

< o p e n - s o u r c e >

OpenXava

< j2ee - development >

V E R S I Ó N 2 . 0 . 1

G U I A D E R E F E R E N C I A

Índice de contenido

0 Prologo.....	5
0.1 ¿Qué hay nuevo en OpenXava 2.0.1?.....	5
0.2 ¿Qué hay nuevo en OpenXava 2.0?.....	5
1 Visión general.....	7
1.1 Presentación.....	7
1.2 Componente de negocio.....	7
1.3 Controladores.....	7
1.4 Aplicación.....	8
1.5 Estructura de un proyecto.....	8
1.6 Conclusión.....	9
2 Mi primer proyecto OpenXava.....	10
2.1 Crear un proyecto nuevo.....	10
2.2 Configurar base de datos.....	10
2.3 Nuestro primer componente.....	11
2.4 La aplicación.....	12
2.5 La tabla.....	13
2.6 Ejecutar nuestra aplicación.....	14
2.7 Automatizando las pruebas.....	14
2.8 Las etiquetas.....	17
2.9 Conclusión.....	17
3 Modelo.....	18
3.1 Implementación en Java.....	18
3.2 Componente de negocio.....	18
3.3 Entidad y agregados.....	19
3.4 Entidad.....	19
3.5 Bean (1).....	21
3.6 EJB (2).....	23
3.7 Implementa (3).....	25
3.8 Propiedad (4).....	27
3.8.1 Estereotipo.....	28
3.8.2 Estereotipo GALERIA_IMAGENES (nuevo en v2.0).....	29
3.8.3 Valores posibles.....	30
3.8.4 Calculador.....	31
3.8.5 Calculador valor defecto.....	36
3.8.6 Validador.....	37
3.9 Referencia (5).....	39
3.9.1 Calculador valor por defecto para una referencia.....	40
3.10 Colección (6).....	41
3.11 Método (7).....	46
3.12 Buscador (8).....	49
3.13 Calculador poscrear (9).....	51
3.14 Calculador posmodificar (11).....	52
3.15 Calculadores poscargar y preborrar (10, 12).....	53
3.16 Validador (13).....	53
3.17 Validador borrar (14).....	56

3.18 Agregado.....	57
3.18.1 Referencia a agregado.....	58
3.18.2 Colección de agregados.....	58
4 Vista.....	61
4.1 Disposición.....	62
4.2 Vista propiedad.....	67
4.2.1 Formato de etiqueta.....	67
4.2.2 Evento de cambio de valor.....	68
4.2.3 Acciones de la propiedad.....	68
4.3 Vista referencia.....	69
4.3.1 Escoger vista.....	71
4.3.2 Personalizar el enmarcado.....	71
4.3.3 Acción de búsqueda propia.....	72
4.3.4 Acción de creación propia.....	73
4.3.5 Lista descripciones (combos).....	74
4.4 Vista colección.....	76
4.4.1 Acción de editar/ver detalle propia.....	78
4.4.2 Acciones de lista propias.....	79
4.4.3 Acciones de detalle propias.....	80
4.5 Propiedad de vista.....	82
4.6 Configuración de editores.....	83
4.7 Editores para valores múltiples.....	86
4.8 Editores personalizables y estereotipos para crear combos.....	88
4.9 Vista sin modelo asociado.....	90
5 Datos tabulares.....	93
5.1 Propiedades iniciales y resaltar filas.....	94
5.2 Filtros y condición base.....	95
5.3 Select íntegro.....	98
5.4 Orden por defecto.....	98
6 Mapeo objeto/relacional.....	99
6.1 Mapeo de entidad.....	99
6.2 Mapeo propiedad.....	100
6.3 Mapeo referencia.....	102
6.4 Mapeo propiedad multiple.....	104
6.5 Mapeo de referencias a agregados.....	107
6.6 Mapeo de agregados usados en colecciones.....	108
6.7 Conversores por defecto.....	110
6.8 Filosofía del mapeo objeto-relacional.....	111
7 Controladores.....	113
7.1 Variables de entorno y objetos de sesión.....	113
7.2 El controlador y sus acciones.....	114
7.3 Herencia de controladores.....	118
7.4 Acciones en modo lista.....	118
7.5 Sobrecribir búsqueda por defecto.....	121
7.6 Inicializando un módulo con una acción.....	123
7.7 Llamar a otro módulo.....	124
7.8 Cambiar el modelo de la vista actual.....	127

7.9 Ir a una página JSP.....	128
7.10 Generar un informe propio con JasperReports.....	128
7.11 Cargar y procesar un fichero desde el cliente (formulario multipart)	130
7.12 Todos los tipos de acciones.....	133
8 Aplicación.....	135
8.1 Un módulo típico.....	136
8.2 Módulo con solo detalle.....	137
8.3 Módulo con solo lista.....	137
8.4 Módulo de documentación.....	137
9 Aspectos.....	139
9.1 Introducción a AOP.....	139
9.2 Definición de aspectos.....	139
9.3 AccessTracking: Una aplicación práctica de los aspectos.....	141
9.3.1 La definición del aspecto.....	141
9.3.2 Configurar AccessTracking.....	142
10 Miscelánea.....	144
10.1 Relaciones de muchos-a-muchos.....	144
10.2 Programar con Hibernate.....	146
10.3 Vistas JSP propias y taglibs de OpenXava.....	146
10.3.1 Ejemplo.....	147
10.3.2 xava:editor.....	148
10.3.3 xava:action, xava:link, xava:image, xava:button.....	148

Esta guía pretende ser una referencia exhaustiva de OpenXava para el desarrollador de aplicaciones. Se centra especialmente en la sintaxis de los archivos XML usados en OpenXava. Está llena de ejemplos de XML y código Java que se puede escribir.

Este documento no pretende ser una introducción a OpenXava sino una guía completa para el desarrollador, aunque los dos primeros capítulos tienen cierto carácter introductorio. Como introducción aconsejamos seguir el tutorial (que podemos encontrar en el sitio web de OpenXava o en la propia distribución de OpenXava). Por otra parte la referencia definitiva es el proyecto OpenXavaTest que contiene una muestra de todas las posibilidades que ofrece OpenXava.

Esta guía supone que el desarrollador usa Eclipse como IDE, aunque OpenXava no usa recursos de Eclipse y puede ser usado sin problemas con otro IDE e incluso con un editor y línea de órdenes. También se supone que el Eclipse apunta al *workspace* que tenemos en el directorio raíz de OpenXava. Si uno sigue las instrucciones del tutorial debe tenerlo todo configurado correctamente. Aunque es una guía de referencia no es mala idea leerla secuencialmente al menos una vez, para poder comprender las posibilidades de OpenXava.

En esta guía no se incluye una definición de las APIs de OpenXava ni tampoco trata temas de configuración o aspectos filosóficos. Se puede encontrar información de estos asuntos en <http://www.openxava.org>.

Todas las sugerencias y críticas son bienvenidas. Las podéis enviar a javierpaniza@gestion400.com.

0.1 ¿Qué hay nuevo en OpenXava 2.0.1?

- `SequenceCalculator` para generación automática de identificadores usando *sequence* de la base de datos: Sección 3.8.5
- Acciones arbitrarias disponibles en `<reference-view/>`: Sección 4.3
- Soporte de atajos de teclado (keystrokes) para las acciones: Sección 7.2
- `<xava:editor/>` soporta propiedades calificadas: Sección 10.3

0.2 ¿Qué hay nuevo en OpenXava 2.0?

- Capa del model ahora es generado en formato POJO (Plain Old Java Object, Clases de Java Convencionales).
- La persistencia es manejada por Hibernate, esto permite desplegar las aplicaciones OpenXava en un simple Tomcat (or cualquier otro servidor de servlets que deseemos).
- Estereotype GALERIA_IMAGENES: Sección 3.8.2, página 28
- Añadido `IModelCalculator`, como alternativa mejor a `IEntityCalculator`: Sección 3.8.4, página 32
- Las secciones anidadas están soportadas en las vistas: Sección 4.1, page 64
- Es posible acceder a variables de entorno desde un filtro: Sección 5.2, página 95
- `tipo-cmp` disponible para usar en mapeo de referencias: Sección 6.3, page 102

- Es posible ocultar y mostrar acciones en la Interfaz de Usuario: Sección 7.12, página 131
- Nuevo capítulo 10 Miscelanea: con explicaciones sobre relaciones muchos-a-muchos, uso de Hibernate dentro de OpenXava, vista personalizadas con JSP y la taglibs de OpenXava.

1.1 Presentación

OpenXava es un marco de trabajo para desarrollar aplicaciones J2EE rápida y fácilmente.

OpenXava es acrónimo de **Open** source, **XML** y **Java**. Y la filosofía subyacente es definir con XML y programar con Java, pero cuanto más definimos (es decir cuanto más X,Ñ) y menos programamos (menos Java) 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 específicas.

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

1.2 Componente de negocio

Las piezas fundamentales para crear una aplicación OpenXava son los componentes, en el contexto de OpenXava un componente de negocio es un archivo 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.xml*. 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.

1.3 Controladores

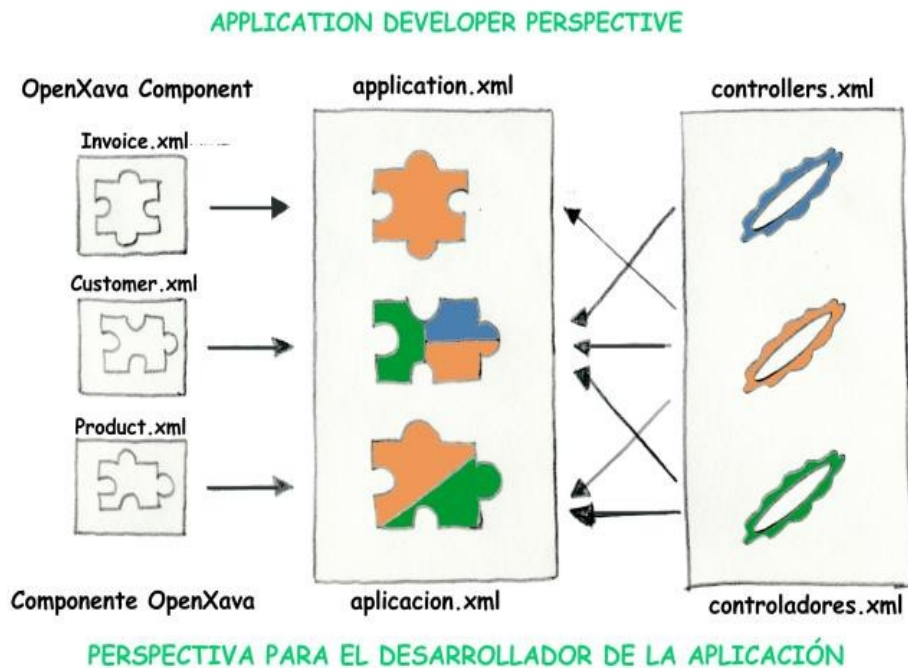
Los componentes de negocio no definen lo que un usuario puede hacer con la aplicación; esto se define con los controladores. Los controladores están en el archivo *xava/controladores.xml* de cada proyecto; además OpenXava tiene un conjunto de controladores predefinidos en *OpenXava/xava/controllers.xml*.

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.

1.4 Aplicación

Una aplicación OpenXava es un conjunto de módulos. Un módulo une un componente de negocio con uno o más controladores. Visualmente sería:



Cada módulo de la aplicación es lo que al final utiliza el usuario, y generalmente se configura como un portlet dentro de un portal.

1.5 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) y los archivos de configuración para éste (normalmente uno por cliente).
- `src[carpeta fuente]`: contiene el código fuente Java escrito por nosotros.
- `components`: Los archivos XML con las definiciones de nuestros componentes de negocio.
- `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.
- `gen-src[carpeta fuente]`: Código generado por XDoclet. Solo necesario si usamos EJB.
- `gen-src-xava[carpeta fuente]`: Código generado por OpenXava.
- `properties[carpeta fuente]`: Archivos de propiedades para configurar nuestra aplicación.
- `build`: Archivos XML necesarios en una aplicación J2EE, algunos son generados y otros pueden

ser editados manualmente.

- `filtered-files[carpeta fuente]`: Es para uso interno de las tareas de construcción. Tiene que se una carpeta de código fuente. *(nuevo en v2.0)*
- `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.

1.6 Conclusión

Este capítulo nos introduce un poco en algunos conceptos de OpenXava, el resto de la guía ampliará toda esta información, pero desde un punto de vista técnico.

2.1 Crear un proyecto nuevo

Una vez abierto el Eclipse y apuntando al *workspace* que viene en la distribución de OpenXava (*openxava.zip*). Para crear un nuevo proyecto hemos de editar el archivo *CrearNuevoProyecto.xml* en el proyecto *OpenXavaPlantilla* de esta forma:

```
<property name="proyecto" value="Gestion" />
<property name="paquete" value="gestion" />
<property name="componente" value="Almacen" />
<property name="modulo" value="Almacenes" />
<property name="fuentedatos" value="MiGestionDS" />
```

Ahora hemos de ejecutar *CrearNuevoProyecto.xml* usando Ant. Podemos hacerlo con *Botón Derecho en CrearNuevoProyecto.xml > Run as > Ant Build*

Usando el asistente de Eclipse apropiado hemos de crear un nuevo proyecto Java llamado *Gestion*.

Con esto ya tenemos nuestro proyecto listo para empezar a trabajar, pero antes de nada tenemos que tener una base de datos configurada.

2.2 Configurar base de datos

OpenXava genera una aplicación J2EE pensada para ser desplegada en un servidor de aplicaciones J2EE (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 de 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 Tomcat e Hypersonic.

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/MiGestionDS" auth="Container" type="javax.sql.DataSource"
    maxActive="20" maxIdle="5" maxWait="10000"
    username="sa" password="" driverClassName="org.hsqldb.jdbcDriver"
    url="jdbc:hsqldb:file:../data/migestion-db"/>
```

Lo importante aquí es el nombre JNDI, que es a lo único que se hace referencia desde OpenXava, en este caso *MiGestionDS*, también hemos de escoger un nombre de base de datos, aquí *migestion-db*, que referencia al archivo físico en donde están los datos (en realidad un script SQL).

2.3 Nuestro primer componente

Crear un componente OpenXava es sencillo, la definición de cada componente está en un archivo XML de sintaxis sencilla. Para empezar podemos editar el archivo *Almacen.xml*, que ya tenemos creado al crear el proyecto. Lo editamos para dejarlo como sigue:

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

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

<componente nombre="Almacen">

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

    <mapeo-entidad tabla="GESTION@separator@ALMACENES">
        <mapeo-propiedad
            propiedad-modelo="codigoZona" columna-tabla="ZONA"/>
        <mapeo-propiedad
            propiedad-modelo="codigo" columna-tabla="CODIGO"/>
        <mapeo-propiedad
            propiedad-modelo="nombre" columna-tabla="NOMBRE"/>
    </mapeo-entidad>

</componente>
```

En esta definición observamos 2 partes diferenciadas, la primera es *entidad*, *entidad* sirve para definir el modelo principal para este componente, la información que ponemos aquí se usa para crear las clases Java y demás recursos para trabajar con el concepto de Almacén. El código generado puede usar POJO (clases de java convencionales) + Hibernate (*nuevo en v2.0*) o tecnología EJB (con EntityBeans CMP2). En esta parte no solo se define la estructura de los datos sino también la lógica de negocio asociada.

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

```
<propiedad
    nombre="codigoZona"          (1)
    clave="true"                 (2)
    longitud="3"                 (3)
```

```
requerido="true"           (4)
tipo="int"                 (5)

/>
```

Este es su significado:

- (1)**nombre**: Es el nombre de la propiedad en el código Java generado y también sirve como identificador de esta propiedad dentro de los archivos OpenXava.
- (2)**clave**: 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.
- (3)**longitud**: Longitud de los datos visualizados. Es opcional, pero suele ser útil para hacer mejores interfaces gráficos.
- (4)**requerido**: Indica si hay que validar la existencia de información en esta propiedad antes de crear o modificar.
- (5)**tipo**: Es el tipo de la propiedad. Todo tipo válido para una propiedad Java se puede poner, lo que incluye tipos integrados, clases del JDK, clases propias, etc.

Las posibilidades de `propiedad` van mucho más allá de lo que aquí se muestra, se puede ver una explicación más completa en el capítulo 3.

Por otra parte tenemos el mapeo, en donde relacionamos nuestro componente con una tabla de la base de datos, la sintaxis es obvia. El uso de `@separator@` en el nombre de tabla nos permite que la misma aplicación funcione con bases de datos que soporten colecciones o esquemas y las que no, simplemente tenemos que dar en nuestro *build.xml* el valor '.' o '_' a `separator`.

Ahora estamos listos para generar código, para eso hemos de ejecutar la tarea ant *generarCodigo*. La forma más práctica de ejecutar una tarea ant en Eclipse es dándola de alta en *Externals Tools*, de esta forma cuando queremos volver a ejecutarla lo podemos hacer simplemente escogiéndola del menú. Es muy importante configurar la tarea para que **refresque** el proyecto *Gestion* **después de ejecutarse**. El *build.xml* que tenemos en nuestro proyecto tiene ya configuradas las tareas ant más usadas.

Después de generar el código podemos ejecutar un *Build*, y comprobar que no hay errores.

Con un componente hecho ya podemos definir una aplicación OpenXava.

2.4 La aplicación

En OpenXava se entiende que una aplicación es el producto final que el usuario va a utilizar. Una aplicación está compuesta por un conjunto de módulos. Un módulo es la unión entre un componente (sobre qué se quiere trabajar) y un conjunto de controladores (qué es lo que se quiere usar). Como era de esperar una aplicación se define con un archivo XML, el archivo *xava/aplicacion.xml*. Si hemos hecho bien el paso de crear el proyecto, ya deberíamos tener a punto el archivo, vamos a revisarlo de todas formas:

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

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

```

<aplicacion nombre="Gestion">

    <modulo nombre="Almacenes">
        <modelo nombre="Almacen"/>
        <controlador nombre="Typical"/>
    </modulo>

</aplicacion>

```

En este caso tenemos definido un solo módulo `Almacenes`, este nombre de módulo es el que se usará en la URL de nuestro navegador para ejecutarlo, o el nombre del portlet si desplegamos la aplicación en un portal. Definimos como modelo `Almacen`, el componente que hemos definido antes, y como controlador `Typical`, este controlador predefinido nos permite hacer un mantenimiento (altas, bajas, modificaciones y consultas), además de generar un informe en PDF y exportar a Excel. Desde un punto de vista visual se puede decir que con `controlador` definimos los botones que van a aparecer y con `modelo` los datos; aunque esta visión es excesivamente simple.

2.5 La tabla

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

Hemos de crear el archivo *migestion-db.script* dentro de la carpeta *data* del Tomcat, y poner:

```

CREATE SCHEMA PUBLIC AUTHORIZATION DBA

CREATE MEMORY TABLE GESTION_ALMACENES(ZONA INTEGER,CODIGO INTEGER,NOMBRE VARCHAR(40),
PRIMARY KEY(ZONA,CODIGO))

CREATE USER SA PASSWORD ""

GRANT DBA TO SA

SET WRITE_DELAY 20

```

También tenemos que crear el archivo *migestion-db.properties* dentro de la carpeta *data* del Tomcat, con este contenido:

```

#HSQL Database Engine
hsqldb.script_format=0
runtime.gc_interval=0
sql.enforce_strict_size=false
hsqldb.cache_size_scale=8
readonly=false
hsqldb.nio_data_file=true
hsqldb.cache_scale=14
version=1.8.0
hsqldb.default_table_type=memory
hsqldb.cache_file_scale=1
hsqldb.log_size=200
modified=yes

```

```
hsqldb.cache_version=1.7.0
hsqldb.original_version=1.8.0
hsqldb.compatible_version=1.8.0
```

Arrancamos el Tomcat y ya está todo listo.

2.6 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=Almacenes>

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 *deplegarPortlets*.
- Abrimos un navegador de internet y vamos a la dirección <http://localhost:8080/>

Ahora podemos entrar como 'admin' y probar el módulo.

2.7 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 `PruebaAlmacenes`, y pegaremos en ella el siguiente código:

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

import org.openxava.tests.*;

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

```

public class PruebaAlmacenes extends ModuleTestBase {

    public PruebaAlmacenes(String testName) {
        super(testName, "Gestion", "Almacenes");    // (1)
    }

    public void testCrearLeerModificarBorrar() throws Exception {
        // Creamos
        execute("CRUD.new");    // (2)
        setValue("codigoZona", "1");    // (3)
        setValue("codigo", "7");
        setValue("nombre", "Almacen JUNIT");
        execute("CRUD.save");
        assertNoErrors();    // (4)
        assertValue("codigoZona", "");    // (5)
        assertValue("codigo", "");
        assertValue("nombre", "");

        // Leeemos
        setValue("codigoZona", "1");
        setValue("codigo", "7");
        execute("CRUD.search");
        assertValue("codigoZona", "1");
        assertValue("codigo", "7");
        assertValue("nombre", "Almacen JUNIT");

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

        // Comprobamos modificado
        setValue("codigoZona", "1");
        setValue("codigo", "7");
        execute("CRUD.search");
        assertValue("codigoZona", "1");
        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/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 `deployPortlets`). Solo necesitamos editar el archivo `properties/xava-junit.properties` y escribir:

```

jetspeed2.url=openxava
#jetspeed2.username=demo
#jetspeed2.password=demo

```

El `username` y `password` son opcionales, si no los especificamos la prueba no hace un 'login' en el portal (esto es útil si asignamos nuestro módulo al usuario 'guest').

2.8 Las etiquetas

Ya nos funciona, pero hay un pequeño detalle que se ha quedado suelto. Las etiquetas que aparecen al usuario no son apropiadas (p. ej. `codigoZona`). Podemos asignar una etiqueta a cada propiedad con el atributo `etiqueta`, pero no es una buena solución. Lo ideal es escribir un archivo con todas las etiquetas, y así podemos traducir nuestro producto a otro idioma con facilidad.

Para definir las etiqueta solo tenemos que crear un archivo llamado *EtiquetasGestion_es.properties* en la carpeta *i18n*. Editar ese archivo y añadir:

```
Almacen=Almacén  
codigoZona=Código zona
```

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, 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  
codigoZona=Zone number
```

Para saber más sobre como definir las etiquetas de nuestros elementos OpenXava podemos echar un vistazo a los archivos de *OpenXavaTest/xava/i18n*.

2.9 Conclusión

En este capítulo hemos visto como crear un nuevo proyecto, y hemos echado un vistazo inicial a algunas características de OpenXava. Por supuesto OpenXava ofrece mucha más posibilidades, y en el resto del libro se van a ir viendo más minuciosamente.

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

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

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

3.1 Implementación en Java

Actualmente OpenXava genera código para las siguiente 3 alternativas:

1. Clases de Java convencionales (los famosos POJOs) para el modelo usando Hibernate para la persistencia.
2. Los clásicos EntityBeans de EJB2 para el modelo y la persistencia.
3. POJOs + Hibernate dentro de un contenedor EJB.

La opción 1 es la opción por defecto y la más adecuada para la mayoría de los casos. La opción 2 es para poder soportar las aplicaciones OpenXava escritas hasta ahora con EJB. La opción 3 puede ser útil en ciertas circunstancias. Podemos ver como configurar esto en *OpenXavaTest/properties/xava.properties*.

3.2 Componente de negocio

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

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

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

<componente nombre="NombreComponente">

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

```

```

<!-- Vista -->
<vista>...</vista>
<vista nombre="...">...</vista>
<vista nombre="...">...</vista>
...

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

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

</component>

```

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

3.3 Entidad y agregados

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

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

3.4 Entidad

La sintaxis de la entidad es como sigue:

```

<entidad>
    <bean ... />                                ( 1 )
    <ejb ... />                                  ( 2 )

```

```

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

```

- (1)bean (uno, opcional): Permite usar un JavaBean existente (una simple clase Java, uno de esos famosos POJOs). Esto aplica si usamos Hibernate como gestor de persistencia. En este caso la generación de código para POJO y mapeo de Hibernate para este componente no se produce.
- (2)ejb (uno, opcional): Permite usar un EJB existente. Esto solo aplica si usamos EJB CMP2 para gestionar la persistencia. En este caso la generación de código EJB para este componente no se produce.
- (3)implementa (varios, opcional): Para que el código generado implemente una o varias interfaces Java cualquiera.
- (4)propiedad (varias, opcional): Las propiedades representan propiedades Java (con su *setter* y *getter*) en el código generado.
- (5)referencia (varias, opcional): Referencias a otros modelos, podemos referenciar a la entidad de otro componente o a un agregado del nuestro.
- (6)coleccion (varias, opcional): Colecciones de referencias, en el código generado se convierten en una propiedad que devuelve un `java.util.Collection`.
- (7)metodo (varios, opcional): Para crear un método en el código generado, en este caso la lógica del método estará contenida en un calculador (`Icalculator`).
- (8)buscador (varios, opcional): Usado para crear métodos de búsqueda, en el caso de EJB genera un *finder*.
- (9)calculador-poscrear (varios, opcional): Lógica a ejecutar después de hacer el objeto persistente. En Hibernate en el evento `PreInsertEvent`, en EJB2 en el método `ejbPostCreate`.
- (10)calculador-poscargar (varios, opcional): Lógica a ejecutar justo después de cargar el estado del objeto desde el almacenamiento persistente. En Hibernate en el evento `PostLoadEvent`, en EJB2 en el método `ejbLoad`.
- (11)calculador-posmodificar (varios, opcional): Lógica a ejecutar después de modificar el objeto persistente y antes de grabarlo en el almacenamiento persistente. En Hibernate en el evento `PreUpdateEvent`, en EJB2 en el método `ejbStore`.
- (12)calculador-preborrar (varios, opcional): Lógica a ejecutar justo antes de borrar el objeto

persistente. En Hibernate en el evento `PreDeleteEvent`, en EJB2 en el método `ejbRemove`.

- (13) `validador` (varios, opcional): Ejecuta una validación a nivel de modelo. Este validador puede recibir el valor de varias propiedades del modelo. Para validar una sola propiedad es preferible poner el validador a nivel de propiedad.
- (14) `validador-borrar` (varios, opcional): Se ejecuta antes de borrar, y tiene la posibilidad de vetar el borrado del objeto.

3.5 Bean (1)

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

Por ejemplo:

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

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

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

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

import java.io.*;

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

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

    public String getOid() {
        return oid;
    }

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

    public int getCodigo() {
        return codigo;
    }

}
```

```

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

    public String getDescripcion() {
        return descripcion;
    }

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

```

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

Adicionalmente tenemos que definir el mapeo usando Hibernate:

```

<?xml version="1.0"?>

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

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

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

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

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

    </class>

</hibernate-mapping>

```

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

...

```

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

```

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

3.6 EJB (2)

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

Por ejemplo:

```

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

```

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

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

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

```

package org.openxava.test.ejb.xejb;

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

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

/**
 * @ejb:bean name="Familia" type="CMP" view-type="remote"
 *     jndi-name="OpenXavaTest/ejb/openxava.test/Familia"

```

```

* @ejb:interface extends="org.openxava.ejbx.EJBReplicable"
* @ejb:data-object extends="java.lang.Object"
* @ejb:home extends="javax.ejb.EJBHome"
* @ejb:pk extends="java.lang.Object"
*
* @jboss:table-name "XAVATEST@separator@FAMILIA"
*
* @author Javier Paniza
*/
abstract public class FamiliaBean
    extends org.openxava.ejbx.EJBReplicableBase // (1)
    implements javax.ejb.EntityBean {

    private UUIDCalculator calculadorOid = new UUIDCalculator();

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

    /**
     * @ejb:interface-method
     * @ejb:persistent-field
     *
     * @jboss:column-name "CODIGO"
     */
    public abstract int getCodigo();
    /**
     * @ejb:interface-method
     */
    public abstract void setCodigo(int newCodigo);

    /**
     * @ejb:interface-method
     * @ejb:persistent-field
     *
     * @jboss:column-name "DESCRIPCION"

```



```

    */
    public abstract String getDescripcion();
    /**
     * @ejb:interface-method
     */
    public abstract void setDescripcion(String newDescripcion);

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

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

```

Al escribir nuestro EJB tenemos que observar dos pequeñas restricciones:

- (1) La clase ha de descender de `org.openxava.ejbx.EJBReplicableBase`
- (2) Ha de haber al menos un `ejbCreate` (con su `ejbPostCreate`) que reciba como argumento un mapa y asigne sus valores al bean como en este ejemplo.

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

3.7 Implementa (3)

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

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

Y podemos hacer nuestra interfaz Java:

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

import java.rmi.*;

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

    String getNombre() throws RemoteException;

}
```

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

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

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

In the POJO generated code you can see:

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

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

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

y la clase del bean también se ve afectada

```
abstract public class ClienteBean extends EJBReplicableBase
    implements
```

```
org.openxava.test.modelo.ICliente,  
EntityBean
```

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

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

3.8 Propiedad (4)

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

La sintaxis para definir una propiedad es:

```
<propiedad  
    nombre="nombrePropiedad"           (1)  
    etiqueta="etiqueta"                (2)  
    tipo="tipo"                        (3)  
    estereotipo="ESTEREOTIPO"          (4)  
    longitud="longitud"                 (5)  
    requerido="true|false"             (6)  
    clave="true|false"                  (7)  
    oculta="true|false"                 (8)  
>  
    <valores-posibles .../>             (9)  
    <calculador .../>                   (10)  
    <calculador-valor-defecto .../>     (11)  
    <validador .../> ....                (12)  
</propiedad>
```

- (1)nombre (obligado): Es el nombre que tendrá la propiedad en Java, por lo tanto ha de seguir la normativa para nombres de propiedad Java, entre ellas empezar por minúscula. Se desaconseja el uso de subrayados (_).
- (2)etiqueta (opcional): Etiqueta que se mostrará al usuario final. Es **mucho mejor** usar los archivos *i18n*.
- (3)tipo (opcional): Corresponde a un tipo Java. Todos los tipos válidos para una propiedad Java son válidos aquí, esto incluye clases definidas por nosotros mismos. Solo hemos de proveer un conversor para grabar en la base de datos y un editor para visualizar en HTML; así que cosas como una `java.sql.Connection` o así puede que sean complicadas de tratar, pero no imposible. Es opcional, pero solo si hemos especificado `<bean/>` o `<ejb/>` antes o asignamos un estereotipo con un tipo asociado.

- (4)`estereotipo` (opcional): Permite especificar un comportamiento especial para ciertas propiedades.
- (5)`longitud` (opcional): Longitud en caracteres de la propiedad. Especialmente útil a la hora de generar interfaces gráficas. Si no especificamos longitud asume un valor por defecto asociado a tipo o estereotipo que se obtiene de *default-size.xml* o *longitud-defecto.xml*.
- (6)`requerido` (opcional): Indica si esa propiedad es requerida. Por defecto es `true` para las propiedades clave sin calculador valor por defecto al crear y `false` en todos los demás casos. Al grabar OpenXava comprobará si las propiedades requeridas están presentes, si no lo están no se producirá la grabación y se devolverá una lista de errores de validación. La lógica para determinar si una propiedad está presente o no se puede configurar creando un archivo *validators.xml* o *validadores.xml* en nuestro proyecto. Podemos ver la sintaxis en *OpenXava/xava/validators.xml*.
- (7)`clave` (opcional): Para indicar si una propiedad forma parte de la clave. Al menos una propiedad (o referencia) ha de ser clave. La combinación de propiedades (y referencias) clave se debe mapear a un conjunto de campos en la base de datos que no tengan valores repetidos, típicamente con la clave primaria.
- (8)`oculta` (opcional): Una propiedad oculta es aquella que tiene sentido para el desarrollador pero no para el usuario. Las propiedades ocultas se excluyen cuando se generan interfaces gráficas automáticas, sin embargo a nivel de código generado están presentes y son totalmente funcionales, incluso si se les hace alusión explícita podrían aparecer en una interfaz gráfica.
- (9)`valores-posibles` (uno, opcional): Para indicar que la propiedad en cuestión solo puede tener un conjunto de valores fijos.
- (10)`calculador` (uno, opcional): Para implementar la lógica de una propiedad calculada. Una propiedad calculada solo tiene *getter* y no se almacena en la base de datos.
- (11)`calculador-valor-defecto` (uno, opcional): Para implementar la lógica para calcular el valor inicial de la propiedad. Una propiedad con `calculador-valor-defecto` sí tiene *setter* y es persistente.
- (12)`validador` (varios, opcional): Indica la lógica de validación a ejecutar sobre el valor a asignar a esta propiedad antes de crear o modificar.

3.8.1 Estereotipo

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

El OpenXava viene configurado con los siguientes estereotipos:

- `DINERO`, `MONEY`
- `FOTO`, `PHOTO`, `IMAGEN`, `IMAGE`
- `TEXTO_GRANDE`, `MEMO`, `TEXT_AREA`

- ETIQUETA, LABEL
- ETIQUETA_NEGRITA, BOLD_LABEL
- HORA, TIME
- FECHAHORA, DATETIME
- GALERIA_IMAGENES, IMAGES_GALLERY (instrucciones en 3.8.2) *nuevo en v2.0*

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

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

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

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

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

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

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

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

Y así si no ponemos longitud asumirá 40 por defecto.

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

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

Ahora podemos definir propiedades con estereotipo `NOMBRE_PERSONA`:

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

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

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

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

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

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

Y ya está todo.

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

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

```
CREATE TABLE IMAGENES (  
    ID VARCHAR(32) NOT NULL PRIMARY KEY,  
    GALLERY VARCHAR(32) NOT NULL,  
    IMAGE BLOB);  
  
CREATE INDEX IMAGENES01  
    ON IMAGENES (GALLERY);
```

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

El nombre de la tabla puede ser cualquiera que queramos. Especificamos el nombre de la tabla en el archivo de configuración (un archivo `.properties` en la raíz de nuestro proyecto OpenXava). De esta forma:

```
images.table=IMAGENES
```

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

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

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

3.8.3 Valores posibles

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

Es fácil de usar, veamos un ejemplo:

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

```

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

```

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

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

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

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

3.8.4 Calculador

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

Así se define una propiedad calculada:

```

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

```

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

Puede ser instructivo ver el código del calculador:

```

package org.openxava.test.calculadores;

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

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

```

```

private BigDecimal euros;

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

public BigDecimal getEuros() {
    return euros;
}

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

```

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

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

```

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

```

Y `result` contendría 332.772.

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

```

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

```

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

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

```

<propiedad nombre="nombreCompleto" tipo="String">
    <calculador clase="org.openxava.calculators.ConcatCalculator">
        <poner propiedad="string1" desde="id"/>
    </calculador>
</propiedad>

```



```
<poner propiedad="separator" valor=" - "/>
<poner propiedad="string2" desde="name"/>
</calculador>
</propiedad>
```

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

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

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

Y el calculador:

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

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

import javax.rmi.*;

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

/**
 * @author Javier Paniza
 */

public class CalculadorSumaImportes implements IModelCalculator { // (1)

    private IFactura factura;

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

```

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

```

Este calculador implementa `IModelCalculator` (1) (*nuevo en v2.0*) y por ello tiene un método `setModel` (2), este método se llama antes de llamar al método `calculate()` y así desde el método `calculate()` podemos acceder al modelo (en este caso a la factura) que contiene la propiedad o método. A pesar de su nombre también puede recibir un objeto que esté actuando como agregado.

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

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

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

```

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

```

Y la clase del calculador:

```

package org.openxava.test.calculadores;

import java.sql.*;

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

/**

```

```

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

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

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

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

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

    public int getAño() {
        return año;
    }

    public int getNumero() {
        return numero;
    }

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

```

```

    }

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

```

Para usar JDBC tenemos que implementar `IJBDBCalculator` (1) y entonces recibiremos un `IConnectionProvider` (2) que podremos usar dentro de `calculate()`. Ya sé que el código JDBC es feo y engorroso, pero en ocasiones puede ayudar a resolver algún problema de rendimiento.

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

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

3.8.5 Calculador valor defecto

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

```

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

```

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

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

```

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

```

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

Un uso típico de `al-crear="true"` es para generar identificadores automáticamente. En el ejemplo anterior, un identificador único de tipo `String` y 32 caracteres es generado. Podemos usar otras

técnicas de generación, por ejemplo, una *sequence* de base de datos se puede definir así:

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

SequenceCalculator (nuevo en v2.0.1) no funciona con EJB2. Funciona con Hibernate y EJB3.

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

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

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

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

3.8.6 Validador

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

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

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

El código del validador es:

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

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

/**
 * @author Javier Paniza
```

```

*/

public class ValidadorExcluirCadena implements IPropertyValidator { // (1)

    private String cadena;

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

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

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

}

```

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

- (2)`Messages errores`: Un objeto de tipo `Messages` que representa un conjunto de mensajes (una especie de colección inteligente) y es donde podemos añadir los problemas de validación que encontremos.
- (3)`Object valor`: El valor a validar.
- (4)`String nombreObjeto`: Nombre del objeto al que pertenece la propiedad a validar. Útil para usarlo en los mensajes de error.
- (5)`String nombrePropiedad`: Nombre de la propiedad a validar. Útil para usarlo en los mensajes de error.

Como se ve cuando encontramos un error de validación solo tenemos que añadirlo (con `errores.add()`) enviando un identificador de mensaje y los argumentos. Para que este validador

produzca un mensaje significativo tenemos que tener en nuestro archivo de mensajes i18n la siguiente entrada:

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

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

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

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

3.9 Referencia (5)

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

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

para acceder al nombre del comercial de ese cliente.

La sintaxis para definir referencias es:

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

- (1) `nombre` (opcional, obligada si no se especifica `modelo`): Es el nombre que tendrá la referencia en Java, por lo tanto ha de seguir la normativa para nombres de miembros Java, entre ellas empezar por minúscula. Si no especificamos `nombre` asume el nombre del modelo pero en con la primera letra minúscula. Se desaconseja usar subrayado (`_`).
- (2) `etiqueta` (opcional): Etiqueta que se mostrará al usuario final. Es **mucho mejor** usar los archivos i18n.
- (3) `modelo` (opcional, obligada si no se especifica `nombre`): Es el nombre del modelo a referenciar. Puede ser el nombre de otro componente, en cuyo caso es una referencia a entidad, o el nombre de un agregado del componente en el que estamos. Si no especificamos `modelo` asume el `nombre` de la referencia pero con la primera letra en mayúscula.

- (4)`requerido` (opcional): Indica si la referencia es requerida. Al grabar OpenXava comprobará si las referencias requeridas están presentes, si no lo están no se producirá la grabación y se devolverá una lista de errores de validación.
- (5)`clave` (opcional): Para indicar si la referencia forma parte de la clave. La combinación de propiedades y referencias clave se debe mapear a un conjunto de campos en la base de datos que no tengan valores repetidos, típicamente con la clave primaria.
- (6)`cometido-destino` (opcional): Usado solo en referencia dentro de colecciones. Ver más adelante.
- (7)`calculador-valor-defecto` (uno, opcional): Para implementar la lógica para calcular el valor inicial de la referencia. Este calculador ha de devolver el valor de la clave, que puede ser un dato simple (solo si la clave del objeto referenciado es simple) o un objeto clave (un objeto especial que envuelve la clave primaria y que genera OpenXava).

Un pequeño ejemplo de uso de referencias:

```
<referencia modelo="Direccion" requerido="true"/> (1)
<referencia nombre="comercial"/> (2)
<referencia nombre="comercialAlternativo" model="Comercial"/> (3)
```

- (1)Una referencia a un agregado llamado `Direccion`, el nombre de la referencia será `direccion`.
- (2)Una referencia a la entidad del componente `Comercial`. Se deduce el modelo a partir del nombre.
- (3)Una referencia llamada `comercialAlternativo` a la entidad del componente `Comercial`.

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

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

3.9.1 Calculador valor por defecto para una referencia

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

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

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

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

```
public Object calculate() throws Exception {
```



```
        return new Integer(value);  
    }  
}
```

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

En el caso de clave compuesta sería así:

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

Y el código del calculador:

```
package org.openxava.test.calculadores;  
  
import org.openxava.calculators.*;  
import org.openxava.test.ejb.*;  
  
/**  
 * @author Javier Paniza  
 */  
public class CalculadorDefectoAlmacen implements ICalculator {  
  
    public Object calculate() throws Exception {  
        Almacen clave = new Almacen();  
        clave.setCodigo(4);  
        clave.setCodigoZona(4);  
        return clave; // Funciona con POJOs y EJB  
        // return new AlmacenKey(new Integer(4), 4); // Funciona solo con EJB  
    }  
  
}
```

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

3.10 Colección (6)

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

Aquí la sintaxis para definir una colección:

```
<coleccion  
    nombre="nombre"                (1)  
    etiqueta="etiqueta"           (2)
```

```

    minimo="N"                                (3)
>
    <referencia ... />                        (4)
    <condicion ... />                        (5)
    <orden ... />                            (6)
    <calculador ... />                      (7)
    <calculador-posborrar ... />           (8)
</coleccion>

```

- (1)nombre (obligado): Es el nombre que tendrá la colección en Java, por lo tanto ha de seguir la normativa para nombres de miembro Java, entre ellas empezar por minúscula. Se desaconseja usar subrayado (_).
- (2)etiqueta (opcional): Etiqueta que se mostrará al usuario final. Es **mucho mejor** usar los archivos *i18n*.
- (3)minimo (opcional): Indica el número mínimo de elementos esperados. Esto se valida antes de grabar.
- (4)referencia (obligada): Con la sintaxis vista en el subtema anterior.
- (5)condicion (opcional): Para restringir los elementos que aparecen en la colección.
- (6)orden (opcional): Para que los elementos de la colección aparezcan en un determinado orden.
- (7)calculador (opcional): Permite especificar nuestra propia lógica Java para generar la colección. Si se especifica `calculador` no se puede poner ni `condicion` ni `orden`.
- (8)calculador-posborrar (opcional): Permite ejecutar cierta lógica de negocio justo después de haber eliminado un elemento de la colección.

Vamos a ver algunos ejemplos. Empecemos por uno simple:

```

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

```

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

Ahora podemos escribir código como este:

```

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

```

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

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

```

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

```

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

- (1) La restricción de `minimo="1"` hace que sea obligado que haya al menos una línea para que la factura sea válida.
- (2) Con `orden` obligamos a que las líneas se devuelvan ordenadas por `tipoServicio`.
- (3) Con `calculador-posborrar` indicamos un calculador a ejecutar justo después de borrar un elemento de la colección. Veamos el código de este calculador:

```

package org.openxava.test.calculadores;

import java.rmi.*;

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

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

    private IFactura factura;

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

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

}

```

Como se ve es un calculador normal y corriente, como el que hemos visto que se puede asignar a

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

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

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

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

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

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

Y aquí el código del calculador:

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

import java.rmi.*;

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

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

```

private ITransportista transportista;

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

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

```

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

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

```

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

```

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

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

3.11 Método (7)

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

La sintaxis para definir un método es:

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

- (1)nombre (obligado): Es el nombre que tendrá el método en Java, por lo tanto ha de seguir la normativa para miembros Java, entre ellas empezar por minúscula.
- (2)tipo (opcional, por defecto `void`): Corresponde a un tipo Java. Todos los tipos válidos como tipo de retorno de un método en Java son válidos aquí.
- (3)argumentos (opcional): Lista de argumentos del método en formato Java.
- (4)excepciones (opcional): Lista de excepciones que puede lanzar el método en formato Java.
- (5)calculador (obligado): Implementa la lógica que ejecuta el método.

Definir un método es sencillo:

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

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

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

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

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

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

```

public class CalculadorIncrementarPrecio implements IModelCalculator {

    private IProducto producto;

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

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

}

```

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

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

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

Ahora podemos usar el método de la forma esperada:

```

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

```

Y en `nuevoPrecio` tendremos 102.

Otro ejemplo de método, ahora un poco más complejo:

```

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

```

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

El calculador quedaría de la siguiente forma:

```

package org.openxava.test.calculadores;

import java.math.*;

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

/**
 * @author Javier Paniza
 */

public class CalculadorPrecioExportacion implements ICalculator {

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

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

    public BigDecimal getEuros() {
        return euros;
    }

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

    public BigDecimal getTarifa() {
        return tarifa;
    }

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

    public String getPais() {
        return pais;
    }

```



```

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

```

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

Y para usar el método:

```

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

```

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

3.12 Buscador (8)

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

La sintaxis para definir buscadores es:

```

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

```

- (1)`nombre` (obligado): Es el nombre que tendrá el método *finder* en Java, por lo tanto ha de seguir la normativa para miembros Java, entre ellas empezar por minúscula.
- (2)`argumentos` (obligado): Lista de argumentos del método en formato Java. Lo más aconsejable es usar tipos de datos simples.
- (3)`coleccion` (opcional, por defecto `false`): Indica si el resultado va a ser un solo objeto o una colección.
- (4)`condicion` (opcional): Una condición con sintaxis SQL/EJBQL en la que podemos usar los nombres de las propiedades entre `{ }`.

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

Algunos ejemplos:

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

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

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

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

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

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

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

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

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

3.13 Calculador poscrear (9)

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

Su sintaxis es:

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

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

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

Un ejemplo sencillo sería:

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

Y ahora la clase del calculador:

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

import java.rmi.*;

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

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

    private ITipoAlbaran tipoAlbaran;
    private String sufijo;

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

```

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

    public String getSufijo() {
        return sufijo;
    }

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

```

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

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

3.14 *Calculador posmodificar (11)*

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

Su sintaxis es:

```

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

```

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

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

Un ejemplo sencillo sería:

```

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

```

Y ahora la clase del calculador:

```

package org.openxava.test.calculadores;

import java.rmi.*;

```

```

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

/**
 * @author Javier Paniza
 */

public class CalculadorPosmodificarTipoAlbaran implements IModelCalculator {

    private ITipoAlbaran tipoAlbaran;

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

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

}

```

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

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

3.15 Calculadores poscargar y preborrar (10, 12)

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

3.16 Validador (13)

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

Su sintaxis es:

```

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

```

```
</validador>
```

- (1) `clase` (opcional, obligada si no se especifica nombre): Clase que implementa la validación. Ha de ser del tipo `IValidator`.
- (2) `nombre` (opcional, obligada si no se especifica clase): Este nombre es el que tenga el validador en el archivos `xava/validators.xml` o `xava/validadores.xml`, del proyecto OpenXava o de nuestro propio proyecto.
- (3) `solo-al-crear` (opcional): Si `true` el validador es ejecutado solo cuando estamos creando un objeto nuevo, no cuando modificamos uno existente. El valor por defecto es `false`.
- (4) `poner` (varios, opcional): Para establecer valor a las propiedades del validador antes de ejecutarse.

Un ejemplo:

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

Y el código del validador:

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

import java.math.*;

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

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

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

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

```

        }

    }

    public BigDecimal getPrecioUnitario() {
        return precioUnitario;
    }

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

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

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

    public int getLimite() {
        return limite;
    }

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

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

}

```

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

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

3.17 Validador borrar (14)

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

Su sintaxis es:

```
<validador-borrar
    clase="validador"           (1)
    nombre="nombre"           (2)
>
    <poner ... /> ...          (3)
</validador-borrar>
```

- (1) `clase` (opcional, obligada si no se especifica `nombre`): Clase que implementa la validación. Ha de ser del tipo `IRemoveValidator`.
- (2) `nombre` (opcional, obligada si no se especifica `clase`): Este nombre es el que tenga el validador en el archivos `xava/validators.xml` o `xava/validadores.xml`, del proyecto OpenXava o de nuestro propio proyecto.
- (3) `poner` (varios, opcional): Para establecer valor a las propiedades del calculador antes de ejecutarse.

Un ejemplo puede ser:

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

Y el validador:

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

import java.util.*;

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

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

    private ITipoAlbaran tipoAlbaran;

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



```

    }

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

```

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

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

3.18 Agregado

La sintaxis de un agregado es como sigue:

```

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

```

- (1) **nombre** (obligado): Cada agregado tiene que tener un nombre único. La normativa para poner el nombre es la misma que para un nombre de clase Java, es decir, empieza por mayúscula y cada palabra nueva también.
- (2) **bean** (uno, opcional): Permite especificar una clase escrita por nosotros para implementar el agregado. La clase ha de ser un `JavaBean`, es decir una clase de Java normal y corriente con *getters* y *setters* para las propiedades. Normalmente no se suele usar ya que es mucho mejor que OpenXava genere el código por nosotros.
- (3) **ejb** (uno, opcional): Permite usar un EJB existente para implementar un agregado. Esto se usa en el caso de querer tener una colección de agregados. Normalmente no se suele usar ya que es

mucho mejor que OpenXava genere el código por nosotros.

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

3.18.1 Referencia a agregado

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

En la entidad principal pondremos:

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

Y a nivel de componente definiremos el agregado.

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

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

El código resultante se puede usar así, para leer:

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

O así para establecer una nueva dirección

```
// para establecer una nueva dirección
Direccion direccion = new Direccion(); // es un JavaBean, nunca un EJB
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`, jamás tendrá vida propia ni podrá ser compartida por otro `Cliente`.

3.18.2 Colección de agregados

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

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

Y definimos el agregado LineaFactura:

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

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

```
</validador>

</agregado>
```

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

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

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

La sintaxis para definir una vista es:

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

- (1) `nombre` (opcional): El nombre identifica a la vista, y puede ser usado desde otro lugares de OpenXava (por ejemplo desde *aplicacion.xml*) o desde otro componente. Si no se pone nombre se asume que es la vista por defecto, es decir la forma normal de visualizar el objeto.
- (2) `etiqueta` (opcional): Etiqueta que se muestra al usuario si es necesario, cuando se visualiza la vista. Es **mucho mejor** usar los archivos *i18n*.
- (3) `modelo` (opcional): Si es una vista de un agregado de este componente se pone aquí el nombre del agregado. Si no se pone nada se supone que es una vista de la entidad.
- (4) `miembros` (opcional): Lista de miembros a visualizar. Por defecto visualiza todos los miembros no ocultos en el orden en que están declarados en el modelo. Este atributo es excluyente con el elemento `miembros` visto más adelante.
- (5) `propiedad` (varios, opcional): Permite definir propiedades de la vista, es decir, información que se puede visualizar de cara al usuario y tratar programáticamente, pero no forma parte del modelo.
- (6) `vista-propiedad` (varias, opcional): Para definir la forma en que queremos que se visualice cierta propiedad.
- (7) `vista-referencia` (varias, opcional): Para definir la forma en que queremos que se visualice cierta referencia.
- (8) `vista-coleccion` (varias, opcional): Para definir la forma en que queremos que se visualice cierta colección.

(9)`miembros` (uno, opcional): Indica los miembros que tienen que salir y como tienen que estar dispuestos en la interfaz gráfica. Es excluyente con el atributo `miembros`.

4.1 Disposición

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

Por ejemplo, un modelo así:

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

Generaría una vista con este aspecto:



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

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

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

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

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

Podemos observar como separamos los nombres de miembros con comas y punto y coma, 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:

Zona 1 Oficina 1 Número 1

Nombre PEPE

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

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

En este caso el resultado sería:

Id

Zona 1 Oficina 1 Número 1

Nombre PEPE

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

Podemos poner varios grupos en una vista:

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

En este caso se visualizan uno al lado del otro:

Cliente

Tipo Normal

Nombre

Comercial

Comercial

Número Añadir

Nombre

Relacion con comercial GOOD

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

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

En este caso se visualizaría así:

The image shows a graphical user interface with two main sections. The top section is titled 'Cliente' and contains two fields: 'Tipo' with a dropdown menu showing 'Normal' and a checkmark icon, and 'Nombre' with a text input field and a checkmark icon. The bottom section is titled 'Comercial' and contains a sub-form titled 'Comercial' with two fields: 'Número' with a text input field, a key icon, and an 'Añadir' button, and 'Nombre' with a text input field and a checkmark icon. Below the sub-form is a field for 'Relacion con comercial' with the value 'GOOD'.

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

```
<miembros>
    factura;
    <grupo nombre="datosAlbaran">
        tipo, codigo;
        fecha;
        descripcion;
        envio;
        <grupo nombre="datosTransporte">
            distancia vehiculo; modoTransporte; tipoConductor;
        </grupo>
        <grupo nombre="datosEntregadoPor">
            entregadoPor;
            transportista;
            empleado;
        </grupo>
    </grupo>
```



```

    </grupo>

</grupo>
</miembros>

```

Obtendremos lo siguiente:

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

```

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

```

El resultado visual sería:

Las secciones se convierten en pestañitas que el usuario puede pulsar para ver la información contenida en esa sección. Podemos observar también como en la vista indicamos todo tipo de miembros (y no solo propiedades), así `cliente` es una referencia, `lineas` una colección de agregados y `albaranes` una colección de referencias a entidad.

Año Número Fecha Pagada ☐

Descuento cliente € Descuento tipo cliente € Descuento por año €

Comentario

Comercial Líneas Importes Albaranes

Codiguito [Añadir](#)

Tipo

Nombre

Dirección

ViewProperty

Via pública ☐ Código postal State

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

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

En este caso podemos obtener una interfaz gráfica como esta:

Año Número

Cliente **Datos** Albaranes

Líneas **Importes**

I.V.A. Suma importes

% IVA ☐

I.V.A. €

Es de notar tenemos grupos y no marcos y secciones y no pestañas. Porque en las vista de OpenXava intentamos mantener un nivel de abstracción alto, es decir, un grupo es un conjunto de propiedades relacionadas semánticamente, y las secciones nos permite dividir la información en

partes cuando tenemos mucha y posiblemente no se pueda visualizar toda a la vez, el que los grupos se representen con marquitos y las secciones con pestañas es una cuestión de implementación, pero el generador del interfaz gráfico podría escoger usar un árbol u otro control gráfico para representar las secciones, por ejemplo.

4.2 Vista propiedad

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

Tiene esta sintaxis:

```
<vista-propiedad
  propiedad="nombrePropiedad"           (1)
  etiqueta="etiqueta"                   (2)
  solo-lectura="true|false"             (3)
  formato-etiqueta="NORMAL|PEQUENA|SIN_ETIQUETA" (4)
>
  <al-cambiar ... />                     (5)
  <accion ... /> ...                     (6)
</vista-propiedad>
```

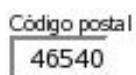
- (1) `propiedad` (obligado): Normalmente el nombre de una propiedad del modelo, aunque también puede ser el nombre de una propiedad propia de la vista.
- (2) `etiqueta` (opcional): Para modificar la etiqueta que se sacará en esta vista para esta propiedad. Para esto es **mucho mejor** usar los archivos `i18n`.
- (3) `solo-lectura` (opcional): Si la ponemos a `true` esta propiedad no será nunca editable por el usuario en esta vista. Una alternativa a esto es hacer la propiedad editable/no editable programáticamente usando `org.openxava.view.View`.
- (4) `formato-etiqueta` (opcional): Forma en que se visualiza la etiqueta para esta propiedad.
- (5) `al-cambiar` (uno, opcional): Acción a realizar cuando cambia el valor de esta propiedad.
- (6) `accion` (varias, opcional): Acciones (mostradas como vínculos, botones o imágenes al usuario) asociadas (visualmente) a esta propiedad y que el usuario final puede ejecutar.

4.2.1 Formato de etiqueta

Un ejemplo sencillo para cambiar el formato de la etiqueta:

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

En este caso el código postal lo visualiza así:



El formato `NORMAL` es el que hemos visto hasta ahora (con la etiqueta grande y la izquierda) y el

formato `SIN_ETIQUETA` simplemente hace que no salga etiqueta.

4.2.2 Evento de cambio de valor

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

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

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

El código que se ejecutará será:

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

import org.openxava.actions.*;

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

    public void execute() throws Exception {
        if (getNewValue() == null) return; // (2)
        getView().setValue("observaciones", // (3)
            "El transportista es " + getNewValue());
        addMessage("transportista_cambiado");
    }

}
```

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

4.2.3 Acciones de la propiedad

También podemos especificar acciones que el usuario puede pulsar directamente:

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

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

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

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

Número  [Generar](#)

El vínculo de la acción aparece solo cuando la propiedad es editable, ahora bien si la propiedad es de solo-lectura o calculada entonces está siempre disponible.

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.

4.3 Vista referencia

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

Su sintaxis es:

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

```

vista="vista" (2)
solo-lectura="true|false" (3)
marco="true|false" (4)
crear="true|false" (5)
buscar="true|false" (6)
>
<accion-buscar ... /> (7)
<lista-descripciones ... /> (8)
<accion ... /> ... (9) // Nuevo en v2.0.1
</vista-referencia>

```

- (1)referencia (obligado): Nombre de la referencia del modelo de la que se quiere personalizar la visualización.
- (2)vista (opcional): Si omitimos este atributo usa la vista por defecto del objeto referenciado para visualizarlo, con este atributo podemos indicar que use otra vista.
- (3)solo-lectura (opcional): Si la ponemos a `true` esta referencia no será nunca editable por el usuario en esta vista. Una alternativa a esto es hacer la propiedad editable/no editable programáticamente usando `org.openxava.view.View`.
- (4)marco (opcional): Si el dibujador de la interfaz gráfica usa un marco para envolver todos los datos de la referencia con este atributo se puede indicar que dibuje o no ese marco, por defecto sí que sacará el marco.
- (5)crear (opcional): Indica si el usuario ha de tener opción para crear o no un nuevo objeto del tipo referenciado. Por defecto vale `true`.
- (6)buscar (opcional): Indica si el usuario va a tener un vínculo para poder realizar búsquedas con una lista, filtros, etc. Por defecto vale `true`.
- (7)accion-buscar (una, opcional): Nos permite especificar nuestra propia acción de búsqueda.
- (8)lista-descripciones: Permite visualizar los datos como una lista descripciones, típicamente un combo. Práctico cuando hay pocos elementos del objeto referenciado.
- (9)accion (varias, opcional): (*nuevo en v2.0.1*) Acciones (mostradas como vínculos, botones o imágenes al usuario) asociadas (visualmente) a esta referencia y que el usuario final puede ejecutar. Funciona como en el caso de `<vista-propiedad/>`, mirar la sección 4.2.3.

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

```

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

```

La interfaz gráfica tendrá el siguiente aspecto:

Familia	
Número	<input type="text" value="1"/> Añadir
Descripción	<input type="text" value="SOFTWARE"/>
Products	<input type="text"/>

4.3.1 Escoger vista

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

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

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

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

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

Factura							
Año	<input type="text" value="2002"/>	Número	<input type="text" value="1"/>	Fecha	<input type="text" value="01/01/2002"/>	Descuento por año	<input type="text" value="200"/> €

4.3.2 Personalizar el enmarcado

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

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

Así obtendríamos:

Comercial	
Número	<input type="text"/>   Añadir
Nombre	<input type="text"/>
Relacion con comercial	GOOD

4.3.3 Acción de búsqueda propia

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

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

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

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

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

Lo que hagamos en `MiAccionBuscar` ya es cosa nuestra. Podemos, por ejemplo, refinar la acción por defecto de búsqueda 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.

4.3.4 Acción de creación propia

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

```

<!--
Puesto que su nombre es Almacen2Creation (nombre modelo + Creation) es usado
por defecto para crear desde referencias, en vez de NewCreation.
La accion 'new' es ejecutada automáticamente.
-->
<controlador nombre="Almacen2Creation">
    <hereda-de controlador="NewCreation"/>
    <accion nombre="new" oculta="true"
        clase="org.openxava.test.actions.CrearNuevoAlmacenDesdeReferencia"
        imagen="images/new.gif"
        atajo-de-teclado="F2">
        <usa-objeto nombre="xava_view"/>
    </accion>
</controlador>

```

En este caso cuando en una referencia a `Almacen2` pulsemos el vínculo 'crear' irá a la vista por defecto de `Almacen2` y mostrará las acciones de `Almacen2Creation`.

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

4.3.5 Lista descripciones (combos)

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

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

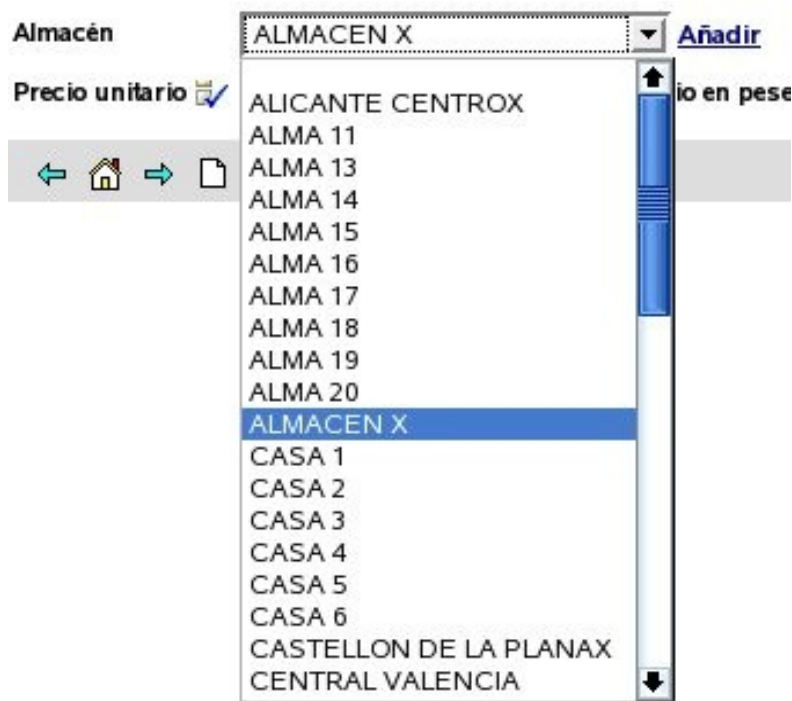
- (1) `propiedad-descripcion` (opcional): Indica que propiedad es la que tiene que aparecer en la lista, si no se especifica asume la propiedad `description`, `descripcion`, `name` o `nombre`. Si el objeto referencia no tiene ninguna propiedad llamada así entonces es obligado especificar aquí un nombre de propiedad.
- (2) `propiedades-descripciones` (opcional): Como `propiedad-descripcion` (y además exclusiva con ella) pero permite poner una lista de propiedades separadas por comas. Al usuario le aparecen concatenadas.
- (3) `depende-de` (opcional): Se usa junto con `condicion` para hacer que el contenido de la lista dependa del valor de otro miembro visualizado en la vista principal (si simplemente ponemos el nombre del miembro) o en la misma vista (si ponemos `this.` delante del nombre de miembro).
- (4) `condicion` (opcional): Permite poner una condición (al estilo SQL) para filtrar los valores que aparecen en la lista de descripciones.
- (5) `ordenado-por-clave` (opcional): Por defecto los datos salen ordenados por descripción, pero si ponemos esta propiedad a `true` saldrán ordenados por clave.
- (6) `orden` (opcional): Permite poner un orden (al estilo SQL) para los valores que aparecen en la lista de descripciones.
- (7) `formato-etiqueta` (opcional): Forma en que se visualiza la etiqueta para esta referencia. Ver sección 4.2.1.

El uso más simple es:

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

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

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



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

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

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

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

En el caso de Familia (1) se visualiza la propiedad `descripcion` de Familia, ya que si no lo indicamos por defecto visualiza una propiedad llamada 'descripcion' o 'nombre'. En este caso los datos aparecen ordenados por clave y no por descripción. En el caso de Subfamilia indicamos que no muestre el vínculo para crear una nueva subfamilia (2) y que la propiedad a visualizar es `descripcion` (aunque esto lo podríamos haber omitido). Con `depende-de` (4) hacemos que este combo dependa de la referencia `familia`, cuando cambia `familia` en la interfaz gráfica, rellena esta lista de descripciones aplicando la condición de `condicion` (5) y enviando como argumento (para rellena el interrogante) el nuevo valor de `familia`. Y las entradas están ordenadas descendientemente por descripción (6).

En `condicion` y `orden` ponemos los nombres de las propiedades entre `${}` y los argumentos como

?, los operadores de comparación son los de SQL.

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

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

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

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

4.4 Vista colección

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

```
<vista-coleccion
  coleccion="coleccion"           (1)
  vista="vista"                   (2)
  solo-lectura="true|false"       (3)
  solo-edicion="true|false"       (4)
  crear-referencia="true|false"   (5)
>
  <propiedades-lista ... />       (6)
  <accion-editar ... />          (7)
  <accion-ver ... />             (8)
  <accion-lista ... /> ...       (9)
  <accion-detalle ... /> ...    (10)
</vista-coleccion>
```

- (1) `coleccion` (obligado): Indica la colección de la que se quiere personalizar la presentación.
- (2) `vista` (opcional): La vista del objeto referenciado que se ha de usar para representar el detalle. Por defecto usa la vista por defecto.
- (3) `solo-lectura` (opcional): Por defecto `false`, si la ponemos a `true` solo podremos visualizar los elementos de la colección, no podremos ni añadir, ni borrar, ni modificar los elementos.
- (4) `solo-edicion` (opcional): Por defecto `false`, si la ponemos a `true` podemos modificar los elementos existentes, pero no podemos añadir nuevos ni eliminar.
- (5) `crear-referencia` (opcional): Por defecto `true`, si la ponemos a `false` el usuario final no tendrá el vínculo que le permite crear objetos del tipo del objeto referenciado. Esto solo aplica en el caso de colecciones de referencias a entidad.
- (6) `propiedades-lista` (una, opcional): Indica las propiedades que han de salir en la lista al visualizar la colección. Podemos calificar las propiedades. Por defecto saca todas las propiedades persistentes del objeto referenciado (sin incluir referencias ni calculadas).

- (7)accion-editar (una, opcional): Permite sobrescribir 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.
- (8)accion-ver (una, opcional): Permite sobrescribir la acción para visualizar un elemento de la colección. Esta es la acción mostrada en cada fila cuando la colección es de solo lectura.
- (9)accion-lista (varias, opcional): Para poder añadir acciones en el modo lista; normalmente acciones cuyo alcance es la colección entera.
- (10)accion-detalle (varias, opcional): Para poder añadir acciones en detalle, normalmente acciones cuyo alcance es el detalle que se está editando.

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

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

De esta forma la colección se visualiza así:

Clientes					
	Número	Nombre	Observaciones	Relacion con comercial	Descripción
Editar	1	Javi		BUENA	MANAGER
Editar	2	Juanillo			MANAGER
Añadir					

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

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


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


```
<view name="Simple" members="number; type; name; address"/>
```


Al pulsar detalle aparecerá:

Clientes					
	Número	Nombre	Observaciones	Relacion con comercial	Descripción
Editar	1	Javi		BUENA	MANAGER
Editar	2	Juanillo			MANAGER

Cliente



Número 

Tipo  ☒ Fijo




Nombre 

Dirección

ViewProperty

Via pública  ☒ DOCTOR PESSET  ☒ Código postal

Población

 ☒ EL PUIG  ☒ Estado  ☒ New York

[Grabar detalle](#) [Cerrar](#) [Quitar detalle](#)

4.4.1 Acción de editar/ver detalle propia

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

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

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

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

Y nuestra acción puede ser así:

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

import java.text.*;
```

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

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

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

En este caso queremos solamente refinar y por eso nuestra acción descende 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).

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

4.4.2 Acciones de lista propias

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

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

Ahora aparece un nuevo vínculo al usuario:

Fellow Carriers					
		Número	Nombre	Calculated	Observaciones
Editar	<input type="checkbox"/>	2	DOS	TR	
Editar	<input type="checkbox"/>	3	THREE	TR	
Editar	<input type="checkbox"/>	4	FOUR	TR	
Añadir Traducir nombre					

Vemos además como ahora hay una casilla de chequeo en cada línea.

Falta definir la acción en `controladores.xml`:

```
<controlador nombre="Transportistas">
    <accion nombre="traducirNombre"
```

```
        clase="org.openxava.test.acciones.TraducirNombreTransportista">
    </accion>
</controladores>
```

Y el código de nuestra acción:

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

import java.util.*;

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

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

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

La acción descende 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).

4.4.3 Acciones de detalle propias

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

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

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

Debemos definir la acción en *controladores.xml*:

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

Y el código de nuestra acción:

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

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

import org.openxava.actions.*;

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

    private Map valoresFactura;

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

```

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

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

public String getCustomView() {
    return SAME_VIEW;
}

public Map getValoresFactura() {
    return valoresFactura;
}

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

```

Vemos como descende 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.

4.5 Propiedad de vista

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

Un ejemplo:

```
<vista>
```

```

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

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

    ...
</vista>

```

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

4.6 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%>&nbsp;
<%
}
%>

<% 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 sobrescribiendo 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.

4.7 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:

```
<propiedad nombre="regiones" estereotipo="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`).

4.8 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:

```
<entidad>
    ...
    <propiedad nombre="codigoFamilia" estereotipo="FAMILIA" requerido="true"/>
    <propiedad nombre="codigoSubfamilia" estereotipo="SUBFAMILIA" requerido="true"/>
    ...
</entidad>
```

Y en nuestro *editores.xml* ponemos:

```
<editor url="descriptionsEditor.jsp"> (1)
    <propiedad nombre="modelo" valor="Familia"/> (2)
    <propiedad nombre="propiedadClave" valor="codigo"/> (3)
    <propiedad nombre="propiedadDescripcion" valor="descripcion"/> (4)
    <propiedad nombre="ordenadoPorClave" valor="true"/> (5)
    <propiedad nombre="readOnlyAsLabel" valor="true"/> (6)
    <para-estereotipo estereotipo="FAMILIA"/> (11)
</editor>

<!-- Es posible especificar dependencias de estereotipos o propiedades -->
<editor url="descriptionsEditor.jsp" (1)
    depende-de-estereotipos="FAMILIA"> (12)
<!--
<editor url="descriptionsEditor.jsp" depende-de-propiedades="codigoFamilia"> (13)
-->
    <propiedad nombre="modelo" valor="Subfamilia"/> (2)
    <propiedad nombre="propiedadClave" valor="codigo"/> (3)
    <propiedad nombre="propiedadesDescripcion" valor="codigo, descripcion"/> (4)
    <propiedad nombre="condicion" value="{codigoFamilia} = ?"/> (7)
    <propiedad nombre="estereotiposValoresParametros" valor="FAMILIA"/> (8)
    <!--
    <propiedad nombre="propiedadesValoresParametros" valor="codigoFamilia"/> (9)
    -->
    <propiedad nombre="formateadorDescripciones" (10)
        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* (1) y lo configuramos asignándole valores a sus propiedades. Algunas

propiedades con las que podemos configurar estos editores son:

- (2)`modelo`: Modelo del que se obtiene los datos. Puede ser el nombre de un componente (`Invoice`) o el nombre de un agregado usado en colección (`Invoice.InvoiceDetail`).
- (3)`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.
- (4)`propiedadDescripcion` o `propiedadesDescripcion`: Propiedad o lista de propiedades a visualizar en el combo.
- (5)`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.
- (6)`readOnlyAsLabel`: Si cuando es de solo lectura se ha de visualizar como una etiqueta. Por defecto es `false`.
- (7)`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.
- (8)`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)
- (9)`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)
- (10)`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 `lista-descripciones`; pero siempre tenemos la opción de los estereotipos disponible.

4.9 Vista sin modelo asociado

En OpenXava no se puede tener vistas que no estén asociadas a un modelo. Así que si queremos dibujar una interfaz gráfica arbitraria, lo que hemos de hacer es crear un componente y asociar éste a una tabla inexistente (mientras que no tratemos grabar o leer no pasa nada) y a partir de él definir una vista.

Un ejemplo puede ser:

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

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

<!--
```

Ejemplo de componente OpenXava no persistente.

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

Por supuesto, si tratamos de grabar o leer desde la db con este componente fallará.

De momento es necesario especificar la parte del mapeo.

-->

```
<componente nombre="FiltrarPorSubfamilia">

  <entidad>
    <propiedad nombre="oid" clave="true" tipo="String" oculta="true"/>
    <referencia nombre="subfamilia" modelo="Subfamilia2" requerido="true"/>
  </entidad>

  <vista nombre="Familiar1">
    <vista-referencia referencia="subfamilia" crear="false">
      <lista-descripciones condicion="{familia.codigo} = 1"/>
    </vista-referencia>
  </vista>

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

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

  <mapeo-entidad tabla="XAVATEST@separator@MOCKTABLE">
    <mapeo-propiedad propiedad-modelo="oid" columna-tabla="OID"/>
    <mapeo-referencia referencia-modelo="subfamilia">
      <detalle-mapeo-referencia
        columna-tabla="SUBFAMILIA"
        propiedad-modelo-referenciado="codigo"/>
    </mapeo-referencia>
  </mapeo-entidad>

</componente>
```

```
</componente>
```

De esta forma podemos hacer un diálogo que puede servir, por ejemplo, para lanzar un listado de familias o productos filtrado por subfamilias. La tabla **MOCKTABLE** no existe.

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

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:

		Borrar seleccionados		Marcados a minúsculas		Detalle - Lista	
Filtrar		Zona	Código almacen	Nombre			
=		=	=	empieza por			
Detalle	<input type="checkbox"/>	1	1	CENTRAL VALENCIA			
Detalle	<input type="checkbox"/>	1	2	VALENCIA SURETE			
Detalle	<input type="checkbox"/>	1	3	VALENCIA NORTE			
Detalle	<input type="checkbox"/>	2	1	CASTELLON DE LA PLANAX			
Detalle	<input type="checkbox"/>	3	1	ALICANTE CENTROX			
Detalle	<input type="checkbox"/>	4	2	ALMA 2			
Detalle	<input type="checkbox"/>	4	3	ALMA 3			
Detalle	<input type="checkbox"/>	4	4	ALMA 4			
Detalle	<input type="checkbox"/>	4	5	ALMA 5			
Detalle	<input type="checkbox"/>	4	6	ALMA 6			
1 2 3 4 5		Hay 49 registros en la lista					
		Borrar seleccionados		Marcados a minúsculas		Detalle - Lista	

Esta lista permite al usuario:

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

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

La sintaxis de `tab` es:

```

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

```

```
<orden-defecto ... />      (6)
</tab>
```

- (1) **nombre** (opcional): Podemos definir varios tabs para un componente, y ponerle un nombre a cada uno. Este nombre se usará después para indicar que tab queremos usar (normalmente en *aplicación.xml* al definir un módulo).
- (2) **filtro** (uno, opcional): Permite definir programáticamente un filtro a realizar sobre los valores que introduce el usuario cuando quiere filtrar.
- (3) **estilo-fila** (varios, opcional): Una forma sencilla de especificar una estilo de visualización diferente para ciertas filas. Normalmente para resaltar filas que cumplen cierta condición.
- (4) **propiedades** (uno, opcional): La lista de propiedades a visualizar inicialmente. Pueden ser calificadas.
- (5) **condicion-base** (una, opcional): Es una condición que aplicará siempre a los datos visualizados añadiéndose a las que pueda poner el usuario.
- (6) **orden-defecto** (uno, opcional): Para especificar el orden en que aparece los datos en la lista inicialmente.

5.1 Propiedades iniciales y resaltar filas

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

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

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

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

	Nombre	Tipo	Comercial	Población	Nivel comercial
Filtrar	empieza por ▼	▼	empieza por ▼	empieza por ▼	empieza por ▼
Detalle	<input type="checkbox"/> Javi	Fijo	MANUEL CHAVARRI	EL PUIG	MANAGER
Detalle	<input type="checkbox"/> Juanillo	Normal	MANUEL CHAVARRI	VALENCIA	MANAGER
Detalle	<input type="checkbox"/> Carmelo	Normal		EL PUIG	
Detalle	<input type="checkbox"/> Cuatrero	Normal	JUANVI LLAVADOR	VALENCIA	MANAGER

1 Hay 4 registros en la lista

5.2 Filtros y condición base

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

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

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

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

import java.util.*;

import org.openxava.filters.*;

/**
 * @author Javier Paniza
 */

public class FiltroAñoActual implements IFilter { // (1)

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

```

        else {
            r = new Object[2];
            r[0] = año;
            r[1] = o;
        }

        return r;
    }
}

```

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

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

Podemos ver otro caso:

```

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

```

En este caso el filtro es:

```

package org.openxava.test.filtros;

import java.util.*;

import org.openxava.filters.*;

/**
 * @author Javier Paniza
 */

public class FiltroAñoDefecto extends BaseContextFilter {
    // (1)

```



```

public Object filter(Object o) throws FilterException {
    if (o == null) {
        return new Object [] { getAñoDefecto() };           // (2)
    }
    if (o instanceof Object []) {
        List c = new ArrayList(Arrays.asList((Object []) o));
        c.add(0, getAñoDefecto());                         // (2)
        return c.toArray();
    }
    else {
        return new Object [] { getAñoDefecto(), o };       // (2)
    }
}

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

```

Este filtro descende 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`. Este objeto lo definimos en nuestro archivo *controladores.xml* de esta forma:

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

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

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

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

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

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

5.3 Select íntegro

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

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

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

5.4 Orden por defecto

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

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

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

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

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

- Hibernate 3.1.
- EntityBeans CMP 2 de JBoss 3.2.x y 4.0.x.
- EntityBeans CMP 2 de Websphere 5, 5.1y 6.

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

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

6.1 Mapeo de entidad

La sintaxis para mapear la entidad principal es:

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

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

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

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

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

Un ejemplo sencillo de mapeo puede ser:

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

```
<mapeo-propiedad propiedad-modelo="descripcion" columna-tabla="DESCRIPCION" />
</mapeo-entidad>
```

Nada más fácil.

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

6.2 Mapeo propiedad

La sintaxis para mapear una propiedad es:

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

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

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

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

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

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

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

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

El código del conversor será:

```

package org.openxava.converters;

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

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

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

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

```

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

Ante cualquier problemas podemos lanzar una `ConversionException`.

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

```
<entidad>
```

```

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

```

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

```

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

```

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

6.3 Mapeo referencia

La sintaxis para mapear una referencia es:

```

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

```

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

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

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

```

<entidad>
    ...

```

```

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

```

Podemos mapearla de esta forma:

```

<mapeo-entidad tabla="XAVATEST@separator@ALBARAN">
  <mapeo-referencia referencia-modelo="factura">
    <detalle-mapeo-referencia
      columna-tabla="FACTURA_AÑO"
      propiedad-modelo-referenciado="año" />
    <detalle-mapeo-referencia
      columna-tabla="FACTURA_NUMERO"
      propiedad-modelo-referenciado="numero" />
  </mapeo-referencia>
  ...
</mapeo-entidad>

```

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

```

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

```

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

```

<mapeo-referencia referencia-modelo="albaran">
  <detalle-mapeo-referencia
    columna-tabla="ALBARAN_FACTURA_AÑO"
    propiedad-modelo-referenciado="factura.año" />
  <detalle-mapeo-referencia
    columna-tabla="ALBARAN_FACTURA_NUMERO"
    propiedad-modelo-referenciado="factura.numero" />
  <detalle-mapeo-referencia
    columna-tabla="ALBARAN_TIPO"

```

```

        propiedad-modelo-referenciado="tipo.codigo"/>
    <detalle-mapeo-referencia
        columna-tabla="ALBARAN_NUMERO"
        propiedad-modelo-referenciado="numero"/>
</mapeo-referencia>

```

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

También es posible usar conversores cuando mapeamos referencias:

```

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

```

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

6.4 Mapeo propiedad multiple

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

La sintaxis de este tipo de mapeo es:

```

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

```


- (1) `propiedad-modelo` (obligada): Nombre de la propiedad que se quiere mapear.
- (2) `conversor` (uno, obligado): Clase con la lógica para hacer la conversión de Java a base de datos y viceversa. Ha de implementar `IMultipleConverter`.
- (3) `campo-cmp` (varios, obligado): Hace corresponder cada columna de la base de datos con una propiedad del conversor.

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

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

`DIAENTREGA`, `MESENTREGA` y `AÑOENTREGA` son las tres columnas que en la base de datos guardan la fecha de entrega, y `day`, `month` y `year` son propiedades que podemos encontrar en `Date3Converter`. Y aquí `Date3Converter`:

```
package org.openxava.converters;

import java.util.*;

import org.openxava.util.*;

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

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

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

```

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

public int getYear() {
    return year;
}

public int getDay() {
    return day;
}

public int getMonth() {
    return month;
}

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

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

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

```

```
}
```

El conversor tiene que implementar `IMultipleConverter` (1) lo que le obliga a tener un método `toJava()` (2) que a partir de lo contenido en sus propiedades (en este caso `year`, `month` y `day`) ha de devolver un objeto Java con el valor de la propiedad mapeada (en nuestro ejemplo el valor de la `fechaEntrega`); y el método `toDB()` (3) que recibe el valor de la propiedad Java (el de `fechaEntrega`) y tiene que desglosarlo para que se queda almacenado en las propiedades del conversor (`year`, `month` y `day`).

6.5 Mapeo de referencias a agregados

Una referencia a un agregado contiene información que en el modelo relacional se guarda en la misma tabla que la entidad principal. Por ejemplo si tenemos un agregado `Direccion` asociado a un `Cliente`, los datos de la dirección se guardan en la misma tabla que los del cliente. ¿Cómo se expresa eso en OpenXava? Es muy sencillo. En el modelo nos encontramos:

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

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

Sencillamente una referencia a un agregado, y para mapearlo lo hacemos así:

```
<mapeo-entidad tabla="XAVATEST@separator@CLIENTE">
    ...
    <mapeo-propiedad propiedad-modelo="direccion_calle" column-table="CALLE"/>
    <mapeo-propiedad
        propiedad-modelo="direccion_codigoPostal"
        column-table="CP" tipo-cmp="String">
        <conversor clase="org.openxava.converters.IntegerStringConverter"/>
    </mapeo-propiedad>
    <mapeo-propiedad
        propiedad-modelo="direccion_municipio" column-table="MUNICIPIO"/>
    <mapeo-referencia referencia-modelo="direccion_municipio">
        <detalle-mapeo-referencia
```

```

        column-tabla="PROVINCIA" propiedad-modelo-referenciado="codigo"/>
    </mapeo-referencia>
</mapeo-entidad>

```

Vemos como los miembros del agregado se mapean en el mapeo de la entidad que lo contienen, solo que ponemos como prefijo el nombre de la referencia a agregado con un subrayado (en esta caso `direccion_`). Podemos observar como podemos mapear referencias y propiedades y usar conversores.

6.6 Mapeo de agregados usados en colecciones

En el caso de que tengamos una colección de agregados, supongamos las líneas de una factura, obviamente la información de las líneas se guarda en una tabla diferente que la información de cabecera de la factura. En este caso los agregados han de tener su propio mapeo. Veamos el ejemplo de las líneas de una factura.

En la parte de modelo del componente `Factura` tenemos:

```

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

<agregado nombre="LineaFactura">
    <propiedad nombre="oid" tipo="String" clave="true" oculta="true">
        <calculador-valor-defecto
            clase="org.openxava.test.calculadores.CalculadorOidLineaFactura"
            al-crear="true"/>
        </calculador-valor-defecto>
    </propiedad>
    <propiedad nombre="tipoServicio">
        <valores-posibles>
            <valor-posible valor="especial"/>
            <valor-posible valor="urgente"/>
        </valores-posibles>
    </propiedad>
    <propiedad nombre="cantidad" tipo="int" longitud="4" requerido="true"/>
    <propiedad nombre="precioUnitario" estereotipo="DINERO" requerido="true"/>
    <propiedad nombre="importe" estereotipo="DINERO">
        <calculador clase="org.openxava.test.calculadores.CalculadorImporteLinea">
            <poner propiedad="precioUnitario"/>
            <poner propiedad="cantidad"/>
        </calculador>
    </propiedad>
</agregado>

```

```

</propiedad>
<referencia modelo="Producto" requerido="true"/>
<propiedad nombre="fechaEntrega" tipo="java.util.Date">
    <calculador-valor-defecto
        clase="org.openxava.calculators.CurrentDateCalculator"/>
</propiedad>
<referencia nombre="vendidoPor" modelo="Comercial"/>
<propiedad nombre="observaciones" estereotipo="MEMO"/>
</agregado>

```

Vemos una colección de LineaFactura que es un agregado, LineaFactura se ha de mapear así:

```

<mapeo-agregado agregado="LineaFactura" tabla="XAVATEST@separator@LINEAFACTURA">
    <mapeo-referencia referencia="factura">                (1)
        <detalle-mapeo-referencia
            columna-tabla="FACTURA_AÃ'O"
            propiedad-modelo-referenciado="aÃto"/>
        <detalle-mapeo-referencia
            columna-tabla="FACTURA_NUMERO"
            propiedad-modelo-referenciado="numero"/>
    </mapeo-referencia>
    <mapeo-propiedad propiedad-modelo="oid" columna-tabla="OID"/>
    <mapeo-propiedad propiedad-modelo="tipoServicio" columna-tabla="TIPOSERVICIO"/>
    <mapeo-propiedad
        propiedad-modelo="precioUnitario" columna-tabla="PRECIOUNITARIO"/>
    <mapeo-propiedad propiedad-modelo="cantidad" columna-tabla="CANTIDAD"/>
    <mapeo-referencia referencia="producto">
        <detalle-mapeo-referencia
            columna-tabla="PRODUCTO_CODIGO"
            propiedad-modelo-referenciado="codigo"/>
    </mapeo-referencia>
    <mapeo-propiedad-multiple propiedad-modelo="fechaEntrega">
        <conversor clase="org.openxava.converters.Date3Converter"/>
        <campo-cmp
            propiedad-conversor="day"
            columna-tabla="DIAENTREGA" tipo-cmp="int"/>
        <campo-cmp
            propiedad-conversor="month"
            columna-tabla="MESENTREGA" tipo-cmp="int"/>
        <campo-cmp
            propiedad-conversor="year"
            columna-tabla="AÃ'OENTREGA" tipo-cmp="int"/>
    </mapeo-propiedad-multiple>

```

```

    <mapeo-referencia referencia="vendidoPor">
        <detalle-mapeo-referencia
            columna-tabla="VENDIDOPOR_CODIGO"
            propiedad-modelo-referenciado="codigo"/>
    </mapeo-referencia>
    <mapeo-propiedad
        propiedad-modelo="observaciones" columna-tabla="OBSERVACIONES"
    </aggregate-mapping>

```

Los mapeos de agregado se ponen a continuación del mapeo de entidad, y debe haber tantos como agregados usados en colecciones. El mapeo de un agregado tiene exactamente las mismas posibilidades que el mapeo de entidad ya visto, con la salvedad de que necesitamos definir el mapeo de la referencia al objeto contenedor aunque ella no esté en el modelo. Esto es aunque nosotros no definamos en `LineaFactura` una referencia a `Factura`, el OpenXava la añade automáticamente y por ende nosotros en el mapeo tenemos que reflejarlo (1).

6.7 Conversores por defecto

Vemos como podemos declarar un conversor a cada mapeo de propiedad. Pero ¿qué pasa cuando no declaramos conversor? En realidad en OpenXava todas las propiedades (a excepción de las que son clave) tienen un conversor aunque no se indique explícitamente. Los conversores por defecto están definidos en el archivo `OpenXava/xava/default-converters.xml`, que tiene un contenido como este:

```

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

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

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

<converters>

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

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

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

```

```

        cmp-type="java.lang.Integer"/>

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

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

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

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

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

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

</converters>

```

Si usamos una propiedad de un tipo que no tenemos definido aquí por defecto se le asigna el conversor `NoConversionConverter`, que es un conversor tonto que no hace nada.

En el caso de las propiedades clave no se asigna conversor en absoluto, aplicar conversor a las propiedades claves puede ser problemático en ciertas circunstancias, pero si aun así lo queremos podemos declarar explícitamente en nuestro mapeo un conversor para una propiedad clave y se le aplicará.

Si queremos modificar el comportamiento de los conversores por defecto para nuestra aplicación no debemos modificar este archivo sino crear uno llamado *converters.xml* o *conversores.xml* en el directorio *xava* de nuestro proyecto. Podemos asignar también conversor por defecto a un estereotipo (usando `<para-estereotipo/>` o `<for-stereotype/>`).

6.8 Filosofía del mapeo objeto-relacional

OpenXava ha nacido y se ha desarrollado en un entorno en el que nos hemos visto obligados a trabajar con bases de datos existentes sin poder modificar su estructura, esto hace que:

- Ofrezca gran flexibilidad para mapear contra bases de datos legadas.
- No ofrezca algunas características propias de la OOT y que requieren poder manipular el esquema, como por ejemplo el soporte de herencia o consultas polimórficas.

Otras característica importante del mapeo de OpenXava es que las aplicaciones son 100% portables entre JBoss CMP2 y Websphere CMP2 sin tener que reescribir nada por parte del desarrollador. Además, la portabilidad entre Hibernate y EJB2 de una aplicación es muy alta, los mapeos y todos los controladores automáticos son portables al 100%, obviamente el código propio escrito con EJB2 o Hibernate no lo es tanto.

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 *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 sobrescrito 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.

7.1 Variables de entorno y objetos de sesión

Definir variables de entorno y objetos de sesión es muy sencillo, podemos observar las que hay definidas en *OpenXava/xava/controllers.xml*:

```
<env-var name="XAVA_SEARCH_ACTION" value="CRUD.searchByViewKey"/>
<env-var name="XAVA_LIST_ACTION" value="List.viewDetail"/>

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

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

Vemos una sintaxis sencilla, nombre y valor para las variables de entorno y nombre, clase y valor para los objetos de sesión. En cuanto al estilo de los nombre ponemos delante el prefijo de nuestra aplicación, como estos son las variables y objetos del núcleo de OpenXava ponemos `XAVA_` o `xava_`; y las variables de entorno las ponemos en mayúsculas y los objetos en minúsculas.

Estos objetos y variables son los que usa OpenXava para funcionar, aunque es bastante normal que nosotros usemos algunos de estos desde nuestras acciones. Si queremos crear nuestras propias variables y objetos lo podemos hacer en nuestro propio *controladores.xml* en el directorio *xava* de nuestro proyecto.

7.2 El controlador y sus acciones

La sintaxis de un controlador es:

```
<controlador
    nombre="nombre"           (1)
>
    <hereda-de ... /> ...     (2)
    <accion ... /> ...       (3)
</controlador>
```

(0)nombre (obligado): Nombre del controlador.

(1)hereda-de (varios, opcional): Permite usar herencia múltiple, para que este controlador herede todas las acciones de otro (u otros) controlador.

(2)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)
    por-defecto="nunca|si-posible|casi-siempre|siempre" (9)
    cuesta="true|false"       (10)
    confirmar="true|false"    (11)
    atajo-de-teclado="atajo-de-teclado" (12) nuevo en v2.0.1
>
```

<code><poner ... /> ...</code>	(13)
<code><usa-objeto ... /> ...</code>	(14)
<code></accion></code>	

- (1) **nombre** (obligado): Nombre identificativo de la acción tiene que ser único dentro del controlador, pero puede repetirse el nombre en diferentes controladores. Cuando referenciamos 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) **por-defecto** (opcional): Indica el peso de esta acción a la hora de seleccionar cual es la acción por defecto. La acción por defecto es la que se ejecuta cuando el usuario pulsa ENTER. Por defecto vale `nunca`.
- (10) **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`.
- (11) **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`.
- (12) **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)
- (13) **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.
- (14) **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 los 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.acciones.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>
</controlador>
```

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 Remarks. La barra de botones tendrá el siguiente aspecto:



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.

7.3 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/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 hacer altas, bajas, modificaciones y consultas).

Podemos usar la herencia para refinar la forma de trabajar de un controlador estándar, como sigue:

```
<controlador nombre="Familias">
  <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 descende `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.

7.4 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">
        <usa-objeto nombre="xava_tab"/> (3)
    </accion>

```

Ponemos `mode="list"` para que solo aparezca en modo lista (1), y usamos el objeto de sesión `xava_tab` que es el que nos permite acceder a los datos visualizados en la lista (3). Ya que esta acción borra registros hacemos que el usuario tenga que confirmar antes de ejecutarse (2).

Programar la acción sería así:

```

package org.openxava.actions;

import java.util.*;

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

/**
 * @author Javier Paniza
 */

public class DeleteSelectedAction extends BaseAction implements IModelAction { // (1)

    private Tab tab; // (2)
    private String model;

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

```

```

        }
    }
    getTab().deselectAll(); // (6)
    resetDescriptionsCache(); // (7)
}

}

public Tab getTab() { // (2)
    return tab;
}

public void setTab(Tab web) { // (2)
    tab = web;
}

public void setModel(String modelName) { // (8)
    this.model = modelName;
}

}

```

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 descende de `BaseAction` e implementa `IModelAction`, al descender de `BaseAction` 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 método `setModel()` (8) con el que recibirá el nombre del modelo (del componente OpenXava) antes de ejecutarse.

Tenemos una propiedad `tab` de tipo `org.openxava.tab.Tab` (2), y esto unido a que hemos puesto al definir nuestra acción...

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

... nos va a permitir manipular la lista de objetos visualizados. Por ejemplo, con `tab.getSelected()` (3) 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` (2). `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 procesos automáticos pero cuando queremos hacer cosas específicas es mejor usar directamente los objetos del modelo (los

POJOs o EJB generados por OpenXava) . 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 in *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()` (7) 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`.

7.5 Sobrecribir 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>
```

```
</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 SearchByKeyAction { // (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 EJB, JDO o Hibernate) y llenar la vista. Muchas veces lo más práctico es hacer que extienda de `SearchByKeyAction` (1) y dentro del `execute()` hacer un `super.execute()` (2).

OpenXava viene con 2 acciones de búsquedas:

- `CRUD.searchByKey`: 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 al-cambiar asociadas a las propiedades de la vista.

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`.

7.6 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.

7.7 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.

7.8 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.

7.9 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 `INavigationAction` (con `ICustomViewAction` 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).

7.10 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";
}

private ISubfamilia2 getSubfamilia() throws Exception {
    if (subfamilia == null) {
        int codigoSubfamilia = getView().getValueInt("subfamilia.codigo");
        // Usando Hibernate, the usual case
        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 sobrescribir 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 EJBs sé cuidadoso y no hagas un bucle sobre una colección de EJB 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 tiene que estar en el classpath. Puedes tener para esto una carpeta de código fuente llamada *informes* en tu proyecto.

Por defecto el informe es visualizado en una ventana emergente, pero si lo deseas puedes sobrescribir 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.

7.11 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 tecnológicamente 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.

7.12 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.
- `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 `IJBCCalculator` (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 EJBs. 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 ocultas 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.

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 ... /> ...             (3)
</aplicacion>
```

(1)nombre (obligado): Nombre de la aplicación.

(2)etiqueta (opcional): Mucho **mejor** usar archivos *i18n*.

(3)modulo (varios, obligado): 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 nuestra 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.

8.1 Un módulo típico

Definir un módulo sencillo puede ser como sigue:

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

También se genera un portlet para poder desplegar el módulo como un portlet JSR-168 en un portal Java.

8.2 Módulo con solo detalle

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

```
<modulo nombre="FacturasSinLista">
  <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.

8.3 Módulo con solo lista

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

```
<modulo nombre="FacturasSoloLista">
  <modelo nombre="Factura"/>
  <controlador nombre="Typical"/>
  <controlador nombre="ListOnly"/>      (1)
  <controlador-modo nombre="Void"/>      (2)
</modulo>
```

El controlador de modo `Void` (2) es para que no aparezcan los vínculos “detalle – lista”. Además al definir `ListOnly` (1) como controlador el módulo cambia a modo lista al iniciar, por lo tanto éste es un módulo de solo lista.

8.4 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.

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

9.1 Introducción a AOP

AOP (Programación Orientada a Aspectos) introduce una nueva forma de reutilizar código. Realmente complementa algunas deficiencias de la programación orientada a objetos tradicional.

¿Qué problema resuelve AOP? A veces tenemos funcionalidad que es común a un grupo de clases pero usar herencia no es práctico (en Java solo contamos con herencia simple) ni ético (porque se rompe la relación *es-un*). Además el sistema puede estar ya escrito, o quizá necesitamos poder incluir o no la funcionalidad bajo demanda. AOP es una manera fácil de resolver estos problemas.

¿Qué es un aspecto? Un aspecto es un puñado de código que puede ser esparcido a nuestro gusto por la aplicación.

El lenguaje Java ofrece un soporte completo de AOP mediante el proyecto AspectJ.

OpenXava añade algún soporte para el concepto de *aspectos* desde la versión 1.2.1. De momento el soporte es bastante limitado y OpenXava está todavía muy lejos de un marco de trabajo AOP al uso, pero el soporte de aspectos dentro de OpenXava se ha mostrado útil.

9.2 Definición de aspectos

El archivo *aspectos.xml* dentro de la carpeta *xava* de nuestro proyecto se usa para definir los aspectos.

Su sintaxis es:

```
<aspectos>
  <aspecto ... /> ...           (1)
  <aplicar ... /> ...           (2)
</aspectos>
```

(1) *aspecto* (varios, opcional): Para definir aspectos.

(2) *aplicar* (varios, opcional): Para aplicar los aspectos definidos a los modelos seleccionados.

Con *aspecto* (1) podemos definir un aspecto (es decir un grupo de características) con un nombre, y usando *aplicar* (2) conseguimos que ese conjunto de modelos (entidades o agregados) obtengan esas características automáticamente.

Veamos la sintaxis para un aspecto:

```
<aspecto
  nombre="nombre"               (1)
>
  <calculador-poscrear .../> ... (2)
```

```

    <calculador-poscargar .../> ...      (3)
    <calculador-posmodificar .../> ...   (4)
    <calculador-preborrar .../> ...     (5)
</aspecto>

```

- (1)nombre (obligado): Nombre para este aspecto. Tiene que ser único.
- (2)calculador-poscrear (varios, opcional): Todos los modelos con este aspecto tendrán este calculador-poscrear implícitamente.
- (3)calculador-poscargar (varios, opcional): Todos los modelos con este aspecto tendrán este calculador-poscargar implícitamente.
- (4)calculador-posmodify (varios, opcional): Todos los modelos con este aspecto tendrán este calculador-posmodify implícitamente.
- (5)calculador-preborrar (varios, opcional): Todos los modelos con este aspecto tendrán este calculador-preborrar implícitamente.

Además, tenemos que asignar el aspecto que hemos definido a nuestros modelos. La sintaxis para eso es:

```

<aplicar
    aspecto="aspecto"           (1)
    para-modelos="modelos"     (2)
    excepto-para-modelos="modelos" (3)
/>

```

- (1)aspecto (obligado): El nombre del aspecto que queremos aplicar.
- (2)para-modelos (opcional): Una lista de modelos, separados por coma, para aplicarles este aspecto. Es exclusivo con el atributo excepto-para-modelos.
- (3)excepto-para-modelos (opcional): Una lista de modelos, separados por comas, para excluir cuando se aplique este aspecto. De esta forma el aspecto es aplicado a todos los modelos menos los indicados. Es exclusivo con el atributo para-modelos.

Si no usamos ni para-modelos ni excepto-para-modelos entonces el aspecto se aplicará a todos los modelos de la aplicación. Los modelos son los nombres de componente (para sus entidades) o agregados.

Un ejemplo simple puede ser:

```

<aspecto nombre="MiAspecto">
    <calculador-poscrear
        clase="com.miempresa.miaplicacion.calculadores.MiCalculador"/>
</aspecto>
<aplicar aspecto="MiAspecto"/>

```

De esta forma tan sencilla podemos hacer que cuando un nuevo objeto se cree (se grabe en la base de datos por primera vez) la lógica en MiCalculador sea ejecutada. Y esto para todos los modelos.

Como se ve, solo unos pocos calculadores son soportados de momento. Esperamos extender las posibilidades de los *aspectos* en OpenXava en el futuro. De todas formas, estos calculadores ahora ofrecen posibilidades interesantes. Veamos un ejemplo en la siguientes sección.

9.3 AccessTracking: Una aplicación práctica de los aspectos

La distribución actual de OpenXava incluye el proyecto *AccessTracking*. Este proyecto define un aspecto que nos permite llevar la pista de todos los acceso a la información en nuestra aplicación. Actualmente, este proyecto permite que nuestras aplicaciones cumplan con la Ley de Protección de Datos española, incluyendo los datos con un nivel de seguridad alto. Aunque es lo suficientemente genérico para ser útil en una amplia variedad de circunstancia.

9.3.1 La definición del aspecto

Podemos encontrar la definición del aspecto en *AccessTracking/xava/aspects.xml*:

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

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

<!-- AccessTracking -->

<aspects>

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

</aspects>
```

Cuando aplicamos este aspecto a nuestro componentes el código de *AccessTrackingCalculator*

es ejecutado cada vez que un objeto es creado, cargado, modificado o borrado. `AccessTrackingCalculator` escribe un registro en una tabla de la base de datos con información acerca del acceso.

Para poder aplicar este aspecto necesitamos escribir en nuestro *aspectos.xml* algo como esto:

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

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

<aspectos>

    <aplicar aspecto="AccessTracking" para-modelos="Almacen, Factura"/>

</aspectos>
```

De esta forma este aspecto es aplicado a `Almacen` y `Factura`. Todos los accesos a estas entidades serán registrados en una tabla de base de datos.

9.3.2 Configurar AccessTracking

Si queremos usar el aspecto `AccessTracking` en nuestro proyecto hemos de seguir los siguientes pasos de configuración:

- Añadir *AccessTracking* como proyecto referenciado.
- Crear la tabla en nuestra base de datos para almacenar el registro de los accesos. Podemos encontrar los `CREATE TABLEs` en el archivo *AccessTracking/data/access-tracking-db.script*.
- Hemos de incluir la propiedad `hibernate.dialect` en nuestros archivos de configuración. Puedes ver ejemplos de esto en *OpenXavaTest/jboss-hypersonic.properties* y otros archivos *OpenXavaTest/xxx.properties*.
- Dentro del proyecto *AccessTracking* necesitamos seleccionar una configuración (editando *build.xml*) y regenerar código hibernate (usando la tarea `ant generateHibernate`) para *AccessTracking*.
- Editamos el archivo de nuestro proyecto *build/ejb/META-INF/MANIFEST.MF* para añadir los siguientes jars al classpath: `./lib/tracking.jar ./lib/ehcache.jar ./lib/antlr.jar ./lib/asm.jar ./lib/cglib.jar ./lib/hibernate3.jar ./lib/dom4j.jar`. (Este paso no es necesario si solo usamos POJOs, y no usamos EJB CMP2, nuevo en v2.0)

También necesitamos modificar la tarea `ant crearJarEJB` (solo si usamos EJB2 CMP) y `desplegarWar` de nuestro *build.xml* de esta forma:

```
<target name="crearJarEJB">    <!-- 'crearJarEJB' solo si usamos EJB2 CMP -->
    ...
    <ant antfile="../AccessTracking/build.xml" target="createEJBTracker"/>
</target>

<target name="desplegarWar">
```

```
<ant antfile="../../AccessTracking/build.xml" target="createTracker"/>
...
</target>
```

Después de todo esto, hemos de aplicar el aspecto en nuestra aplicación. Creamos un archivo *xava/aspectos.xml* en nuestro proyecto:

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

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

<aspectos>

    <aplicar aspecto="AccessTracking"/>

</aspectos>
```

Ahora solo nos queda desplegar el war para nuestro proyecto. (*nuevo en v2.0*)

En el caso de que usemos EJB2 CMP tenemos que regenerar código, desplegar EJB y desplegar el war para nuestro proyecto.

Los accesos son registrados en una tabla con el nombre TRACKING.ACCESS. Si lo deseamos podemos desplegar el módulo web o el portlet del proyecto *AccessTracking* para tener una aplicación web para examinar los accesos.

Para más detalle podemos echar un vistazo al proyecto *OpenXavaTest*.

10.1 Relaciones de muchos-a-muchos

En OpenXava no existe el concepto directo de relación muchos-a-muchos, en lugar de eso en OpenXava solo existen colecciones. Aun así modelar una relación muchos-a-muchos con OpenXava es fácil. Solo necesitamos definir colecciones en ambas partes de la relación.

Por ejemplo, si tenemos clientes y provincias, y un cliente puede trabajar en varias provincias, y, obviamente, en una provincia pueden trabajar varios clientes, tenemos un caso de relación muchos-a-muchos (usando la nomenclatura relacional). Suponiendo que tenemos una tabla CLIENTE (sin referencia a provincia), una tabla PROVINCIA (sin referencia a clientes) y una tabla CLIENTE_PROVINCIA (para vincular ambas tablas), este caso podemos modelarlo así:

```
<componente nombre="Cliente">
  <entidad>
    ...
    <coleccion nombre="provincias"> (1)
      <referencia modelo="ClienteProvincia"/>
    </coleccion>
    ...
  </entidad>

  <agregado nombre="ClienteProvincia"> (2)
    <referencia nombre="cliente" clave="true"/> (3)
    <referencia nombre="provincia" clave="true"/> (4)
  </agregado>
  ...
</componente>
```

Estamos definiendo en Cliente una colección de agregados (1), cada agregado (ClienteProvincia) (2) contiene una referencia a Provincia (4) y, por supuesto, una referencia a su entidad contenedora (Cliente) (3).

Ahora podemos mapear la colección de la forma habitual:

```
<componente nombre="Cliente">
  ...
  <mapeo-agregado agregado="ClienteProvincia" tabla="CLIENTE_PROVINCIA">
    <mapeo-referencia referencia="cliente">
      <detalle-mapeo-referencia
        column-tabla="CLIENTE" (1)
      </detalle-mapeo-referencia>
    </mapeo-referencia>
  </mapeo-agregado>
</componente>
```



```

        propiedad-modelo-referenciado="codigo"/>
    </mapeo-referencia>
    <mapeo-referencia referencia="provincia">
        <detalle-mapeo-referencia
            columna-tabla="PROVINCIA" (2)
            propiedad-modelo-referenciado="id"/>
    </mapeo-referencia>
</mapeo-agregado>
</componente>

```

ClienteProvincia lo mapeamos a CLIENTE_PROVINCIA, una tabla que contiene dos columnas, una para apuntar a CLIENTE (1) y otra para apuntar a PROVINCIA (2).

A nivel de modelo y mapeo ya lo tenemos todo listo, pero la interfaz gráfica que genera OpenXava por defecto es un poco fea en este caso. Aunque con los siguientes retoques en la parte de la vista nuestra colección muchos-a-muchos puede quedar bastante bien:

```

<componente nombre="Cliente">
    ...
    <vista>
        ...
        <vista-coleccion coleccion="provincias">
            <propiedades-lista>
                provincia.id, provincia.nombre (1)
            </list-properties>
        </vista-coleccion>

        <miembros>
            ...
            provincias
            ...
        </miembros>

    </vista>

    <vista modelo="ClienteProvincia">
        <vista-referencia referencia="provincia" marco="false"/> (2)
    </vista>
    ...
</componente>

```

En esta vista podemos ver como definimos explícitamente (1) las propiedades a mostrar en la lista de la colección provincias. Esto es necesario porque tenemos que mostrar las propiedades de Provincia, no las de ClienteProvincia. Adicionalmente, definimos que la referencia a Provincia en ClienteProvincia se muestre sin marcos (2), de esta forma evitamos dos feos

marcos anidados.

De esta manera podemos definir una colección que mapea a una relación de muchos a muchos en la base de datos. Si queremos que la relación sea bidireccional solo tenemos que crear una colección `clientes` en la entidad `Provincia`, esta colección puede ser de type agregado `ProvinciaCliente` y tiene que mapearse a la tabla `CLIENTE_PROVINCIA`. Todo exactamente igual que en el ejemplo aquí mostrado.

10.2 Programar con Hibernate

Podemos usar las APIs de Hibernate en cualquier parte de una aplicación OpenXava, esto es, dentro de calculadores, acciones, filtros, etc.

Para facilitar el uso de Hibernate OpenXava provee la clase `XHibernate`. Por ejemplo, si queremos guardar un objeto en la base de datos usando Hibernate, la manera típica sería:

```
Session sesion = HibernateUtil.getSessionFactory().getCurrentSession();
sesion.beginTransaction();
Cliente cliente = ... ;
sesion.save(cliente);
sesion.getTransaction().commit();
sesion.close();
```

Pero, dentro de OpenXava y usando `XHibernate` podemos escribir:

```
Cliente cliente = ... ;
XHibernate.getSession().save(cliente);
```

Nada más.

La primera vez que llamamos a `XHibernate.getSession()` una sesión es creada y asignada al hilo actual y una transacción es creada también; la siguiente vez que lo llamemos, la misma sesión de Hibernate es usada. Al final del ciclo completo de la ejecución de la acción, OpenXava confirma automáticamente la transacción y cierra la sesión. Además, `XHibernate.getSession()` funciona bien dentro y fuera de un entorno CMT.

Podemos, opcionalmente, confirmar la transacción en cualquier momento llamando a `XHibernate.commit()`, si después de esto llamamos a `XHibernate.getSession()` una nueva sesión y transacción son creadas.

Podemos ver más acerca de esto consultando el API de `org.openxava.hibernate.XHibernate`.

10.3 Vistas JSP propias y taglibs de OpenXava

Obviamente la mejor forma de crear interfaces de usuario es usando la sección vista de los componentes tal y como se ve 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.

10.3.1 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-proprijs/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 del componente; 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 explícitamente. Podemos poner nuestro JSP dentro de la carpeta `web/jsp-proprijs` de nuestro proyecto, y este JSP puede ser así:

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

<table>
<tr>
  <td>Código: </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.

10.3.2 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*).

10.3.3 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 determinar el estilo de visualización podemos usar directamente las siguientes marcas: `<xava:button/>`, `<xava:image/>` o `<xava:link/>` similares a `<xava:action/>`.