DefaultValueCalculator* funciona como en una propiedad, solo que hay que devolver el valor de la clave de la referencia.
Por ejemplo, en el caso de una referencia con clave simple podemos poner:

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

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:

```
@ManyToOne(fetch=FetchType.LAZY)
@JoinColumns({
    @JoinColumn(name="ZONA", referencedColumnName="ZONA"),
    @JoinColumn(name="ALMACEN", referencedColumnName="CODIGO")
})
@DefaultValueCalculator(CalculadorDefectoAlmacen.class)
private Almacen almacen;
```

Y el código del calculador:

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

import org.openxava.calculators.*;

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

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

}
```

Devuelve un objeto de tipo *AlmacenKey*.

Usar referencias como clave

Podemos usar referencias como clave, o como parte de la clave. Hemos de declarar la referencia como *@Id*, y usar una clase clave, como sigue:

```
@Entity
@IdClass(DetalleAdicionalKey.class)
public class DetalleAdicional {

    // JoinColumn se especifica también en DetalleAdicionalKey por un
    // bug de Hibernate, ver http://opensource.atlassian.com/projects/hibernate/browse/ANN-361
    @Id @ManyToOne(fetch=FetchType.LAZY)
    @JoinColumn(name="SERVICIO")
    private Servicio servicio;

    @Id @Hidden
    private int contador;

    ...

}
```

Además, necesitamos escribir la clase clave:

```
public class DetalleAdicionalKey implements java.io.Serializable {

    @ManyToOne(fetch=FetchType.LAZY)
    @JoinColumn(name="SERVICIO")
    private Servicio servicio;
```

```
    @Hidden
    private int contador;

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

}
```

Necesitamos escribir la clase clave aunque la clave sea solo una referencia con una sola columna clave.
Es mejor usar esta característica sólo cuando estemos trabajando contra bases de datos legadas, si tenemos control sobre el esquema es mejor usar un id autogenerado.

Colecciones

Podemos definir colecciones de referencias a entidades. Una colección es una propiedad Java que devuelve *java.util.Collection*.
Aquí la sintaxis para definir una colección:

```
@Size                                                          // 1
@Condition                                                      // 2
@OrderBy                                                       // 3
@XOrderBy                                                      // 4
@OneToMany/@ManyToMany                                           // 5
private Collection<tipo> nombreColeccion;                            // 5
public Collection<tipo> getNombreColeccion() { ... }                 // 5
public void setNombreColeccion(Collection<tipo> nuevoValor) { ... }  // 5
```

1. **@Size** (HV, opcional): Cantidad mínima (*min*) y/o máxima (*max*) de elementos esperados. Esto se valida antes de grabar.
2. **@Condition** (OX, opcional): Para restringir los elementos que aparecen en la colección.
3. **@OrderBy** (JPA, opcional): Para que los elementos de la colección aparezcan en un determinado orden.
4. **@XOrderBy** (OX, opcional): *@OrderBy* de JPA no permite usar propiedades calificadas (propiedades de referencias). *@XOrderBy* sí lo permite.
5. **Declaracion de la colección**: Una declaración de colección convencional de Java con sus *getters* y *setters*. La colección se marca con *@OneToMany (JPA)* o *@ManyToMany (JPA)* y el tipo ha de ser otra entidad.

Vamos a ver algunos ejemplos. Empecemos por uno simple:

```
@OneToMany (mappedBy="factura")
private Collection<Albaran> albaranes;
public Collection<Albaran> getAlbaranes() {
    return albaranes;
}
public void setAlbaranes(Collection<Albaran> albaranes) {
    this.albaranes = albaranes;
}
```

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. Usamos *mappedBy="factura"* para indicar que la referencia *factura* de *Albaran* se usa para mapear esta colección.
Ahora podemos escribir código como este:

```
Factura factura = …
for (Albaran albaran: factura.getAlbaranes()) {
    albaran.hacerAlgo();
}
```

Para hacer algo con todos los albaranes asociados a una factura.
Vamos a ver otro ejemplo más complejo, también dentro de *Factura*:

```
@OneToMany (mappedBy="factura", cascade=CascadeType.REMOVE)  // 1
@OrderBy("tipoServicio desc")                                // 2
@org.hibernate.validator.Size(min=1)                          // 3
private Collection<LineaFactural> facturas;
```

1. Usar REMOVE como tipo de cascadaas cascade type hace que cuando el usuario borra una factura sus líneas también se borran.
2. Con *@OrderBy* obligamos a que las lineas se devuelvan ordenadas por *tipoServicio*.
3. La restricción de *@Size(min=1)* hace que sea obligado que haya al menos una línea para que la factura sea válida.

Tenemos libertad completa para definir como se obtienen los datos de una colección, con *@Condition* podemos sobreescribir la condición por defecto:

```
@Condition(
    "${almacen.codigoZona} = ${this.almacen.codigoZona} AND " +
    "${almacen.codigo} = ${this.almacen.codigo} AND " +
    "NOT (${codigo} = ${this.codigo})"
)
public Collection<Transportista> getCompañeros() {
    return null;
}
```

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. *@Condition* solo aplica a la interfaz de usuario generada por OpenXava, si llamamos directamente a *getFellowCarriers()* it will be returns null.
Si con esto no tenemos suficiente, podemos escribir completamente la lógica que devuelve la colección. La colección anterior también se podría haber definido así:

```
public Collection<Transportista> getCompañeros() {
    Query query = XPersistence.getManager().createQuery("from Transportista t where " +
        "t.almacen.codigoZona = :zona AND " +
        "t.almacen.codigo = :codigoAlmacen AND " +
        "NOT (t.codigo = :codigo) ");
    query.setParameter("zona", getAlmacen().getCodigoZona());
    query.setParameter("codigoAlmacen", getAlmacen().getCodigo());
    query.setParameter("codigo",  getCodigo());
    return query.getResultList();
}
```

Como se ve es un método *getter*. Obviamente ha de devolver una *java.util.Collection* cuyos elementos sean de tipo *Transportista*.
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 puede ocurrir que en *Cliente* tenga más de una referencia a *Comercial* (por ejemplo, *comercial* y *comercialAlternativo*) y entonce JPA no sabe cual escoger, por eso tenemos el atributo *mappedBy* de *@OneToMany*. En este caso pondríamos:

```
@OneToMany(mappedBy="comercial")
private Collection<Cliente> clientes;
```

Para indicar que es la referencia *comercial* y no *comercialAlternativo* la que vamos a usar para esta colección.
La anotación *@ManyToMany (JPA)* permite definir una colección con una multiciplidad de muchos-a-muchos. Como sigue:

```
@Entity
public class Cliente {
    ...
    @ManyToMany
    private Collection<Provincia> provincias;
    ...
}
```

En este caso un cliente tiene una colección de provincias, pero una misma provincia puede estar presente en varios clientes.

Métodos

Los métodos se definen en una entidad OpenXava (mejor dicho, en una entidad JPA) como una clase de Java convencional. Por ejemplo:

```
public void incrementarPrecio() {
    setPrecioUnitario(getPrecioUnitario().multiply(new BigDecimal("1.02")).setScale(2));
}
```

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).

Buscadores

Un buscador es método estático especial que nos permite buscar un objeto o una colección de objetos que sigue algún criterio.
Algunos ejemplos:

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

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

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

Estos métodos se pueden usar de esta manera:

```
Cliente cliente = Cliente.findByCodigo(8);
Collection javieres = Cliente.findByNombreLike("%JAVI%");
```

Como se ve, usar método buscadores produce un código más legible que usando la verbosa API de JPA. Pero esto es solo una recomendación de estilo, podemos escoger no escribir métodos buscadores y usar directamente consultas de JPA.

Validador de entidad

Este validador (*@EntityValidator*) 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:

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

1. **value** (opcional, obligada si no se especifica nombre): Clase que implementa la validación. Ha de ser del tipo *IValidator*.
2. **onlyOnCreate** (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.
3. **properties** (varios *@PropertyValue*, opcional): Para establecer valor a las propiedades del validador antes de ejecutarse.

Un ejemplo:

```
@EntityValidator(value=org.openxava.test.validadores.ValidadorProductoBarato.class, properties= {
    @PropertyValue(name="limite", value="100"),
    @PropertyValue(name="descripcion"),
    @PropertyValue(name="precioUnitario")
})
public class Producto {
```

Y el código del validador:

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

import java.math.*;

/**
 * @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) {
                errors.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(Integer.toString(limite));
    }

}
```

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.
Podemos definir más de un validador por entidad usando *@EntityValidators*, como sigue:

```
@EntityValidators({
    @EntityValidator(value=org.openxava.test.validadores.ValidadorProductoBarato.class, properties= {
        @PropertyValue(name="limite", value="100"),
        @PropertyValue(name="descripcion"),
        @PropertyValue(name="precioUnitario")
    }),
    @EntityValidator(value=org.openxava.test.validadores.ValidadorProductoCaro.class, properties= {
        @PropertyValue(name="limite", value="1000"),
        @PropertyValue(name="descripcion"),
        @PropertyValue(name="precioUnitario")
    }),
    @EntityValidator(value=org.openxava.test.validadores.ValidadorPrecioProhibido.class,
        properties= {
            @PropertyValue(name="precioProhibido", value="555"),
            @PropertyValue(name="precioUnitario")
        },
        onlyOnCreate=true
    )
})
public class Product {
```

Validador al borrar

El *@RemoveValidator* 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:

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

1. **clase** (obligada): Clase que implementa la validación. Ha de ser del tipo *IRemoveValidator*.
2. **properties** (varios *@PropertyValue*, opcional): Para establecer valor a las propiedades del calculador antes de ejecutarse.

Un ejemplo puede ser:

```
@RemoveValidator(value=ValidadorBorrarTipoAlbaran.class,
    properties=@PropertyValue(name="codigo")
```

```
)
public class TipoAlbaran {
```

Y el validador:

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

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

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

    private TipoAlbaran tipoAlbaran;
    private int codigo; // Usamos esto (en vez de obtenerlo de tipoAlbaran)
              // para probar @PropertyValue con propiedades simples

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

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

    public int getCodigo() {
        return codigo;
    }

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

}
```

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 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.

Tal y como ocurre con *@EntityValidator* podemos usar varios *@RemoveValidator* por entidad usando la anotación *@RemoveValidators*.

Métodos de retrollamada de JPA

Con *@PrePersist* podemos indicar que se ejecute cierta lógica justo antes de crear el objeto como persistente.
Como sigue:

```
@PrePersist
private void antesDeCrear() {
    setDescripcion(getDescripcion() + " CREADO");
}
```

En este caso cada vez que se graba por primera vez un *TipoAlbaran* se añade un sufijo a su descripción.
Como se ve es exactamente igual que cualquier otro método solo que este se ejecuta automáticamente antes de crear.
Con *@PreUpdate* 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.
Como sigue:

```
@PreUpdate
private void antesDeModificar() {
    setDescripcion(getDescripcion() + " MODIFICADO");
}
```

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 método solo que este se ejecuta automáticamente antes de modificar.
Podemos usar todas las anotaciones JPA de retrollamada: *@PrePersist*, *@PostPersist*, *@PreRemove*, *@PostRemove*, *@PreUpdate*, *@PostUpdate* y *@PostLoad*.

Clases incrustables (Embeddable)

Tal y como indica la especificación JPA:
"*Una entidad puede usar otras clases finamente granuladas para representar su estado. Instancias de estas clases, no como en el caso de las entidades, no tiene identidad persistente. En vez de eso, existen solo como objetos incrustados de una entidad a la que pertenecen. Estos objetos incrustados son propiedad exclusiva de sus entidades dueñas, y no se comparten entre entidades persistentes.*"
La sintaxis para una clase incrustada es:

```
@Embeddable                    // 1
public class NombreIncrustada {  // 2
  // Propiedades               // 3
  // Referencias               // 4
  // Metodos                   // 5
}
```

1. **@Embeddable** (JPA, una, requerido): Indica que esta clase es una clase incrustada de JPA, en otras palabras, sus instancias serán parte de objetos persistente.
2. **Declaración de la clase**: Como una clase Java convencional. Podemos uar *extends* y *implements*.
3. **Properties**: Propiedades Java convencionales.
4. **References**: Referencias a entidades. Esto no esta soportado en JPA 1.0 (EJB 3.0), pero la implementación de Hibernate lo soporta.
5. **Métodos**: Métodos Java con lógica de negocio.

Referencias incrustadas

Este ejemplo es una *Direccion* incrustada (anotada con *@Embedded*) que es referenciada desde la entidad principal.
En la entidad principal podemos escribir:

```
@Embedded
private Direccion direccion;
```

Y hemos de definir la clase *Direccion* como incrustable:

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

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

/**
 *
 * @author Javier Paniza
 */

@Embeddable
public class Direccion implements IConPoblacion {

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

    @Required @Column(length=5)
    private int codigoPostal;
```

```
    @Required @Column(length=20)
    private String poblacion;

    // ManyToOne dentro de un Embeddable no está soportado en JPA 1.0 (ver en 9.1.34),
    // pero la implementación de Hibernate lo soporta.
    @ManyToOne(fetch=FetchType.LAZY, optional=false) @JoinColumn(name="STATE")
    private Provincia provincia;

    public String getPoblacion() {
        return poblacion;
    }

    public void setPoblacion(String poblacion) {
        this.poblacion = poblacion;
    }

    public String getCalle() {
        return calle;
    }

    public void setCalle(String calle) {
        this.calle = calle;
    }

    public int getCodigoPostal() {
        return codigoPostal;
    }

    public void setCodigoPostal(int codigoPostal) {
        this.codigoPostal = codigoPostal;
    }

    public Provincia getProvincia() {
        return provincia;
    }

    public void setProvincia(Provincia provincia) {
        this.provincia = provincia;
    }

}
```

Como se ve una clase incrustable puede implementar una interfaz (1) y contener referencias (2), entre otras cosas, pero no puede tener colecciones persistentes ni usar métodos de retrollamada de JPA.

Este código se puede usar así, para leer:

```
Cliente 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();
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*.

Colecciones incrustadas

Las colecciones incrustadas no se soportan en JPA 1.0. Pero podemos simularlas usando colecciones a entidades con tipo de cascada REMOVE o ALL. OpenXava trata estas colecciones de una manera especial, como si fueran colecciones incrustadas.
Ahora un ejemplo de una colección incrustada. En la entidad principal (por ejemplo de *Factura*) podemos poner:

```
@OneToMany (mappedBy="factura", cascade=CascadeType.REMOVE)
private Collection<LineaFactura> lineas;
```

Es de notar que usamos *CascadeType.REMOVE* y *LineaFactura* es una entidad y no una clase incrustable:

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

import java.math.*;

import javax.persistence.*;

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

/**
 *
 * @author Javier Paniza
 */

@Entity
```

```
@EntityValidator(value=ValidadorLineaFactura.class,
    properties= {
        @PropertyValue(name="factura"),
        @PropertyValue(name="oid"),
        @PropertyValue(name="producto"),
        @PropertyValue(name="precioUnitario")
    }
)
public class LineaFactura {

    @ManyToOne // 'Lazy fetching' produce un falla al borrar una linea desde la factura
    private Factura factura;

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

    private TipoServicio tipoServicio;
    public enum TipoServicio { ESPECIAL, URGENTE }

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

    @Stereotype("DINERO") @Required
    private BigDecimal precioUnitario;

    @ManyToOne(fetch=FetchType.LAZY, optional=false)
    private Producto producto;

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

    @ManyToOne(fetch=FetchType.LAZY)
    private Comercial vendidoPor;

    @Stereotype("MEMO")
    private String observaciones;

    @Stereotype("DINERO") @Depends("precioUnitario, cantidad")
    public BigDecimal getImporte() {
        return getPrecioUnitario().multiply(new BigDecimal(getCantidad()));
    }

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

     @PostRemove
    private void postRemove() {
        factura.setComentario(factura.getComentario() + "DETALLE BORRADO");
    }

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

```
    public void setOid(String oid) {
        this.oid = oid;
    }
    public TipoServicio getTipoServicio() {
        return tipoServicio;
    }
    public void setTipoServicio(TipoServicio tipoServicio) {
        this.tipoServicio = tipoServicio;
    }
    public int getCantidad() {
        return cantidad;
    }
    public void setCantidad(int cantidad) {
        this.cantidad = cantidad;
    }
    public BigDecimal getPrecioUnitario() {
        return precioUnitario==null?BigDecimal.ZERO:precioUnitario;
    }
    public void setPrecioUnitario(BigDecimal precioUnitario) {
        this.precioUnitario = precioUnitario;
    }

    public Product getProducto() {
        return producto;
    }

    public void setProducto(Producto producto) {
        this.producto = producto;
    }

    public java.util.Date getFechaEntrega() {
        return fechaEntrega;
    }

    public void setFechaEntrega(java.util.Date fechaEntrega) {
        this.fechaEntrega = fechaEntrega;
    }

    public Seller getVendidoPor() {
        return vendidoPor;
    }

    public void setVendidoPor(Comercial vendidoPor) {
        this.vendidoPor = vendidoPor;
    }

    public String getObservaciones() {
        return observaciones;
    }

    public void setObservaciones(String observaciones) {
        this.observaciones = observaciones;
    }

    public Invoice getFactura() {
```

```
        return factura;
    }

    public void setFactura(Factura factura) {
        this.factura = factura;
    }

}
```

Como se ve esto es una entidad compleja, con calculadores, validadores, referencias y así por el estilo. También hemos de definir una referencia a su clase contenedora (*factura*). En este caso cuando una factura se borre todas sus líneas se borrarán también. Además hay diferencias a nivel de interface gráfica (podemos aprender más en el capítulo de la vista).

Herencia

OpenXava soporta la herencia de herencia de JPA y Java.
Por ejemplo podemos definer una superclase mapeada (*@MappedSuperclass*) de esta manera:

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

import javax.persistence.*;

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

/**
 * Clase base para definir entidades con un oid UUID. <p>
 *
 * @author Javier Paniza
 */

@MappedSuperclass
public class Identificable {

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

    public String getOid() {
        return oid;
    }

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

```
}
```

Podemos definir otra *@MappedSuperclass* que extienda de esta, por ejemplo:

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

import javax.persistence.*;

import org.openxava.annotations.*;

/**
 * Clase base para entidades con una propiedad 'nombre'. <p>
 *
 * @author Javier Paniza
 */
@MappedSuperclass
public class ConNombre extends Identifiable {

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

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

}
```

Ahora podemos usar *Identificable* y *ConNombre* para definir nuestra entidades, como sigue:

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

import javax.persistence.*;

/**
 *
 * @author Javier Paniza
 */

@Entity
@DiscriminatorColumn(name="TYPE")
@DiscriminatorValue("HUM")
@Table(name="PERSONA")
@AttributeOverrides(
    @AttributeOverride(name="name", column=@Column(name="PNOMBRE"))
)
public class Humano extends ConNombre {
```

```
    @Enumerated(EnumType.STRING)
    private Sexo sexo;
    public enum Sexo { MASCULINO, FEMENINO };

    public Sexo getSexo() {
        return sexo;
    }
    public void setSexo(Sexo sexo) {
        this.sexo = sexo;
    }

}
```

Y ahora, la auténtica herencia de entidades, una entidad que extiende de otra entidad:

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

import javax.persistence.*;

/**
 *
 * @author Javier Paniza
 */

@Entity
@DiscriminatorValue("PRO")
public class Programador extends Humano {

    @Column(length=20)
    private String lenguajePrincipal;

    public String getLenguajePrincipal() {
        return lenguajePrincipal;
    }

    public void setLenguajePrincipal(String lenguajePrincipal) {
        this.lenguajePrincipal = lenguajePrincipal;
    }

}
```

Podemo crear un módulo OpenXava para *Humano* y *Programador* (no para *Identificable* ni *ConNombre* directamente). En el módulo de *Programador* el usuario puede acceder solo a programadores, por otra parte usando el módulo de *Humano* el usuario puede acceder a objetos de tipo *Humano* y *Programador*. Además cuando el usuario trata de visualizar el detalle de un *Programador* desde el módulo de *Humano* se mostrará la vista de *Programador*. Polimorfismo puro.

Sobre el mapeo, se soporta *@AttributeOverrides* , pero, de momento, solo la estrategia una única tabla por jerarquía de clases funciona.

Clave múltiple

La forma preferida para definir la clave de una entidad es una clave única autogenerada (anotada con *@Id* y *@GeneratedValue*), pero a veces, por ejemplo cuando vamos contra bases de datos legadas, necesitamos tener una entidad mapeada a una tabla que usa varias columnas como clave. Este caso se pude resolver con JPA (y por tanto con OpenXava) de dos formas, usando *@IdClass* o usando *@EmbeddedId*

Clase id

En este caso usamos *@IdClass* en nuestra entidad para indicar una clase clave, y marcamos las propiedades clave como *@Id* en nuestra entidad:

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

import javax.persistence.*;

import org.openxava.annotations.*;
import org.openxava.jpa.*;

/**
 *
 * @author Javier Paniza
 */

@Entity
@IdClass(AlmacenKey.class)
public class Almacen {

    @Id
    // Column también se especifica en AlmacenKey por un bug en Hibernate, ver
    // http://opensource.atlassian.com/projects/hibernate/browse/ANN-361
    @Column(length=3, name="ZONA")
    private int codigoZona;

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

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


    public String getNombre() {
        return nombre;
```

```
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public int getCodigo() {
        return codigo;
    }

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

    public int getCodigoZona() {
        return codigoZona;
    }

    public void setCodigoZona(int codigoZona) {
        this.codigoZona = codigoZona;
    }

}
```

También necesitamos declarar una clase id, una clase serializable normal y corriente con todas las propiedades clave de la entidad:

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

import java.io.*;
import javax.persistence.*;

/**
 *
 * @author Javier Paniza
 */

public class AlmacenKey implements Serializable {

    @Column(name="ZONE")
    private int codigoZona;
    private int codigo;

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

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

```
    @Override
    public String toString() {
        return "AlmacenKey::" + codigoZona + ":" + codigo;
    }

    public int getCodigo() {
        return codigo;
    }

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

    public int getCodigoZona() {
        return codigoZona;
    }

    public void setCodigoZona(int codigoZona) {
        this.codigoZona = codigoZona;
    }

}
```

Id inscrustado

En este case tenemos una referencia a un objeto incrustado (*@Embeddable*) marcada como *@EmbeddedId*:

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

import javax.persistence.*;

import org.openxava.annotations.*;

/**
 *
 * @author Javier Paniza
 */

@Entity
public class Almacen {

    @EmbeddedId
    private AlmacenKey clave;

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

    public AlmacenKey getClave() {
        return clave;
    }
```

```
    public void setClave(AlmacenKey clave) {
        this.clave = clave;
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

}
```

Y nuestra clave es una clase incrustable que contiene las propiedades clave:

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

import javax.persistence.*;

/**
 *
 * @author Javier Paniza
 */

@Embeddable
public class AlmacenKey implements java.io.Serializable {


    @Column(length=3, name="ZONA")
    private int codigoZona;

    @Column(length=3)
    private int codigo;

    public int getCodigo() {
        return codigo;
    }

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

    public int getCodigoZona() {
        return codigoZona;
    }

    public void setCodigoZona(int codigoZona) {
        this.codigoZona = codigoZona;
    }

}
```

Capítulo 4: 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.

Disposición

La anotación *@View* se puede usar en una entidad o una clase incrustable para definir la disposición de sus miembros en la interfaz de usuario.
La sintaxis para definir una vista (*@View)* es:

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

1. **name** (opcional): El nombre identifica a la vista, y puede ser usado desde otro lugares de OpenXava (por ejemplo desde *aplicacion.xml*) o desde otra entidad. Si no se pone nombre se asume que es la vista por defecto, es decir la forma normal de visualizar el objeto.
2. **members** (opcional): Indica los miembros que tienen que salir y como tienen que estar dispuestos en la interfaz gráfica. Por defecto visualiza todos los miembros no ocultos en el orden en que están declarados en el modelo. Dentro de miembros podemos usar los elementos seccion y grupo para indicar la disposición; o el elemento accion para mostrar un vínculo asociado a una acción propia dentro de la vista.

Podemos definir varias vistas para una entidad usando la anotación *@Views*.
Por defecto (es decir si no definimos ni siquiera el elemento *@View* en nuestra clase) 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:

```
@Entity
@IdClass(OficinistaKey.class)
```

```
public class Oficinista {

    @Id @Required
    @Column(length=3, name="ZONA")
    private int codigoZona;

    @Id @Required
    @Column(length=3, name="OFICINA")
    private int codigoOficina;

    @Id  @Required
    @Column(length=3, name="CODIGO")
    private int codigo;

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

    // Getters and setters
    ...

 }
```

Generaría una vista con este aspecto:

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

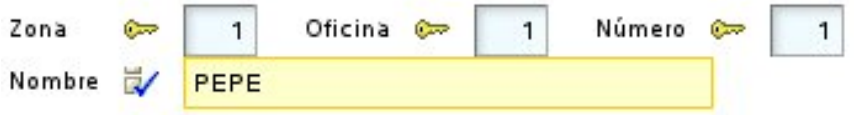
```
@Entity
@IdClass(OficinistaKey.class)
@View(members="codigoZona; codigoOficina; codigo")
public class Oficinista {
```

En este caso ya no aparece el *nombre* en la vista.
También se puede usar *members* para refinar la disposición:

```
@View(members=
    "codigoZona, codigoOficina, codigo;" +
    "nombre"
)
```

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:



Grupos

Con los grupos podemos agrupar un conjunto de propiedades relacionadas, y esto tiene un efecto visual. Para definir un grupo solo necesitamos poner el nombre del grupo y después sus miembros entre corchetes. Justo de esta forma:

```
@View(members=
    "id [ codigoZona, codigoOficina, codigo ];" +
    "nombre"
)
```

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:

```
@View(members=
    "general [" +
    "    codigo;" +
    "    tipo;" +
    "    nombre;" +
    "]" +
    "contacto [" +
    "    telefono;" +
    "    correoElectronico;" +
```

```
    "    sitioWeb;" +
    "]"
)
```

En este caso se visualizaría así:

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

```
@View(members=
    "general [" +
    "    codigo;" +
    "    tipo;" +
    "    nombre;" +
    "];" +
    "contacto [" +
    "    telefono;" +
    "    correoElectronico;" +
    "    sitioWeb;" +
    "]"
)
```

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:

```
@View(members=
    "factura;" +
    "datosAlbaran [" +
    "    tipo, codigo;" +
    "    fecha;" +
    "    descripcion;" +
    "    envio;" +
    "    datosTransporte [" +
    "        distancia; vehiculo; modoTransporte; tipoConductor;" +
    "    ]" +
    "    datosEntregadoPor [" +
    "        entregadoPor;" +
    "        transportista;" +
    "        empleado;" +
    "    ]" +
    "]"
)
```

Obtendremos lo siguiente:

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

```
@View(name="Amounts", members=
    "año, numero;" +
    "importes [" +
        "descuentoCliente, descuentoTipoCliente, descuentoAño;" +
        "sumaImportes, porcentajeIVA, iva;" +
    "]"
)
```

...será visualizada como sigue:

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

```
@View(name="Amounts", members=
    "año, numero;" +
    "importes [#" +
        "descuentoCliente, descuentoTipoCliente, descuentoAño;" +
        "sumaImportes, porcentajeIVA, iva;" +
    "]"
)
```

Notemos que usamos *[#* en vez de *[*. Ahora obtenemos este resultado:

Ahora, gracias al #, los miembros están alineado por columnas.
Esta prestación esta disponible también para las secciones (ver abajo).

Secciones

Además de en grupo los miembros se pueden organizar en secciones. Para definir una sección solo necesitamos poner el nombre de la sección y después sus miembros entre llaves. Veamos un ejemplo en la entidad *Factura*:

```
@View(members=
    "año, numero, fecha, pagada;" +
    "comentario;" +
    "cliente { cliente }" +
    "lineas { lineas }" +
    "importes { sumaImportes; porcentajeIVA; iva }" +
    "albaranes { albaranes }"
)
```

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* y *albaranes* son colecciones.
Se permiten secciones anidadas. Por ejemplo, podemos definir una vista como ésta:

```
@View(name="SeccionesAnidadas", members=
    "año, numero, fecha;" +
    "cliente { cliente }" +
    "datos {" +
    "    lineas { lineas }" +
    "    importes {" +
    "        iva { porcentajeIVA; iva }" +
    "        sumaImportes { sumaImportes }" +
    "    }" +
    "}" +
    "albaranes { albaranes }"
)
```

En este caso podemos obtener una interfaz gráfica como esta:

Al igual que en los grupos, las secciones permiten usar # para conseguir alineado por columnas, así:

```
@View(name="ImportesAlineadosEnSeccion", members=
    "año, numero;" +
    "cliente { cliente }" +
    "lineas { lineas }" +
    "importes {#" +
    "descuentoCliente, descuentoTipoCliente, descuentoAño;" +
        "sumaImportes, porcentajeIVA, iva;" +
    "}"
)
```

Con el mismo efecto que en el caso de los grupos.

Filosofía para la disposición

Es de notar tenemos grupos y no marcos y secciones y no pestañas. Porque en las vistas 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.

Normas para las anotaciones de vista

Podemos anotar un miembro (propiedad, referencia o colección) con varias anotaciones que refinan su estilo de visualización y comportamiento. Además podemos definir que el efecto de estas anotaciones solo aplica a algunas vistas.
Por ejemplo, si tenemos una entidad como esta:

```
@Entity
@Views({
    @View( members="codigo; tipo; nombre; direccion" ),
    @View( name="A", members="codigo; tipo; nombre; direccion; comercial" ),
    @View( name="B", members="codigo; tipo; nombre; comercial; comercialAlternativo" ),
    @View( name="C", members="codigo; tipo; nombre; direccion; lugaresEntrega" )
})
public class Cliente {
```

Y queremos que la propiedad *nombre* sea de solo lectura. Podemos anotarlo de esta manera:

```
@ReadOnly
private String nombre;
```

De esta forma *nombre* es de solo lectura en todas las vistas. Ahora bien, puede que queramos que *nombre* sea de solo lectura solo en las vistas B y C, entonces podemos definir el miembro como sigue:

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

Otra forma para definir este mismo caso es:

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

Usando *notForViews* indicamos las vistas donde la propiedad *nombre* es de solo lectura. DEFAULT se usa para referenciar a la vista por defecto, la vista sin nombre.
Algunas anotaciones tiene uno o más valores, por ejemplo para indicar que vista del tipo referenciado se usará para visualizar una referencia usamos la anotación *@ReferenceView*:

```
@ReferenceView("Simple")
private Comercial comercial;
```

En este caso cuando se visualiza el comercial se usa la vista *Simple*, definida en la clase *Comercial*.

¿Qué ocurre si queremos usar la vista *Simple* de *Comercial* solo en la vista *B* de *Cliente*? Es fácil:

```
@ReferenceView(forViews="B", value="Simple")
private Comercial comercial;
```

¿Qué ocurre si lo que queremos es usar la vista *Simple* de *Comercial* solo en la vista *B* de *Cliente* y la vista *MuySimple* de *Comercial* para la vista *A* de *Cliente*? En este caso hemos de usar varias *@ReferenceView* agrupandolas con *@ReferenceViews*, justo así:

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

Estas normas aplican a todas las anotaciones de este capítulo, excepto *@View* y *@Views*.

Personalización de propiedad

Podemos refinar la forma de visualización y comportamiento de una propiedad en la vista usando las siguientes anotaciones:

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

Todas estas anotaciones siguen las normas para anotaciones de vista y todas ellas son opcionales. OpenXava siempre asume valores por defecto correcto si se omiten.

1. **@ReadOnly** (OX): Si marcas una propiedad con esta anotaciones 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.
2. **@LabelFormat** (OX): Forma en que se visualiza la etiqueta para esta propiedad. Su valor puede ser *LabelFormatType.NORMAL*, *LabelFormatType.SMALL* o *LabelFormatType.NO_LABEL*.
3. **@DisplaySize** (OX): 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 *@DisplaySize* no se especifica se asume el valor de la longitud de la propiedad.

4. **@OnChange** (OX): Acción a realizar cuando cambia el valor de esta propiedad. Solo una acción *@OnChange* por vista está permitida.
5. **@Action** (OX): Acciones (mostradas como vínculos, botones o imágenes al usuario) asociadas (visualmente) a esta propiedad y que el usuario final puede ejecutar. Es posible definir varias *@Action* por cada vista.
6. **@Editor** (OX): 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.

Formato de etiqueta

Un ejemplo sencillo para cambiar el formato de la etiqueta (*@LabelFormat*):

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

En este caso el código postal lo visualiza así:

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

Evento de cambio de valor de propiedad

Si queremos reaccionar al evento de cambio de valor de una propiedad podemos user *@OnChange* como sigue:

```
@OnChange(AlCambiarNombreCliente.class)
private String nombre;
```

El código que se ejecutará será:

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

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

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

    public void execute() throws Exception {
        String valor = (String) getNewValue();                          // 2
        if (valor == null) return;
        if (valor.startsWith("Javi")) {
            getView().setValue("tipo", Cliente.Tipo.FIJO);              // 3
        }
    }

}
```

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 (*View*) (cambiar valores, ocultar miembros, hacerlos editables, o lo que queramos).

Acciones de la propiedad

También podemos especificar acciones (*@Action*) que el usuario puede pulsar directamente:

```
@Action("Albaran.generarNumero")
private int numero;
```

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="Albaran">
    ...
    <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 (*@ReadOnly*) o calculada entonces está siempre disponible. Podemos usar el atributo *alwaysEnabled* a *true* para que el vínculo esté siempre presente, incluso si la propiedad no es editable. Como sigue:

```
@Action(value="Albaran.generarNumero", alwaysEnabled=true)
```

El atributo *alwaysEnabled* 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* (esto está disponible solo para acciones usadas en *@Action* de propiedades, 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.

Escoger un editor

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

```
@Editor(forViews="TipoConRadioButton", value="ValidValuesRadioButton")
private Tipo tipo;
public enum Tipo { NORMAL, FIJO, ESPECIAL };
```

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*.

Personalización de referencia

Podemos refinar la forma de visualización y comportamiento de una referencia en la vista usando las siguientes anotaciones:

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

Todas estas anotaciones siguen las normas para anotaciones de vista y todas ellas son opcionales. OpenXava siempre asume valores por defecto correcto si se omiten.

1. **@ReferenceView** (OX): Si omitimos esta anotación usa la vista por defecto del objeto referenciado para visualizarlo, con este anotación podemos indicar que use otra vista.
2. **@ReadOnly** (OX): Si usamos esta anotación 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.
3. **@NoFrame** (OX): El dibujador de la interfaz gráfica usa un marco para envolver todos los datos de la referencia. Con esta anotación se puede indicar que no se use ese marco.
4. **@NoCreate** (OX): Por defecto el usuario tiene opción para crear un nuevo objeto del tipo referenciado. Con esta anotación anulamos esta posibilidad.

5. **@NoModify** (OX): Por defecto el usuario tiene opción para modificar el objeto actualmente referenciado. Con esta anotación anulamos esta posibilidad.
6. **@NoSearch** (OX): Por defecto el usuario tiene un vínculo para poder realizar búsquedas con una lista, filtros, etc. Con esta anotación anulamos esta posibilidad.
7. **@AsEmbedded** (OX): Por defecto en el caso de una referencia a una clase incrustable 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 *@AsEmbedded* entonce la interfaz de usuario para referencias a entidad se comporta como en el caso de los incrustados, permitiendo al usuario crear un nuevo objeto y editar sus datos directamente. No tiene efecto en el caso de una referencia a un objeto incrustado. ¡Ojo! Si borramos una entidad sus entidades referenciadas no se borran, incluso si estamos usando *@AsEmbedded*.
8. **@SearchAction** (OX): Nos permite especificar nuestra propia acción de búsqueda cuando se pulsa al vínculo de buscar. Solo es posible una por vista.
9. **@DescriptionsList** (OX): Permite visualizar los datos como una lista descripciones, típicamente un combo. Práctico cuando hay pocos elementos del objeto referenciado.
10. **@LabelFormat** (OX): Formato de la etiqueta de la referencia. Solo aplica si esta referencia se ha anotado con *@DescriptionsList*. Funciona como en el caso de las propiedades.
11. **@Action** (OX): Acciones (mostradas como vínculos, botones o imágenes al usuario) asociadas (visualmente) a esta referencia y que el usuario final puede ejecutar. Funciona como en el caso de las propiedades. Podemos definir varias acciones a la misma referencia en una vista.
12. **@OnChange** (OX): Acción a realizar cuando cambia el valor de esta propiedad. Solo una acción *@OnChange* por vista está permitida.
13. **@OnChangeSearch** (OX): Nos permite especificar nuestra propia acción de búsqueda cuando el usuario teclea una clave nueva. Solo es posible una por vista.

Si no usamos ninguna de estas anotaciones OpenXava dibuja la referencia usando su vista por defecto. Por ejemplo si tenemos una referencia así:

```
@ManyToOne
private Familia familia;
```

La interfaz gráfica tendrá el siguiente aspecto:

Escoger vista

La modificación más sencilla sería especificar que vista del objeto referenciado queremos usar. Esto se hace mediante *@ReferenceView*:

```
@ManyToOne(fetch=FetchType.LAZY)
@ReferenceView("Simple")
private Factura factura;
```

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

```
@Entity
@Views({
    ...
    @View(name="Simple", members="año, numero, fecha, descuentoAño;"),
    ...
})
public class Factura {
```

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

Personalizar el enmarcado

Si combinamos *@NoFrame* con un grupo podemos agrupar visualmente una propiedad que no forma parte de la referencia, por ejemplo:

```
@View( members=
    ...
    "comercial [" +
    "    comercial; " +
    "    relacionConComercial;" +
    "]" +
    ...
)
public class Cliente {
    ...
    @ManyToOne(fetch=FetchType.LAZY)
    @NoFrame
    private Comercial comercial;
    ...
}
```

Así obtendríamos:

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 *@SearchAction*, como sigue:

```
@ManyToOne(fetch=FetchType.LAZY) @SearchAction("MiReferencia.buscar")
private Comercial comercial;
```

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.

Acción de creación propia

Si no hemos puesto *@NoCreate* 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.

Acción de modificación propia

Si no hemos puesto *@NoModify* 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.

Lista descripciones (combos)

Con *@DescriptionsList* 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:

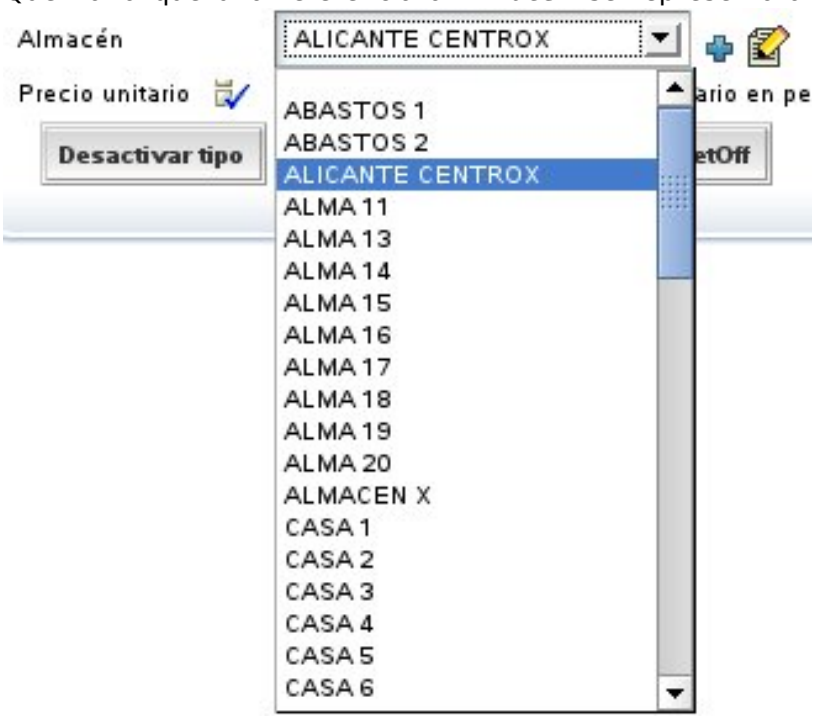
```
@DescriptionsList(
    descriptionProperties="propiedades",  // 1
    depends="depende de",                 // 2
    condition="condición",                // 3
    orderByKey="true|false",              // 4
    order="orden"                         // 5
)
```

1. **descriptionProperties** (opcional): Indica que propiedad o propiedades tienen 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. Permite poner una lista de propiedades separadas por comas. Al usuario le aparecen concatenadas.
2. **depends** (opcional): Se usa junto con *condition* 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).
3. **condition** (opcional): Permite poner una condición (al estilo SQL) para filtrar los valores que aparecen en la lista de descripciones.
4. **orderByKey** (opcional): Por defecto los datos salen ordenados por descripción, pero si ponemos está propiedad a *true* saldrán ordenados por clave.
5. **order** (opcional): Permite poner un orden (al estilo SQL) para los valores que aparecen en la lista de descripciones.

El uso más simple es:

```
@ManyToOne(fetch=FetchType.LAZY)
@DescriptionsList
private Almacen almacen;
```

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



En un principio saca todos los almacenes, aunque en realidad usa la *baseCondition* y *filter* 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:

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

@ManyToOne(fetch=FetchType.LAZY) @NoCreate    // 2
@DescriptionsList(
    descriptionProperties="descripcion",  // 3
    depends="familia",                    // 4
    condition="${familia.codigo} = ?"     // 5
    order="${descripcion} desc"           // 6
```

```
  )
private Subfamilia subfamilia;
```

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 *depends* (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 *condition* (5) y enviando como argumento (para rellenar el interrogante) el nuevo valor de familia. Y las entradas están ordenadas descendentemente por *descripcion* (6).
En *condition* y *order* 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:

```
@ManyToOne(fetch=FetchType.LAZY)
@ReadOnly
@DescriptionsList(descriptionProperties="nivel.descripcion, nombre")
private Comercial comercialAlternativo;
```

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 (*@DescriptionsList*) como solo lectura (*@ReadOnly*) se visualizará la descripción (en este caso *nivel.descripcion + nombre*) como si fuera una propiedad simple de texto y no como un combo.

Evento de cambio de valor de referencia

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

```
@ManyToOne(fetch=FetchType.LAZY)
@OnChange(AlCambiarTransportistaEnAlbaran.class)
private Transportista transportista;
```

En este caso nuestra acción escucha al cambio del código de transportista.
El código a ejecutar es:

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

}
```

La acción implementa *IOnChangePropertyAction*, mediante *OnChangePropertyBaseAction* (1), aunque es una referencia. Recibimos el cambio de la propiedad clave de la referencia; en este caso *transportista.codigo*. El resto es como en el caso de una propiedad.

Búsqueda de referencia al cambiar

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 la anotación *@OnChangeSearch*, justo así:

```
@ManyToOne(fetch=FetchType.LAZY)
@OnChangeSearch(BuscarAlCambiarSubfamilia.class)
private Subfamilia subfamilia;
```

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.

Personalización de colección

Podemos refinar la forma de visualización y comportamiento de una colección en la vista usando las siguientes anotaciones:

```
@CollectionView       //  1
@ReadOnly             //  2
@EditOnly             //  3
@NoCreate             //  4
@NoModify             //  5
@AsEmbedded           //  6
@ListProperties       //  7
@RowStyle             //  8
@EditAction           //  9
@ViewAction           // 10
@NewAction            // 11
@SaveAction           // 12
@HideDetailAction     // 13
```

```
@RemoveAction          // 14
@RemoveSelectedAction   // 15
@ListAction           // 16
@DetailAction          // 17
@OneToMany/@ManyToMany
private Collection nombreColeccion;
```

Todas estas anotaciones siguen las normas para anotaciones de vista y todas ellas son opcionales. OpenXava siempre asume valores por defecto correcto si se omiten.

1. **@CollectionView** (OX): La vista del objeto referenciado que se ha de usar para representar el detalle. Por defecto usa la vista por defecto.
2. **@ReadOnly** (OX): Si la ponemos solo podremos visualizar los elementos de la colección, no podremos ni añadir, ni borrar, ni modificar los elementos.
3. **@EditOnly** (OX): Si la ponemos podemos modificar los elementos existentes, pero no podemos añadir nuevos ni eliminar.
4. **@NoCreate** (OX): Si la ponemos el usuario final no tendrá el vínculo que le permite crear objetos del tipo del objeto referenciado. No aplica a colecciones incrustadas.
5. **@NoModify** (OX): Si la ponemos el usuario final no tendrá el vínculo que le permite modificar objetos del tipo del objeto referenciado. No aplica a colecciones incrustadas.
6. **@AsEmbedded** (OX): Por defecto las colecciones incrustadas permiten al usuario crear y añadir elementos, mientras que las colecciones convencionales permiten solo escoger entidades existentes para añadir (o quitar) de la colección. Si ponemos *@AsEmbedded* 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 incrustada.
7. **@ListProperties** (OX): 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). Solo una *@ListProperties* por vista está permitida.
8. **@RowStyle** (OX): Para dar un estilo especial a algunas filas. Se comporta igual que en el caso del Tab. No funciona para colecciones calculadas. Es posible definir varias *@RowStyle* por cada vista.
9. **@EditAction** (OX): 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. Solo una *@EditAction* por vista está permitida.

10. **@ViewAction** (OX): 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. Solo una *@ViewAction* por vista está permitida.
11. **@NewAction** (OX): 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'. Solo una *@NewAction* por vista está permitida.
12. **@SaveAction** (OX): 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'. Solo una *@SaveAction* por vista está permitida.
13. **@HideDetailAction** (OX): Permite definir nuestra propia acción para ocultar la vista de detalle. Ésta es la acción que se ejecuta al pulsar el vínculo 'Cerrar'. Solo una *@HideDetailAction* por vista está permitida.
14. **@RemoveAction** (OX): 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'. Solo una *@RemoveAction* por vista está permitida.
15. **@RemoveSelectedAction** (OX): 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'. Solo una *@RemoveSelectedAction* por vista está permitida.
16. **@ListAction** (OX): Para poder añadir acciones en el modo lista; normalmente acciones cuyo alcance es la colección entera. Es posible definir varias *@ListAction* por cada vista.
17. **@DetailAction** (OX): Para poder añadir acciones en detalle, normalmente acciones cuyo alcance es el detalle que se está editando. Es posible definir varias *@DetailAction* por cada vista.

Si no usamos ninguna de estas anotaciones 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:

```
@CollectionView("Simple"),
@ListProperties("codigo, nombre, observaciones, relacionConComercial, comercial.nivel.descripcion, tipo")
@OneToMany(mappedBy="comercial")
private Collection<Cliente> clientes;
```

De esta forma la colección se visualiza así:

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

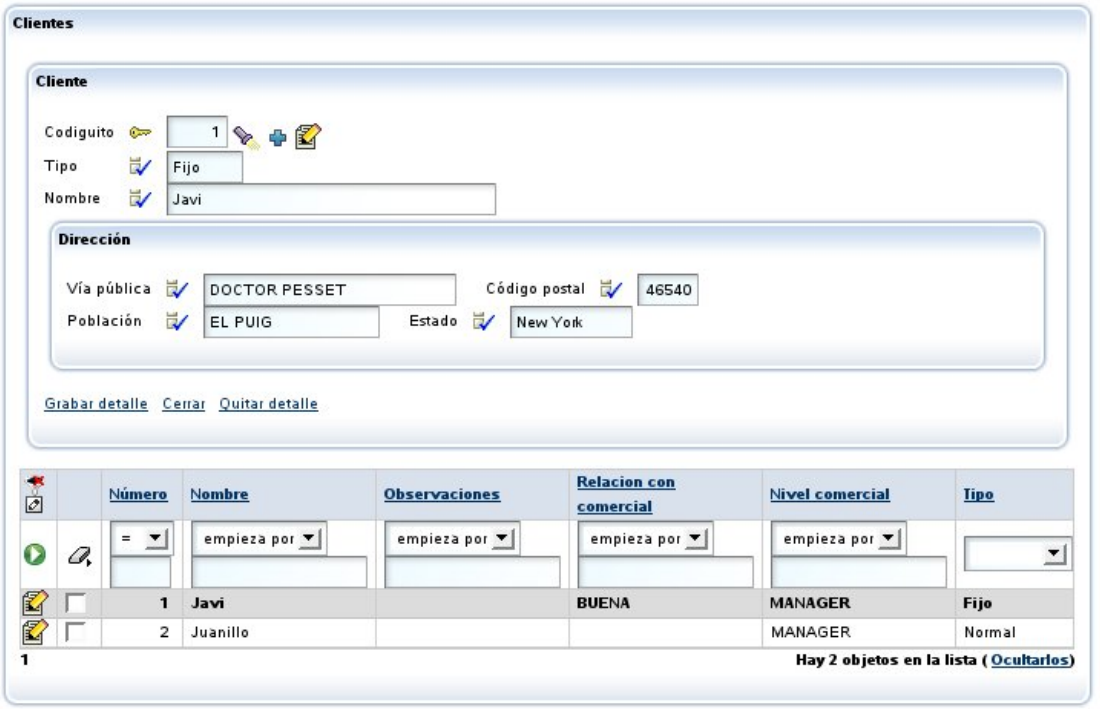
Al pulsar ('Editar') se visualizará el detalle usando la vista *Simple* de *Cliente*; para eso
hemos de tener una vista llamada *Simple* en la entidad *Cliente* (el modelo de los elementos de la colección).

Este vista se usa también cuando el usuario pulsa en 'Añadir' en una colección incrustada, en caso contrario OpenXava no muestra esta vista, en su lugar muestra una lista de entidades a añadir.

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

```
@View(name="Simple", members="codigo; tipo; nombre; direccion")
```

Al pulsar detalle aparecerá:



Acción de editar/ver detalle propia

Podemos refinar fácilmente el comportamiento cuando se pulse el vínculo  ('Editar') usando *@EditAction*:

```
@EditAction("Factura.editarLinea")
@OneToMany (mappedBy="factura", cascade=CascadeType.REMOVE)
private Collection<LineaFactura> lineas;
```

Hemos de definir *Factura.editarLinea* en *controladores.xml*:

```
<controlador nombre="Factura">
    ...
    <accion nombre="editarLinea" oculta="true"
        imagen="images/edit.gif"
        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, de esta manera:

```
@EditAction("")
@OneToMany (mappedBy="factura", cascade=CascadeType.REMOVE)
private Collection<LineaFactura> lineas;
```

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 (*@ReadOnly*).

La técnica para refinar una acción 'ver' (la acción para cada fila cuando la colección es de solo lectura) es la misma pero usando *@ViewAction* en vez de *@EditAction*.

Acciones de lista propias

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

```
@ListAction("Transportista.traducirNombre"),
private Collection<Transportista> compañeros;
```

Ahora aparecen un nuevo vínculo al usuario:

Falta definir la acción en *controladores.xml*:

```
<controlador nombre="Transportista">
    ...
    <accion nombre="traducirNombre" oculta="true"
        clase="org.openxava.test.actiones.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()) {
            Transportista transportista = (Transportista) 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 como acciones de lista para una colección.

Acciones de lista por defecto

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 *@ListAction* (ver la sección *acciones de lista propias*) pero aplica a todas las colecciones a la vez.
Esta característica no aplica a las colecciones calculadas.

Acciones de detalle propias

También podemos añadir nuestras propias acciones a la vista de detalle usada para editar cada elemento. Esto se consigue mediante la anotación *@DetailAction*. Estas sería acciones que aplican a un solo elemento de la colección. Por ejemplo:

```
@DetailAction("Factura.verProducto")
@OneToMany (mappedBy="factura", cascade=CascadeType.REMOVE)
private Collection<InvoiceDetail> lineas;
```

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 con *addDetailAction()*, *removeDetailAction()*, *addListAction()* y *removeListAction()*, ver API doc para *org.openxava.view.View*.

Refinar comportamiento por defecto para la vista de colección

Usando *@NewAction*, *@SaveAction*, *@HideDetailAction*, *@RemoveAction* y *@RemoveSelectedAction* 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:

```
@SaveAction("LineaAlbaran.grabar")
@OneToMany (mappedBy="albaran", cascade=CascadeType.REMOVE)
private Collection<LineaAlbaran> lineas;
```

Debemos tener la acción *LineaAlbaran.grabar* en *controladores.xml*:

```
<controlador nombre="LineaAlbaran">
    ...
    <accion nombre="grabar"
        clase="org.openxava.test.acciones.GrabarLineaAlbaran">
        <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.

También es posible eliminar cualquiera de estas acciones de la interfaz gráfica, por ejemplo, podemos defina una colección de esta manera:

```
@RemoveSelectedAction("")
@OneToMany (mappedBy="albaran", cascade=CascadeType.REMOVE)
private Collection<LineaAlbaran> lineas;
```

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.

Propiedades transitorias para controles gráficos

Con *@Transient (JPA)* podemos usar una propiedad que no se guarde en la base de datos, pero que sí nos interesa que se visualice al usuario. Podemos usarlas para proporcionar controles al usuario para manejar la interfaz gráfica. Un ejemplo:

```
@Transient
@DefaultValueCalculator(value=EnumCalculator.class,
    properties={
        @PropertyValue(name="enumType", value="org.openxava.test.modelo.Albaran$EntregadoPor")
        @PropertyValue(name="value", value="TRANSPORTISTA")
    }
)
@OnChange(AlCambiarEntradoPor.class)
private EntragadoPor entregadoPor;
public enum EntregadoPor { TRABAJADOR, TRANSPORTISTA }
```

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 *enum* y que tenga un *@DefaultValueCalculator*. Después de haber definido la propiedad podemos usarla en la vista como una propiedad más, asignandole una acción *@OnChange* por ejemplo y por supuesto poniendola como miembro de una vista.

Acciones de la vista

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 ponemos el nombre calificado de la acción seguido de paréntesis (), de esta manera:

```
@View( members=
    "codigo;" +
    "tipo;" +
    "nombre, Cliente.cambiarEtiquetaDeNombre();" +
    ...
```

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 poner la palabra ALWAYS entre los paréntesis, como sigue:

```
@View( members=
    "codigo;" +
    "tipo;" +
    "nombre, Cliente.cambiarEtiquetaDeNombre(ALWAYS);" +
    ...
```

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.

Clase transitoria: Solo para crear vistas

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 una clase, no marcarla como entidad y a partir de ésta definir una vista.
Una clase transitoria 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:

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

import javax.persistence.*;

import org.openxava.annotations.*;

/**
 * Ejemplo de una clase OpenXava transitoria (no persistente) del modelo. <p>
 *
 * Esto se puede usar, por ejemplo, para visualizar un diálogo,
 * o cualquier otro interfaz gráfica.<p>
 *
 * Notemos como no está marcada con @Entity <br>
 *
 * @author Javier Paniza
```

```
 */

@Views({
    @View(name="Familia1", members="subfamilia"),
    @View(name="Familia2", members="subfamilia"),
    @View(name="ConFormularioSubfamilia", members="subfamilia"),
    @View(name="Rango", members="subfamilia; subfamiliaHasta")
})
public class FiltroPorSubfamilia {

    @ManyToOne(fetch=FetchType.LAZY) @Required
    @NoCreate(forViews="Familia1, Familia2")
    @NoModify(forViews="Familia2, ConFormularioSubfamilia")
    @NoSearch(forViews="ConFormularioSubfamilia")
    @DescriptionsLists({
        @DescriptionsList(forViews="Familia1",
            condition="${familia.codigo} = 1", order="${codigo} desc"
        ),
        @DescriptionsList(forViews="Familia2",
            condition="${familia.codigo} = 2"
        )
    })
    private Subfamilia subfamilia;

    @ManyToOne(fetch=FetchType.LAZY)
    private Subfamilia subfamiliaHasta;

    public Subfamilia getSubfamilia() {
        return subfamilia;
    }

    public void setSubfamilia(Subfamilia subfamilia) {
        this.subfamilia = subfamilia;
    }

    public Subfamilia getSubfamiliaHasta() {
        return subfamiliaHasta;
    }

    public void setSubfamiliaHasta(Subfamilia subfamiliaHasta) {
        this.subfamiliaHasta = subfamiliaHasta;
    }

}
```

Para definir una clase del modelo como transitorio solo necesitamos definir una clase convencional sin *@Entity*. No hemos de poner el mapeo ni declarar propiedades como clave.

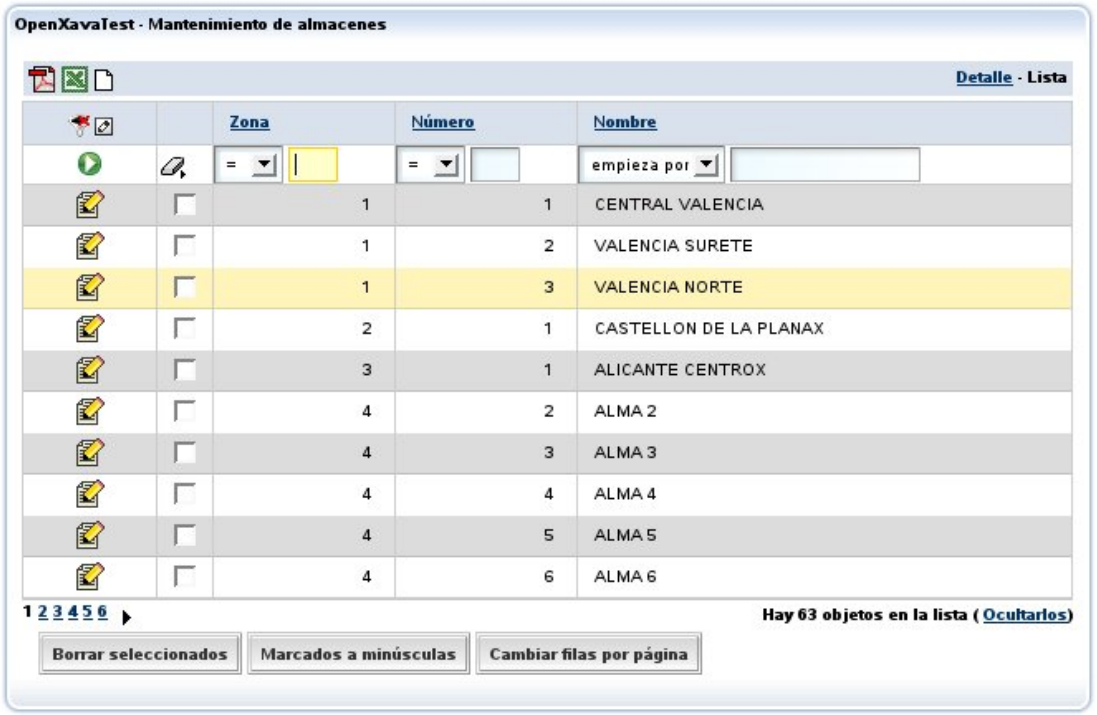
De esta forma podemos hacer un diálogo que puede servir, por ejemplo, para lanzar un listado de familias o productos filtrado por subfamilias.

Podemos así tener un generador de cualquier tipo de interfaz gráficas sencillo

y bastante flexible, aunque no queramos que la información visualizada sea persistente.

Capítulo 5: 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 la anotación *@Tab* dentro de la definición de la entidad.
La sintaxis de *@Tab* es:

```
@Tab(
    name="nombre",                  // 1
    filter=clase del filtro,          // 2
    rowStyles=array de @RowStyle,     // 3
    properties="propiedades",         // 4
    baseCondition="condición base",   // 5
    defaultOrder="orden por defecto"  // 6
)
public class MyEntity {
```

1. **name** (opcional): Podemos definir varios tabs para una entidad (mediante la anotación *@Tabs*), 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. **filter** (opcional): Permite definir programáticamente un filtro a realizar sobre los valores que introduce el usuario cuando quiere filtrar.
3. **rowStyles** (varios, opcional): Una forma sencilla de especificar una estilo de visualización diferente para ciertas filas. Normalmente para resaltar filas que cumplen cierta condición. Especificamos un array de *@RowStyle*, así podemos usar varios estilo por tab.
4. **properties** (opcional): La lista de propiedades a visualizar inicialmente. Pueden ser calificadas.
5. **baseCondition** (opcional): Es una condición que aplicará siempre a los datos visualizados añadiendose a las que pueda poner el usuario.
6. **defaultOrder** (opcional): Para especificar el orden en que aparece los datos en la lista inicialmente.

Propiedades iniciales y resaltar filas

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

```
@Tab(
    rowStyles=@RowStyle(style="highlight", property="tipo", value="fijo"),
    properties="nombre, tipo, comercial.nombre, direccion.municipio," +
        "comercial.nivel.descripcion, direccion.estado.nombre"
)
```

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 *@RowStyle*; 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 de Dirección	Nivel comercial	Estado de Dirección
		empieza por		empieza por	empieza por	empieza por	empieza por
		Javi	Fijo	MANUEL CHAVARRI	EL PUIG	MANAGER	NEW YORK
		Juanillo	Normal	MANUEL CHAVARRI	VALENCIA	MANAGER	COLORADO
		Carmelo	Normal		EL PUIG		NEW YORK
		Cuatrero	Normal		VALENCIA		NEW YORK

1 Hay 4 objetos en la lista (Ocultarlos)

Filtros y condición base

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

```
@Tab(name="Actuales",
    filter=FiltroAñoActual.class,
    properties="año, numero, sumaImportes, iva, cantidadLineas, pagada, cliente.nombre",
    baseCondition="${año} = ?"
)
```

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(name="AñoDefecto",
    filter=FiltroAñoDefecto.class,
    properties="año, numero, cliente.numero, cliente.nombre, sumaImportes, " +
        "iva, cantidadLineas, pagada, importancia",
    baseCondition="${año} = ?"
)
```

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 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 sobre controladores.

Select íntegro

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

```
@Tab(name="SelectIntegro",
    properties="codigo, descripcion, familia",
    baseCondition=
        "select" +
        "    ${codigo}, ${descripcion}, XAVATEST@separator@FAMILIA.DESCRIPCION " +
        "from " +
        "    XAVATEST@separator@SUBFAMILIA, XAVATEST@separator@FAMILIA " +
        "where " +
        "    XAVATEST@separator@SUBFAMILIA.FAMILIA = " +
        "    XAVATEST@separator@FAMILIA.CODIGO"
)
```

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

Orden por defecto

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

```
@Tab(name="Simple", properties="año, numero, fecha",
    defaultOrder="${año} desc, ${numero} desc"
```

```
}
```

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

Capítulo 6: 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 nuestra entidad. Las herramientas O/R nos permiten trabajar con objetos, en vez de con tablas y columnas y generan automáticamente el código SQL necesario para leer y actualizar la base de datos. De esta forma no necesitamos 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 las anotaciones de mapeo JPA.
Las entidades OpenXava son entidades JPA, por lo tanto el mapeo objeto/relacional en OpenXava se hace mediante Java Persistence API (JPA). Este capítulo muestra las técnicas más básicas y algunos casos especiales. Si queremos aprender más sobre JPA podemos consultar la documentación de Hibernate Annotations (la implementación de JPA usada por OpenXava por defecto), o cualquier otro manual de JPA que queramos.

Mapeo de entidad

La anotación *@Table* especifica la tabla principal para la entidad. Se pueden especificar tablas adicionales usando *@SecondaryTable* o *@SecondaryTables*.
Si no se especifica *@Table* para una entidad se aplicaran los valores por defecto.
Ejemplo:

```
@Entity
@Table(name="CLI", schema="XAVATEST")
public class Cliente {
```

Mapeo propiedad

La anotación *@Column* se usa para especificar como mapear una propiedad persistente. Si no se especifica *@Column* se aplican los valores por defecto.
Un ejemplo sencillo:

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

Un ejemplo anotando el *getter*:

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

Otros ejemplos:

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

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

Mapeo de referencia

La anotación *@JoinColumn* se usa para especificar el mapeo de una columna para una referencia.
Ejemplo:

```
@ManyToOne
@JoinColumn(name="CLI_ID")
private Cliente cliente;
```

Si necesitamos definir un mapeo para una clave foranea compuesta hemos de usar *@JoinColumns*. Esta anotación agrupa anotaciones *@JoinColumn* para la misma reference.
Cuando se usa la anotación *@JoinColumns*, tanto el atributo *nombre* como *referencedColumnName* tienen que especificarse en cada anotación *@JoinColumn*.
Ejemplo:

```
@ManyToOne
@JoinColumns({
    @JoinColumn(name="FAC_AÑO", referencedColumnName="AÑO"),
    @JoinColumn(name="FAC_NUMERO", referencedColumnName="NUMERO")
})
private Factura factura;
```

Mapeo de colección

Cuando usamos *@OneToMany* para una colección el mapeo depende de la referencia usada en la otra parte de la asociación, es decir, normalmente no es necesario hacer nada. Pero si estamos usando *@ManyToMany*, quizás nos sea útil declarar la tabla de unión (*@JoinTable*), como sigue:

```
@ManyToMany
@JoinTable(name="CLIENTE_PROVINCIA",
    joinColumns=@JoinColumn(name="CLIENTE"),
    inverseJoinColumns=@JoinColumn(name="PROVINCIA")
)
private Collection<Provincia> provincias;
```

Si omitimos *@JoinTable* se aplican los valores por defecto.

Mapeo de referencia incrustada

Una referencia incrustada contiene información que en el modelo relacional se guarda en la misma tabla que la entidad principal. Por ejemplo si tenemos un incrustable *Direccion* asociado a un *Cliente*, los datos de la dirección se guardan en la misma tabla que los del cliente. ¿Cómo se expresa eso con JPA? Es muy sencillo, usando la anotación *@AttributeOverrides*, de esta forma:

```
@Embedded
@AttributeOverrides({
    @AttributeOverride(name="calle", column=@Column("DIR_CALLE")),
    @AttributeOverride(name="codigoPostal", column=@Column("DIR_CP"))
    @AttributeOverride(name="poblacion", column=@Column("DIR_POB")),
    @AttributeOverride(name="pais", column=@Column("DIR_PAIS"))
})
private Direccion direccion;
```

Si no usamos *@AttributeOverrides* se asumen valores por defectos.

Conversión de tipo

La conversión de tipos entre Java y la base de datos relacional es un trabajo de la implementación de JPA (OpenXava usa Hibernate por defecto). Normalmente, la conversión de tipos por defecto es buena para la mayoría de los casos, pero si trabajamos con bases de datos legadas quizás necesitemos algunos de los trucos que aquí se muestran.

Dado que OpenXava usa la facilidad de conversión de tipos de Hibernate podemos aprender más en la documentación de Hibernate.

Conversión de propiedad

Cuando el tipo de una propiedad Java y el tipo de su columna correspondiente en la base de datos no coincide necesitamos escribir un *Hibernate Type* para poder hacer nuestra conversión de tipo personalizada.
Por ejemplo, si tenemos una propiedad de tipo *String []*, y queremos almacenar su valor concatenándolo en una sola columna de base de datos de tipo VARCHAR. Entonces tenemos que declarar la conversión para nuestra propiedad de esta manera:

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

La lógica de conversión en RegionesType es:

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

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

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

/**
 *
 * @author Javier Paniza
 */

public class RegionesType implements UserType {  // 1

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

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

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

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

```
    public Object nullSafeGet(ResultSet resultSet, String[] names, Object owner)  // 2
        throws HibernateException, SQLException
    {
        Object o = resultSet.getObject(names[0]);
           if (o == null) return new String[0];
           String dbValue = (String) o;
           String [] javaValue = new String [dbValue.length()];
           for (int i = 0; i < javaValue.length; i++) {
               javaValue[i] = String.valueOf(dbValue.charAt(i));
           }
           return javaValue;
    }

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

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

    public boolean isMutable() {
        return true;
    }

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

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

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

}
```

El conversor de tipo ha de implementar *org.hibernate.usertype.UserType* (1). Los métodos principales son *nullSafeGet* (2) para leer de la base de datos y

convertir a Java, y *nullSafeSet* (3) para escribir el valor Java en la base de datos.
OpenXava tiene conversores de tipo de Hibernate genéricos en el paquete *org.openxava.types* listos para usar. Uno de ellos es *EnumLetterType*, que permite mapear propiedades de tipo *enum*. Por ejemplo, si tenemos una propiedad como esta:

```
private Distancia distancia;
public enum Distancia { LOCAL, NACIONAL, INTERNACIONAL };
```

En esta propiedad Java 'LOCAL' es 1, 'NATIONAL' es 2 and 'INTERNATIONAL' es 3 cuando la propiedad se almacena en la base de datos. Pero, ¿qué ocurre, si en la base de datos se almacena una única letra ('L', 'N' or 'I')? En este caso podemos usar *EnumLetterType* de esta forma:

```
@Type(type="org.openxava.types.EnumLetterType",
    parameters={
        @Parameter(name="letters", value="LNI"),
        @Parameter(name="enumType", value="org.openxava.test.modelo.Albaran$Distancia")
    }
)
private Distancia distancia;
public enum Distancia { LOCAL, NACIONAL, INTERNACIONAL }
```

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 de tipos nos permite hacer conversores reutilizables.

Conversión con multiples columnas

Con *CompositeUserType* 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.
Un ejemplo típico sería usar el conversor genérico *Date3Type*, que permite almacenar en la base de datos 3 columnas y en Java una propiedad *java.util.Date*.

```
@Type(type="org.openxava.types.Date3Type")
@Columns(columns = {
    @Column(name="AÑOENTREGA"),
    @Column(name="MESENTREGA"),
```

```
    @Column(name="DIAENTREGA")
})
private java.util.Date fechaEntrega;
```

DIAENTREGA, MESENTREGA y AÑOENTREGA son las tres columnas que en la base de datos guardan la fecha de entrega. Y aquí *Date3Type*:

```
package org.openxava.types;

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

import org.hibernate.*;
import org.hibernate.engine.*;
import org.hibernate.type.*;
import org.hibernate.usertype.*;
import org.openxava.util.*;

/**
 * In java a <tt>java.util.Date</tt> and in database 3 columns of
 * integer type. <p>
 *
 * @author Javier Paniza
 */

public class Date3Type implements CompositeUserType {  // 1

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

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

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

    public void setPropertyValue(Object component, int property, Object value)
        throws HibernateException    // 3
    {
        java.util.Date date = (java.util.Date) component;
        int intValue = value == null?0:((Number) value).intValue();
        switch (property) {
```

```
            case 0:
                Dates.setYear(date, intValue);
            case 1:
                Dates.setMonth(date, intValue);
            case 2:
                Dates.setYear(date, intValue);
        }
        throw new HibernateException(XavaResources.getString("date3_type_only_3_properties"));
    }

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

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

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

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

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

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

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

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

    public boolean isMutable() {
        return true;
```

```
    }

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

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

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

}
```

Como se ve el conversor de tipo implementa *CompositeUserType* (1). Los métodos clave son *getPropertyValue* (2) y *setPropertyValue* (3) para coger y poner valores en las propiedades del objeto del tipo compuesto, y *nullSafeGet* (4) y *nullSafeSet* (5) para leer y grabar este objeto en la base de datos.

Conversión de referencia

La conversión de referencias no se soporta directamente por Hibernate. Pero en alguna circunstancias extremas puede ser que necesitemos hacer conversión de referencias. En esta sección se explica como hacerlo.

Por ejemplo, puede que tengamos una referencia a permiso de conducir usando dos columnas, PERMISOCONDUCIR_NIVEL y PERMISOCONDUCIR_TIPO, y la columna PERMISOCONDUCIR_TIPO no admita nulos, pero es posible que el objeto puede no tener permiso de conducir, en cuyo caso la columna PERMISOCONDUCIR_TIPO almacena una cadena vacía. Esto no es algo normal si nosotros diseñamos la base de datos usando claves foráneas, pero si la base de datos fue diseñada por un programador RPG, por ejemplo, esto se habrá hecho de esta forma, porque los programadores RPG no están acostumbrados a lidiar con nulos.

Es decir, necesitamos una conversión para PERMISOCONDUCIR_TIPO, para transformar el nulo en una cadena vacía. Esto se puede conseguir con un código como este:

```
// Aplicamos conversión (nulo en una cadena vacía) a la columna PERMISOCONDUCIR_TIPO
// Para hacerlo, creamos permisoConducir_nivel y permisoConducir_tipo
// Hacemos JoinColumns no insertable ni modificable, modificamos el método get/setPermisoConducir
// y creamos un método conversionPermisoConducir().
@ManyToOne(fetch=FetchType.LAZY)
@JoinColumns({  // 1
    @JoinColumn(name="PERMISOCONDUCIR_NIVEL", referencedColumnName="NIVEL",
         insertable=false, updatable=false),
    @JoinColumn(name="PERMISOCONDUCIR_TIPO", referencedColumnName="TIPO",
         insertable=false, updatable=false)
})
private PermisoConducir permisoConducir;
private Integer permisoConducir_nivel;  // 2
private String permisoConducir_tipo;    // 2

public PermisoConducir getPermisoConducir() {  // 3
    // De esta manera porque la columna tipo de permiso de conducir no admite nulos
    try {
        if (permisoConducir != null) permisoConducir.toString(); // para forzar la carga
        return permisoConducir;
    }
    catch (EntityNotFoundException ex) {
        return null;
    }
}

public void setPermisoConducir(PermisoConducir permiso) {  // 4
    // De esta manera porque la columna tipo de permiso de conducir no admite nulos
    this.permisoConducir = permiso;
    this.permisoConducir_nivel = permiso==null?null:permiso.getNivel();
    this.permisoConducir_tipo = permiso==null?null:permiso.getTipo();
}

@PrePersist @PreUpdate
private void conversionPermisoConducir() {  // 5
    if (this.permisoConducir_tipo == null) this.permisoConducir_tipo = "";
}
```

Lo primero poner *@JoinColumns* con *insertable=false* y *updatable=false* en todas las *@JoinColumn* (1), de esta manera la referencia es leida de la base de datos, pero no escrita. También tenemos que definir propiedades planas para almacenar la clave foránea de la referencia (2).

Ahora tenemos que escribir un *getter*, *getPermisoConducir()* (3), para devolver nulo cuand la referencia no se encuentre, y un *setter*, *setPermisoConducir()* (4), para asignar la clave de la referencia a las propiedades planas correspondientes.

Finalmente, hemos de escribir un método de retrollamada, *conversionPermisoConducir()* (5), para hacer el trabajo de conversión. Este método será automáticamente ejecutado al crear y actualizar.

Este ejemplo shows como es posible envolver bases de datos legadas simplemente usando un poco de programación y algunos recursos básicos de JPA.

Capítulo 7: Controladores

Los controladores sirven para definir las acciones (botones, vínculos, imágenes) que el usuario final puede pulsar. Los controladores se definen en un archivo llamado *controladores.xml* que ha de estar en el directorio *xava* de nuestro proyecto. No definimos las acciones junto con los componentes porque hay muchas acciones de uso genérico que pueden ser aplicadas a cualquier componente.
En *OpenXava/xava* tenemos un *default-controllers.xml* que contiene un grupo de componente de uso genérico que podemos usar en nuestras aplicaciones.
El archivo *controladores.xml* contiene un elemento de tipo *<controladores/>* con la sintaxis:

```
<controladores>
    <var-entorno … /> …     <!-- 1 -->
    <objeto … /> …          <!-- 2 -->
    <controlador … /> …     <!-- 3 -->
</controladores>
```

1. **var-entorno** (varias, opcional): Variable que contienen información de configuración. Estas variables pueden ser accedidas desde las acciones y filtros, y su valor puede ser sobreescrito para cada módulo.
2. **objeto** (varios, opcional): Define objetos Java de sesión, es decir objetos que se crean y existen durante toda la sesión del usuario.
3. **controlador** (varios, obligado): Los controladores son agrupaciones de acciones.

Variable de entorno

Las variables de entorno contienen información de configuración. Estas variables pueden ser accedidas desde las acciones y los filtros, y su valor puede ser sobrescrito en cada módulo. Su sintaxis es:

```
<var-entorno
    nombre="nombre"          <!-- 1 -->
    valor="valor"            <!-- 2 -->
/>
```

1. **nombre** (obligado): Nombre de la variable de entorno en mayúsculas y usando subrayados para separar palabras.
2. **valor** (obligado): Valor para la variable de entorno.

Estos son algunos ejemplos:

```
<var-entorno nombre="MIAPLICACION_AÑO_DEFECTO" value="2007"/>
<var-entorno nombre="MIAPLICACION_COLOR" valor="ROJO"/>
```

Objectos de sesión

Los objetos Java declarados en *controladores.xml* tienen ámbito de sesión; es decir, son objetos que son creado para un usuario y existen durante toda su sesión. Su sintaxis es:

```
<objeto
   nombre="nombreObjeto"          <!-- 1 -->
   clase="tipoObjeto"            <!-- 2 -->
   valor="valorInicial"           <!-- 3 -->
   ambito="modulo|global"         <!-- 4  Muevo en v2.1 -->
/>
```

1. **nombre** (obligado): Nombre del objeto, normalmente usaremos el nombre de la aplicación como prefijo para evitar colisión de nombres en proyectos grandes.
2. **clase** (obligado): Nombre calificado de la clase Java para esto objeto.
3. **valor** (opcional): Valor inicial para el objeto.
4. **ambito** (opcional): *(Nuevo en v2.1)* El valor por defecto es module. Si usamos modulo como ámbito cada módulo tendrá su propia copia de este objeto. Si usamos global como ámbito el mismo objeto será compartido por todos los módulos de todas la aplicaciones OpenXava (que se ejecuten dentro del mismo war).

Definir objetos de sesión es muy fácil, podemos ver los que están definidos en *OpenXava/xava/default-controllers.xml*:

```
<object name="xava_view" class="org.openxava.view.View"/>
<object name="xava_referenceSubview" class="org.openxava.view.View"/>
<object name="xava_tab" class="org.openxava.tab.Tab"/>
<object name="xava_mainTab" class="org.openxava.tab.Tab"/>
<object name="xava_row" class="java.lang.Integer" value="0"/>
<object name="xava_language" class="org.openxava.session.Language"/>
<object name="xava_newImageProperty" class="java.lang.String"/>
<object name="xava_currentReferenceLabel" class="java.lang.String"/>
<object name="xava_activeSection" class="java.lang.Integer" value="0"/>
```

```
<object name="xava_previousControllers" class="java.util.Stack"/>
<object name="xava_previousViews" class="java.util.Stack"/>
```

Estos objetos son usado por OpenXava para su funcionamiento interno, aunque es bastante normal que los usemos en nuestras propias acciones. Para definir nuestro propios objetos podemos hacerlo en *controladores.xml* en el directorio *xava* de nuestro proyecto.

El controlador y sus acciones

La sintaxis de un controlador es:

```
<controlador
    nombre="nombre"           <!-- 1 -->
>
    <hereda-de … /> …      <!-- 2 -->
    <accion … /> …         <!-- 3 -->
</controlador>
```

1. **nombre** (obligado): Nombre del controlador.
2. **hereda-de** (varios, opcional): Permite usar herencia múltiple, para que este controlador herede todas las acciones de otro (u otros) controlador.
3. **accion** (varios, obligada): Definición de la lógica a ejecutar cuando el usuario pulse un botón o vínculo.

Obviamente los controladores los formas las acciones, que son en sí lo importante. Aquí su sintaxis:

```
<accion
    nombre="nombre"                                  <!--  1 -->
    etiqueta="etiqueta"                               <!--  2 -->
    descripcion="descripcion"                          <!--  3 -->
    modo="detail|list|ALL"                            <!--  4 -->
    imagen="imagen"                                  <!--  5 -->
    clase="clase"                                    <!--  6 -->
    oculta="true|false"                               <!--  7 -->
    al-iniciar="true|false"                           <!--  8 -->
    en-cada-peticion="true|false"                      <!--  9  Nuevo en v2.1.2 -->
    antes-de-cada-peticion="true|false"                <!-- 10  Nuevo en v2.2.5 -->
    por-defecto="nunca|si-posible|casi-siempre|siempre"  <!-- 11 -->
    cuesta="true|false"                               <!-- 12 -->
    confirmar="true|false"                            <!-- 13 -->
    atajo-de-teclado="atajo-de-teclado"               <!-- 14  Nuevo en v2.0.1 -->
>
    <poner … /> …                                  <!-- 15 -->
    <usa-objeto … /> …                             <!-- 16 -->
</accion>
```

1. **nombre** (obligado): Nombre identificativo de la acción tiene que ser único dentro del controlador, pero puede repetirse el nombre en diferentes controladores. Cuando referenciemos a una acción desde fuera lo haremos siempre especificando *NombreControlador.nombreAccion*.
2. **etiqueta** (opcional): Etiqueta del botón o texto del vínculo. Es mucho mejor usar los archivos i18n.
3. **descripcion** (opcional): Texto descriptivo de la acción. Es mucho mejor usar los archivos i18n.
4. **modo** (opcional): Indica en que modo ha de ser visible esta acción para el usuario. Por defecto es ALL, que quiere decir que esta acción es siempre visible.
5. **imagen** (opcional): URL de la imagen asociada a la acción. En la implementación actual si especificamos imagen aparece la imagen como un vínculo en el que el usuario puede pulsar.
6. **clase** (opcional): Clase que implementa la lógica a ejecutar. Ha de implementar la interfaz *IAction*.
7. **oculta** (opcional): Una acción oculta no aparece por defecto en la barra de botones, aunque sí que se puede usar para todo lo demás, por ejemplo como acción asociada a un evento de cambio de valor, acción de propiedad, en las colecciones, etc. Por defecto vale *false*.
8. **al-iniciar** (opcional): Si la ponemos a *true* esta acción se ejecutará automáticamente al iniciar el módulo. Por defecto vale *false*.
9. **en-cada-peticion** (opcional): *(Nuevo en v2.1.2)* Si la ponemos a *true* esta acción se ejecutará automáticamente en cada petición del usuario, es decir, en la primera ejecución del módulo y antes de la ejecución de cada acción del usuario. En el momento de la ejecución todos los objetos de sesión de OpenXava están configurados y listos para usar. Es decir, desde esta acción podemos usar *xava_view* y *xava_tab*. Por defecto vale *false*.
10. **antes-de-cada-peticion** (opcional): *(Nuevo en v2.2.5)* Si la ponemos a true esta acción se ejecutará automáticamente antes de cada petición del usuario, es decir, en la primera ejecución del módulo y antes de la ejecución de cada acción del usuario, pero antes de que los objetos de sesión de OpenXava estén configurados y listos para usar. Es decir, desde esta acción no podemos usar *xava_view* ni *xava_tab*. Por defecto vale false.
11. **por-defecto** (opcional): Indica el peso de esta acción a la hora de seleccionar cual es la acción por defecto. Las acción por defecto es la

que se ejecuta cuando el usuario pulsa ENTER. Por defecto vale *nunca*.

12. **cuesta** (opcional): Si la ponemos a *true* indicamos que esta acción cuesta tiempo en ejecutarse (minutos u horas), en la implementación actual OpenXava visualiza una barra de progreso. Por defecto vale *false*.
13. **confirmar** (opcional): Si la ponemos a **true** antes de ejecutarse la acción un diálogo le preguntará al usuario si está seguro de querer ejecutarla. Por defecto vale *false*.
14. **atajo-de-teclado** (opcional): Define una atajo de teclado que el usuario puede pulsar para ejecutar esta acción. Los valores posibles son los mismos que para *javax.swing.KeyStroke*. Ejemplos: "control A", "alt x", "F7" *(nuevo en v2.0.1)*.
15. **poner** (varios, opcional): Sirve para dar valor a las propiedades de la acción. De esta forma una misma acción configurada de forma diferente puede usarse en varios controladores.
16. **usa-objeto** (varios, opcional): Asigna un objeto de sesión a una propiedad de la acción antes de ejecutarse, y al acabar recoge el valor de la propiedad y lo coloca en el contexto (actualiza el objeto de sesión).

Las acciones son objetos de corta vida, cuando el usuario pulsa un botón se crea el objeto acción, se configura con lo valores de *poner* y *usa-objeto*, se ejecuta y se actualiza los objetos de sesión, y después de eso se desprecia.
Un controlador sencillo puede ser:

```
<controlador nombre="Observaciones">
    <accion nombre="ocultarObservaciones"
        clase="org.openxava.test.acciones.OcultarMostrarPropiedad">
        <poner propiedad="propiedad" valor="observaciones" />
        <poner propiedad="ocultar" valor="true" />
        <usa-objeto nombre="xava_view"/>
    </accion>
    <accion nombre="mostrarObservaciones" modo="detail"
        clase="org.openxava.test.actiones.OcultarMostrarPropiedad">
        <poner propiedad="propiedad" valor="observaciones" />
        <poner propiedad="ocultar" valor="false" />
        <usa-objeto nombre="xava_view"/>
    </accion>
    <accion nombre="ponerObservaciones" modo="detail"
        clase="org.openxava.test.acciones.PonerValorPropiedad">
        <poner propiedad="propiedad" valor="observaciones" />
        <poner propiedad="valor" valor="Demonios tus ojos" />
        <usa-objeto nombre="xava_view"/>
    </accion>
</controladores>
```

Podemos ahora incluir este controlador en el módulo deseado; esto se hace editando en *xava/aplicacion.xml* el módulo en el que deseemos usar estas acciones:

```
<modulo nombre="Albaranes">
    <modelo nombre="Albaran"/>
    <controlador nombre="Typical"/>
    <controlador nombre="Observaciones"/>
</modulo>
```

De esta forma en este módulo tendremos disponibles las acciones de *Typical* (mantenimiento e impresión) más las que nosotros hemos definido en nuestro controlador *Observaciones*. La barra de botones superior del módulo tendrá el siguiente aspecto:

Y la barra de botones inferior:

Vemos como las acciones con imagen se colocan arriba, y las acciones sin imagen abajo.
Podemos observar el código *ocultarObservaciones* por ejemplo:

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

import org.openxava.actions.*;

/**
 * @author Javier Paniza
 */

public class OcultarMostrarPropiedad extends ViewBaseAction {    // 1

    private boolean ocultar;
    private String propiedad;

    public void execute() throws Exception {                    // 2
        getView().setHidden(propiedad, ocultar);                // 3
    }
```

```
    public boolean isOcultar() {
        return ocultar;
    }

    public void setOcultar(boolean b) {
        ocultar = b;
    }

    public String getPropiedad() {
        return propiedad;
    }

    public void setPropiedad(String string) {
        propiedad = string;
    }

}
```

Una acción ha de implementar *IAction*, pero normalmente se hace que descienda de una clase base que a su vez implemente esta interfaz. La acción base básica es *BaseAction* que implementa la mayoría de los métodos de *IAction* a excepción de *execute()*. En este caso usamos *ViewBaseAction* como clase base. *ViewBaseAction* tiene una propiedad view de tipo *View*. Esto unido a que al declarar la acción hemos puesto...

```
<usa-objeto nombre="xava_view"/>
```

...permite desde esta acción manipular mediante view la vista, o dicho de otra forma la interfaz de usuario que éste está viendo.
El *<usa-objeto />* coge el objeto de sesión *xava_view* y lo asigna a la propiedad *view* (quita el prefijo *xava_*, y en general quita el prefijo *miaplicacion_* antes de asignar el objeto) de nuestra acción justo antes de llamar a *execute()*.
Ahora dentro del método *execute()* podemos usar *getView()* a placer (3), en este caso para ocultar una propiedad. Todas las posibilidades de *View* las podemos ver consultando la documentación JavaDoc de *org.openxava.view.View*.
Con...

```
        <poner propiedad="propiedad" valor="observaciones" />
        <poner propiedad="ocultar" valor="true" />
```

establecemos valores fijos a las propiedades de nuestra acción.

Herencia de controladores

Podemos crear un controlador que herede todas sus acciones de uno o más controladores. Un ejemplo de esto lo encontramos en el controlador genérico más típico *Typical*, este controlador se encuentra en *OpenXava/xava/default-controllers.xml*:

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

A partir de ahora cuando indiquemos que un módulo usa el controlador *Typical* este módulo tendrá a su disposición todas las acciones de *Print* (para generar informes PDF y Excel) y *CRUD* (para hace altas, bajas, modificaciones y consultas).
Podemos usar la herencia para refinar la forma de trabajar de un controlador estándar, como sigue:

```
<controlador nombre="Familia">
   <hereda-de controlador="Typical"/>
   <accion nombre="new" imagen="images/new.gif"
      clase="org.openxava.test.acciones.CrearNuevaFamilia">
      <usa-objeto nombre="xava_view"/>
   </accion>
</controlador>
```

Como el nombre de nuestra acción *new* coincide con la de *Typical* (en realidad la de *CRUD* del cual desciende *Typical*) se anula la original y se usará la nuestra. Así de fácil podemos indicar que ha de hacer nuestro módulo cuando el usuario pulse nuevo.

Acciones en modo lista

Podemos hacer acciones que apliquen a varios objetos. Estas acciones normalmente solo se visualizan en modo lista y suelen actuar sobre los objetos que el usuario haya escogido.
Un ejemplo puede ser:

```
<accion nombre="borrarSeleccionados" modo="list"        <!-- 1 -->
   confirmar="true"                                  <!-- 2 -->
   clase="org.openxava.actions.DeleteSelectedAction">
</accion>
```

Ponemos mode="list" para que solo aparezca en modo lista (1). Ya que esta acción borra registros hacemos que el usuario tenga que confirmar antes de ejecutarse (2). No es necesario incluir un *<usa-objeto/>* para *xava_tab (nuevo en v2.1.4)*.
Programar la acción sería así:

```
package org.openxava.actions;

import java.util.*;

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

/**
 * @author Javier Paniza
 */

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

    public void execute() throws Exception {
        int [] selectedOnes = getTab().getSelected();                              // 2
        if (selectedOnes != null) {
            for (int i = 0; i < selectedOnes.length; i++) {
                Map clave = (Map)
                    getTab().getTableModel().getObjectAt(selectedOnes[i]);
                try {
                    MapFacade.remove(model, clave);                                // 3
                }
                catch (ValidationException ex) {
                    addError("no_delete_row", new Integer(i), clave);// (4)
                    addErrors(ex.getErrors());
                }
                catch (Exception ex) {
                    addError("no_delete_row", new Integer(i), clave);
                }
            }
            getTab().deselectAll();                                                // 5
            resetDescriptionsCache();                                              // 6
        }
    }

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

Esta acción es una acción estándar de OpenXava, pero nos sirve para ver que cosas podemos hacer dentro de nuestras acciones de modo lista. Observamos (1) como desciende de *TabBaseAction* e implementa *IModelAction*, al descender de *TabBaseAction (new in v2.1.4)* tiene un conjunto de utilidades disponible y no estamos obligados a implementar todos los

métodos de *IAction*; y al implementar *IModelAction* nuestra acción tendrá un metoto *setModel()* (7) con el que recibirá el nombre del modelo (del componente OpenXava) antes de ejecutarse.
Puedes acceder al *Tab* usando el método *getTab()* (2); este método está implementado en *TabBaseAction* y permite acceder al objeto de sesión *xava_tab*. Mediante *getTab()* podemos manipular la lista de objetos visualizados. Por ejemplo, con *getTab().getSelected()* (2) obtenemos los índices de las filas seleccionadas, con *getTab().getTableModel()* un table model para acceder a los datos, y con *getTab().deselectAll()* deseleccionar las filas. Podemos echar un vistazo a la documentación JavaDoc de *org.openxava.tab.Tab* para más detalles sobre sus posibilidades.
Algo muy interesante que se ve en este ejemplo es el uso de la clase *MapFacade* (3). *MapFacade* permite acceder a la información del modelo mediante mapas de Java (*java.util.Map*), esto es conveniente cuando obtenemos datos de *Tab* o *View* en formato *Map* y queremos con ellos actualizar el modelo (y por ende la base de datos) o viceversa. Todas las clases genéricas de OpenXava interactúan con el modelo mediante *MapFacade* y nosotros también lo podemos usar, pero como consejo general de diseño decir que trabajar con mapas es práctico para proceso automáticos pero cuando queremos hacer cosas específicas es mejor usar directamente los objetos del modelo. Para más detalles podemos ver la documentación JavaDoc de *org.openxava.model.MapFacade*.
Observamos como añadir mensajes que serán visualizados al usuario con *addError()*. El método *addError()* recibe el id de una entrada en nuestros archivos i18n y los argumentos que el mensaje pueda usar. Los mensajes añadidos se visualizaran al usuario como errores. Si queremos añadir mensajes de advertencia podemos usar *addMessage()* que tiene exactamente el mismo funcionamiento que *addError()*. Los archivos i18n para errores y mensajes han de llamarse *MiProyecto-messages.properties* o *MensajeMiProyecto.properties* y el sufijo del idioma (_en, _ca, _es, _it, etc). Podemos ver como ejemplos los archivos que hay en *OpenXavaTest/xava/i18n*. Todas las excepciones no atrapadas producen un mensaje de error genérico, excepto si la excepción es una *ValidationException* en cuyo caso visualiza el mensaje de error de la excepción.
El método *resetDescriptionsCache()* (6) borra los caché usados por OpenXava para visualizar listas de descripciones (combos), es conveniente llamarlo siempre que se actualicen datos.
Podemos ver más posibilidades si vemos la documentación JavaDoc de *org.openxava.actions.BaseAction* y *org.openxava.actions.TabBaseAction*.

Desde v2.1.4 este tipo de acciones también pueden ser usadas como @ListAction *(<accion-lista/>* de una *<vista-coleccion/>)*.

Sobreescribir búsqueda por defecto

Cuando en un módulo nos aparece el modo lista y pulsamos para visualizar un detalle, entonces OpenXava busca el objeto correspondiente y lo visualiza en el detalle. Ahora bien si en modo detalle rellenamos la clave y pulsamos a buscar (unos prismático) también hace lo mismo. Y cuando navegamos por los registros pulsando siguiente o anterior hace la misma búsqueda. ¿Cómo podemos personalizar las búsqueda? Vamos a ver cómo.
Lo único que hemos de hacer es definir nuestro módulo en *xava/aplicacion.xml* de la siguiente forma:

```
<modulo nombre="Albaranes">
    <var-entorno nombre="XAVA_SEARCH_ACTION" valor="Albaranes.buscar"/>
    <modelo nombre="Albaran"/>
    <controlador nombre="Typical"/>
    <controlador nombre="Observaciones"/>
    <controlador nombre="Albaranes"/>
</modulo>
```

Podemos observar que definimos una variable de entorno XAVA_SEARCH_ACTION que tiene el valor de la acción que queremos usar para buscar. Esa acción está definida en *xava/controladores.xml* así:

```
<controlador nombre="Albaranes">
    <accion nombre="buscar" modo="detail"
        por-defecto="si-posible" oculta="true"
        clase="org.openxava.test.acciones.BuscarAlbaran"
        atajo-de-teclado="F8">
        <usa-objeto nombre="xava_view"/>
    </accion>
    ...
</controlador>
```

Y su código es:

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

import java.util.*;

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

/**
 * @author Javier Paniza
```

```
 */

public class BuscarAlbaran extends SearchByViewKeyAction {                  // 1

    public void execute() throws Exception {
        super.execute();                                                    // 2
        if (!Is.emptyString(getView().getValueString("empleado"))) {
            getView().setValue("entregadoPor", new Integer(1));
            getView().setHidden("transportista", true);
            getView().setHidden("empleado", false);
        }
        else {
            Map transportista = (Map) getView().getValue("transportista");
            if (!(transportista == null || transportista.isEmpty())) {
                getView().setValue("entregadoPor", new Integer(2));
                getView().setHidden("transportista", false);
                getView().setHidden("empleado", true);
            }
            else {
                getView().setHidden("transportista", true);
                getView().setHidden("empleado", true);
            }
        }
    }

}
```

Básicamente hemos de buscar en la base de datos (o mediante las APIs de EJB2, EJB3 JPA o Hibernate) y llenar la vista. Muchas veces lo más práctico es hacer que extienda de *SearchByViewKeyAction* (1) y dentro del *execute()* hacer un *super.execute()* (2).

OpenXava viene con 3 acciones de búsquedas:

- **CRUD.searchByViewKey**: Esta es la configurada por defecto. Hace una búsqueda a partir de la clave que hay ese momento en la vista, no ejecuta ningún evento.
- **CRUD.searchExecutingOnChange**: Funciona como la anterior pero al buscar ejecuta las acciones *@OnChange/al-cambiar* asociadas a las propiedades de la vista.
- **CRUD.searchReadOnly**: Funciona como *searchByViewKey* pero pone la vista de detalle a estado no editable al buscar. Útil para crear módulos de consulta.

Si queremos que al buscar ejecute las acciones al cambiar tenemos que definir nuestro módulo de la siguiente forma:

```
<modulo nombre="ProductosAccionesAlCambiarAlBuscar">
    <var-entorno nombre="XAVA_SEARCH_ACTION" valor="CRUD.searchExecutingOnChange"/>
    <modelo nombre="Producto"/>
    <controlador nombre="Typical"/>
```

```
    <controlador nombre="Productos"/>
    <controlador-modo nombre="Void"/>
</modulo>
```

Como se ve, simplemente poniendo valor a la variable de entorno XAVA_SEARCH_ACTION.

Inicializando un módulo con una acción

Con solo poner *al-iniciar*="*true*" cuando definimos una acción hacemos que esta acción se ejecute automáticamente cuando se ejecuta el módulo por primera vez. Esto nos da una oportunidad para inicializar nuestro módulo. Veamos un ejemplo. En nuestro *controladores.xml* ponemos:

```
<controlador nombre="Facturas2002">
    <accion nombre="iniciar" al-iniciar="true" oculta="true"
        clase="org.openxava.test.acciones.IniciarAñoDefectoA2002">
        <usa-objeto nombre="xavatest_añoDefecto"/>
        <usa-objeto nombre="xava_tab"/>
    </accion>
    ...
</controlador>
```

Y en nuestra acción:

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

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

/**
 * @author Javier Paniza
 */

public class IniciarAñoDefectoA2002 extends BaseAction {

    private int añoDefecto;
    private Tab tab;

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

    public int getAñoDefecto() {
        return añoDefecto;
    }

    public void setAñoDefecto(int i) {
```

```
        añoDefecto = i;
    }

    public Tab getTab() {
        return tab;
    }

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

}
```

Establecemos el año por defecto a 2002 (1), hacemos que el título de la lista sea visible (2) y asignamos un valor como argumento para ese título (3). El título de la lista está definido en los archivos i18n, normalmente se usa para los informes, pero podemos visualizarlos también en modo lista.

Llamar a otro módulo

A veces resulta conveniente llamar programáticamente desde un módulo a otro. Por ejemplo, imaginemos que queremos sacar una lista de clientes y al pulsar en uno nos aparezca una lista de sus facturas y al pulsar en la factura poder editarla. Una manera de conseguir esto es tener un módulo de clientes que tenga solo la lista y al pulsar vayamos al modulo de facturas haciendo que el tab filtre para mostrar solo las de ese cliente. Vamos a verlo. Primero definiríamos el módulo en *aplicacion.xml* de la siguiente forma:

```
<modulo nombre="FacturasDeClientes">
    <var-entorno nombre="XAVA_LIST_ACTION" valor="Facturas.listarDeCliente"/>  <!-- 1 -->
    <modelo nombre="Cliente"/>
    <controlador nombre="Print"/>
    <controlador nombre="ListOnly"/>                                          <!-- 2 -->
    <controlador-modo nombre="Void"/>                                         <!-- 3 -->
</modulo>
```

En este modulo solo aparece la lista (sin la parte de detalle) para eso decimos que el controlador de modo ha de ser *Void* (3) y así no aparece lo de detalle y lista, y añadimos un controlador llamado *ListOnly* (2) para que sea el modo lista el que aparezca (si ponemos controlador de modo *Void* y nada más por defecto aparecería solo el detalle). Además declaramos la variable XAVA_LIST_ACTION para que apunte a una acción nuestra, ahora cuando el usuario pulse en el vínculo que aparece en cada fila de la lista ejecutará nuestra propia acción. Esta acción hemos de declararla en *controladores.xml*:

```
<controlador nombre="Facturas">
    <accion nombre="listarDeCliente" oculta="true"
        clase="org.openxava.test.acciones.ListarFacturasDeCliente">
        <usa-objeto nombre="xava_tab"/>
    </accion>
    ...
</controlador>
```

Y el código de la acción:

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

import java.util.*;

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

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

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

    public void execute() throws Exception {
        Map claveCliente = (Map) tab.getTableModel().getObjectAt(row);          // 4
        int codigoCliente = ((Integer) claveCliente.get("codigo")).intValue();
        Tab tabFacturas = (Tab)
            context.get("OpenXavaTest", getNextModule(), "xava_tab");           // 5
        tabFacturas.setBaseCondition("${cliente.codigo} = "+codigoCliente);     // 6
    }

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

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

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

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

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

}
```

Para poder cambiar de módulo la acción implementa *IChangeModuleAction* (1) esto hace que tenga que tener un método *getNextModule()* (7) que sirve para indicar cual será el módulo al que cambiaremos después de ejecutar la acción, y *hasReinitNextModule()* (9) para indicar si queremos que se reinicie el módulo al cambiar a él.
Por otra parte hace que implemente *IModuleContextAction* (2) que hace que esta acción reciba un objeto de tipo *ModuleContext* con el método *setContext()* (8). *ModuleContext* nos permite acceder a objetos de sesión de otros módulos, es útil para poder configurar el módulo al que vamos a cambiar.
Otro detalle es que la acción que se pone como valor para XAVA_LIST_ACTION ha de tener un propiedad llamada *row* (3); antes de ejecuta la acción se llena esta propiedad con la fila en la que el usuario ha pulsado.
Teniendo esto en cuenta es fácil entender lo que hace la acción:

- Coge la clave del objeto asociada a la fila pulsada (4), para ello usa el tab del modulo actual.
- Accede al tab del módulo al que vamos usando *context* (5).
- Establece la condición base del tab del módulo al que vamos a ir (6) usando la clave obtenida del tab actual.

Cambiar el modelo de la vista actual

Como alternativa a cambiar de módulo podemos optar por cambiar el modelo de la vista actual. Hacer esto es muy sencillo solo hemos de usar las APIs disponible en *View*. Un ejemplo:

```
public void execute() throws Exception {
    try {
        setValoresFactura(getView().getValues());                            // 1
        Object codigo = getCollectionElementView().getValue("producto.codigo");
```

```
        Map clave = new HashMap();
        clave.put("codigo", codigo);
        getView().setModelName("Producto");                                    // 2
        getView().setValues(clave);                                            // 3
        getView().findObject();                                                // 4
        getView().setKeyEditable(false);
        getView().setEditable(false);
    }
    catch (ObjectNotFoundException ex) {
        getView().clear();
        addError("object_not_found");
    }
    catch (Exception ex) {
        ex.printStackTrace();
        addError("system_error");
    }
}
```

Este es un extracto de una acción que permite visualizar pulsando la acción un objeto de otro tipo. Lo primero que hacemos es guardarnos los datos visualizados actualmente (1), para poder dejar la vista como estaba cuando volvamos. Después cambiamos el módulo de la vista (2), esto es la parte clave. Ahora solo llenamos los valores clave (3) y con *findObject()* (4) hacemos que se rellene lo demás.

Cuando usamos esta técnica hemos de tener presente que cada módulo tiene un solo objeto *xava_view* activo a la vez, así que si queremos volver hacia atrás tenemos que ocuparnos nosotros de poner el modelo y vista original en la vista así como de restaurar la información que tenía.

Ir a una página JSP

El generador automático de vista de OpenXava suele ir bien para la inmensa mayoría de los casos, pero puede que nos interese visualizar al usuario una página JSP diseñada manualmente por nosotros. Podemos hacer esto con una acción como esta:

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

import org.openxava.actions.*;

/**
 * @author Javier Paniza
 */

public class MiAccionBuscar extends BaseAction implements INavigationAction {  // 1
```

```
    public void execute() throws Exception {
    }

    public String[] getNextControllers() {                                  // 2
        return new String [] { "MiReferencia" } ;
    }

    public String getCustomView() {                                         // 3
        return "quieresBuscar.jsp";
    }

    public void setKeyProperty(String s) {
    }

}
```

Para ir a una vista personalizada (a una página JSP en este caso) hacemos que nuestra acción implemente *INavigationActionICustomViewAction* (con hubiera bastado) y de esta forma podemos indicar con *getNextControllers()* (2) los siguientes controladores a usar y con *getCustomView()* (3) la página JSP que ha de visualizarse (3).

Generar un informe propio con JasperReports

OpenXava permite al usuario final generar sus propios informes desde el modo lista. El usuario puede filtrar, ordenar, añadir/quitar campos, cambiar la posición de los campos y entonces generar un informe PDF.
Pero todas las aplicaciones de gestión no triviales necesitan sus propios informes creados programáticamente. Puedes hacer esto fácilmente usando JasperReports e integrando tu informe en tu aplicación OpenXava con la acción *JasperReportBaseAction*.
En primer lugar tienes que diseñar tu informe JasperReports, puedes hacerlo usando el excelente diseñador iReport.
Una vez hecho eso puedes escribir tu acción de impresión de esta manera:

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

import java.util.*;

import net.sf.jasperreports.engine.*;
import net.sf.jasperreports.engine.data.*;

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

```
import org.openxava.validators.*;

/**
 * Informe de productos de la subfamilia seleccionada. <p>
 *
 * Usa JasperReports. <br>
 *
 * @author Javier Paniza
 */
public class InformeProductosDeFamiliaAction extends JasperReportBaseAction {  // 1

    private ISubfamilia2 subfamilia;

    public Map getParameters() throws Exception  {                          // 2
        Messages errores =
            MapFacade.validate("FiltroPorSubfamilia", getView().getValues());
        if (errores.contains()) throw new ValidationException(errores);       // 3
        Map parametros = new HashMap();
        parametros.put("familia", getSubfamilia().getFamilia().getDescripcion());
        parametros.put("subfamilia", getSubfamilia().getDescripcion());
        return parametros;
    }

    protected JRDataSource getDataSource() throws Exception {                 // 4
        return new JRBeanCollectionDataSource(
            getSubfamilia().getProductosValues());
    }

    protected String getJRXML() {                                             // 5
        return "Productos.jrxml"; // Para leer del classpath
        //return "/home/javi/Products.jrxml"; // Para leer del sistema de ficheros
    }

    private ISubfamilia2 getSubfamilia() throws Exception {
        if (subfamilia == null) {
            int codigoSubfamilia = getView().getValueInt("subfamilia.codigo");
            // Usando Hibernate, lo más típico
            subfamilia = (ISubfamilia2)
                 XHibernate.getSession().get(
                    Subfamilia2.class, new Integer(codigoSubfamilia));
            // Usando EJB
            // subfamilia = Subfamilia2Util.getHome().
            //    findByPrimaryKey(new Subfamilia2Key(codigoSubfamilia));
        }
        return subfamilia;
    }

}
```

Solo necesitas que tu acción extienda de *JasperReportBaseAction* (1) y sobreescribir los siguientes 3 métodos:

- **getParameters**() (2): Un *Map* con los parámetros a enviar al informe, en este caso hacemos también la validación de los datos entrados (usando *MapFacade.validate()*) (3).
- **getDataSource**() (4): Un *JRDataSource* con los dato a imprimir. En este caso una colección de JavaBeans obtenidos llamando a un objeto modelo. Si usas EJB EntityBeans CMP2 sé cuidadoso y no hagas un bucle sobre una colección de EntityBeans EJB2 dentro de este método, como en este caso obtén los datos con una sola llamada EJB.
- **getJRXML**() (5): El XML con el diseño JasperReports, este archivo puede estar en el classpath. Puedes tener para esto una carpeta de código fuente llamada informes en tu proyecto. Otra opción es poner este archivo en el sistema de ficheros *(nuevo en v2.0.3)*, esto se consigue especificando la ruta completa del archivo, por ejemplo: */home/javi/Products.jrxml*.

Por defecto el informe es visualizado en una ventana emergente, pero si lo deseas puedes sobreescribir el método *inNewWindow()* para que el informa aparezca en la ventana actual.
Podemos encontrar más ejemplos de acciones JasperReport en el proyecto OpenXavaTest, como *InvoiceReportAction* para imprimir una Factura.

Cargar y procesar un fichero desde el cliente (formulario multipart)

Esta característica nos permite procesar en nuestra aplicación OpenXava un archivo binario (o varios) enviado desde el cliente. Esto está implementado en un contexto HTTP/HTML con formularios multipart de HTML, aunque el código OpenXava es tecnologicamente neutral, por ende nuestra acción será portable a otros entornos sin recodificar.
Para cargar un archivo lo primero es crear una acción para ir al formulario en donde el usuario pueda escoger su archivo. Esta acción tiene que implementar *ILoadFileAction*, de esta forma:

```
public class CambiarImagen extends BaseAction implements ILoadFileAction {  // 1
    ...
    public void execute() throws Exception {                                 // 2
    }

    public String[] getNextControllers() {                                   // 3
        return new String [] { "CargarImagen" };
    }
```

```
    public String getCustomView() {                                    // 4
        return "xava/editors/cambiarImagen";
    }

    public boolean isLoadFile() {                                      // 5
        return true;
    }

    ...
}
```

Una acción *ILoadFileAction* (1) es también una *INavigationAction* que nos permite navegar a otros controladores (3) y a otra vista personalizada (4). El nuevo controlador (3) normalmente tendrá un acción del tipo *IProcessLoadedFileAction*. El método *isLoadFile()* (5) devuelve true en el caso de que queramos navegar al formulario para cargar el archivo, puedes usar la lógica en *execute()* (2) para determinar este valor. La vista personalizada es (4) un JSP con tu propio formulario para cargar el fichero.
Un ejemplo de JSP para una vista personalizada puede ser:

```
<%@ include file="../imports.jsp"%>

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

<table>
<th align='left' class=<%=style.getLabel()%>>
<fmt:message key="introducir_nueva_imagen"/>
</th>
<td>
<input name = "nuevaImagen" class=<%=style.getEditor()%> type="file" size='60'/>
</td>
</table>
```

Como se puede ver, no se especifica el formulario HTML, porque el módulo OpenXava ya tiene uno incluido.
La última pieza es la acción para procesar los archivos cargados:

```
public class CargarImagen extends BaseAction
    implements INavigationAction, IProcessLoadedFileAction {          // 1

    private List fileItems;
    private View view;
    private String newImageProperty;

    public void execute() throws Exception {
        Iterator i = getFileItems().iterator();                        // 2
        while (i.hasNext()) {
            FileItem fi = (FileItem)i.next();                          // 3
            String fileName = fi.getName();
            if (!Is.emptyString(fileName)) {
```

```
                getView().setValue(getNewImageProperty(), fi.get());   // 4
            }
        }
    }

    public String[] getNextControllers() {
        return DEFAULT_CONTROLLERS;
    }

    public String getCustomView() {
        return DEFAULT_VIEW;
    }

    public List getFileItems() {
        return fileItems;
    }

    public void setFileItems(List fileItems) {                        // 5
        this.fileItems = fileItems;
    }
    ...
}
```

La acción implementa *IProcessLoadedFileAction* (1), así la acción tiene que tener un método *setFileItem()* (5) para recibir la lista de los archivos descargados. Esta lista puede procesarse en *execute()* (2). Los elementos de la colección son del tipo *org.apache.commons.fileupload.FileItem* (4) (del proyecto fileupload de apache commons). Llamando a *get()* (4) en el *file item* podemos acceder al contenido del archivo cargado.

Sobreescribir los controladores por defecto (nuevo en v2.0.3)

Los controladores en *OpenXava/xava/default-controllers.xml* (antes de v2.0.3 era *OpenXava/xava/controllers.xml*) son usados por OpenXava para dar a la aplicación un comportamiento por defecto. Muchas veces la forma más fácil de modificar el comportamiento de OpenXava es creando nuestros propios controladores y usandolos en nuestras aplicaciones, es decir, podemos crear un controlador llamado *MiTipico*, y usarlo en vez del *Typical* que viene con OpenXava.
Otra opción es sobreescribir un controlador por defecto de OpenXava. Para poder sobreescribir un controlador por defecto solo necesitamos crear en nuestra aplicación un controlador con el mismo nombre que el de defecto. Por ejemplo, si queremos refinar el comportamiento de las colecciones para

nuestra aplicación tenemos que crear un controlador *Collection* en nuestro *xava/controladores.xml*, como sigue:

```
<controlador nombre="Collection">
    <accion nombre="new"
        clase="org.openxava.actions.CreateNewElementInCollectionAction">
    </accion>
    <accion nombre="hideDetail"                                    <!-- 1 -->
        clase="org.openxava.test.acciones.MiOcultarDetalle">
    </accion>
    <accion nombre="save"
        clase="org.openxava.actions.SaveElementInCollectionAction">
        <usa-objeto nombre="xava_view"/>
    </accion>
    <accion nombre="remove"
        clase="org.openxava.actions.RemoveElementFromCollectionAction">
        <usa-objeto nombre="xava_view"/>
    </accion>
    <accion nombre="edit"
        clase="org.openxava.actions.EditElementInCollectionAction">
        <usa-objeto nombre="xava_view"/>
    </accion>
    <accion nombre="view"
        clase="org.openxava.actions.EditElementInCollectionAction">
        <usa-objeto nombre="xava_view"/>
    </accion>
</controlador>
```

En este case solo sobreescribimos el comportamiento de la accion *hideDetail* (1). Pero tenemos que declarar todas las acciones del controlador original, porque OpenXava confia en todas estas acciones para funcionar; no podemos borrar o renombrar acciones.

Todos los tipos de acciones

Se puede observar por lo visto hasta ahora que nosotros podemos hacer que nuestra acción implemente una interfaz u otra para hacer que se comporte de una manera u otra. A continuación se enumeran las interfaces que tenemos disponibles para nuestras acciones:

- **IAction**: Interfaz básica que obligatoriamente ha de implementar toda acción.
- **IChainAction**: Permite encadenar acciones, es decir que cuando se termine de ejecutar nuestra acción ejecute otra inmediatamente.
- **IChainActionWithArgv**: *(Nuevo en v2.2)* Es un refinamiento de *IChainAction*. Permite enviar como argumentos valores para llenar las propiedades de la acción encadenada antes de ejecutarla.

- **IChangeControllersAction**: Para cambiar los controladores (y por ende las acciones) disponible al usuario.
- **IChangeModeAction**: Para cambiar de modo, de lista a detalle o viceversa.
- **IChangeModuleAction**: Para cambiar de módulo.
- **ICustomViewAction**: Para que la vista sea una página JSP propia.
- **IForwardAction**: Redirecciona a un Servlet o página JSP. No es como *ICustomViewAction*, *ICustomViewAction* hace que la vista que está dentro de nuestro interfaz generado con OpenXava (que a su vez puede estar dentro de un portal) sea nuestro JSP, mientras que *IForwardAction* redirecciona de forma completa a la URI indicada.
- **IHideActionAction**, **IHideActionsAction**: Permite ocultar una acción o un conjunto de acciones en la interfaz de usuario *(nuevo en v2.0)*.
- **IJDBCAction**: Permite usar directamente JDBC en una acción. Recibe un *IConnectionProvider*. Funciona de forma parecida a un *IJDBCCalculator* (ver capítulo 3).
- **ILoadFileAction**: Permite navegar a una vista con la posibilidad de cargar un archivo.
- **IModelAction**: Una acción que recibe el nombre del modelo.
- **IModuleContextAction**: Recibe un *ModuleContext* para poder acceder a objetos de sesión de otros módulos, por ejemplo.
- **INavigationAction**: Extiende de *IChangeControllersAction* y *ICustomViewAction*.
- **IOnChangePropertyAction**: Este interfaz lo ha de implementar las acciones que reaccionan a un cambio de valor de propiedad en la interfaz gráfica.
- **IProcessLoadedFileAction**: Procesa una lista de archivos cargados desde el cliente al servidor.
- **IRemoteAction**: Útil para cuando se usa EJB2. Bien usada puede ser un buen sustituto de un *SessionBean*.
- **IRequestAction**: Recibe un request de Servlets. Hace que nuestras acciones se vinculen a la tecnología de servlets/jsp, por lo que es mejor evitarla. Pero a veces es necesario cierta flexibilidad.
- **IShowActionAction**, **IShowActionsAction**: Permite mostrar una acción o un grupo de acciones previamente ocultadas en una *IHideAction(s)Action (nuevo en v2.0)*.

Para saber más como funcionan las acciones lo ideal es mirar la API JavaDoc

del paquete *org.openxava.actions* y ver los ejemplos disponibles en el proyecto *OpenXavaTest*.

Capítulo 8: Aplicación

Una aplicación es el software que el usuario final puede usar. Hasta ahora hemos visto como definir las piezas que forman una aplicación (los componentes y las acciones principalmente), ahora vamos a ver como ensamblarlas para crear aplicaciones.
La definición de una aplicación OpenXava se hace en el archivo *aplicacion.xml* que encontramos en el directorio *xava* de nuestro proyecto.
La sintaxis de este archivo es:

```
<aplicacion
    nombre="nombre"                 <!-- 1 -->
    etiqueta="etiqueta"              <!-- 2 -->
>
    <modulo-defecto … /> …        <!-- 3  Nuevo en v2.2.2 -->
    <modulo … /> …               <!-- 4 -->
</aplicacion>
```

1. **nombre** (obligado): Nombre de la aplicación.
2. **etiqueta** (opcional): Mucho mejor usar archivos i18n.
3. **modulo-defecto** (uno, opcional): *Nuevo en v2.2.2*. Para definir los controladores para los módulos por defecto (generados automáticamente para cada componentes).
4. **modulo** (varios, opcionales): Cada módulo es ejecutable directamente por el usuario final.

Se ve claramente que una aplicación es un conjunto de módulos. Vamos a ver como se define un módulo:

```
<modulo
    nombre="nombre"                <!--  1 -->
    carpeta="carpeta"              <!--  2 -->
    etiqueta="etiqueta"            <!--  3 -->
    descripcion="descripcion"       <!--  4 -->
>
    <var-entorno … /> …          <!--  5 -->
    <modelo … />                  <!--  6 -->
    <vista … />                   <!--  7 -->
    <vista-web … />               <!--  8 -->
    <tab … />                    <!--  9 -->
    <controlador … /> …          <!-- 10 -->
    <controlador-modo … />         <!-- 11 -->
    <doc … />                    <!-- 12 -->
</modulo>
```

1. **nombre** (obligado): Identificador único del módulo dentro de esta aplicación.
2. **carpeta** (opcional): Carpeta en la cual residirá el módulo. Es una sugerencia para clasificar los módulos. De momento es usado para generar la estructura de carpetas para JetSpeed2 pero su uso puede ser ampliado en el futuro. Podemos usar / o . para indicar subcarpetas (por ejemplo, "facturacion/informes" o "facturacion.informes").
3. **etiqueta** (opcional): Nombre corto que se visualizará al usuario. Mucho mejor usar archivos i18n.
4. **descripcion** (opcional): Descripción larga que se visualizará al usuario.
5. **var-entorno** (varias, opcional): Permite definir una variable con un valor que podrán ser accedidos posteriormente desde las acciones. Así podemos tener acciones configurables según el módulo.
6. **modelo** (uno, opcional): Indica el nombre de componente usado en este módulo. Si no lo ponemos estamos obligados a usar *vista-web*.
7. **vista** (una, opcional): El nombre de la vista que se va a usar para dibujar el detalle. Si no lo ponemos usará la vista por defecto para ese modelo.
8. **vista-web** (una, opcional): Nos permite indicar nuestro propia página JSP que será usada como vista.
9. **tab** (uno, opcional): El nombre del tab que usará la el modo lista. Si no lo ponemos usará el tab por defecto.
10. **controlador** (varios, opcional): Controladores con las acciones que aparecen en el módulo al iniciarse.
11. **controlador-modo** (uno, opcional): Permite definir el comportamiento para pasar de detalle a lista, o bien definir un módulo que no tenga detalle y lista.
12. **doc** (uno, opcional): Es exclusivo con todos los demás elementos. Permite definir módulos que solo contienen documentación, no lógica. Útil para generar portlets informativos para nuestras aplicaciones.

Un módulo típico

Definir un módulo sencillo puede ser como sigue:

```
<aplicacion nombre="Gestion">
   <modulo nombre="Almacen" carpeta="almacen">
      <modelo nombre="Almacen"/>
      <controlador nombre="Typical"/>
      <controlador nombre="Almacen"/>
```

```
    </modulo>
    …
</aplicacion>
```

En este caso tenemos un módulo que nos permite hacer altas, bajas modificaciones, consultas, listados en PDF y exportación a Excel de los datos de los almacenes (gracias a *Typical*) y acciones propias que aplican solo a almacenes (gracias al controlador *Almacen*). En el caso en que el sistema genere una estructura de módulos (como en JetSpeed2) este módulo estará en la carpeta "almacen".
Para ejecutar este módulo podemos desde nuestro navegador escribir:
http://localhost:8080/Gestion/xava/
modulo.jsp?application=Gestion&module=Almacen
También se genera un portlet para poder desplegar el módulo como un portlet JSR-168 en un portal Java.

Módulos por defecto (nuevo en v2.2.2)

OpenXava asume un módulo por defecto para cada componente en la aplicación, aunque el módulo no se defina explícitamente en *aplicacion.xml*.
Es decir, si definimos un componente Factura.xml, podemos abrir nuestro navegador e ir a:
http://localhost:8080/Gestion/xava/
modulo.jsp?application=Gestion&module=Factura
También un portlet es generado para permitir desplegar el módulo como un portlet JSR-168 en un portal Java.
Y todo esto sin necesidad de definirlo en *aplicacion.xml*.
El controlador para estos módulos por defecto será *Typical*, pero podemos cambiar este valor por defecto usando el elemento *modulo-defecto* en *aplicacion.xml*, de esta manera:

```
<aplicacion nombre="Gestion">

    <modulo-defecto>
        <controlador nombre="MantenimientoGestion"/>
    </modulo-defecto>

</aplicacion>
```

En este caso todos los módulos por defecto de la aplicación *Gestion* tendrán el controlador *MantenimientoGestion* asignado a ellos.

Si queremos que cierto módulo no use estos controladores por defecto, tenemos dos opciones:

1. Definir un controlador en nuestro *controladores.xml* con el mismo nombre que el componente.
2. Definir explícitamente el módulo en *aplicacion.xml*, tal y como se explica arriba.

Resumiendo, si definimos un componente, llamado *Cliente* por ejemplo, entonces tenemos un módulo llamado *Cliente*, y también un portlet. Este módulo se definirá de una de la siguiente formas:

1. Si definimos un módulo llamado *Cliente* en *aplicacion.xml* entonces este módulo será el válido, si no...
2. Si definimos un controlador llamado *Cliente* en *controlladores.xml* un módulo será generado usando el controlador *Cliente* como controlador y el componente *Cliente* como modelo, si no...
3. Si definimos un elemento *modulo-defecto* en nuestro *aplicacion.xml* entonces un modulo se generará usando los controladores en *modulo-defecto* y el componente *Cliente* como modelo, si no ...
4. un módulo con *Typical* como controlador y *Cliente* como modelo se asumirá en última instancia.

Módulo con solo detalle

Un módulo con solo tenga modo detalle, sin lista se define así:

```
<modulo nombre="FacturaSinLista">
    <modelo nombre="Factura"/>
    <controlador nombre="Typical"/>
    <controlador-modo nombre="Void"/>    <!-- 1 -->
</modulo>
```

El controlador de modo *Void* (1) es para que no aparezcan los vínculos "detalle – lista"; en esta caso el módulo usa por defecto el modo detalle únicamente.

Módulo con solo lista

Un módulo con solo modo lista, sin detalle se define así:

```
<modulo nombre="FamiliaSoloLista">
    <var-entorno nombre="XAVA_LIST_ACTION" valor=""/>    <!-- 1  Nuevo en v2.0.4 -->
    <modelo nombre="Familia"/>
```

```
    <controlador nombre="Typical"/>
    <controlador nombre="ListOnly"/>                  <!-- 2 -->
    <controlador-modo nombre="Void"/>                 <!-- 3 -->
</modulo>
```

El controlador de modo *Void* (3) es para que no aparezcan los vínculos "detalle – lista". Además al definir *ListOnly* (2) como controlador el módulo cambia a modo lista al iniciar, por lo tanto éste es un módulo de solo lista. Finalmente, poniendo XAVA_LIST_ACTION a cadena vacía (1) el vínculo de detalle en cada fila no aparece *(nuevo en v2.0.4)*.

Módulo de documentación

Un módulo de documentación solo visualiza un documento HTML. Es fácil de definir:

```
<modulo nombre="Descripcion">
    <doc url="doc/descripcion" idiomas="es,en"/>
</modulo>
```

Este módulo muestra el documento *web/doc/descripcion_en.html* o *web/doc/descripcion_es.html* según el idioma del navegador. Si el idioma del navegador no es inglés o español entonces asume español (el primer idioma especificado). Si no especificamos idioma entonces el documento a visualizar será *web/doc/descripcion.html*.
Esto es útil para portlets informativos. Este tipo de módulos no tiene efecto fuera de un portal.

Módulo de solo lectura

Un módulo de solo lectura, es decir solo para consultar no para modificar, puede ser definido como sigue:

```
<modulo nombre="ConsultaClientes">
    <var-entorno nombre="XAVA_SEARCH_ACTION" valor="CRUD.searchReadOnly"/>  <!-- 1 -->
    <modelo nombre="Cliente"/>
    <controlador nombre="Print"/>                                          <!-- 2 -->
</modulo>
```

Usando *CRUD.searchReadOnly* (1) el usuario no puede editar los datos, y usando solo el controlador *Print* (2) (sin *CRUD* ni *Typical*) las acciones para grabar, borrar, etc. no están presentes. Esto es un simple módulo de consulta.

La sintaxis de *aplicacion.xml* no tiene mucha complicación. Podemos ver más ejemplos en *OpenXavaTest/xava/application.xml*.

La interfaz de usuario generada por OpenXava es buena para la mayoría de los casos, pero a veces puede que necesitemos personalizar alguna parte de la interfaz de usuario (creando nuestros propios editores) o crear nuestra interfaz de usuario íntegramente a mano (usando vistas personalizadas con JSP).

Editores

Configuración de editores

Vemos como el nivel de abstracción usado para definir las vista es alto, nosotros especificamos las propiedades que aparecen y como se distribuyen, pero no cómo se visualizan. Para visualizar las propiedades OpenXava utiliza editores.
Un editor indica como visualizar una propiedad. Consiste en una definición XML junto con un fragmento de código JSP.
Para refinar el comportamiento de los editores de OpenXava o añadir los nuestros podemos crear en el directorio *xava* de nuestro proyecto un archivo llamado *editores.xml*. Este archivo es como sigue:

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

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

<editores>
    <editor …/> …
</editores>
```

Simplemente contiene la definición de un conjunto de editores, y un editor se define así:

```
<editor
    url="url"                              <!-- 1 -->
    formatear="true|false"                   <!-- 2 -->
    depende-de-estereotipos="estereotipos"   <!-- 3 -->
    depende-de-propiedades="propiedades"     <!-- 4 -->
    enmarcable="true|false"                  <!-- 5 -->
>
```

```
    <propiedad … /> …                    <!-- 6 -->
    <formateador … />                      <!-- 7 -->
    <para-estereotipo … /> …             <!-- 8 -->
    <para-tipo … /> …                   <!-- 8 -->
    <para-propiedad-modelo … /> …        <!-- 8 -->
</editor>
```

1. **url** (obligado): URL de la página JSP que implementa el editor.
2. **formatear** (opcional): Si es *true* es OpenXava el que tiene la responsabilidad de formatear los datos desde HTML hasta Java y viceversa, si vale *false* tiene que hacerlo el propio editor (generalmente recogiendo información del *request* y asignandolo a *org.openxava.view.View* y viceversa). Por defecto vale *true*.
3. **depende-de-estereotipos** (opcional): Lista de estereotipos separados por comas de los cuales depende este editor. Si en la misma vista hay algún editor para estos estereotipos éstos lanzarán un evento de cambio si cambian.
4. **depende-de-propiedades** (opcional): Lista de propiedades separadas por comas de los cuales depende este editor. Si en la misma vista se está visualizando alguna de estas propiedades éstas lanzarán un evento de cambio si cambian.
5. **enmarcable** (opcional): Si vale *true* enmarca visualmente el editor. Por defecto vale *false*. Es útil para cuando hacemos editores grandes (de más de una línea) que pueden quedar más bonitos de esta manera.
6. **propiedad** (varias, opcional): Permite enviar valores al editor, de esta forma podemos configurar un editor y poder usarlo en diferente situaciones.
7. **formateador** (uno, opcional): Clase java para definir la conversión de Java a HTML y de HTML a Java.
8. **para-estereotipo** o **para-tipo** o **para-propiedad-modelo** (obligada una de ellas, y solo una): Asocia este editor a un estereotipo, tipo o a una propiedad concreta de un modelo. Tiene preferencia cuando asociamos un editor a una propiedad de un modelo, después por estereotipo y como último por tipo.

Podemos ver un ejemplo de definición de editor, este ejemplo es uno de los editores que vienen incluidos con OpenXava, pero es un buen ejemplo para aprender como hacer nuestros propios editores:

```
<editor url="textEditor.jsp">
    <for-type type="java.lang.String"/>
```

```
    <for-type type="java.math.BigDecimal"/>
    <for-type type="int"/>
    <for-type type="java.lang.Integer"/>
    <for-type type="long"/>
    <for-type type="java.lang.Long"/>
</editor>
```

Aquí asignamos a un grupo de tipos básicos el editor *textEditor.jsp*. El código JSP de este editor es:

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

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

Un editor JSP recibe un conjunto de parámetros y tiene accesos a atributos que le permiten configurarse adecuadamente para encajar bien en una vista OpenXava. En primer lugar vemos como cogemos *propertyKey* (1) que después usaremos como id HTML. A partir de ese id podemos acceder a la *MetaProperty* (2) (que contiene toda la meta información de la propiedad a editar). El atributo *fvalue* (3) contiene el valor ya formateado y listo para visualizar. Averiguamos también la alineación (4) y si es o no editable (5).

También recibimos el trozo de script de javascript (6) que hemos de poner en el editor.
Aunque crear un editor directamente con JSP es sencillo no es una tarea muy habitual, es más habitual configurar JSPs ya existentes. Por ejemplo si en nuestro *xava/editores.xml* ponemos:

```
<editor url="textEditor.jsp">
    <formatedor clase="org.openxava.formatters.UpperCaseFormatter"/>
    <para-tipo tipo="java.lang.String"/>
</editor>
```

Estaremos sobreescribiendo el comportamiento de OpenXava para las propiedades de tipo *String*, ahora todas las cadenas se visualizaran y aceptaran en mayúsculas. Podemos ver el código del formateador:

```
package org.openxava.formatters;

import javax.servlet.http.*;

/**
 * @author Javier Paniza
 */

public class UpperCaseFormatter implements IFormatter {              // 1

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

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

}
```

Un formateado ha de implementar *IFormatter* (1) lo que lo obliga a tener un método *format()* (2) que convierte el valor de la propiedad que puede ser un objeto Java cualquiera en una cadena para ser visualizada en un documento HTML; y un método *parse()* (3) que convierte la cadena recibida de un *submit* del formulario HTML en un objeto Java listo para asignar a la propiedad.

Editores para valores múltiples

Definir un editor para editar valores múltiples es parecido a hacerlo para valores simples. Veamos.
Por ejemplo, si queremos definir un estereotipo REGIONES que permita al usuario seleccionar más de una región para una propiedad. Ese estereotipo se puede usar de esta manera:

```
@Stereotype("REGIONES")
private String [] regiones;
```

Entonces podemos añadir una entrada en el archivo *tipo-estereotipo-defecto.xml* como sigue:

```
<para estereotipo="REGIONES" tipo="String []"/>
```

Y definir nuestro editor en el *editores.xml* de nuestro proyecto:

```
<editor url="editorRegiones.jsp">                                                    <!-- 1 -->
   <propiedad nombre="cantidadRegiones" valor="3"/>                                 <!-- 2 -->
   <formateador clase="org.openxava.formatters.MultipleValuesByPassFormatter"/>  <!-- 3 -->
   <para-estereotipo estereotipo="REGIONES"/>
</editor>
```

editorRegiones.jsp (1) es el archivo JSP que dibuja nuestro editor. Podemos definir propiedades que serán enviada al JSP como parámetros del *request* (2). El formateador tiene que implementar *IMultipleValuesFormatter*, que es similar a *IFormatter* pero usa *String []* en vez de *String*. En este caso usamos un formateador genérico que simplemente deja pasar el dato.
Y para terminar escribimos nuestro editor JSP:

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

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

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

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

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

Como se puede ver es como definir un editor para un valor simple, la principal diferencia es que el valor formateado (1) es un array de cadenas (*String []*) y no una cadena simple (*String*).

Editores personalizables y estereotipos para crear combos

Podemos hacer que propiedades simples que se visualicen como combos que rellenen sus datos desde la base datos. Veámoslo.
Definimos las propiedades así en nuestro componente:

```
@Stereotype("FAMILY")
private int familyNumber;

@Stereotype("SUBFAMILY")
private int subfamilyNumber;
```

Y en nuestro *editores.xml* ponemos:

```
<editor url="descriptionsEditor.jsp">                                          <!-- 10 -->
    <propiedad nombre="modelo" valor="Familia"/>                                <!--  1 -->
    <propiedad nombre="propiedadClave" valor="codigo"/>                           <!--  2 -->
    <propiedad nombre="propiedadDescripcion" valor="descripcion"/>                   <!--  3 -->
    <propiedad nombre="ordenadoPorClave" valor="true"/>                           <!--  4 -->
```

```
    <propiedad nombre="readOnlyAsLabel" valor="true"/>                          <!--  5 -->
    <para-estereotipo estereotipo="FAMILIA"/>                                  <!-- 11 -->
</editor>

<!--  Es posible especificar dependencias de estereotipos o propiedades -->
<editor url="descriptionsEditor.jsp"                                           <!-- 10 -->
    depende-de-estereotipos="FAMILIA">                                         <!-- 12 -->
<!--
<editor url="descriptionsEditor.jsp" depende-de-propiedades="codigoFamilia">       <!-- 13 -->
-->
    <propiedad nombre="modelo" valor="Subfamilia"/>                            <!--  1 -->
    <propiedad nombre="propiedadClave" valor="codigo"/>                         <!--  2 -->
    <propiedad nombre="propiedadesDescripcion" valor="codigo, descripcion"/>       <!--  3 -->
    <propiedad nombre="condicion" value="${codigoFamilia} = ?"/>                <!--  6 -->
    <propiedad nombre="estereotiposValoresParametros" valor="FAMILIA"/>            <!--  7 -->
    <!--
    <propiedad nombre="propiedadesValoresParametros" valor="codigoFamilia"/>       <!--  8 -->
    -->
    <propiedad nombre="formateadorDescripciones"                                <!--  9 -->
        valor="org.openxava.test.formatters.FormateadorDescripcionesFamilia"/>
    <para-estereotipo estereotipo="SUBFAMILIA"/>                               <!-- 11 -->
</editor>
```

Al visualizar una vista con estas dos propiedades *codigoFamilia* y *codigoSubfamilia* sacará un combo para cada una de ellas, el de familias con todas las familias disponible y el de subfamilias estará vacío y al escoger una familia se rellenará con sus subfamilias correspondientes.

Para hacer eso asignamos a los estereotipos (FAMILIA y SUBFAMILIA en este caso(11)) el editor *descriptionsEditor.jsp* (10) y lo configuramos asignandole valores a sus propiedades. Algunas propiedades con las que podemos configurar estos editores son:

1. **modelo**: Modelo del que se obtiene los datos. Puede ser el nombre de una entidad (*Factura*) o el nombre de un modelo usado en una colección incrustada (*Factura.LineaFactura*).
2. **propiedadClave** o **propiedadesClave**: Propiedad clave o lista de propiedades clave que es lo que se va a usar para asignar valor a la propiedad actual. No es obligado que sean las propiedades clave del modelo, aunque sí que suele ser así.
3. **propiedadDescripcion** o **propiedadesDescripcion**: Propiedad o lista de propiedades a visualizar en el combo.
4. **ordenadoPorClave**: Si ha de estar ordenador por clave, por defecto sale ordenado por descripción. También se puede usar order con un orden al estilo SQL, si lo necesitas.
5. **readOnlyAsLabel**: Si cuando es de solo lectura se ha de visualizar como una etiqueta. Por defecto es *false*.

6. **condicion**: Condición para restringir los datos a obtener. Tiene formato SQL, pero podemos poner nombres de propiedades con *${}*, incluso calificadas. Podemos poner argumentos con *?*. En ese caso es cuando dependemos de otras propiedades y solo se obtienen los datos cuando estas propiedades cambian.
7. **estereotiposValoresParametros**: Lista de estereotipos de cuyas propiedades dependemos. Sirven para rellenar los argumentos de la condición y deben coincidir con el atributo *depende-de-estereotipos*. (12)
8. **propiedadesValoresParametros**: Lista de propiedades de las que dependemos. Sirven para rellenar los argumentos de la condición y deben coincidir con el atributo *depende-de-propiedades*. (13)
9. **formateadorDescripciones**: Formateador para las descripciones visualizadas en el combo. Ha de implementar *IFormatter*.

Siguiendo este ejemplo podemos hacer fácilmente nuestro propios estereotipos que visualicen una propiedad simple con un combo con datos dinámicos. Sin embargo, en la mayoría de los casos es más conveniente usar referencias visualizadas como @DescriptionsList; pero siempre tenemos la opción de los estereotipos disponible.

Vistas JSP propias y taglibs de OpenXava

Obviamente la mejor forma de crear interfaces de usuario es usando las anotaciones de vista que se ven en el capítulo 4. Pero, en casos extremos quizás necesitemos definir nuestra propia vista usando JSP. OpenXava nos permite hacerlo. Y para hacer más fácil la labor podemos usar algunas taglibs JSP provistas por OpenXava. Veamos un ejemplo.

Ejemplo

Lo primero es indicar en nuestro módulo que queremos usar nuestro propio JSP, en *aplicacion.xml*:

```
<modulo nombre="ComercialJSP" carpeta="facturacion.variaciones">
   <modelo nombre="Comercial"/>
   <vista nombre="ParaJSPPropio"/>                          <!-- 1 -->
   <vista-web url="jsp-propios/comercial.jsp"/>             <!-- 2 -->
   <controlador nombre="Typical"/>
</modulo>
```

Si usamos *vista-web* (2) al definir el módulo, OpenXava usa nuestro JSP para dibujar el detalle, en vez de usar la vista generada automáticamente. Opcionalmente podemos definir una vista OpenXava con *vista* (1), esta vista es usada para saber que eventos lanzar y que propiedades llenar, si no se especifica se usa la vista por defecto de la entidad; aunque es aconsejable crear una vista OpenXava explícita para nuestra vista JSP, de esta manera podemos controlar los eventos, las propiedades a rellenar, el orden del foco, etc explicitamente. Podemos poner nuestro JSP dentro de la carpeta *web/jsp-propios* (u otra de nuestra elección) de nuestro proyecto, y este JSP puede ser así:

```
<%@ include file="../xava/imports.jsp"%>

<table>
<tr>
    <td>C&oacute;digo: </td>
    <td>
        <xava:editor property="codigo"/>
    </td>
</tr>
<tr>
    <td>Nombre: </td>
    <td>
        <xava:editor property="nombre"/>
    </td>
</tr>

<tr>
    <td>Nivel: </td>
    <td>
        <xava:editor property="nivel.id"/>
        <xava:editor property="nivel.descripcion"/>
    </td>
</tr>
</table>
```

Somos libres de crear el archivo JSP como queramos, pero puede ser práctico usar las taglibs de OpenXava, en este caso, por ejemplo, se usa *<xava:editor/>*, esto dibuja un editor apto para la propiedad indicada, además añade el JavaScript necesario para lanzar los eventos. Si usamos *<xava:editor/>*, podemos manejar la información visualizada usando el objeto *xava_view* (del tipo *org.openxava.view.View*), por lo tanto todos los controladores estándar de OpenXava (*CRUD* incluido) funcionan.

Podemos observar como las propiedades cualificadas están soportadas (como *nivel.id* o *nivel.descripcion*) *(nuevo en v2.0.1)*, además cuando el usuario rellena *nivel.id*, *nivel.descripcion* se llena con su valor correspondiente. Sí, todo el comportamiento de una vista OpenXava está disponible dentro de

nuestros JSPs si usamos las taglibs de OpenXava.
Veamos las taglib de OpenXava.

xava:**editor**

La marca (tag) *<xava:editor/>* permite visualizar un editor (un control HTML) para nuestra propiedad, de la misma forma que lo hace OpenXava cuando genera automáticamente la interfaz de usuario.

```
<xava:editor
   property="nombrePropiedad"        <!-- 1 -->
   editable="true|false"            <!-- 2  Nuevo en v2.0.1 -->
   throwPropertyChanged="true|false"  <!-- 3  Nuevo en v2.0.1 -->
/>
```

1. **property** (obligado): Propiedad del modelo asociado al módulo actual.
2. **editable** (opcional): *Nuevo en v2.0.1*. Fuerza a este editor a ser editable, de otra forma se asume un valor por defecto apropiado.
3. **throwPropertyChanged** (opcional): *Nuevo en v2.0.1*. Fuerza a este editor a lanzar el evento de propiedad cambiada, de otra forma se asume un valor por defecto apropiado.

Esta marca genera el JavaScript para permitir a nuestra vista trabajar de la misma forma que una vista automática. Las propiedades calificadas (propiedades de referencias) están soportadas *(nuevo en v2.0.1)*.

xava:**action**, **xava**:**link**, **xava**:**image**, **xava**:**button**

La marca (tag) *<xava:action/>* permite dibujar una acción (un botón o una imagen que el usuario puede pulsar).

```
<xava:action action="controlador.accion" argv="argv"/>
```

El atributo action indica la acción a ejecutar, y el atributo *argv* (opcional) nos permite establecer valores a algunas propiedades de la acción antes de ejecutarla. Un ejemplo:

```
<xava:action action="CRUD.save" argv="resetAfter=true"/>
```

Cuando el usuario pulse en la acción se ejecutará *CRUD.save*, antes pone a true la propiedad *resetAfter* de la acción.
La acción se visualiza como una imagen si tiene una imagen asociada y como un botón si no tiene imagen asociada. Si queremos detereminar el estilo de visualización podemos usar directamente las siguientes marcas:

<xava:button/>, *<xava:image/>* o *<xava:link/>* similares a *<xava:action/>*.
Podemos especificar una cadena vacía para la acción *(nuevo en v2.2.1)*, como sigue:

```
<xava:action action=""/>
```

En este caso la marca (tag) no tiene efecto y no se produce error. Esta característica puede ser útil cuando el nombre de la acción lo obtenemos dinámicamente (es decir *action="<%=micodigo()%>"*), y el valor pueda estar vacío en ciertos casos.

xava:**message** (**nuevo en v2**.**0**.**3**)

La marca (tag) <xava:message/> permite mostrar en HTML un mensaje de los archivos de recursos i18n de OpenXava.

```
<xava:message key="clave_mensaje" param="parametroMensaje" intParam="paramMensaje"/>
```

El mensaje es buscado primero en los archivos de recursos de nuestro proyecto (*MiProyecto/i18n/MensajesMiProyecto.properties*) y si no se encuentra ahí es buscado en los mensajes por defecto de OpenXava (*OpenXava/i18n/Messages.properties*).
Los atributos *param* y *intParam* son opcionales. El atributo *intParam* es usado cuando el valor a enviar como parametro es de tipo int. Si usamos Java 5 podemos usar siempre param porque int es automáticamente convertido por autoboxing.
Esta marca solo genera el texto del mensaje, sin ningun tipo de formateo HTML.
Un ejemplo:

```
<xava:message key="cantidad_lista" intParam="<%=cantidadTotal%>"/>
```

xava:**descriptionsList** (**nuevo en v2**.**0**.**3**)

La marca (tab) *<xava:descriptionsList/>* permite visualizar una lista descripciones (un combo HTML) para una referencia, del mismo modo que lo hace OpenXava cuando genera la interfaz de usuario automáticamente.

```
<xava:descriptionsList
    reference="nombreReferencia"  <!-- 1 -->
/>
```

1. **reference** (obligado): Una referencia del modelo asociado con el módulo actual.

Esta marca genera el JavaScript necesario para permitir a la vista personalizada trabajar de la misma forma que una automática.
Un ejemplo:

```
<tr>
    <td>Nivel: </td>
    <td>
        <xava:descriptionsList reference="nivel"/>
    </td>
</tr>
```

En este caso *nivel* es una referencia al modelo actual (por ejemplo *Comercial*). Un combo es mostrado con todos los niveles disponibles.